

Hiago de Oliveira Mendes e Lucas Sales Salvo Petrucci

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 Petrucci

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 Petrucci

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, 19 de maio 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

O crescimento exponencial da web, com milhares de sites surgindo diariamente, tem elevado a complexidade do desenvolvimento de aplicações web e evidenciado a importância da escolha adequada da estratégia de renderização de conteúdo. Dentre as principais abordagens estão o *Client-Side Rendering* (CSR) e o *Server-Side Rendering* (SSR), cada uma com características distintas que impactam diretamente a performance, a experiência do usuário (UX), a escalabilidade e a otimização para motores de busca (SEO). Este trabalho tem como objetivo principal realizar uma análise comparativa entre CSR e SSR, destacando seus efeitos técnicos e funcionais em aplicações web modernas. A justificativa fundamenta-se na lacuna de estudos de caso práticos e aprofundados, especialmente no contexto nacional, que analisem criticamente os impactos reais da adoção dessas abordagens. A metodologia adotada envolveu fundamentação teórica, mapeamento sistemático da literatura com uso da técnica *PICOC*, e a implementação de um estudo de caso realista, com coleta de métricas de desempenho, tempo de carregamento, impacto em SEO e UX. Os resultados apontaram que, embora o CSR seja mais vantajoso em aplicações altamente interativas, o SSR apresenta melhor desempenho inicial e maior indexabilidade. Conclui-se que a escolha entre CSR e SSR deve considerar o perfil do sistema, a infraestrutura disponível e os objetivos estratégicos do projeto, sendo muitas vezes recomendável a adoção de soluções híbridas.

Palavras-chave: Renderização do lado do cliente, CSR, Renderização do lado do servidor, SSR, Renderização Web, Desempenho, SEO, UX.

Abstract

The exponential growth of the web, with thousands of new websites emerging daily, has increased the complexity of web application development and highlighted the importance of choosing the appropriate content rendering strategy. Among the main approaches are Client-Side Rendering (CSR) and Server-Side Rendering (SSR), each with distinct characteristics that directly affect performance, user experience (UX), scalability, and search engine optimization (SEO). This work aims to conduct a comparative analysis between CSR and SSR, emphasizing their technical and functional effects on modern web applications. The justification lies in the lack of practical and in-depth case studies, especially in the Brazilian context, that critically examine the real impact of adopting these strategies. The methodology included a theoretical foundation, a systematic literature review using the PICOC technique, and the implementation of a realistic case study, collecting performance metrics, load times, SEO impact, and UX data. The results show that while CSR is more advantageous for highly interactive applications, SSR provides better initial performance and greater indexability. It is concluded that the choice between CSR and SSR should consider the system profile, infrastructure, and strategic goals, with hybrid solutions often being the most appropriate.

Keywords: Client-side rendering, CSR, Server-side rendering, SSR, Web Rendering, Performance, SEO, UX.

Lista de Figuras

Figura 1 – Comunicação entre cliente e servidor.	16
Figura 2 – Analogia entre HTML, CSS e JavaScript e os componentes de um corpo humano.	21
Figura 3 – Comparativo entre estratégias de renderização	24
Figura 4 – Etapas do método de renderização no lado do cliente	26
Figura 5 – Etapas do método de renderização no lado do servidor	28
Figura 6 – Etapas do método de geração estática de páginas (SSG)	31
Figura 7 – Funcionamento do modelo Incremental Static Regeneration (ISR)	33
Figura 8 – Funcionamento da estratégia Deferred Static Generation (DSG)	34
Figura 9 – Tempo de Rastreamento e Posicionamento da Página	39
Figura 10 – Número de publicações ao longo dos anos	51
Figura 11 – Distribuição das publicações por país	52

Lista de quadros

Quadro 1 – Estrutura PICOC aplicada à pesquisa	49
Quadro 2 – Expressão de busca utilizada	50
Quadro 3 – Artigos selecionados sobre estratégias de renderização e desempenho em aplicações web	53
Quadro 4 – Cronograma de execução da primeira etapa do TCC	57
Quadro 5 – Cronograma de execução da segunda etapa do TCC	58

Lista de codigos

2.1	Exemplo de HTML mínimo em aplicação Angular com CSR	27
2.2	Exemplo de HTML mínimo em aplicação Next.js com SSR	30
2.3	Exemplo de HTML estático gerado com SSG	31
2.4	Exemplo de revalidação de conteúdo com ISR no Next.js	33
2.5	Exemplo de definição de página DSG no Gatsby	35

Siglas

CSR	Client-Side Rendering
CSS	Cascading Style Sheets
DOM	Document Object Model
DSG	Deferred Static Generation
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ISR	Incremental Static Regeneration
MPA	Multi Page Application
QUIC	Quick UDP Internet Connections
SEO	Search Engine Optimization
SPA	Single Page Application
SSG	Static Site Generation
SSL	Secure Sockets Layer
SSR	Server-Side Rendering
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UX	User Experience

Sumário

1	INTRODUÇÃO	12
1.1	Problema e contexto	12
1.2	Justificativa	13
1.3	Objetivos	14
1.3.1	Objetivo Geral	14
1.3.2	Objetivos Específicos	14
2	FUNDAMENTAÇÃO TEÓRICA	15
2.1	Fundamentos de Desenvolvimento Web	15
2.1.1	Arquitetura Cliente-Servidor	15
2.1.2	Protocolo HTTP	16
2.1.3	HTML, CSS e JavaScript	19
2.2	Renderização na Web	21
2.2.1	Single Page Application (SPA) e Multi Page Application (MPA)	21
2.2.2	<i>Client-Side Rendering</i> (CSR)	25
2.2.3	<i>Server-Side Rendering</i> (SSR)	27
2.2.4	<i>Static Site Generation</i> (SSG)	30
2.2.5	<i>Incremental Static Regeneration</i> (ISR)	32
2.2.6	<i>Deferred Static Generation</i> (DSG)	33
2.3	Frameworks Web	35
2.3.1	Bibliotecas JavaScript	35
2.3.2	Frameworks para CSR	36
2.3.3	Meta-frameworks para SSR	36
2.3.4	Comparativo entre Frameworks CSR e SSR	38
2.4	Experiência do Usuário	38
2.4.1	<i>Search Engine Optimization</i> (SEO)	38
2.4.2	<i>Velocidade de carregamento</i>	39
2.4.3	Interatividade	40
2.4.4	Acessibilidade	40
3	METODOLOGIA	42
3.1	Levantamento Teórico	42

3.2	Mapeamento Sistemático da Literatura	43
3.3	Estudo de Caso Prático	44
3.4	Coleta de Dados e Testes	44
3.5	Síntese e Discussão	45
4	TRABALHOS RELACIONADOS	47
4.1	Questões de pesquisa	47
4.2	Estratégia de busca	47
4.3	Quadro PICOC	49
4.3.1	Expressão de busca	50
4.4	Estratégia de seleção	50
4.5	Caracterização de pesquisa	50
4.6	Artigos selecionados	52
4.7	Resultados e Discussão	53
4.7.1	Q1: De que maneira a escolha entre CSR e SSR influencia a experiência do usuário?	54
4.7.2	Q2: Como as abordagens CSR e SSR afetam métricas de performance, tempo de carregamento e tempo até a interatividade em aplicações web?	54
4.7.3	Q3: Quais são os principais desafios e <i>trade-offs</i> na implementação de CSR e SSR?	55
4.7.4	Q4: Quais trabalhos relacionados existem na literatura que abordam recomendações sobre quando usar o CSR ou o SSR?	55
5	CRONOGRAMA	57
6	ESTUDO DE CASO	59
6.1	Contexto	59
6.2	Ciclo de Vida do Desenvolvimento de Software	59
7	RESULTADOS E DISCUSSÕES	60
7.1	Resultados	60
8	CONCLUSÃO	61
	REFERÊNCIAS	62

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, 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.

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 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](#)).

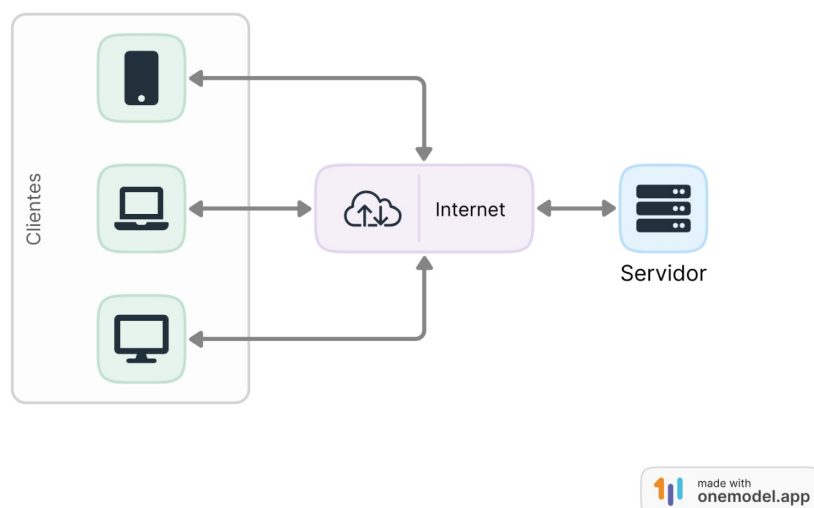
2.1.1 Arquitetura Cliente-Servidor

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 1 ilustra, de forma simplificada, esse fluxo de comunicação entre cliente e servidor.

Figura 1 – 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).

2.1.2 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 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*¹.

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](#)).

2.1.3 [HTML](#), [CSS](#) e JavaScript

O desenvolvimento frontend, conforme definido por [Amazon Web Services \(2024\)](#), refere-se à camada de apresentação de uma aplicação web — a interface gráfica com a qual os usuários interagem diretamente, composta por menus, botões, formulários e outros elementos visuais. Essa camada baseia-se em um conjunto de tecnologias fundamentais que operam em conjunto para fornecer estrutura, estilo e interatividade às páginas: [HTML](#), [CSS](#) e JavaScript. Cada uma dessas linguagens desempenha um papel específico e complementar, sendo essenciais tanto em abordagens tradicionais quanto em técnicas modernas como o [CSR](#).

HyperText Markup Language (HTML) é a linguagem de marcação padrão para a criação da estrutura de páginas web. Através de um conjunto de elementos (ou *tags*), o [HTML](#) organiza e define o conteúdo exibido ao usuário, como textos, imagens, links, formulários e tabelas. Além de estruturar visualmente o documento, o [HTML](#) também confere semântica aos elementos, facilitando a indexação por motores de busca e promovendo acessibilidade para leitores de tela. Elementos como `<header>`, `<main>`, `<article>` e `<footer>` exemplificam essa função semântica ([BALLERINI, 2023](#)).

Cascading Style Sheets (CSS) é a linguagem responsável pela estilização das páginas web. Com o [CSS](#), define-se a aparência dos elementos estruturados no [HTML](#), controlando propriedades visuais como cores, fontes, espaçamentos, tamanhos e posicionamentos. O [CSS](#) permite ainda a construção de layouts complexos e responsivos, adaptando o conteúdo para diferentes tamanhos de tela e dispositivos. A separação entre estrutura ([HTML](#)) e estilo ([CSS](#)) é um dos pilares das boas práticas em desenvolvimento web, promovendo manutenibilidade, reutilização e modularidade do código.

Entre os recursos modernos do [CSS](#), destacam-se os seletores avançados, variáveis [CSS](#), pseudo-classes, animações e as funcionalidades de *Flexbox* e *Grid*, que facilitam a

¹ 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](#))

criação de interfaces ricas e adaptáveis (KATTAH, 2023).

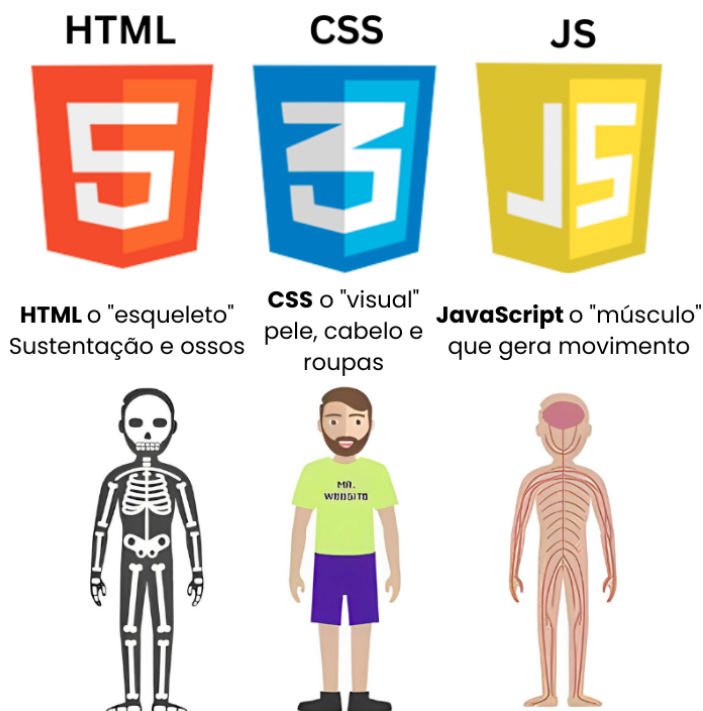
JavaScript é uma linguagem de programação interpretada, orientada a objetos e baseada em eventos, amplamente utilizada para adicionar interatividade e dinamismo às páginas web. Por meio da manipulação da *Document Object Model (DOM)*, permite implementar funcionalidades como respostas a cliques, envio de formulários, movimentações do mouse, digitação, animações, validações e atualizações em tempo real, enriquecendo significativamente a experiência do usuário (BALLERINI, 2023). Além disso, possibilita o carregamento assíncrono de dados com a técnica *AJAX (Asynchronous JavaScript and XML)*, evitando recarregamentos completos da página.

JavaScript é uma das três principais tecnologias da World Wide Web, juntamente com **HTML** e **CSS**, sendo essencial tanto em abordagens tradicionais quanto modernas. Nas aplicações baseadas em **CSR**, essa linguagem tem papel central, pois a renderização das páginas ocorre diretamente no navegador do usuário. Com a evolução do ecossistema JavaScript, surgiram bibliotecas e frameworks robustos como a biblioteca React e os frameworks Vue.js e Angular que facilitam o desenvolvimento de aplicações complexas com componentes reutilizáveis e gerenciamento eficiente de estado.

Adicionalmente, o JavaScript também pode ser executado no lado do servidor (**SSR**) por meio de ambientes como o Node.js — uma plataforma de código aberto e multiplataforma baseada em eventos e não bloqueante, ideal para aplicações escaláveis e em tempo real (NODE..., 2025; JAVASCRIPT, 2025). Isso permite o desenvolvimento de aplicações completas utilizando uma única linguagem em ambas as camadas, cliente e servidor.

A interação entre essas três tecnologias pode ser compreendida por meio de uma analogia: o **HTML** representa a estrutura de um corpo (esqueleto), o **CSS** corresponde à sua aparência externa (pele, roupas, estilo), enquanto o JavaScript age como os músculos e o sistema nervoso, controlando os movimentos e respostas interativas da aplicação. A Figura 2 ilustra essa relação.

Figura 2 – Analogia entre HTML, CSS e JavaScript e os componentes de um corpo humano.



Fonte: (KATTAH, 2023).

Portanto, o domínio dessas três tecnologias é indispensável para qualquer desenvolvedor web. Elas formam o alicerce sobre o qual se constroem interfaces acessíveis, performáticas e envolventes, sendo empregadas tanto em aplicações renderizadas no servidor (SSR) quanto no cliente (CSR), com adaptações específicas conforme a abordagem escolhida.

2.2 Renderização na Web

A renderização na Web diz respeito ao processo de transformar dados em conteúdo visual interpretável pelo navegador. A escolha sobre onde e como essa renderização será realizada (seja no cliente, no servidor ou em tempo de build), isso impacta diretamente métricas como tempo de carregamento, interatividade e indexação por mecanismos de busca (OSMANI; MILLER, 2025).

2.2.1 Single Page Application (SPA) e Multi Page Application (MPA)

As arquiteturas Single Page Application (SPA) e Multi Page Application (MPA) representam duas abordagens distintas para a estrutura de navegação e carregamento de conteúdo em aplicações web.

As **SPAs** são aplicações em que a navegação entre páginas ocorre sem recarregamentos completos do navegador. Após o carregamento inicial, todo o conteúdo adicional é gerenciado dinamicamente com JavaScript, o que proporciona uma experiência mais fluida e interativa. Essa abordagem é comumente utilizada em conjunto com a renderização no lado do cliente (**CSR**) e frameworks como React, Angular ou Vue.js (**EMADAMERHO-ATORI, 2024**).

Já as **MPAs** seguem o modelo tradicional de navegação, em que cada clique em um link leva a uma nova requisição **HTTP** e recarregamento completo da página. Essa arquitetura é naturalmente mais compatível com a renderização no lado do servidor (**SSR**) e favorece aspectos como **SEO**, acessibilidade e previsibilidade de comportamento (**OSMANI; MILLER, 2025**).

A escolha entre **SPA** e **MPA** está diretamente ligada à estratégia de renderização adotada. As **SPAs** tendem a oferecer experiências mais ricas e responsivas, mas exigem cuidados extras com desempenho e indexação. Por outro lado, as **MPAs** são mais robustas em cenários com grande volume de tráfego e requisitos de otimização para mecanismos de busca.

Conceitos e Terminologia

Segundo **Osmani e Miller (2025)**, é importante distinguir os principais modelos:

Client-Side Rendering (CSR) O conteúdo da aplicação é gerado dinamicamente no navegador, utilizando JavaScript. A página HTML inicial contém apenas uma estrutura básica com os scripts necessários para montar a interface após o carregamento. É comum em aplicações do tipo SPA (Single Page Application).

Server-Side Rendering (SSR) O servidor monta todo o conteúdo da página em HTML antes de enviá-lo ao cliente. Isso permite uma exibição mais rápida do conteúdo, mesmo em conexões lentas, e melhora a indexação por mecanismos de busca.

Static Site Generation (SSG) As páginas são geradas de forma estática em tempo de build, com base em dados disponíveis no momento da compilação. O conteúdo é entregue diretamente por uma CDN, garantindo alto desempenho.

Incremental Static Regeneration (ISR) Introduzido pelo Next.js, o ISR permite que páginas geradas estaticamente possam ser atualizadas de forma incremental, após um período de tempo definido. Isso é feito em segundo plano, sem bloquear o carregamento da página atual. Ideal para sites com atualizações frequentes, mas não críticas em tempo real.

Deferred Static Generation (DSG) Proposto pelo Gatsby, o DSG difere do ISR por não gerar certas páginas no momento do build. Em vez disso, elas são geradas apenas na primeira requisição (on-demand). Após isso, são armazenadas em cache e servidas como estáticas nas requisições seguintes. É útil em projetos com milhares de páginas de baixo acesso, reduzindo significativamente o tempo de build.

Além dessas abordagens, destaca-se o conceito de **reidratação**, que consiste em ativar a interatividade de páginas SSR ou SSG no cliente. Esse processo utiliza JavaScript para associar os eventos dinâmicos à estrutura HTML previamente renderizada, sendo essencial para tornar a página interativa após a exibição inicial (OSMANI; MILLER, 2025).

Esses métodos influenciam diretamente métricas como:

- **TTFB (Time to First Byte)**: tempo até o primeiro byte da resposta.
- **FCP (First Contentful Paint)**: quando o conteúdo começa a ser exibido.
- **TTI (Time to Interactive)**: quando a página se torna responsiva.
- **INP (Interaction to Next Paint)** e **TBT (Total Blocking Time)**: indicadores de interatividade e bloqueio da thread principal.

Desempenho e Compensações

A renderização do lado do servidor tende a exibir conteúdo mais rapidamente (menor FCP), favorecendo acessibilidade e *SEO*. No entanto, pode aumentar o TTFB, já que a página precisa ser processada antes de ser enviada (OSMANI; MILLER, 2025). Já a renderização no cliente pode reduzir o tempo de resposta inicial do servidor, mas exige mais do navegador e aumenta o tempo até a página estar interativa (TTI), especialmente em dispositivos móveis.

Modelos híbridos como SSR com *hydration* tentam unir os benefícios de ambas as abordagens, mas podem causar atrasos na interatividade. Técnicas como *hydration progressiva* ou *streaming* reduzem esses impactos ao ativar partes da interface conforme necessário (OSMANI; MILLER, 2025).



Avanços Recentes

Estratégias mais recentes, como a **renderização trimórfica**, permitem que a renderização ocorra em três camadas: servidor, cliente e *service worker*. Isso possibilita desempenho superior em acessos repetidos e melhor controle sobre o cache e atualização de conteúdo dinâmico (OSMANI; MILLER, 2025).

Impacto em SEO e Indexação

Segundo [Osmani e Miller \(2025\)](#), abordagens que entregam HTML completo como SSR e SSG são mais eficazes para indexação por mecanismos de busca. Já modelos que dependem fortemente de JavaScript (CSR) exigem testes adicionais e podem comprometer a visibilidade em sistemas como o Googlebot, especialmente quando há falhas na execução dos scripts.

Figura 3 – Comparativo entre estratégias de renderização

	Server				Browser
					
	Server Rendering	"Static SSR"	SSR with (Re)hydration	CSR with Prerendering	Full CSR
Overview:	An application where input is navigation requests and the output is HTML in response to them.	Built as a Single Page App, but all pages prerendered to static HTML as a build step, and the JS is removed .	Built as a Single Page App. The server prerenders pages, but the full app is also booted on the client.	A Single Page App, where the initial shell/skeleton is prerendered to static HTML at build time.	A Single Page App. All logic, rendering and booting is done on the client. HTML is essentially just script & style tags.
Authoring:	Entirely server-side (request-response, HTML)	Built as if client-side (components, DOM*, fetch)	Built as client-side	Client-side	Client-side
Rendering:	Dynamic HTML	Static HTML	Dynamic HTML and JS/DOM	Partial static HTML, then JS/DOM	Entirely JS/DOM
Server role:	Controls all aspects. (thin client)	Delivers static HTML	Renders pages (navigation requests)	Delivers static HTML	Delivers static HTML
Pros:	👍 TTI = FCP 👍 Fully streaming	👍 Fast TTFB 👍 TTI = FCP 👍 Fully streaming	👍 Flexible	👍 Flexible 👍 Fast TTFB	👍 Flexible 👍 Fast TTFB
Cons:	👎 Slow TTFB 👎 Inflexible	👎 Inflexible 👎 Leads to hydration	👎 Slow TTFB 👎 TTI >>> FCP 👎 Usually buffered	👎 TTI > FCP 👎 Limited streaming	👎 TTI >>> FCP 👎 No streaming
Scales via:	Infra size / cost	build/deploy size	Infra size & JS size	JS size	JS size
Examples:	Gmail Basic HTML view, Hacker News	Docusaurus, Netflix*	Next.js , Razzle , etc	Gatsby, Vuepress, etc	Most apps

Fonte: Adaptado de ([OSMANI; MILLER, 2025](#))

Considerações Finais

Não há abordagem única que sirva para todos os casos. Como sugerem [Osmani e Miller \(2025\)](#), aplicações altamente interativas podem se beneficiar de CSR com otimizações como *code splitting*, enquanto sites estáticos ou com foco em *performance* e SEO podem optar por SSG ou SSR com *streaming*. Avaliar as prioridades da aplicação, tempo de carregamento, interatividade, escalabilidade e indexação é fundamental para a escolha da estratégia mais adequada.

2.2.2 *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. 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:

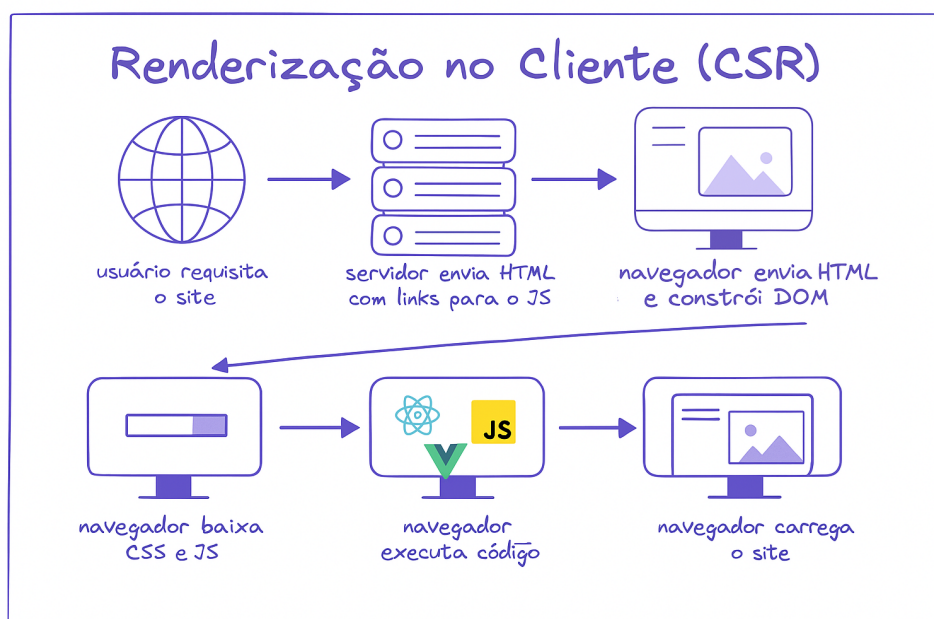
1. O servidor envia uma página *HTML* em branco contendo apenas links para os arquivos *Cascading Style Sheets (CSS)* e JavaScript.
2. O navegador interpreta o *HTML* e constrói a árvore do *Document Object Model (DOM)*
3. Os arquivos de estilo (*CSS*) e script (JavaScript) são baixados pelo navegador.
4. A aplicação é renderizada dinamicamente pelo JavaScript, incluindo elementos visuais como texto, imagens e botões.
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. Ferramentas como a biblioteca React, e frameworks como Vue.js, Angular e Svelte são amplamente utilizadas para implementar *CSR*, permitindo o desenvolvimento de interfaces dinâmicas, interativas e responsivas.

A renderização no lado do cliente (*CSR*) é especialmente vantajosa em aplicações que exigem alta interatividade e atualizações frequentes de conteúdo, como redes sociais, plataformas de streaming e jogos online. No entanto, essa abordagem pode apresentar desvantagens em termos de desempenho inicial e *SEO*, uma vez que o conteúdo só é exibido após a execução do JavaScript, o que pode impactar negativamente a indexação por motores de busca e a experiência do usuário em conexões lentas (EMADAMERHO-ATORI, 2024).

A Figura 4 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.

Figura 4 – Etapas do método de renderização no lado do cliente



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

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. 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.

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

Código 2.1 – Exemplo de HTML mínimo em aplicação Angular com CSR

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](#))

2.2.3 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](#)).

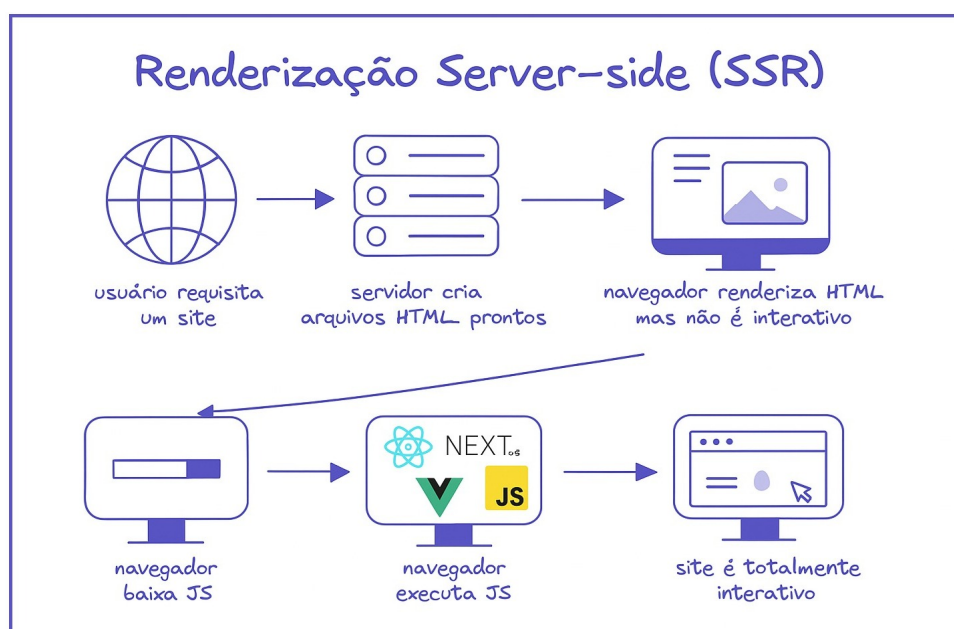
Segundo [Emadamerho-Atori \(2024\)](#), o processo típico de renderização no lado do servidor pode ser descrito em quatro etapas principais:

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 hydration² 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 5 – Etapas do método de renderização no lado do servidor



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

A Figura 5 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 como Next.js, Nuxt.js, SvelteKit, Angular Universal, Remix, Astro e Qwik são amplamente utilizados para construir aplicações com suporte a **SSR**.

² Hydration é uma etapa essencial no **SSR**, em que o JavaScript torna interativo o conteúdo HTML previamente renderizado no servidor.

Esses frameworks operam em um nível superior aos tradicionais (como React, Vue ou Svelte), agregando funcionalidades comuns ao desenvolvimento web, como roteamento, pré-renderização, recuperação de dados e *hydration* podendo oferecer uma estrutura mais completa, opinativa e voltada à escalabilidade.

O [SSR](#) é especialmente útil em aplicações que exigem um carregamento inicial rápido e uma boa indexação por motores de busca, como sites de e-commerce, blogs e páginas de *landing*. Essa abordagem permite que o usuário visualize o conteúdo imediatamente, sem esperar pela execução do JavaScript. Além disso, o [SSR](#) melhora a [SEO](#), pois os mecanismos de busca conseguem indexar o conteúdo completo da página desde o início.

No [Codigo 2.2](#), 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 *time-to-first-byte*³ e de *indexabilidade*⁴ 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.

³ 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](#))

⁴ 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](#))

```
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      <!-- Conteúdo já processado e inserido no servidor -->
19          <div id="__next">
20              <header>
21                  <h1>Ola, mundo!</h1>
22              </header>
23              <main>
24                  <p>Este conteúdo foi renderizado no servidor usando Next.js.</p>
25              </main>
26          </div>
27          <!-- Scripts do Next.js para interacao no cliente -->
28          <script src="/_next/static/chunks/main.js" defer></script>
29      </body>
30      </html>
```

Código 2.2 – Exemplo de HTML mínimo em aplicação Next.js com SSR

2.2.4 Static Site Generation (SSG)

A **Static Site Generation (SSG)** é uma técnica de pré-renderização na qual as páginas da aplicação são geradas estaticamente em tempo de **build** (compilação) e armazenadas como arquivos **HTML**. Ao contrário de abordagens como **CSR** e **SSR**, onde a renderização ocorre no navegador ou sob demanda no servidor, o **SSG** permite que o conteúdo já esteja pronto e otimizado para ser entregue diretamente ao navegador, reduzindo a carga do servidor e otimizando o desempenho de carregamento (PERERA, 2022).

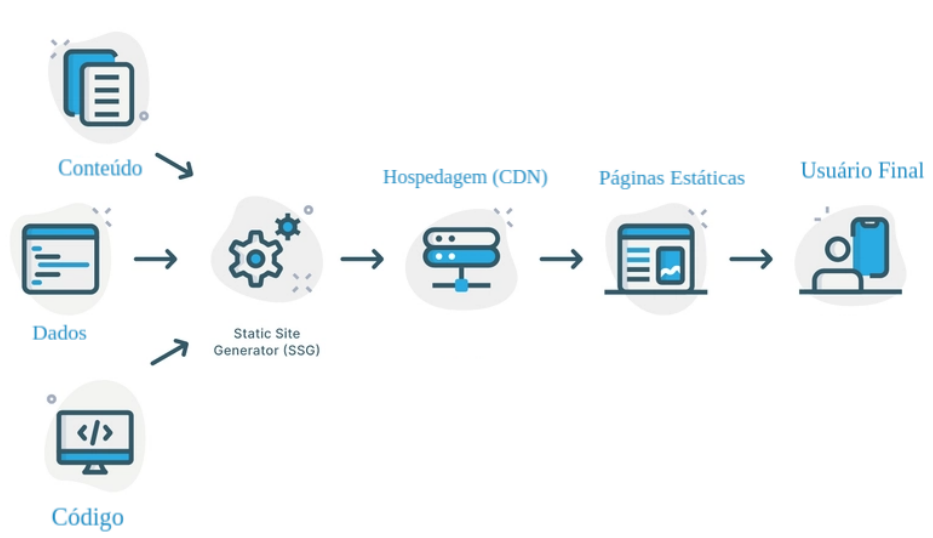
Segundo Bose (2022), a renderização no modelo **SSG** segue estas etapas principais:

1. Durante o processo de construção (build), o gerador de sites estáticos coleta dados de fontes como arquivos locais, *APIs* ou bancos de dados.
2. Esses dados são utilizados para gerar arquivos **HTML** completos para cada rota da aplicação.
3. Os arquivos gerados são armazenados e podem ser servidos diretamente por uma *CDN* (Content Delivery Network).

4. Quando o usuário acessa a aplicação, os arquivos estáticos são entregues instantaneamente, sem necessidade de renderização adicional.

Essa abordagem é ideal para páginas cujo conteúdo não muda com frequência, como blogs, documentações, portfólios e sites institucionais. Como os arquivos são pré-gerados, o tempo de resposta é extremamente rápido, e o [SEO](#) é favorecido, já que os mecanismos de busca encontram o conteúdo pronto para indexação.

Figura 6 – Etapas do método de geração estática de páginas (SSG)



Fonte: (BOSE, 2022) (adaptado)

Frameworks como Next.js, Gatsby, Hugo e Jekyll oferecem suporte completo ao [SSG](#), integrando funcionalidades como roteamento dinâmico, *markdown*, e integração com CMSs. No exemplo a seguir, observa-se um documento [HTML](#) gerado estaticamente por meio de um processo de *build*:

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>Post: SSG Example</title>
7   </head>
8   <body>
9     <article>
10      <h1>Exemplo de página gerada com SSG</h1>
11      <p>Esse conteúdo foi gerado em tempo de build.</p>
12    </article>
13  </body>
14 </html>

```

Código 2.3 – Exemplo de HTML estático gerado com SSG

A principal limitação do [SSG](#) é a dificuldade em lidar com conteúdos altamente dinâmicos. Alterações nos dados requerem um novo processo de build para que as páginas sejam atualizadas, o que pode ser custoso em grandes aplicações ou com frequência de atualização elevada.

2.2.5 *Incremental Static Regeneration (ISR)*

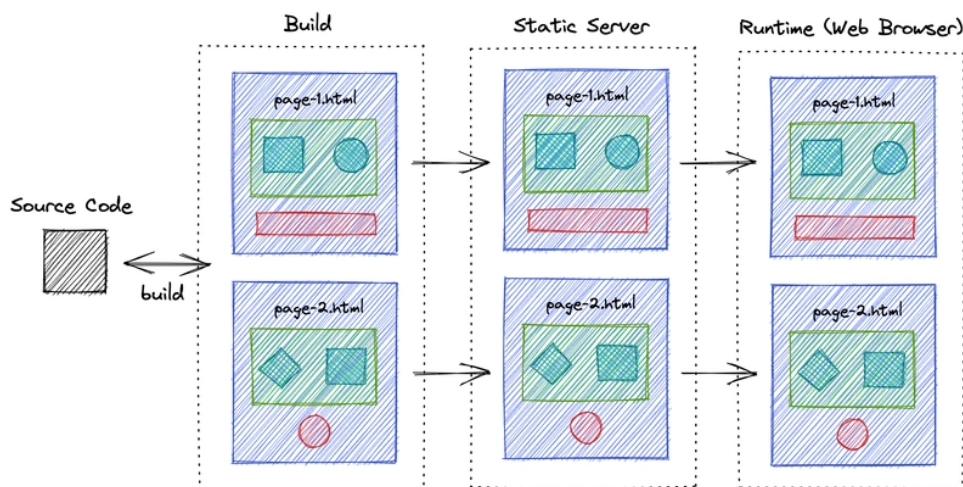
A *Incremental Static Regeneration (ISR)* é uma estratégia híbrida introduzida por frameworks como o Next.js, que combina os benefícios da geração estática ([SSG](#)) com a flexibilidade de atualização dinâmica. Com [ISR](#), as páginas são inicialmente geradas estaticamente em tempo de build, mas podem ser revalidadas e regeneradas no servidor de forma incremental e automática, com base em uma estratégia de tempo (ex: a cada 10 segundos) ou conforme novas requisições são feitas ([PERERA, 2022](#)).

De acordo com [Bose \(2022\)](#), o fluxo típico do [ISR](#) inclui as seguintes etapas:

1. No momento do build inicial, as páginas são geradas e armazenadas como arquivos estáticos.
2. Ao ser requisitada por um usuário, a página é entregue imediatamente, com o conteúdo pré-renderizado.
3. Se o tempo de revalidação definido (ex: `revalidate: 60`) tiver expirado, uma nova requisição ao backend é feita em segundo plano.
4. Essa nova versão da página é armazenada e substitui a anterior, sendo usada em acessos futuros.

Essa abordagem permite obter performance e [SEO](#) semelhantes ao [SSG](#), mas com a vantagem de manter o conteúdo atualizado sem precisar de reconstruções manuais. Por isso, o [ISR](#) é ideal para sites que possuem atualizações regulares, porém não críticas em tempo real, como catálogos de produtos, blogs com comentários ou páginas de notícias.

Figura 7 – Funcionamento do modelo Incremental Static Regeneration (ISR)



Fonte: (PERERA, 2022)

Abaixo, um exemplo típico em Next.js:

```

1      // Funcao usada em getStaticProps
2      export async function getStaticProps() {
3          const res = await fetch('https://api.exemplo.com/posts')
4          const posts = await res.json()
5
6          return {
7              props: { posts },
8              revalidate: 60, // Pagina sera regenerada a cada 60 segundos
9          }
10     }

```

Codigo 2.4 – Exemplo de revalidação de conteúdo com ISR no Next.js

O **ISR** representa um meio-termo eficiente entre a performance do **SSG** e a flexibilidade do **SSR**, oferecendo escalabilidade, **SEO** eficiente e atualização contínua do conteúdo, sem prejudicar a experiência do usuário.

2.2.6 Deferred Static Generation (DSG)

A **Deferred Static Generation (DSG)** é uma extensão da estratégia de geração estática proposta pelo framework Gatsby. Essa abordagem permite que certas páginas do site sejam geradas sob demanda (ou seja, somente no momento em que forem requisitadas pela primeira vez) em vez de serem construídas durante o processo inicial de build, como ocorre no **SSG** tradicional (Gatsby, 2023).

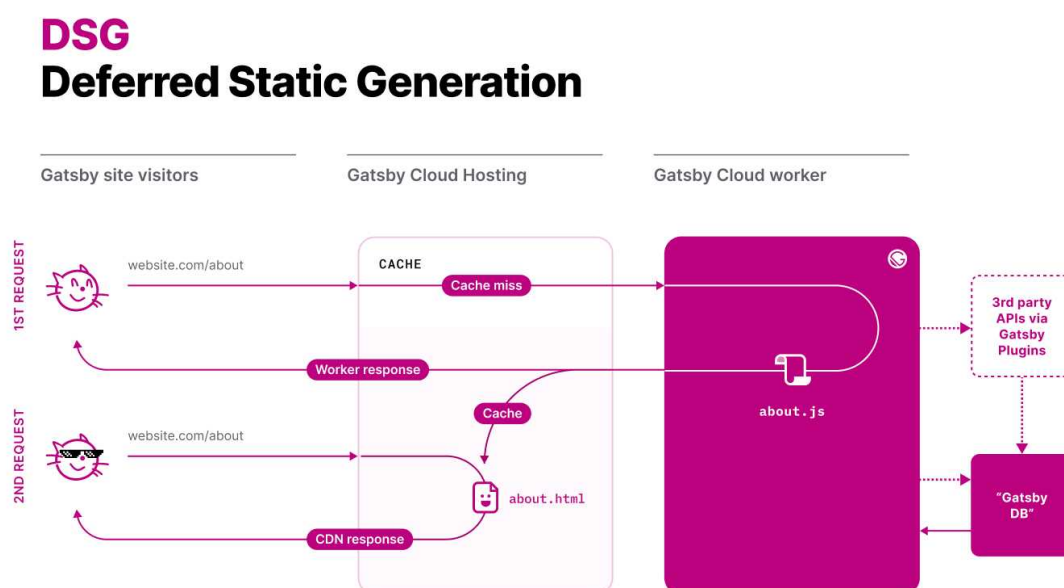
Segundo a Gatsby (2023), o **DSG** tem como principal vantagem a capacidade de reduzir significativamente o tempo de build em projetos com um número elevado de páginas, ao evitar a pré-renderização de rotas que têm baixo tráfego ou que não precisam

estar imediatamente disponíveis. Após a primeira solicitação, a página é armazenada em cache e, a partir daí, servida como conteúdo estático em acessos subsequentes.

O fluxo típico de geração diferida no **DSG** ocorre da seguinte forma:

1. Durante o processo de build, apenas páginas prioritárias são pré-geradas.
2. Páginas marcadas como **defer** são ignoradas temporariamente.
3. Quando uma página **DSG** é acessada pela primeira vez, um *worker* do Gatsby gera a versão **HTML** da página com base em um componente React (ex: `about.js`).
4. A página gerada é armazenada em cache e servida como conteúdo estático nas próximas requisições.

Figura 8 – Funcionamento da estratégia Deferred Static Generation (DSG)



Fonte: (Gatsby, 2023)

Na Figura 8, observa-se que, ao acessar uma página como `/about` pela primeira vez, ocorre um *cache miss*, e um *worker* do Gatsby é acionado para gerar o arquivo `about.html` a partir do componente correspondente (`about.js`). Esse conteúdo pode incluir dados extraídos de plugins, *APIs* de terceiros ou do próprio *Gatsby DB*. Após o processamento, a resposta é enviada ao usuário e armazenada em cache. Requisições futuras à mesma rota são atendidas diretamente pela CDN, com desempenho semelhante ao de páginas **SSG**.

Segundo Anurag (2024), a estratégia do **DSG** é especialmente útil para projetos com milhares de páginas cujo acesso é desigual. Exemplos incluem catálogos de produtos legados, páginas de arquivos antigos ou conteúdos gerados a partir de sistemas CMS.

A seguir, um exemplo de configuração de uma rota com [DSG](#) em Gatsby:

```
1 // gatsby-node.js
2 exports.createPages = async ({ actions }) => {
3   const { createPage } = actions
4   createPage({
5     path: '/produto/exemplo',
6     component: require.resolve('./src/templates/produto.js'),
7     context: { id: 'produto-exemplo' },
8     defer: true, // ativa DSG
9   })
10 }
```

Código 2.5 – Exemplo de definição de página DSG no Gatsby

Assim como o [ISR](#), o [DSG](#) busca combinar desempenho, escalabilidade e atualizações eficientes. No entanto, sua particularidade está em adiar completamente o custo da renderização até o momento da primeira requisição, tornando-o ideal em contextos onde o tempo de build precisa ser otimizado sem prejudicar a entrega do conteúdo a longo prazo.

2.3 Frameworks Web

O uso de bibliotecas e frameworks no desenvolvimento web moderno proporciona ganhos significativos de produtividade, desempenho e organização de código. Eles abstraem operações complexas e oferecem estruturas padronizadas para construção de aplicações escaláveis. A escolha da ferramenta está diretamente relacionada à abordagem de renderização adotada, seja no lado do cliente ([CSR](#)) ou do servidor ([SSR](#)).

2.3.1 Bibliotecas JavaScript

Bibliotecas JavaScript são conjuntos de funcionalidades reutilizáveis que fornecem recursos específicos para o desenvolvimento de aplicações web. Elas diferem dos frameworks por não imporem uma estrutura rígida, oferecendo maior flexibilidade ao desenvolvedor. A seguir, são apresentadas algumas das bibliotecas mais relevantes no contexto de renderização do lado do cliente ([CSR](#)):

- **React:** Desenvolvida pelo Facebook, React é uma biblioteca declarativa focada na construção de interfaces de usuário por meio de componentes reutilizáveis. Seu uso do *virtual DOM* permite renderizações mais eficientes, tornando-a ideal para aplicações interativas com alto desempenho. Apesar de ser comumente chamada de framework, React atua apenas na camada de visualização, exigindo bibliotecas complementares para roteamento e gerenciamento de estado ([REACT](#), 2025).

- **jQuery:** Uma das bibliotecas mais populares da era inicial do JavaScript moderno, jQuery simplifica tarefas comuns como manipulação do DOM, tratamento de eventos e requisições AJAX. Embora sua popularidade tenha diminuído com o surgimento de bibliotecas mais modernas e frameworks reativos, ela ainda é amplamente utilizada em sistemas legados e aplicações de menor complexidade (FOUNDATION, 2025).
- **Alpine.js:** Alpine.js é uma biblioteca leve e reativa voltada para a manipulação de componentes diretamente no HTML, com uma sintaxe declarativa inspirada em Vue.js. Ela é particularmente útil para adicionar interatividade a páginas estáticas ou aplicações simples, sendo uma alternativa eficiente em cenários onde o uso de bibliotecas maiores seria excessivo (PORZIO, 2023).

2.3.2 Frameworks para CSR

No modelo [CSR](#), a renderização da interface é realizada diretamente no navegador do usuário, após o carregamento dos arquivos JavaScript. Frameworks como os listados a seguir são amplamente utilizados para implementar essa abordagem:

- **Vue.js:** framework progressivo para construção de interfaces web interativas. Seu foco está na camada de visualização, com curva de aprendizado acessível e estrutura modular (VUE..., 2025).
- **Angular:** framework completo mantido pelo Google, baseado em TypeScript, que oferece arquitetura robusta e recursos integrados como injeção de dependência e roteamento (ANGULAR, 2025).
- **Svelte:** framework que realiza a compilação dos componentes no momento do build, eliminando a necessidade de um *virtual DOM*, o que reduz o tempo de carregamento e o uso de recursos do navegador (SVELTE, 2025).

Esses frameworks tornam o desenvolvimento com [CSR](#) mais eficiente e sustentável, proporcionando experiências ricas ao usuário com foco em interatividade e responsividade.

2.3.3 Meta-frameworks para SSR

Para aplicações com foco em renderização no lado do servidor, os *meta-frameworks* oferecem soluções completas, otimizando tanto o desempenho inicial quanto a indexabilidade em mecanismos de busca. Eles operam sobre frameworks tradicionais (como Vue ou Svelte) ou bibliotecas (como React), incorporando funcionalidades essenciais como roteamento, pré-renderização, recuperação de dados e *hydration*.

- **Next.js**: baseado em React, fornece recursos para [SSR](#), geração de sites estáticos e suporte a APIs integradas ([NEXT...](#), 2024).
- **Nuxt.js**: extensão do Vue.js que oferece SSR, geração estática e arquitetura modular ([NUXT...](#), 2024).
- **SvelteKit**: baseado em Svelte, permite renderização no servidor e no cliente, com foco em simplicidade e desempenho ([SVELTEKIT](#), 2024).
- **Angular Universal**: solução oficial para SSR em Angular, melhora a indexação e o tempo de carregamento inicial ([ANGULAR...](#), 2024).
- **Remix**: framework full-stack para React que adota um modelo de dados centrado em carregadores e ações ([REMIX](#), 2024).
- **Astro**: framework moderno que carrega apenas o JavaScript necessário, permitindo uso híbrido de componentes React, Vue, Svelte e outros ([ASTRO](#), 2024).
- **Qwik**: introduz o conceito de aplicações *resumíveis*, com SSR e carregamento progressivo de interatividade ([QWIK](#), 2024).

Esses meta-frameworks são especialmente indicados para aplicações que priorizam SEO, acessibilidade e desempenho no primeiro carregamento, como páginas institucionais, lojas virtuais e blogs.

2.3.4 Comparativo entre Frameworks CSR e SSR

Tabela 2 – Comparação entre frameworks para CSR e SSR

Critério	Frameworks CSR (Vue, Angular, Svelte)	Meta-frameworks SSR (Next.js, Nuxt, SvelteKit, etc.)
Renderização Inicial	O conteúdo é montado no navegador após o carregamento do JavaScript	O conteúdo é gerado no servidor e entregue já renderizado ao navegador
Tempo de Carregamento	Maior tempo de carregamento inicial (dependente do JS)	Melhor desempenho no carregamento inicial (TTFB menor)
SEO	Pode ser limitado, pois bots podem não processar JavaScript adequadamente	Excelente, já que o HTML completo está disponível para rastreadores
Interatividade	Alta, com foco em aplicações ricas e dinâmicas	Boa, com necessidade de <i>hydration</i> após o carregamento
Complexidade de Infraestrutura	Menor, geralmente servido por CDNs e arquivos estáticos	Maior, exige servidores para processar cada requisição
Casos de Uso Ideais	SPAs, dashboards, aplicações com muitas interações em tempo real	Landing pages, blogs, e-commerces, sites que dependem de SEO

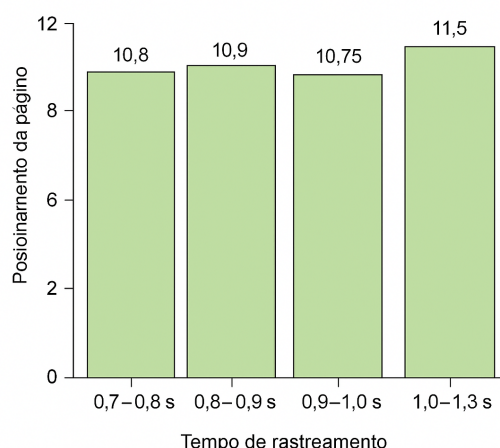
2.4 Experiência do Usuário

A [User Experience \(UX\)](#) é um aspecto crítico no desenvolvimento de aplicações web, influenciando diretamente a satisfação e a eficácia da interação do usuário com o sistema ([EMADAMERHO-ATORI, 2023](#)). Para alcançar uma [UX](#) satisfatória, a escolha entre [CSR](#) e [SSR](#) deve considerar fatores como: [SEO](#), velocidade de carregamento, interatividade e acessibilidade. Conforme [Emadamerho-Atori \(2024\)](#), a [UX](#) vai além da interface gráfica, englobando toda a jornada do usuário desde a navegação até a conclusão de tarefas.

2.4.1 *Search Engine Optimization* (SEO)

O [Search Engine Optimization \(SEO\)](#) consiste em um conjunto integrado de práticas de otimização, tanto no aspecto técnico quanto no de conteúdo, com três objetivos principais: maximizar a visibilidade orgânica nos mecanismos de busca, posicionar estrategicamente páginas-chave e garantir uma experiência de usuário qualificada durante o processo de busca. Essas práticas são essenciais para garantir que o conteúdo de um site seja facilmente encontrado e indexado pelos motores de busca, aumentando a probabilidade de atrair visitantes qualificados. Entre os fatores mais conhecidos, destaca-se a velocidade de carregamento da página, que impacta diretamente a experiência do usuário e a classificação nos resultados de busca ([MURPHY, 2022](#)).

Figura 9 – Tempo de Rastreamento e Posicionamento da Página



Fonte: (WAGNER, 2016)(adaptado)

A Figura 9 ilustra a relação entre o tempo de rastreamento e o posicionamento da página. O tempo de rastreamento refere-se ao tempo que os mecanismos de busca levam para acessar e indexar uma página. Quanto mais rápido o tempo de rastreamento, maior a probabilidade de a página ser indexada rapidamente e, conseqüentemente, melhor seu posicionamento nos resultados de busca. Isso destaca a importância de otimizar o desempenho do site para garantir uma boa classificação nos motores de busca.

2.4.2 Velocidade de carregamento

A velocidade de carregamento de uma página refere-se ao tempo necessário para que todo o seu conteúdo esteja visível e interativo no navegador, desde a solicitação inicial do usuário. Esse tempo pode ser influenciado por diversos fatores, como o tamanho dos arquivos, a complexidade do conteúdo, a qualidade da conexão com a internet e o desempenho do servidor (SHOPIFY, 2024).

Esse fator é determinante tanto para a UX quanto para o SEO. Páginas que carregam rapidamente tendem a apresentar menores taxas de rejeição e melhores resultados em métricas de conversão. Além disso, o SEO utiliza a velocidade de carregamento como um dos critérios de ranqueamento nos mecanismos de busca (MURPHY, 2022).

De acordo com Google (2010), quanto mais rápido um site carregar, melhor tende a ser a experiência do usuário. Sites lentos comprometem a navegação e reduzem o tempo de permanência, afetando negativamente o engajamento.

A percepção de desempenho — muitas vezes chamada de *velocidade percebida* — também é um fator crucial para a usabilidade e pode ser tão relevante quanto o tempo de carregamento real. Nesse sentido, a escolha entre SSR e CSR influencia diretamente

essa percepção. O [SSR](#) geralmente proporciona carregamento inicial mais rápido, pois o conteúdo é renderizado no servidor e entregue ao navegador já pronto para exibição. Isso permite que os usuários visualizem o conteúdo principal imediatamente, mesmo que outros recursos ainda estejam sendo carregados ([EMADAMERHO-ATORI, 2024](#)).

Por outro lado, no [CSR](#), o navegador precisa baixar, interpretar e executar o JavaScript antes de renderizar qualquer conteúdo. Isso pode resultar em uma exibição inicial em branco ou em telas de carregamento, o que compromete a percepção de desempenho — especialmente em conexões lentas ou dispositivos com menor capacidade de processamento ([STUDIO, 2023b](#)).

2.4.3 Interatividade

A interatividade é um fator decisivo na experiência do usuário em aplicações web modernas, pois determina a forma como os usuários percebem a continuidade e a capacidade de resposta durante a navegação. Nas aplicações que utilizam [CSR](#), o código JavaScript é executado diretamente no navegador, permitindo respostas imediatas a interações como cliques, preenchimento de formulários ou navegação entre páginas internas. Essa abordagem possibilita transições de página mais suaves e experiências semelhantes às de aplicativos nativos, sem a necessidade de recarregamentos completos ([STUDIO, 2023b](#)).

Segundo [Emadamerho-Atori \(2024\)](#), a renderização no lado do cliente favorece experiências altamente dinâmicas, oferecendo um nível elevado de controle sobre os elementos da interface. Em contrapartida, o [SSR](#), embora proporcione carregamento inicial mais rápido e visibilidade imediata do conteúdo, apresenta limitações em termos de interatividade. Alterações na interface em aplicações [SSR](#) geralmente demandam comunicações adicionais com o servidor, o que pode comprometer a continuidade da experiência do usuário ([EMADAMERHO-ATORI, 2024](#); [WATTS, 2023](#)).

Para mitigar essas limitações, abordagens híbridas têm sido amplamente adotadas. Nelas, o conteúdo é inicialmente renderizado no servidor e, posteriormente, reativado no cliente com JavaScript, em uma estratégia conhecida como *hydration* ([WATTS, 2023](#)). Essa técnica busca aliar os benefícios de desempenho e [SEO](#) do [SSR](#) com a interatividade aprimorada do [CSR](#).

2.4.4 Acessibilidade

A acessibilidade em aplicações web refere-se à capacidade de tornar conteúdos e funcionalidades utilizáveis por pessoas com deficiência, como visual, auditiva, motora ou cognitiva. É um princípio essencial para garantir a equidade no acesso à informação e à interação digital. De acordo com ([STUDIO, 2023a](#)), acessibilidade diz respeito a assegurar

que todos, independentemente de suas habilidades, possam acessar e interagir com o conteúdo da web. Para pessoas com deficiência, isso pode significar o uso de leitores de tela, navegação por teclado ou a dependência de outras tecnologias assistivas.

As abordagens de renderização, como [CSR](#) e [SSR](#), impactam diretamente a acessibilidade, especialmente na compatibilidade com essas tecnologias. Em aplicações que utilizam [CSR](#), o conteúdo geralmente é carregado de forma assíncrona após a execução do JavaScript, o que pode dificultar a leitura imediata por leitores de tela que dependem de uma estrutura HTML previamente carregada para interpretar a página corretamente ([STUDIO, 2023a](#)). Já no [SSR](#), o conteúdo é entregue completamente no carregamento inicial, facilitando a interpretação por essas ferramentas e proporcionando uma experiência mais estável para usuários com deficiência visual ([EMADAMERHO-ATORI, 2024](#)).

Além disso, em contextos com atualizações dinâmicas de conteúdo — como ocorre em SPAs com [CSR](#) — é necessário adotar práticas específicas para garantir a acessibilidade, como gerenciamento de foco, uso de alertas ARIA e atualização de leitores de tela após mudanças no DOM. Essas medidas são fundamentais para que as mudanças de visualização sejam percebidas corretamente por tecnologias assistivas, uma vez que alterações no DOM nem sempre são reconhecidas automaticamente por leitores de tela. O envio de foco a elementos interativos ou o uso de regiões ARIA ao vivo são técnicas recomendadas para anunciar mudanças de estado ao usuário ([SUTTON, 2018](#)).

Assim, embora o [SSR](#) ofereça uma base naturalmente mais acessível, ambas as abordagens podem ser igualmente inclusivas quando aplicadas com atenção às diretrizes e boas práticas de acessibilidade.

3 Metodologia

3.1 Levantamento Teórico

O levantamento teórico consistiu na revisão e sistematização dos principais conceitos, tecnologias e práticas relacionadas à renderização de conteúdo em aplicações web, com foco nas abordagens de *Client-Side Rendering (CSR)* e *Server-Side Rendering (SSR)*. Essa etapa teve como objetivo fornecer o embasamento necessário para o desenvolvimento do estudo de caso e da análise comparativa proposta neste trabalho.

A fundamentação iniciou-se com a exploração dos princípios do desenvolvimento web moderno, incluindo a arquitetura cliente-servidor, o funcionamento do protocolo **HTTP** e as tecnologias essenciais do *frontend*: **HTML**, **CSS** e **JavaScript**. Estes elementos formam a base para compreender como as estratégias de renderização operam, tanto no lado do cliente quanto no lado do servidor.

Em seguida, foram estudadas em detalhes as abordagens **CSR** e **SSR**. A renderização no lado do cliente (**CSR**) foi analisada quanto ao seu funcionamento típico em aplicações *Single Page Application (SPA)*, caracterizadas por uma única página carregada inicialmente, com atualizações dinâmicas de conteúdo via JavaScript. Essa abordagem oferece vantagens como maior interatividade e fluidez na navegação, além de transições rápidas entre páginas internas. No entanto, apresenta desvantagens como maior tempo de carregamento inicial e limitações de indexação por mecanismos de busca.

Por outro lado, a renderização no lado do servidor (**SSR**) foi abordada sob a ótica de desempenho inicial otimizado e maior compatibilidade com **SEO**, pois o conteúdo é entregue já renderizado ao navegador. Essa abordagem é comumente utilizada em aplicações do tipo *Multi Page Application (MPA)*, que possuem múltiplas páginas distintas e se beneficiam da pré-renderização para melhorar a performance inicial, a acessibilidade e a visibilidade em mecanismos de busca. Como contraponto, o SSR demanda maior processamento no servidor e pode aumentar a complexidade da infraestrutura.

Além dessas duas abordagens principais, o estudo também incluiu métodos híbridos como o *Static Site Generation (SSG)*, *Incremental Static Regeneration (ISR)* e *Deferred Static Generation (DSG)*, que visam equilibrar performance, escalabilidade e atualizações de conteúdo dinâmico, especialmente em aplicações que exigem alta eficiência e atualizações frequentes.

O levantamento teórico foi complementado por uma análise dos principais *fra-*

meworks e bibliotecas utilizados no desenvolvimento web atual, como *React*, *Vue.js*, *Angular*, *Next.js*, *Nuxt* e *SvelteKit*, e por uma discussão sobre os impactos de cada abordagem na experiência do usuário (UX), incluindo aspectos como SEO, acessibilidade, tempo de carregamento e interatividade.

Esse base teórica serviu para as próximas etapas da pesquisa, em especial o mapeamento sistemático da literatura e a condução do estudo de caso prático.

3.2 Mapeamento Sistemático da Literatura

O mapeamento sistemático da literatura teve como objetivo identificar, selecionar e analisar estudos acadêmicos relevantes que abordassem comparações entre as abordagens de renderização CSR e SSR no contexto do desenvolvimento de aplicações web. Essa etapa foi essencial para compreender o estado da arte, bem como identificar lacunas e oportunidades para a realização do estudo de caso proposto neste trabalho.

A estratégia de busca foi estruturada com o apoio da metodologia PICOC, que define os elementos População, Intervenção, Comparação, Resultado e Contexto, com o intuito de guiar a construção das expressões de busca e garantir abrangência e precisão nos resultados. As principais bases de dados utilizadas incluíram Periódicos Capes e Scopus, por oferecerem amplo acervo e suporte a pesquisas refinadas.

Foram utilizadas expressões booleanas combinando termos como *Client-Side Rendering*, *Server-Side Rendering*, *Web Performance*, *SEO*, *UX* e *Frontend Architecture*. Após a aplicação dos critérios de inclusão e exclusão, os artigos resultantes foram classificados e analisados de acordo com sua relevância, tipo de abordagem estudada, metodologias utilizadas e principais conclusões.

A seleção final contemplou trabalhos que abordavam métricas de desempenho, tempo de carregamento, interatividade, SEO e experiência do usuário. Além disso, foram considerados estudos que analisavam o uso de frameworks modernos como *React*, *Next.js*, *Nuxt.js* e *Angular Universal*, além de pesquisas aplicadas em contextos reais de produção.

Como resultado, foi possível consolidar uma visão abrangente sobre os desafios, vantagens e limitações de cada abordagem, fornecendo subsídios importantes para a execução do estudo de caso prático apresentado nos capítulos seguintes. O mapeamento sistemático também evidenciou a escassez de estudos nacionais aplicados ao tema, reforçando a relevância deste trabalho no cenário acadêmico e profissional brasileiro.

3.3 Estudo de Caso Prático

O escopo da aplicação a ser desenvolvida visa avaliar os impactos das abordagens de renderização *CSR* e *SSR* no desenvolvimento de aplicações web. Para isso, será considerado o contexto de uma empresa fictícia cujo time de gestores precisa decidir qual modelo de renderização adotar para a criação de sua nova aplicação web. Serão desenvolvidas duas versões de uma mesma aplicação: uma utilizando *CSR* e outra baseada em *SSR*. Ambas as implementações terão as mesmas funcionalidades, aparência visual e estrutura de dados, de modo a permitir uma análise equitativa quanto ao desempenho, à experiência do usuário e à otimização para mecanismos de busca.

A aplicação a ser desenvolvida simulará um catálogo de produtos, com navegação por páginas, carregamento de dados via *API* e exibição de informações detalhadas. Esse modelo foi escolhido por representar um cenário comum na web moderna, abrangendo interações usuais como carregamento dinâmico de conteúdo, roteamento entre páginas e exibição de listas e detalhes.

Cada versão será implementada conforme os princípios da respectiva abordagem de renderização: a versão *CSR* terá a interface renderizada predominantemente no navegador do usuário, enquanto a versão *SSR* contará com o conteúdo renderizado no servidor e enviado ao cliente já montado.

Durante o desenvolvimento, serão seguidas boas práticas de acessibilidade, responsividade e otimização para *SEO*, garantindo que ambas as versões possam ser avaliadas com base em critérios equivalentes. As métricas a serem analisadas incluirão tempo de carregamento, tempo até a interatividade, consumo de recursos, desempenho percebido, qualidade do código e compatibilidade com ferramentas de análise de *SEO*.

A coleta de dados será realizada por meio de ferramentas como Google Lighthouse, PageSpeed Insights e WebPageTest, além de testes manuais com usuários, a fim de observar qualitativamente a experiência de uso. Esses dados fundamentarão a análise comparativa e a discussão dos resultados, que serão apresentados nas seções seguintes.

3.4 Coleta de Dados e Testes

Após o desenvolvimento das versões da aplicação com *CSR* e *SSR*, será realizada a coleta de dados com o objetivo de mensurar o desempenho e a qualidade da experiência do usuário em cada abordagem.

As aplicações serão hospedadas em ambientes equivalentes, com configurações idênticas de hardware e rede, a fim de garantir condições justas para os testes.

A coleta de dados utilizará as seguintes ferramentas:

- **Google Lighthouse:** Avaliação de desempenho, acessibilidade, melhores práticas e [SEO](#);
- **PageSpeed Insights:** Medição de tempo de carregamento, *First Contentful Paint* (FCP), *Largest Contentful Paint* (LCP), *Time to Interactive* (TTI);
- **Core Web Vitals:** Conjunto de métricas essenciais propostas pelo Google para mensurar a qualidade da experiência do usuário, incluindo LCP (tempo de renderização do maior elemento visível), FID (atraso na primeira interação) e CLS (estabilidade visual da página);
- **WebPageTest:** Análise do tempo de resposta, número de requisições e uso de cache;
- **Chrome DevTools:** Inspeção do tempo de execução de scripts e verificação do processo de hidratação na abordagem [SSR](#).

Além disso, será realizada uma etapa de testes manuais com usuários em diferentes dispositivos e conexões, com o intuito de observar a fluidez da navegação, percepção de velocidade, clareza das interfaces e responsividade.

As principais métricas a serem analisadas incluirão:

- Tempo total de carregamento;
- Tempo até a primeira exibição de conteúdo visível (FCP);
- Tempo até a interatividade plena (TTI);
- Número de requisições HTTP;
- Compatibilidade com práticas de [SEO](#);
- Consumo de recursos do navegador (memória e CPU);
- Percepção subjetiva da experiência do usuário.

3.5 Síntese e Discussão

De maneira geral, será feita uma análise comparativa entre as abordagens [CSR](#) e [SSR](#), com base nas informações que serão obtidas ao longo da implementação descrita

no Capítulo 6. Essa análise terá como foco os aspectos técnicos e funcionais relacionados ao desempenho, tempo de carregamento, interatividade, experiência do usuário e compatibilidade com mecanismos de busca.

A discussão será conduzida de forma imparcial, considerando os dados coletados por meio de ferramentas especializadas, como Google Lighthouse, PageSpeed Insights e WebPageTest, bem como possíveis observações manuais obtidas em testes de usabilidade. A intenção será compreender como cada abordagem impacta diferentes requisitos de projeto e quais implicações essas escolhas podem trazer no contexto do desenvolvimento de aplicações web.

Além da comparação direta entre *CSR* e *SSR*, poderá ser explorada, conforme a relevância no decorrer da análise, a viabilidade de estratégias híbridas, como o uso de *SSG* e *ISR*, no sentido de identificar soluções que conciliem desempenho, escalabilidade e experiência do usuário.

Essa etapa terá caráter investigativo e descritivo, sem antecipar conclusões, e servirá como base para reflexões futuras e para a elaboração de recomendações alinhadas aos objetivos definidos neste trabalho.

4 Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados ao objeto de pesquisa, obtidos por meio de um mapeamento sistemático da literatura. O objetivo desse mapeamento foi identificar, analisar e sintetizar estudos acadêmicos que abordam comparações entre as abordagens de renderização [CSR](#) (Client-Side Rendering) e [SSR](#) (Server-Side Rendering) no contexto do desenvolvimento de aplicações web. Buscou-se compreender como essas estratégias impactam aspectos como desempenho, tempo de carregamento, [SEO](#), experiência do usuário e escalabilidade. O protocolo adotado, descrito nas seções seguintes, foi elaborado para garantir a abrangência, a precisão e a relevância dos resultados encontrados.

4.1 Questões de pesquisa

- Q1: De que maneira a escolha entre [CSR](#) e [SSR](#) influenciam a experiência do usuário?
- Q2: Como as abordagens [CSR](#) e [SSR](#) afetam métricas de performance, tempo de carregamento e tempo até a interatividade em aplicações web?
- Q3: Quais são os principais desafios e *trade-offs* na implementação de [CSR](#) e [SSR](#)?
- Q4: Quais trabalhos relacionados existem na literatura que abordam recomendações sobre quando usar o [CSR](#) ou [SSR](#)?

4.2 Estratégia de busca

Esta seção apresenta a estratégia de buscas de artigos científicos e livros relacionados à pesquisa. As ferramentas utilizadas para realizar as buscas são:

- **Periódicos Capes:** É uma ferramenta disponibilizada pelo governo federal para uso de estudantes e pesquisadores. Acessando através da instituição de ensino ou pesquisa, é possível ter acesso completo a uma grande quantidade de artigos científicos publicados em variadas revistas, conferências e universidades. A principal vantagem dessa ferramenta é a possibilidade de ler o conteúdo integral de grande parte das publicações disponíveis. Por outro lado, as expressões de busca atualmente suportadas são bem limitadas.
- **Scopus:** Trata-se de uma ferramenta similar ao Periódicos Capes. No entanto, o *Scopus* permite a elaboração de expressões de buscas mais complexas e sofisticadas, servindo para descobrir publicações não detectadas pelas outras plataformas. Além

disso, possui um acervo bem mais amplo que o Periódicos Capes. Entretanto, algumas publicações não podem ser vistas na íntegra de forma gratuita.

- **PICOC:** A técnica *PICOC* foi utilizada para estruturar e refinar a estratégia de busca. Essa abordagem consiste em definir cinco elementos principais que auxiliam na formulação da expressão booleana para a pesquisa:
 - **P (População/Problema):** Define os estudos ou o grupo de interesse, ou seja, o problema ou a população que se deseja investigar. Por exemplo, “artigos que tratem da integração de tecnologias digitais na educação.”
 - **I (Intervenção/Interesse):** Refere-se à intervenção, prática ou fenômeno que está sendo analisado. Neste caso, pode ser a “inserção de tecnologias digitais nos processos de ensino e aprendizagem.”
 - **C (Comparação):** Descreve o(s) elemento(s) com os quais a intervenção ou situação é comparada, como “ensino tradicional” ou a comparação entre diferentes estratégias digitais, quando aplicável.
 - **O (Outcome/Desfecho):** Indica os resultados ou efeitos esperados da intervenção. Por exemplo, “melhora do desempenho acadêmico” ou “maior engajamento dos alunos.”
 - **C (Contexto):** Considera o ambiente ou cenário onde a intervenção ocorre, como “instituições de ensino, universidades” ou “publicações indexadas em bases internacionais.”

A partir da definição desses elementos, foi possível construir uma expressão booleana que unisse os principais termos de interesse para a pesquisa. Esse método colaborou para refinar os resultados, tornando a busca mais precisa e abrangente, conforme exemplificado no [Quadro 1](#).

4.3 Quadro PICOC

Quadro 1 – Estrutura PICOC aplicada à pesquisa

Elemento	Descrição
P (População/Problema)	Equipes de desenvolvimento web, arquitetos de software e gestores de TI que precisam escolher estratégias de renderização (CSR ou SSR) para aplicações web modernas, visando otimizar desempenho, SEO e experiência do usuário.
I (Intervenção)	Adoção de técnicas de CSR (Client-Side Rendering): todo (ou quase todo) o conteúdo gerado no lado do cliente, utilizando frameworks/libraries como React, Vue, Angular etc.
C (Comparação)	Implementação de SSR (Server-Side Rendering): conteúdo pré-renderizado no servidor antes de ser enviado ao cliente, usando <i>meta-frameworks</i> como Next.js, Nuxt.js, SvelteKit, Angular Universal, entre outros.
O (Outcome / Resultado)	<ul style="list-style-type: none"> Métricas de desempenho (tempo de carregamento, <i>time-to-first-byte</i>, <i>largest contentful paint</i>, etc.) Impacto no SEO (indexabilidade, posicionamento em buscadores) Experiência do usuário e usabilidade Escalabilidade do sistema (uso de recursos de servidor/cliente)
C (Contexto)	Aplicações web modernas que buscam equilibrar interatividade, rapidez de carregamento, otimização para motores de busca e redução de custos operacionais. O estudo pode ser aplicado a sistemas de e-commerce, portais de conteúdo, <i>landing pages</i> , etc.

Fonte: os autores

4.3.1 Expressão de busca

Quadro 2 – Expressão de busca utilizada

Expressão de Busca
<i>(TITLE-ABS-KEY("Client-Side Rendering" OR "CSR" OR "Server-Side Rendering" OR "SSR")) AND (TITLE-ABS-KEY("web performance" OR "page speed" OR "web optimization" OR "SEO" OR "search engine optimization" OR "user experience" OR "UX" OR "usability"))</i>

Fonte: os autores

4.4 Estratégia de seleção

A estratégia de seleção dos artigos foi baseada em critérios de inclusão e exclusão, conforme descrito a seguir:

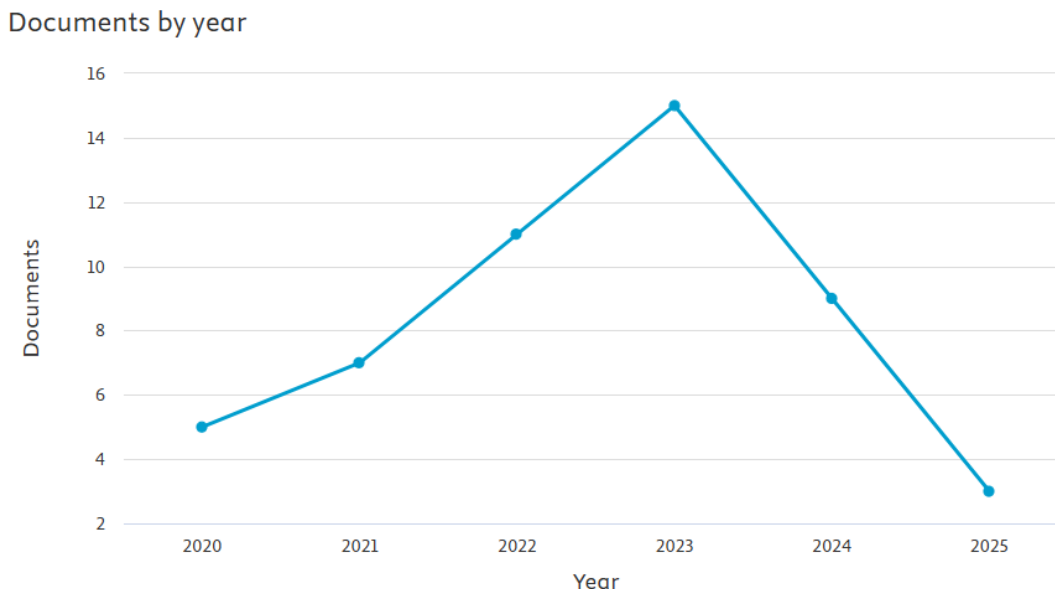
- **Critérios de inclusão:**
 - Artigos publicados entre 2020 e 2025.
 - Artigos que abordem o impacto de [CSR](#) e [SSR](#) em aplicações web.
 - Artigos que apresentem resultados de experimentos ou estudos de caso relacionados a [CSR](#) e [SSR](#).
- **Critérios de exclusão:**
 - Artigos que não estejam disponíveis na íntegra.
 - Artigos que não abordem diretamente o tema da pesquisa.
 - Artigos que sejam duplicados ou muito semelhantes a outros já selecionados.

4.5 Caracterização de pesquisa

Na [Figura 10](#), é apresentado o número de publicações identificadas por ano, no intervalo entre 2020 e 2025. Observa-se um crescimento progressivo de 2020 a 2023, culminando em um pico em 2023, com 15 documentos publicados. A partir desse ponto, nota-se uma queda significativa: em 2024, o número de publicações cai para 9, e em 2025 esse número se reduz ainda mais, atingindo apenas 3 documentos. Essa redução pode ser parcialmente atribuída ao fato de que a coleta foi realizada no mês de abril de 2025, o que possivelmente não contempla todas as publicações previstas para o ano. No total, foram encontrados 50 documentos relevantes, distribuídos de forma desigual, evidenciando uma

tendência crescente de interesse pelo tema até 2023, seguido de uma possível estabilização ou atraso na indexação dos dados mais recentes.

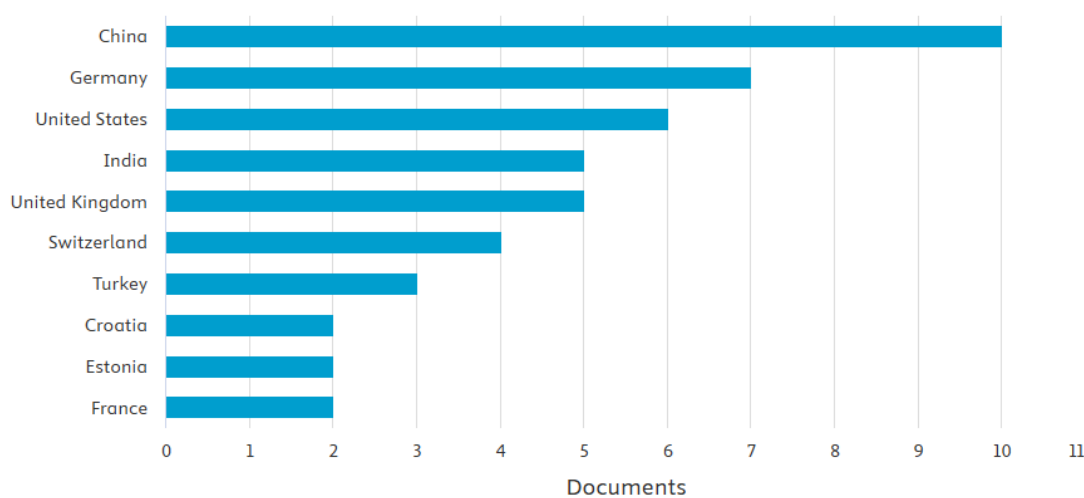
Figura 10 – Número de publicações ao longo dos anos



Fonte: Scopus

Na [Figura 11](#) são apresentados os países com maior número de publicações relacionadas ao tema desta pesquisa. Observa-se que a China lidera com 10 documentos, seguida pela Alemanha (7) e pelos Estados Unidos (6). Em seguida, aparecem Índia, Reino Unido e Suíça, cada um com 5 documentos publicados. A Turquia também se destaca com 4 publicações, enquanto Croácia, Estônia e França completam a lista com 2 documentos cada. Esses dados evidenciam uma concentração relevante de estudos em países com infraestrutura tecnológica consolidada, especialmente na Ásia, Europa Ocidental e América do Norte, o que reforça o caráter global do interesse em torno da comparação entre abordagens de renderização no desenvolvimento de aplicações web.

Figura 11 – Distribuição das publicações por país



Fonte: Scopus

4.6 Artigos selecionados

Após realizadas as leituras preliminares, apenas 13 publicações mostraram-se relevantes para responder às questões de pesquisa e/ou apoiar na elaboração do estudo de caso. Esta seleção considerou critérios de alinhamento temático, profundidade técnica e aplicabilidade ao escopo da pesquisa. A listagem completa dos artigos selecionados pode ser consultada no Quadro 3.

Título	Ano
<i>Progressive Server-Side Rendering with Suspendable Web Templates</i>	2025
<i>Requirements for the Development of a Website Builder with Adaptive Design</i>	2024
<i>Enhancing SEO in Single-Page Web Applications in Contrast With Multi-Page Applications</i>	2024
<i>Web Development Using ReactJS</i>	2023
<i>Improving Universal Rendering Performance on NuxtJS-based Web Application</i>	2023
<i>Comparison between client-side and server-side rendering in the web development</i>	2020
<i>Methods of Improving and Optimizing React Web-applications</i>	2021
<i>Improving ruby on rails-based web application performance</i>	2021
<i>An integrated framework of user experience-oriented smart service requirement analysis for smart product service system development</i>	2022
<i>A Research Framework for B2B Green Marketing Innovation: the Design of Sustainable Websites</i>	2022
<i>Corporate Social Responsibility: Hiring Requisition in Media Companies?</i>	2023
<i>Single page optimization techniques using react</i>	2024
<i>Proceedings of the 19th International Conference on Web Information Systems and Technologies, WEBIST 2023</i>	2023

Quadro 3 – Artigos selecionados sobre estratégias de renderização e desempenho em aplicações web

4.7 Resultados e Discussão

A análise dos 13 artigos selecionados permitiu identificar padrões, lacunas e contribuições relevantes no debate sobre as abordagens de renderização [CSR](#) e [SSR](#) no desenvolvimento de aplicações web. Os resultados foram organizados de acordo com as questões de pesquisa previamente definidas ([seção 4.1](#)).

4.7.1 Q1: De que maneira a escolha entre CSR e SSR influencia a experiência do usuário?

Diversos estudos destacaram o impacto direto da estratégia de renderização na experiência percebida pelo usuário. Por exemplo, o trabalho de (ZHOU et al., 2022) propõe um framework voltado à experiência do usuário em sistemas inteligentes, indicando que tempos de resposta mais rápidos — geralmente proporcionados por renderização no servidor — influenciam positivamente a satisfação dos usuários finais. Embora o estudo não se concentre diretamente em aplicações web tradicionais, os princípios aplicados ao tempo de resposta e fluidez de interação podem ser extrapolados para cenários SSR, destacando o papel do tempo até a interatividade como um fator crítico. Essa abordagem abre espaço para futuras investigações que apliquem o framework em contextos específicos de CSR e SSR.

De forma complementar, (LACOM; SAGOT, 2022) reforça a importância do carregamento eficiente para sites sustentáveis e voltados ao marketing digital, mencionando que usuários tendem a abandonar páginas lentas, o que afeta indicadores de engajamento e conversão.

Além disso, (POKHRIYAL et al., 2024) e (KESHARI et al., 2023) apontam que aplicações construídas com CSR, embora mais interativas após o carregamento inicial, tendem a apresentar maior latência no primeiro carregamento, o que pode prejudicar a primeira impressão do usuário e sua propensão a permanecer na página.

4.7.2 Q2: Como as abordagens CSR e SSR afetam métricas de performance, tempo de carregamento e tempo até a interatividade em aplicações web?

Os estudos de (ISKANDAR et al., 2020) e (ANGKASA et al., 2023) apresentam comparações diretas entre CSR e SSR em relação a métricas como *time-to-first-byte* (TTFB), *first contentful paint* (FCP) e *largest contentful paint* (LCP). Os resultados mostram que a abordagem SSR, especialmente quando combinada com estratégias híbridas (como *static site generation* ou *incremental rendering*), apresenta desempenho superior nos momentos iniciais de carregamento da página.

No entanto, o trabalho de (PAVIC; BRKIC, 2021) chama atenção para a possibilidade de otimização no lado do cliente, sugerindo que ajustes finos em bibliotecas como React podem mitigar parte das desvantagens de CSR em termos de tempo de carregamento.

Já (BEKMANOVA et al., 2024) e (KLOCHKOV; MULAWKA, 2021) demonstram

que a performance final da aplicação pode depender mais da arquitetura geral e das boas práticas de desenvolvimento do que apenas da escolha entre SSR ou CSR. Isso evidencia que decisões arquiteturais e técnicas de otimização, como pré-carregamento de recursos e minimização de dependências, exercem papel crucial na performance percebida pelo usuário.

4.7.3 Q3: Quais são os principais desafios e *trade-offs* na implementação de CSR e SSR?

A literatura aponta diversos desafios na escolha entre as duas abordagens, incluindo complexidade de implementação, compatibilidade com SEO, custo computacional e escalabilidade.

O artigo de (CARVALHO, 2025) destaca que o uso de SSR com *templates* suspensíveis pode melhorar a performance, mas introduz complexidade no código, exigindo mais esforços de manutenção e testes. Por outro lado, (KOWALCZYK; SZANDALA, 2024) aponta que aplicações baseadas em CSR, especialmente SPAs, enfrentam desafios consideráveis em relação à indexabilidade por mecanismos de busca, o que limita seu uso em sites orientados a conteúdo.

Além disso, a adoção de SSR pode impactar negativamente a escalabilidade do sistema, como discutido por (ANGKASA et al., 2023), uma vez que o servidor passa a ter maior carga de processamento por requisição. Em contextos de alto tráfego, isso pode implicar em aumento de custos com infraestrutura e complexidade na distribuição de carga.

4.7.4 Q4: Quais trabalhos relacionados existem na literatura que abordam recomendações sobre quando usar o CSR ou o SSR?

Embora a maioria dos artigos foque em aspectos técnicos, alguns oferecem orientações práticas sobre o uso apropriado de cada abordagem. Por exemplo, (ISKANDAR et al., 2020) sugere que aplicações voltadas a conteúdo estático ou com forte dependência de SEO devem optar por SSR ou SSG. Já aplicações altamente interativas, como dashboards, sistemas internos ou PWAs, se beneficiam mais da flexibilidade e dinamismo do CSR.

O estudo publicado da WEBIST 2023 (PROCEEDINGS..., 2023) também apresenta uma síntese interessante sobre abordagens mistas, propondo que a adoção de técnicas como *server-side hydration* e *isomorphic rendering* pode combinar o melhor dos dois mundos, oferecendo equilíbrio entre tempo de carregamento e interatividade.

Apesar das recomendações, nota-se ainda uma carência de diretrizes sistematizadas

que ajudem desenvolvedores a escolherem a abordagem ideal com base em variáveis como tipo de aplicação, volume de tráfego, perfil dos usuários e metas de negócio. Essa lacuna representa uma oportunidade relevante para pesquisas futuras que proponham modelos de decisão mais robustos para contextos reais de desenvolvimento web.

5 Cronograma

Quadro 4 – Cronograma de execução da primeira etapa do TCC

Atividades	Nov	Dez	Jan	Fev	Mar	Abr	Mai
Definição de tema e análise de artigos relevantes							
Apresentação da proposta de projeto e desenvolvimento da introdução							
Definição do problema de pesquisa e Justificativa							
Fundamentação teórica e trabalhos relacionados							
Revisão de literatura sobre renderização desempenho web em abordagens CSR e SSR							
Apresentação do relatório final de TCC 1, com metodologia detalhada e cronograma atualizado							

Fonte: os autores

Quadro 5 – Cronograma de execução da segunda etapa do TCC

Atividades	Jun	Jul	Ago	Set	Out
Planejamento e definição do escopo da aplicação prática					
Desenvolvimento das aplicações com CSR e SSR					
Execução de testes de desempenho					
Avaliação e comparação dos resultados obtidos					
Consolidação da conclusão do estudo					
Revisão e entrega do relatório final do TCC 2					

Fonte: os autores

6 Estudo de Caso

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

6.1 Contexto

O estudo de caso é realizado em uma empresa fictícia.

6.2 Ciclo de Vida do Desenvolvimento de Software

7 Resultados e Discussões

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

7.1 Resultados

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

8 Conclusão

Este trabalho apresentou um estudo de caso.

Referências

Amazon Web Services. *Front-end x back-end — Diferença entre desenvolvimento de aplicações*. 2024. Acessado em 14 abr. 2025. Disponível em: <<https://aws.amazon.com/pt/compare/the-difference-between-frontend-and-backend/>>. Citado na página 19.

ANGKASA, H. et al. Improving universal rendering performance on nuxtjs-based web application. In: . [s.n.], 2023. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85187776927&doi=10.1109%2fCITSM60085.2023.10455297&partnerID=40&md5=48f7f413665c85291a128368743ec2d1>>. Citado 3 vezes nas páginas 53, 54 e 55.

ANGULAR. 2025. Acessado em: 11 de abril de 2025. Disponível em: <<https://angular.io/guide/architecture>>. Citado na página 36.

ANGULAR Universal. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://angular.io/guide/universal>>. Citado na página 37.

ANURAG. What the heck is web rendering? *Locofy Blog*, 2024. Acesso em: 15 maio 2025. Disponível em: <<https://www.locofy.ai/blog/what-the-heck-is-web-rendering>>. Citado na página 34.

ASTRO. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://astro.build/>>. Citado na página 37.

BALLERINI, R. Html, css e javascript: qual a diferença entre essas linguagens? *Alura*, 2023. Acessado em 14 abr. 2025. Disponível em: <<https://www.alura.com.br/artigos/html-css-e-js-definicoes>>. Citado 2 vezes nas páginas 19 e 20.

BEKMANOVA, G. et al. Requirements for the development of a website builder with adaptive design. In: . [s.n.], 2024. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85215519877&doi=10.1109%2fUBMK63289.2024.10773412&partnerID=40&md5=6192f2d980aa9bbad06a979bd633a64f>>. Citado 2 vezes nas páginas 53 e 54.

BOEHNCKE, G. Corporate social responsibility: Hiring requisition in media companies? *CSR, Sustainability, Ethics and Governance*, 2023. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85151569634&doi=10.1007%2f978-3-031-18976-0_7&partnerID=40&md5=87204e857ad89df0c6a61759a475f0fe>. Citado na página 53.

BOSE, T. Frontend rendering: Ssg vs isr vs ssr vs csr - when to use which? *DEV.to*, 2022. Acessado em 11 maio 2025. Disponível em: <<https://dev.to/ruppysupply/frontend-rendering-ssg-vs-isg-vs-ssr-vs-csr-when-to-use-which-5jp>>. Citado 3 vezes nas páginas 30, 31 e 32.

CARVALHO, F. M. Progressive server-side rendering with suspendable web templates. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2025. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85211240232&doi=10.1007>>

- 2f978-981-96-0576-7_33&partnerID=40&md5=48a28d5d4b14294477317d41fefa0c1f>. Citado 2 vezes nas páginas 53 e 55.
- 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 18.
- 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 15.
- EMADAMERHO-ATORI, N. Ux tips for developers. *Prismic Blog*, 2023. Disponível em: <<https://prismic.io/blog/ux-tips-for-developers>>. Citado na página 38.
- 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 9 vezes nas páginas 13, 22, 25, 26, 27, 28, 38, 40 e 41.
- FOUNDATION, T. jQuery. *jQuery - Fast, small, and feature-rich JavaScript library*. 2025. Acessado em: 11 maio 2025. Disponível em: <<https://jquery.com>>. Citado na página 36.
- Gatsby. Rendering options in gatsby. *Gatsby Documentation*, 2023. Acesso em: 15 maio 2025. Disponível em: <<https://www.gatsbyjs.com/docs/conceptual/rendering-options/>>. Citado 2 vezes nas páginas 33 e 34.
- 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 2 vezes nas páginas 12 e 39.
- 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 29.
- ISKANDAR, T. F. et al. Comparison between client-side and server-side rendering in the web development. In: . [s.n.], 2020. All Open Access, Gold Open Access. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85086446329&doi=10.1088%2f1757-899X%2f801%2f1%2f012136&partnerID=40&md5=1e76e6274991560833731fefce47c3a5>>. Citado 3 vezes nas páginas 53, 54 e 55.
- 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 20.

KATTAH, A. Html, css e javascript: Entenda as diferenças na prática. *HeroCode*, 2023. Acessado em 14 abr. 2025. Disponível em: <<https://herocode.com.br/blog/html-css-javascript-diferencas/>>. Citado 2 vezes nas páginas 20 e 21.

KESHARI, P. et al. Web development using reactjs. In: . [s.n.], 2023. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85195810274&doi=10.1109%2fICAC3N60023.2023.10541743&partnerID=40&md5=7f9f99f1b7f492e6f4f5c43c2b6b5be3>>. Citado 2 vezes nas páginas 53 e 54.

KLOCHKOV, D.; MULAWKA, J. Improving ruby on rails-based web application performance. *Information (Switzerland)*, 2021. All Open Access, Gold Open Access. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85113170428&doi=10.3390%2finfo12080319&partnerID=40&md5=23cc2185b9d757e35194c50632613e1f>>. Citado 2 vezes nas páginas 53 e 54.

KOWALCZYK, K.; SZANDALA, T. Enhancing seo in single-page web applications in contrast with multi-page applications. *IEEE Access*, 2024. All Open Access, Gold Open Access. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85182953693&doi=10.1109%2fACCESS.2024.3355740&partnerID=40&md5=20893a5fc228854aa2846fb5d47a9da4>>. Citado 2 vezes nas páginas 53 e 55.

LACOM, P.; SAGOT, S. A research framework for b2b green marketing innovation: the design of sustainable websites. In: . [s.n.], 2022. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85148699908&doi=10.1109%2fICE%2fITMC-IAMOT55089.2022.10033239&partnerID=40&md5=acca2bbae04d3cd37bc8b94e433f9855>>. Citado 2 vezes nas páginas 53 e 54.

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 2 vezes nas páginas 17 e 18.

MURPHY, C. Guide: React seo considerations and solutions. *Prismic Blog*, 2022. Disponível em: <<https://prismic.io/blog/react-seo-solutions>>. Citado 2 vezes nas páginas 38 e 39.

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 37.

NODE.JS. 2025. Acessado em: 11 de abril de 2025. Disponível em: <<https://nodejs.org/en/docs/>>. Citado na página 20.

NUXT.JS. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://nuxtjs.org/>>. Citado na página 37.

OSMANI, A.; MILLER, J. Rendering on the web. 2025. Acessado em 15 maio 2025. Disponível em: <<https://web.dev/articles/rendering-on-the-web?hl=pt-br>>. Citado 4 vezes nas páginas 21, 22, 23 e 24.

- PAVIC, F.; BRKIC, L. Methods of improving and optimizing react web-applications. In: . [s.n.], 2021. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85123053514&doi=10.23919%2fMIPRO52101.2021.9596762&partnerID=40&md5=104afe6916f3fd1c2a46c8c52ca04519>>. Citado 2 vezes nas páginas 53 e 54.
- PERERA, P. Visual explanation and comparison of csr, ssr, ssg, and isr. *DEV.to*, 2022. Acessado em 11 maio 2025. Disponível em: <<https://dev.to/pahanperera/visual-explanation-and-comparison-of-csr-ssr-ssg-and-isr-34ea>>. Citado 3 vezes nas páginas 30, 32 e 33.
- POKHRIYAL, A. et al. Single page optimization techniques using react. In: . [s.n.], 2024. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85199175705&doi=10.1201%2f9781032644752-63&partnerID=40&md5=667c007334673fbaaedbbbf1c1ce4c>>. Citado 2 vezes nas páginas 53 e 54.
- PORZIO, C. *Alpine.js - A rugged, minimal framework for composing JavaScript behavior in your HTML*. 2023. Acessado em: 11 maio 2025. Disponível em: <<https://alpinejs.dev>>. Citado na página 36.
- PROCEEDINGS of the 19th International Conference on Web Information Systems and Technologies, WEBIST 2023. In: . [s.n.], 2023. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85179780169&partnerID=40&md5=ded8f04f4e9bdb35c331c38030709335>>. Citado 2 vezes nas páginas 53 e 55.
- QWIK. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://qwik.builder.io/>>. Citado na página 37.
- REACT. 2025. Acessado em: 11 de abril de 2025. Disponível em: <<https://react.dev/learn>>. Citado na página 35.
- REMIX. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://remix.run/>>. Citado na página 37.
- SHOPIFY. *Website Load Time Statistics: Why Speed Matters in 2024*. 2024. <https://www.shopify.com/blog/website-load-time-statistics>. Acesso em: 07 maio 2025. Citado na página 39.
- 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 29.
- STUDIO, P. *The Impact of Client-Side Rendering on Accessibility*. 2023. Acessado em: 07 maio 2025. Disponível em: <<https://blog.pixelfreestudio.com/the-impact-of-client-side-rendering-on-accessibility/>>. Citado 2 vezes nas páginas 40 e 41.
- STUDIO, P. *The Impact of Client-Side Rendering on User Experience*. 2023. Acessado em: 07 maio 2025. Disponível em: <<https://blog.pixelfreestudio.com/the-impact-of-client-side-rendering-on-user-experience/>>. Citado na página 40.

- SUTTON, M. *Accessibility Tips in Single-Page Applications*. 2018. <https://www.deque.com/blog/accessibility-tips-in-single-page-applications>. Deque Systems. Acesso em: 07 maio 2025. Citado na página 41.
- SVELTE. 2025. Acesso em: 11 de abril de 2025. Disponível em: <https://svelte.dev/docs>. Citado na página 36.
- SVELTEKIT. 2024. Acesso em: 10 de abril de 2025. Disponível em: <https://kit.svelte.dev/>. Citado na página 37.
- TRADEOFFS in Server Side and Client Side Rendering. 2015. Acesso 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 15 e 16.
- VUE.JS. 2025. Acesso em: 11 de abril de 2025. Disponível em: <https://vuejs.org/guide/introduction.html>. Citado na página 36.
- 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 3 vezes nas páginas 12, 13 e 39.
- WATTS, S. *Server-Side Rendering: Benefiting UX and SEO*. 2023. Acesso em: 07 maio 2025. Disponível em: https://www.splunk.com/en_us/blog/learn/server-side-rendering-ssr.html. Citado na página 40.
- Wikipedia. *Protocolo de Transferência de Hipertexto*. 2024. Acesso em 12 de abril de 2025. Disponível em: https://pt.wikipedia.org/wiki/Hypertext_Transfer_Protocol. Citado 2 vezes nas páginas 17 e 19.
- Wikipedia. *Man-in-the-middle attack*. 2025. Acesso em 13 de abril de 2025. Disponível em: https://en.wikipedia.org/wiki/Man-in-the-middle_attack. Citado na página 19.
- ZHOU, T. et al. An integrated framework of user experience-oriented smart service requirement analysis for smart product service system development. *Advanced Engineering Informatics*, 2022. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85120077519&doi=10.1016%2fj.aei.2021.101458&partnerID=40&md5=97385fab7168f6e32650452fde90c847>. Citado 2 vezes nas páginas 53 e 54.