PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA E DE ARTES
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

WEBSOCKET COMO SOLUÇÃO PARA ENTREGA DE LANCES EM TEMPO REAL EM UMA APLICAÇÃO *WEB* DE LEILÕES

ALUISIO LUCIO DOS SANTOS NETO

GOIÂNIA
2024

ALUISIO LUCIO DOS SANTOS NETO

WEBSOCKET COMO SOLUÇÃO PARA ENTREGA DE LANCES EM TEMPO REAL EM UMA APLICAÇÃO *WEB* DE LEILÕES

Trabalho de Conclusão de Curso apresentado à Escola Politécnica e de Artes da Pontifícia Universidade Católica de Goiás como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação.

Orientadora:
Profa. Ma. Angélica da Silva Nunes.

Banca examinadora:

Prof. Me. Rafael Leal Martins
Prof. Me. Fernando Gonçalves Abadia

GOIÂNIA
2024

ALUISIO LUCIO DOS SANTOS NETO

***WEBSOCKET* COMO SOLUÇÃO PARA ENTREGA DE LANCES EM TEMPO REAL DE UMA APLICAÇÃO *WEB* DE LEILÕES**

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Ciência da Computação, em ____/____/_____.

_____

Orientadora: Profa. Ma. Angélica da Silva Nunes.

_____

Prof. Me. Rafael Leal Martins

_____

Prof. Me. Fernando Gonçalves Abadia

GOIÂNIA

2024

Dedico esta monografia a Deus, pela minha vida e pela graça abundante que me sustenta. Aos meus familiares, pelo incentivo e constante apoio. Aos meus amigos, pelo companheirismo e pelas palavras de encorajamento.

AGRADECIMENTOS

Primeiramente, a Deus pela força e sabedoria concedidas ao longo desta trajetória.

À minha mãe, Solange Gomes de Souza, pela insistência, motivação e por acreditar em mim em todos os momentos.

À Professora Angélica da Silva Nunes, orientadora acadêmica, pelo apoio, incentivo e disposição para corrigir, direcionar e ensinar.

A todos que, direta ou indiretamente, colaboraram para materialização deste trabalho.

“Mera mudança não é crescimento. Crescimento é a síntese de mudança e continuidade, e onde não há continuidade não há crescimento.”

C. S. Lewis

RESUMO

O crescimento da demanda por comunicação em tempo real em aplicações web na internet impulsionou o surgimento do protocolo *WebSocket*. Este é um protocolo de rede utilizado para esse tipo de comunicação, criando uma conexão persistente entre cliente e servidor. O *WebSocket,* introduzido em 2011, se destaca dentre outras abordagens utilizadas na internet para entrega de dados em tempo real, como o SSE e o *HTTP Long Polling*, devido à sua flexibilidade no tráfego de dados e em sua conexão bidirecional. Empresas como Tinder e Dropbox utilizam a tecnologia *WebSocket* em suas aplicações para solucionar o tráfego em tempo real e viabilizar funcionalidades que exigem essa característica. O objetivo deste trabalho é criar uma aplicação de leilões online, baseada em uma arquitetura cliente-servidor, em que a interação e as respostas instantâneas são essenciais durante o processo de lances, utilizando o protocolo *WebSocket*, para entregar os lances e garantir a fluidez, proporcionando uma experiência de uso satisfatória para o usuário final.

Palavras-chave: aplicações *web,* tempo real, *WebSocket*, leilões *online*.

ABSTRACT

The growth in demand for real-time communication in web applications on the internet has driven the emergence of the WebSocket protocol. This is a network protocol used for this type of communication, creating a persistent connection between client and server. Introduced in 2011, WebSocket stands out among other approaches used on the internet for delivering real-time data, such as SSE and HTTP Long Polling, due to its flexibility in data traffic and its bidirectional connection. Companies like Tinder and Dropbox use WebSocket technology in their applications to handle real-time traffic and enable functionalities that require this characteristic. The objective of this work is to create an online auction application, based on a client-server architecture, where interaction and instant responses are essential during the bidding process. Using the WebSocket protocol to deliver bids and ensure smoothness, providing a satisfying user experience for the end user.

Keywords: *web* applications, real-time, *WebSocket*, *online* auction.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS E ABREVIATURAS

AJAX	*Asynchronous JavaScript and XML, JavaScript* Assíncrono e XML
HTTP	*Hypertext Transfer Protocol*, Protocolo de Transferência de Hipertexto
TCP	*Transmission Control Protocol*, Protocolo de Controle de Transmissão
SSE	*Server-Sent Events*, Eventos enviados pelo servidor
HTML	*HyperText Markup Language*, Linguagem de Marcação de Hipertexto
GUID	*Globally Unique Identifier*, Identificador Único Global
AJAX	*Asynchronous JavaScript and XML, JavaScript* Assíncrono e XML
API	*Application Programming Interface, Interface* de Programação de Aplicação
DOM	*Document Object Model,* Modelo de Documento por Objetos
GUID	*Globally Unique Identifier*, Identificador Único Global
HTML	*HyperText Markup Language*, Linguagem de Marcação de Hipertexto
HTTP	*Hypertext Transfer Protocol*, Protocolo de Transferência de Hipertexto
JSON	*JavaScript Object Notation,* Notação de Objeto *JavaScript*
ms	Milissegundos
OSI	*Open System Interconnection,* Interconexão de Sistemas Abertos
SSE	*Server-Sent Events*, Eventos Enviados pelo Servidor
TCP	*Transmission Control Protocol*, Protocolo de Controle de Transmissão
UOL	Universo *Online*
URI	*Uniform Resource Identifier,* Identificador de Recurso Uniforme
URL	*Uniform Resource Locator,* Localizador Uniforme de Recursos
XML	*Extensible Markup Language*
SPA	*Single Page Application,* Aplicativo de página única
AWS	*Amazon web Services,* Serviços *web* Amazon
VPC	*Virtual Private Cloud,* Nuvem privada virtual
EC2	*Elastic Compute Cloud,* Nuvem de computação elástica

SUMÁRIO

1 INTRODUÇÃO

No início da popularização da *Internet* as páginas *web* eram estáticas e com mínima ou nenhuma interação, uma vez que o navegador enviava uma solicitação *Hypertext Transfer Protocol*, Protocolo de Transferência de Hipertexto (HTTP) para o servidor, que retornava ao navegador a página completa. Com o avanço dos navegadores e das tecnologias de comunicação em rede, as páginas *web* tornaram-se dinâmicas, mais interativas e com outras funcionalidades, evoluindo para aplicativos na *web* equivalentes às que já existiam localmente nos computadores. (CHOPRA, 2015).

Os aplicativos *web,* páginas dinâmicas e interativas, puderam ser desenvolvidos devido à popularização do *Asynchronous JavaScript and XML*, *JavaScript* Assíncrono e XML (AJAX), tecnologia que torna as solicitações ao servidor assíncronas sem a necessidade de recarregar a página inteira. Outra tecnologia que surgiu foi o modelo *Comet* que é capaz de tornar as conexões HTTP ativas por um intervalo de tempo maior. Ainda que essas tecnologias funcionem bem, elas são ineficientes e causam alguns problemas, como: o abuso de solicitações HTTP para verificar se há novas atualizações no servidor, a alocação de recursos no servidor *web* em função da maior quantidade de solicitações HTTP, além da latência na comunicação, já que a cada solicitação HTTP ocorre a abertura e o fechamento da conexão entre clientes e servidores. (LOMBARDI, 2015).

O *WebSocket* apresenta-se como uma alternativa para reduzir a latência em aplicações em tempo real, é um protocolo de camada de aplicação que tem por finalidade fornecer uma comunicação bidirecional em tempo real baseado no *Transmission Control Protocol*, Protocolo de Controle de Transmissão (TCP), entre a aplicação cliente *web* (navegador) e o servidor remoto, sem a necessidade da abertura de múltiplas conexões HTTP. Esse protocolo foi projetado para substituir tecnologias de comunicação bidirecional que usam HTTP. (IETF, 2011b).

O protocolo *WebSocket* não é a única solução para conexões de longa duração e em tempo real. Existe também o *HTTP Long Polling* e o *Server-Sent Events*, Eventos enviados pelo servidor (SSE). A vantagem de se utilizar o *WebSocket* é a comunicação bidirecional, de baixa latência e com cabeçalho reduzido, se comparado ao protocolo HTTP. (GRIGORIK, 2013).

Diversas aplicações da *web* utilizam o *WebSocket* para entrega em tempo real. Uma delas é o *site* de notícias Universo *Online* (UOL), que emprega o *WebSocket* para fornecer atualizações instantâneas de eventos importantes, como partidas de futebol. (UNIVERSO *ONLINE* (UOL), 2023).

No aplicativo Tinder, a comunicação instantânea é vital. Assim, o *WebSocket* facilita a entrega imediata de *matches* e mensagens, entre os usuários. (DYANKOV, 2019).

A ferramenta corporativa Slack utiliza o *WebSocket* para a troca rápida de mensagens, para indicar quais contatos estão *online* e para a integração eficiente de outros serviços. (STELDT, 2021).

Diante do contexto apresentado, este trabalho pretende responder a seguinte questão: Como a implementação de uma aplicação *web* de leilões com *WebSocket* pode otimizar a experiência em tempo real dos lances, promovendo eficiência nas transações de leilões *online*?

1.1 Objetivo geral

Implementar uma aplicação *web* de leilões que utiliza a tecnologia *WebSocket* para a transmissão eficiente e em tempo real dos lances.

1.2 Objetivos específicos

- Pesquisar e revisar a literatura relacionada a leilões *online*, tecnologia *WebSocket* e aplicações *web* em tempo real;
- Integrar a tecnologia *WebSocket* para possibilitar a comunicação eficiente e em tempo real entre o servidor e os clientes durante os leilões;
- Avaliar o desempenho da aplicação, especialmente a eficiência da transmissão em tempo real, identificando possíveis melhorias e otimizações.

1.3 Procedimentos metodológicos

O presente estudo, quanto a sua natureza, se trata de um resumo de assunto, visto que condensa, sintetiza, analisa e discute conhecimentos, informações e dados sobre o tema proposto, aplicando o conhecimento abordado. (WAZILAWICK, 2014).

Em relação aos objetivos esta pesquisa classifica-se como explicativa, haja vista que o intuito é estabelecer uma linha de entendimento completo sobre o tema, discorrendo sobre o impacto, vantagens e em quais contextos o uso de *WebSocket* pode ser eficaz, buscando analisar a entrega de conteúdo em tempo real. Nesse sentido, a pesquisa é mais completa por buscar estudar os dados e entender suas causas e explicações. (WAZILAWICK, 2014).

No que tange aos procedimentos técnicos, caracteriza-se como uma pesquisa bibliográfica e experimental. Este procedimento é caracterizado pela manipulação de um aspecto da realidade. A pesquisa experimental implica em se ter uma ou mais variáveis experimentais que podem ser controladas, cuja medição pode levar, possivelmente, à conclusão de que existe algum tipo de dependência com a variável experimental. No contexto do tema abordado tal variável seria o comportamento do sistema baseado em *WebSocket*, em relação a sua entrega de conteúdo em tempo real em aplicativos *web*. (WAZILAWICK, 2014).

1.4 Estrutura da monografia

A estrutura desta monografia foi elaborada com o intuito de proporcionar uma compreensão concisa do uso do protocolo *WebSocket* em uma aplicação web de leilões *online*. A seguir, apresenta-se uma visão geral dos capítulos que compõem este trabalho.

No capítulo “Referencial teórico” é feita uma revisão da literatura sobre as principais tecnologias e métodos de comunicação web em tempo real. Inicialmente, abordam-se as técnicas AJAX e Comet, seguido de uma análise das alternativas tecnológicas, como SSE e *WebSocket*. Também é apresentado um comparativo entre essas tecnologias, destacando suas vantagens e desvantagens. A seguir, o capítulo se aprofunda no *WebSocket*, detalhando sua história, funcionamento, e aspectos técnicos do protocolo.

O capítulo “Estudo de casos” explora casos reais de aplicação do *WebSocket* em plataformas conhecidas. São analisados como o Tinder utiliza o *WebSocket* para gerenciar mensagens, e como o *Dropbox Replay* sincroniza os seus usuários.

No capítulo “Construção de um site de leilões online com *WebSocket”*, é descrito o desenvolvimento de uma aplicação web de leilões *online* utilizando *WebSocket*. A seção inicia detalhando a aplicação web como um todo, seguida de uma explicação da arquitetura da aplicação, abrangendo o *frontend*, *backend*, API, banco de dados, e uso do Docker. Também é abordado o versionamento do código fonte e a infraestrutura necessária, incluindo o ambiente de desenvolvimento, hospedagem e disponibilização da aplicação. Por fim, são apresentados trechos do código da aplicação cliente-servidor, com foco no cliente *WebSocket* e no servidor *WebSocket*.

Em “Resultados obtidos”, são apresentados e analisados os resultados do desenvolvimento da aplicação. São abordadas as validações e testes realizados, incluindo desempenho das páginas web e funcionamento do *WebSocket*, além de um comparativo entre *HTTP Long Polling* e *WebSocket*. O capítulo finaliza com uma discussão sobre os problemas enfrentados durante o desenvolvimento e execução da aplicação.

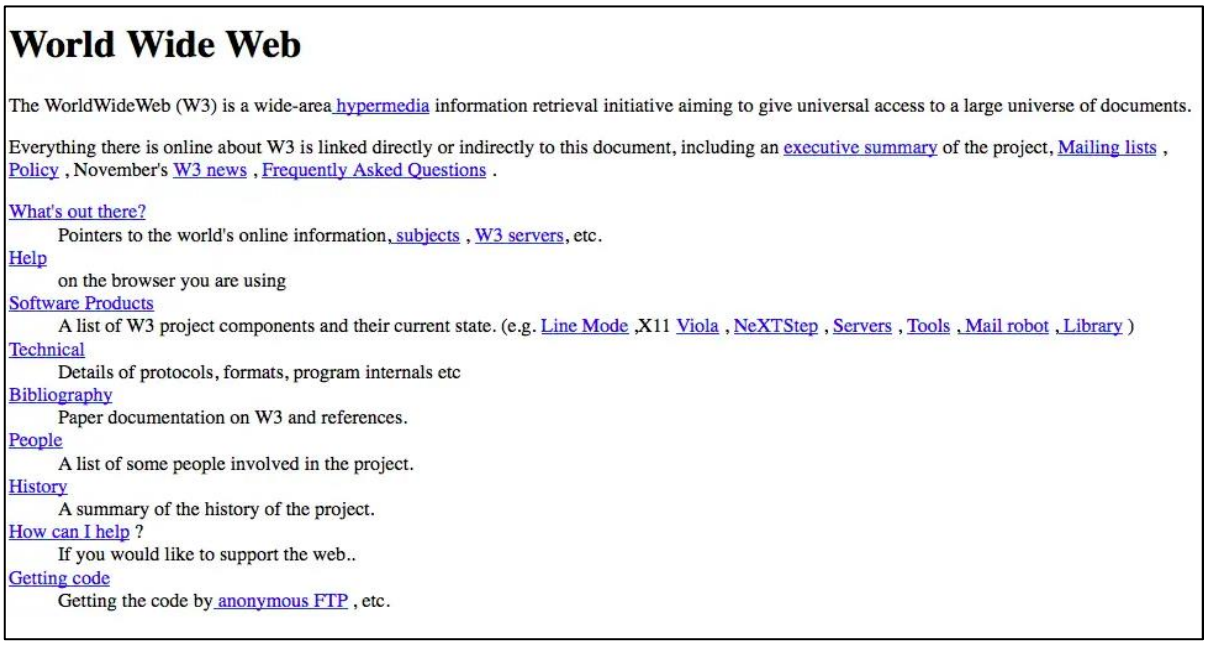
Por fim, o capítulo “Considerações finais”, encerra o assunto do trabalho discorrendo sobre os resultados alcançados e as lições aprendidas. São feitas sugestões de trabalhos futuros, apontando possíveis melhorias e expansões que podem ser realizadas a partir deste estudo.

2 REFERENCIAL TEÓRICO

A *Internet* desde seu surgimento sofreu muitas alterações e evoluiu rapidamente tanto em termos de funcionamento, quanto em termos de alcance e relevância. Segundo Grigorik (2013), pode-se dizer que a *Internet* oferece três tipos de experiência: o documento de hipertexto, a página *web* e o aplicativo *web*.

O documento de hipertexto foi o princípio da *World Wide web,* páginas estáticas de texto simples com nível básico de formatação e suporte a *hiperlinks*. A Figura 1 mostra o exemplo de uma página de documento hipertexto. (GRIGORIK, 2013).

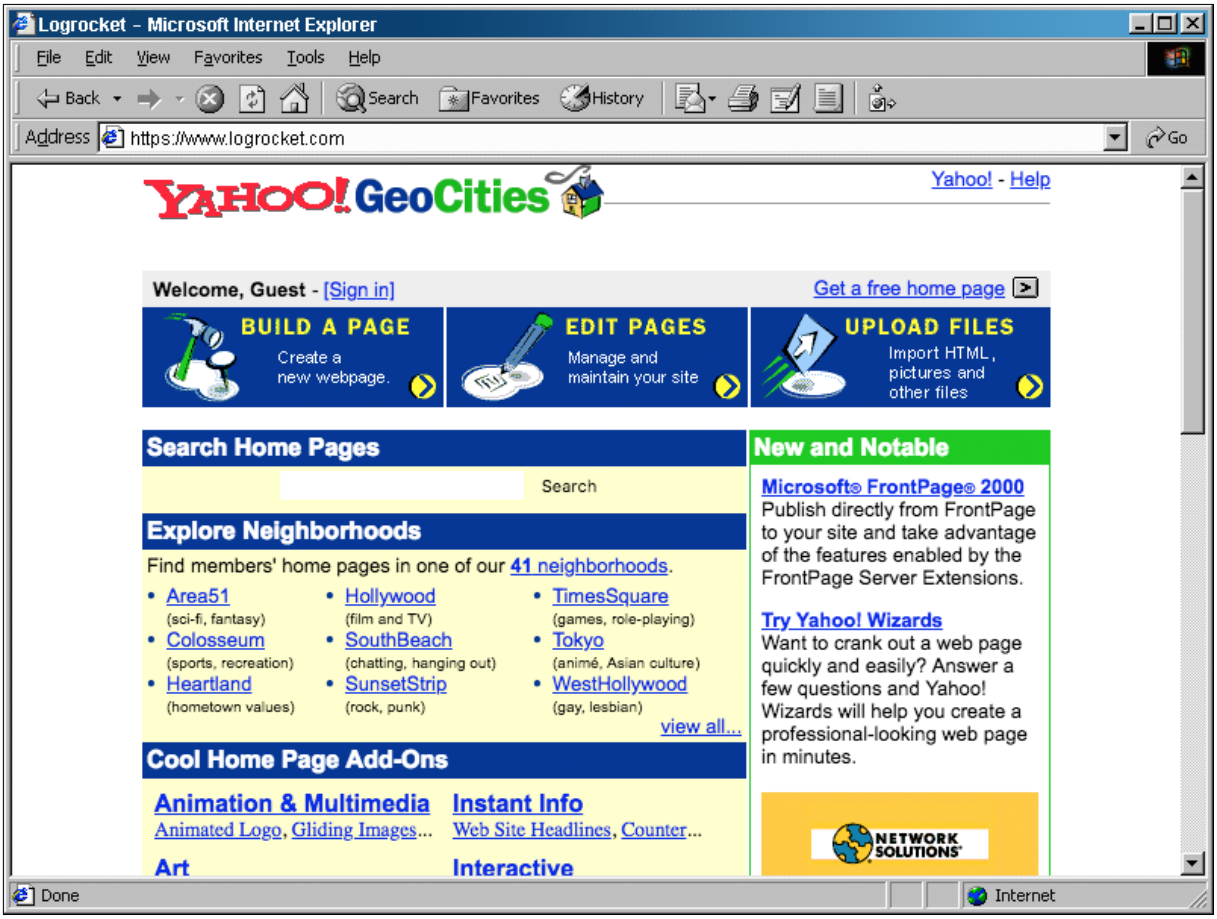
Figura 1: *Site* cópia da primeira página da *web*



Fonte: CERN, 2013

A página *web* se beneficiou do avanço dos navegadores que passaram a suportar mídias adicionais como imagens e áudios, além de ser possível a construção de *layouts* mais ricos, tornando as páginas *web* mais atrativas e visualmente mais bonitas. Contudo, ainda sim pareciam uma página impressa, pois não possuía interatividade. A Figura 2 é um exemplo de uma página *web*. (GRIGORIK, 2013).

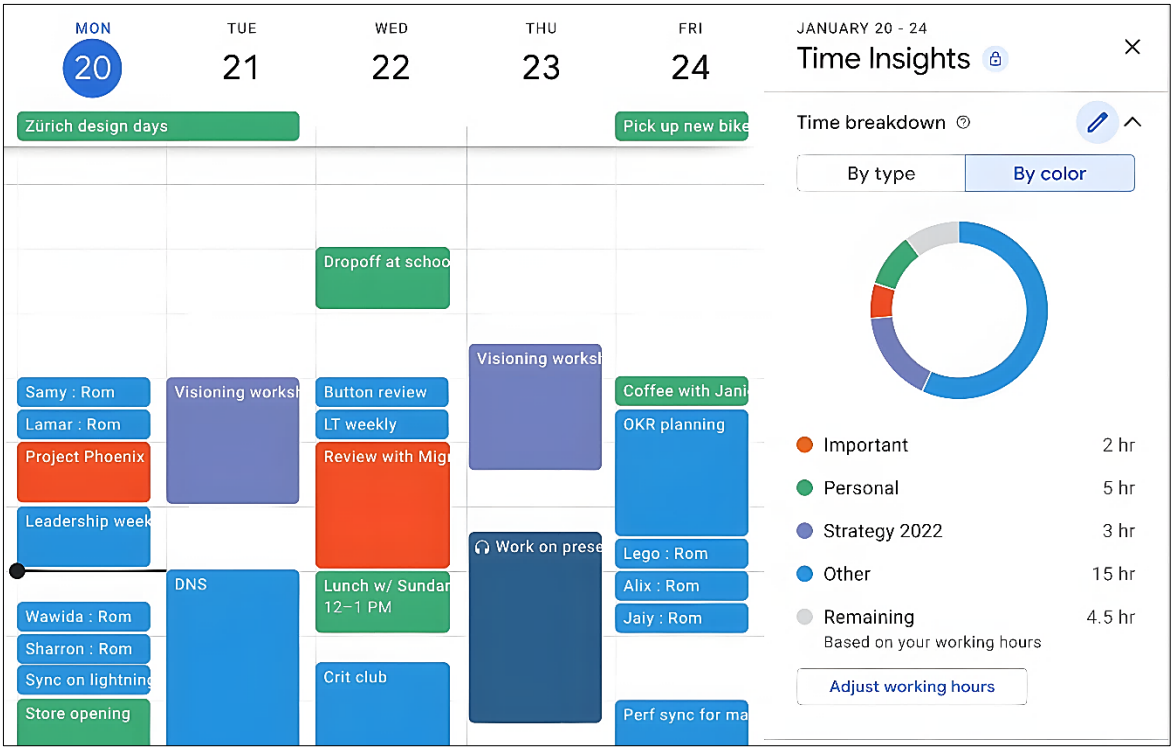
Figura 2: Página principal do *site* Geocities



Fonte: HARRISON, 2018

A incorporação da linguagem JavaScript no navegador, o *HyperText Markup Language*, Linguagem de Marcação de Hipertexto (HTML) dinâmico e o AJAX, tornam possível o aplicativo da *web* que é capaz de fornecer interatividade do usuário no navegador, sendo dessa forma, mais complexa em termos de estilo, além de depender de *scripts* e folhas de marcação e oferecendo a possibilidade de ter funcionalidades encontradas anteriormente apenas em aplicações *desktop*. A Figura 3 mostra o exemplo do Google Agenda. (GRIGORIK, 2013).

Figura 3: Página do Google Agenda



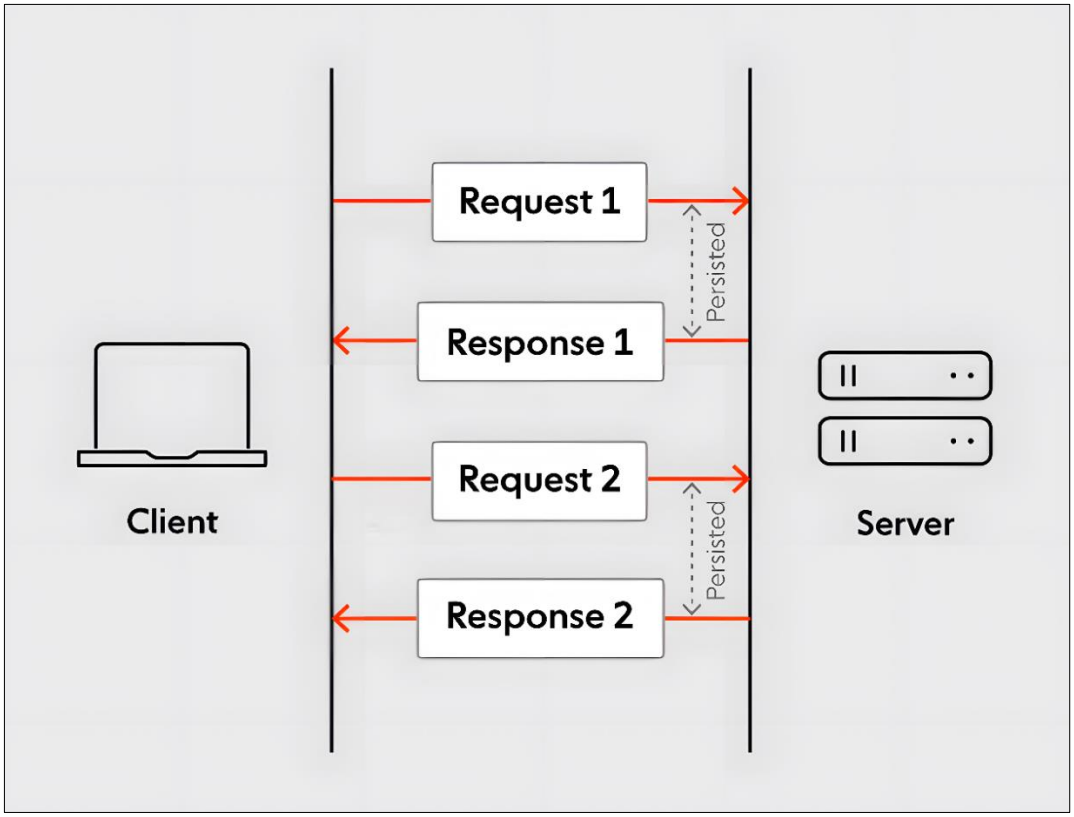
Fonte: *Google Workspace Updates Blog*, 2022

2.1 AJAX e *Comet*

A experiência de interatividade, responsividade e dinamismo que os aplicativos *web* oferecem aos usuários finais, se tornaram viáveis devido a modelos como o AJAX e o *Comet* que passaram a ser utilizados para empregar tais funcionalidades. (DIACONU, 2023).

O protocolo HTTP é baseado em solicitação/resposta, como mostra a Figura 4, e define três entidades envolvidas na comunicação, o cliente, o servidor e os *proxies*. Um cliente cria conexões com um servidor com a finalidade de enviar solicitações HTTP. Um servidor aceita conexões e atende solicitações HTTP de clientes, se aceitas, o servidor envia uma resposta as essas solicitações. Os *proxies* são intermediários entre o cliente e o servidor que se envolvem na entrega das solicitações HTTP, entre o cliente e o servidor e vice-versa. (IETF, 2011a).

Figura 4: Esquema que representa a comunicação HTTP



Fonte: Stichbury, 2023

Em uma comunicação padrão HTTP o servidor não pode iniciar uma conexão com o cliente e nem enviar respostas para clientes dos quais não fizeram uma solicitação HTTP, ou seja, o servidor não pode efetuar eventos assíncronos para os clientes. Para tanto, o deve realizar solicitações ao servidor periodicamente em busca de atualizações e novos conteúdos. (IETF, 2011a).

Com o objetivo de tornar possível eventos assíncronos e melhorar interatividade na *web,* foram desenvolvidos mecanismos e técnicas que foram implementados para serem utilizados com o HTTP, que agrupadas são chamadas de *Comet.* O *Comet* é um modelo de *design* de aplicativos *web* que viabiliza que o servidor envie dados ao navegador, sem que haja a solicitação prévia. Ele permite a comunicação assíncrona, através de conexões HTTP de longa duração, habilitando o servidor a enviar atualizações, quando disponíveis para o navegador. O modelo *Comet* se popularizou graças a organizações como o Google que implementou o modelo no *chat* de bate-papo do Gmail. Por se tratar de um modelo, o *Comet* pode

ser aplicado com o uso de várias técnicas, como *HTTP Long Polling* e HTTP *Streaming*. (DIACONU, 2023).

O *HTTP Long Polling* é uma técnica na qual o servidor tenta manter uma conexão HTTP ativa por uma longa duração com a finalidade de enviar eventos à medida que estes ocorrem, minimizando a latência de entrega das mensagens. Já no *HTTP Streamining* o servidor mantém uma conexão aberta indefinidamente, sem finalizar a conexão, mesmo após responder a solicitação do cliente. (IETF, 2011a).

O AJAX é um modelo que sugere uma forma de utilização do objeto *XMLHttpRequest* para comunicação de *script* no lado do servidor, tanto para enviar quanto para receber dados em diversos formatos, como *JavaScript Object Notation,* Notação de Objeto *JavaScript* (JSON) e *Extensible Markup Language* (XML). Além disso, o AJAX tem o aspecto de ser assíncrono, o que permite que as informações e os dados apareçam sem a necessidade de atualizar a página inteira, permitindo a atualização de partes, conforme a navegação do usuário. (MDN, 2023a; MDN, 2023b).

Os aplicativos *web* que proporcionam dinamismo, interatividade e recursos em tempo real, somente se tornaram viáveis graças ao AJAX e ao *Comet*. Contudo, tanto AJAX quanto o *Comet* têm suas limitações e deficiências. Grande parte de suas limitações são resultantes da utilização de HTTP como protocolo de comunicação. O HTTP foi desenvolvido para servir recursos no modelo solicitação e resposta e não foi pensado e otimizado para lidar com conexões contínuas, para disponibilizar recursos em tempo real para aplicativos *web*. (DIACONU, 2023).

Ao utilizar o HTTP para emular a *web* em tempo real, causa algumas desvantagens, como a escalabilidade limitada, já que a pesquisa HTTP envolve solicitações ao servidor em intervalos fixos de tempo para verificar se há novidades para recuperar o que ocasiona em aumento de tráfego de rede e demanda de servidor e quanto maior o número de usuários, maior será o gargalo e a demanda. (DIACONU, 2023).

Outra desvantagem do HTTP nesse cenário é a ausência de comunicação bidirecional, pois o HTTP por ser baseado em solicitação e resposta, não é capaz de suportar comunicação em ambos os sentidos na mesma conexão ativa, contínua e em tempo real entre cliente e servidor. (DIACONU, 2023).

2.2 Tecnologias e Métodos Alternativos

Existem técnicas e métodos alternativos, mais modernos e eficientes que foram desenvolvidos para implementação de comunicação em tempo real e assíncrona, métodos que são mais adequados atualmente que as abordagens do *Comet*, como o *HTTP Long Polling*, e o AJAX. (GRIGORIK, 2013).

2.2.1 Server Sent Events (SSE)

O SSE permite uma comunicação eficiente com dados de texto enviados do servidor para o cliente. Para alcançar essa finalidade, ele utiliza de dois componentes: a *interface EventSource* no navegador que permite que o cliente receba notificações em forma de eventos da DOM do servidor e o *Event Stream* que é o formato de dados, usado para entregar atualizações individuais. (GRIGORIK, 2013).

No SSE os navegadores assinam um fluxo de eventos por meio da *interface EventSource* e recebem atualizações cada vez que surge um novo evento no servidor. A *EventSource* aceita uma conexão de fluxo HTTP de uma *Uniform Resource Locator,* Localizador Uniforme de Recursos (URL) específica e mantem a conexão aberta enquanto busca eventos novos disponíveis. Os eventos são enviados pelo servidor, em vez de solicitados pelo cliente ou extraídos. (MARTIN, 2023).

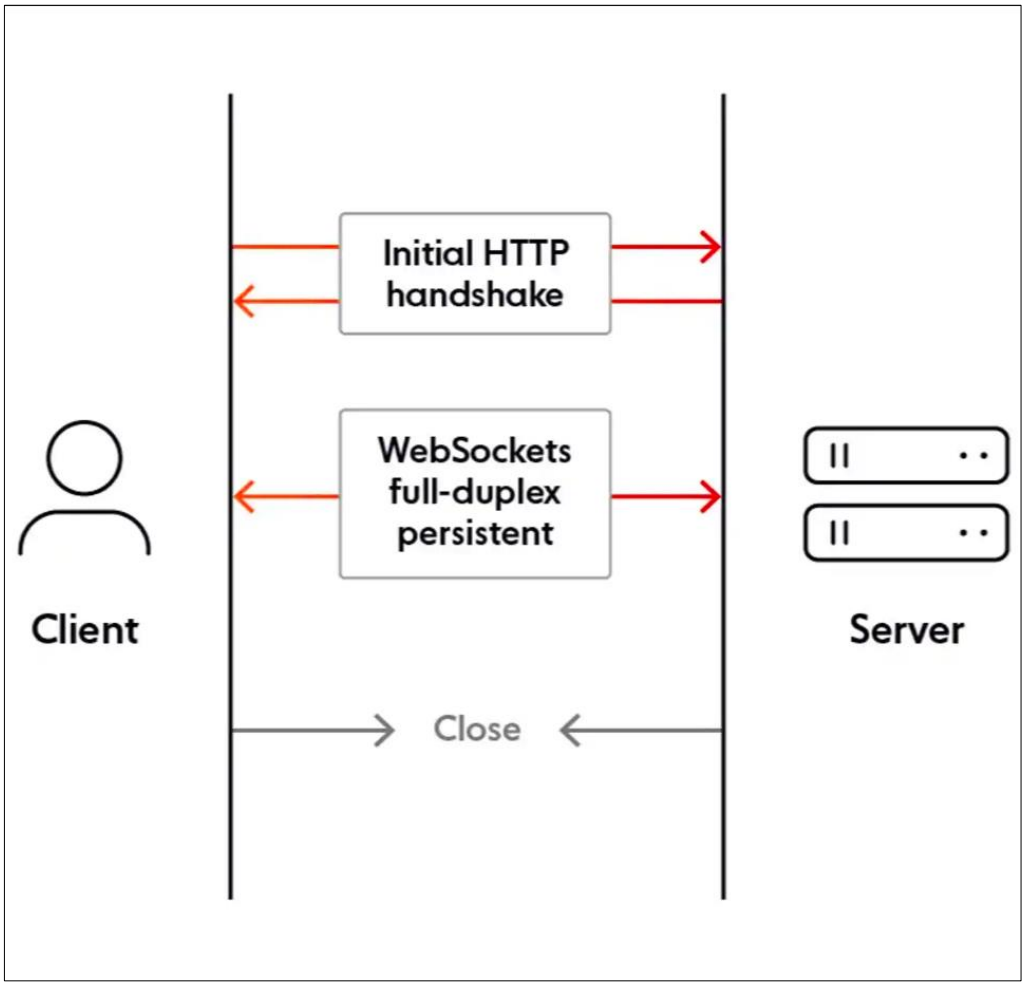
A junção dessas duas tecnologias, o *EventSource* e o *Event Stream*, fazem do SSE uma ferramenta poderosa para lidar com dados em tempo real no navegador, entregando dados com baixa latência em uma conexão HTTP única e persistente. (GRIGORIK, 2013).

2.2.2 WebSocket

O *WebSocket* é um protocolo de comunicação em rede que está presente na camada de aplicação do modelo *Open System Interconnection,* Interconexão de Sistemas Abertos (OSI), e depende do TCP na camada de transporte. Fornece uma comunicação *full-duplex* em uma única conexão TCP. Seu objetivo é fornecer uma comunicação bidirecional (ambos os sentidos) em tempo real baseado em TCP, entre uma aplicação *web* (navegador) e um servidor remoto, sem a necessidade da abertura de múltiplas conexões HTTP. (IETF, 2011b).

A maior diferença do *WebSocket* em relação a outras abordagens, como SSE e *HTTP Long Polling* é sua capacidade de estabelecer conexões bidirecionais, o que permite que ao cliente e ao servidor enviarem atualizações e mensagens assim que estas ocorrem. O *WebSocket* é uma das maneiras de trafegar dados mais versáteis e flexíveis que existe no navegador, já que suporta tanto texto quanto dados binários, além de possuir baixa latência devido a abertura de apenas uma única conexão e mantê-la aberta de forma indefinida até que seja solicitado o seu fechamento. A Figura 5 exemplifica como é uma conexão *WebSocket*. (GRIGORIK, 2013).

Figura 5: Esquema que mostra uma conexão *WebSocket*.



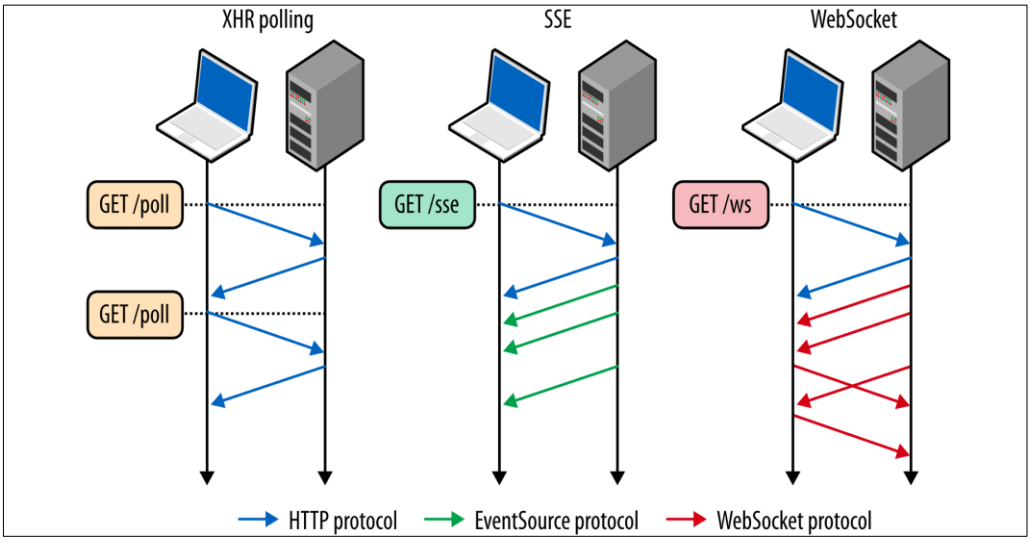
Fonte: Stichbury, 2023

2.2.3 Comparativo

Analisando o mercado de desenvolvimento de *software* verifica-se a tendência crescente da utilização de *cloud computing* para hospedar aplicações *web*. Empresas como Google e Microsoft cobram por esses serviços proporcional ao consumo de recursos, como processamento, memória e banda de rede. Assim, soluções que utilizam modelos como o *Comet* que aplica *HTTP Long Polling*, que segundo Diaconu (2023), necessariamente deve realizar solicitações HTTP periodicamente ao servidor em busca de atualizações, são uma alternativa custosa em termos de recursos de rede e *hardware* e aplicado ao cenário de hospedagem na nuvem, consequentemente, mais onerosos.

Alternativas mais modernas e eficientes que *HTTP Long Polling* são o SSE e o *WebSocket*, como mostra a Figura 6, que se assemelham muito em termos de funcionamento e são ideias para comunicação em tempo real e assíncrona.

Figura 6: Esquema comparativo entre as comunicações HTTP, SSE e *WebSocket*.



Fonte: GRIGORIK, 2013

A diferença entre o SSE e o *WebSocket* está relacionada ao tipo de dado que pode ser trafegado. No caso do SSE é aceito apenas texto e no *WebSocket*, além do texto, o formato binário também é aceito fazendo do *WebSocket* mais versátil. Além do tipo de dados, outra diferença está na comunicação que no SSE é unidirecional, do servidor para o cliente e já no *WebSocket* a comunicação é bidirecional, do cliente

para o servidor, do servidor para o cliente e, se necessário ao mesmo tempo em ambos os sentidos. O Quadro 1 sintetiza as principais características de cada método. (GRIGORIK, 2013).

Quadro 1: Quadro resumo das tecnologias de comunicação *HTTP Long Polling*, SSE e *WebSocket*.

	HTTP Long Polling	SSE	*WebSocket*
Comunicação	Unidirecional	Unidirecional	Bidirecional
Conexão	Estabelece uma nova a cada solicitação	Única e persistente	Única e persistente
Latência	Alta	Baixa	Baixa
Dados aceitos	Texto e binário	Texto	Texto e binário
Custo de banda	Alto	Baixa	Baixa

Fonte: Elaborada pelo autor desse trabalho, dados retirados de GRIGORIK, 2013

2.3 WebSocket

O protocolo de rede *WebSocket* é relativamente novo na *web* tendo surgido em 2011 oficialmente documentado na RFC 6455 e amplamente suportado pelos principais navegadores de *Internet*. O *WebSocket* é um canal de comunicação assíncrono em tempo real e orientado a eventos. (IETF, 2011b).

2.3.1 História

O *WebSocket* foi referenciado pela primeira vez como *TCPConnection* na especificação do HTML5, como um espaço reservado para uma *Application Programming Interface, Interface* de Programação de Aplicação (API) de *sockets* baseado em TCP. Em julho de 2008, Michael Carter liderou uma série de discussões que resultaram na primeira versão do protocolo conhecido como *WebSocket*. O nome "*WebSocket*" seria de fato empregado, logo após, somente quando incluído na especificação do HTML5 por Ian Hickson. Em dezembro de 2009 o Google Chrome 4 foi o primeiro navegador a ter suporte para o protocolo. Em 2011 já aceito por diversos navegadores, por padrão, a RFC 6455 foi concluída por Ian Fette em dezembro do mesmo ano. (WIRESHARK, 2020).

2.3.2 Visão Geral

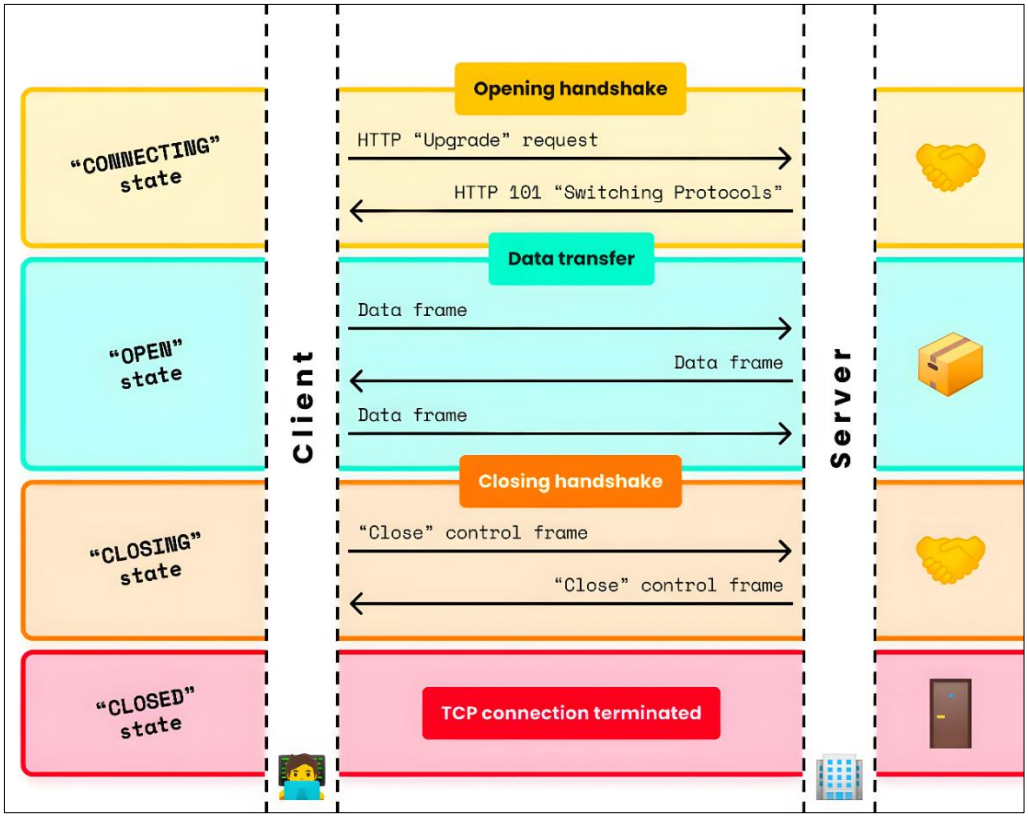
O *WebSocket* é um protocolo de comunicação em rede que está presente na camada de aplicação do modelo OSI, e depende do TCP na camada de transporte. Fornece uma comunicação *full-duplex* em uma única conexão TCP. Seu objetivo é fornecer uma comunicação bidirecional em tempo real baseado em TCP, entre uma aplicação *web* e um servidor remoto, sem a necessidade da abertura de múltiplas conexões HTTP. Foi projetado para funcionar nas portas HTTP 80 e 443 (IETF, 2011b).

Uma comunicação *duplex* é um sistema composto por diversos dispositivos conectados em rede que podem se comunicar uns com os outros. Em um sistema *half-duplex*, ambas as partes, cliente e servidor, podem se comunicar, mas não simultaneamente; a comunicação é em uma direção de cada vez, os dados são enviados por uma das partes (cliente) e recebido por outra (servidor) e um próximo pacote de dados não pode ser enviado até que haja a confirmação do recebimento anterior, esse seria o caso do HTTP. Em um sistema *full-duplex*, como o *WebSocket*, ambas as partes, cliente e servidor, podem se comunicar simultaneamente. (IBM, 2023).

2.3.3 Funcionamento do protocolo

O protocolo *WebSocket* tem duas partes: o *handshake* e a transferência de dados. O *handshake* tem como finalidade estabelecer a conexão entre o navegador e servidor. Após a conexão ser estabelecida, a transferência de dados pode ser feita por meio de quadros, que possuem os dados e outras informações pertinentes. A conexão pode ser encerrada com o fechamento do *handshake,* ao enviar um quadro de fechamento de conexão, que é caracterizado por um código específico, chamado de *opcode*. A Figura 7 exemplifica o ciclo de vida e as etapas da conexão *WebSocket*. (LOMBARDI, 2015).

Figura 7: Esquema gráfico do ciclo de vida da conexão *WebSocket*.



Fonte: MAGRINI, 2021

2.3.3.1 Abertura do *Handshake*

O cliente inicia a abertura do *handshake*, fazendo uma solicitação HTTP com cabeçalhos específicos para estabelecer a conexão. O que indica que a conexão foi estabelecida é o cabeçalho *Sec-WebSocket-Accept* que é retornado pelo servidor e seu valor é um *hash* que é um *Globally Unique Identifier*, Identificador Único Global (GUID), é um número inteiro de 128 *bits* utilizado para identificação. (LOMBARDI, 2015).

O campo de cabeçalho *Sec-WebSocket-Accept* indica se o servidor está disposto a aceitar a conexão. Se presente, este cabeçalho deve incluir um *hash* do *nonce* do cliente enviado em *Sec-WebSocket-Key* junto com um GUID predefinido. Qualquer outro valor não deve ser interpretado como uma aceitação da conexão pelo servidor. (IETF, 2011b).

O servidor ao receber a solicitação do cliente com o campo *Upgrade: Connection* no *handshake,* que indica a alteração do protocolo HTTP para *WebSocket,*

deve enviar uma resposta que o protocolo está sendo alterado. Se o valor no cabeçalho de solicitação for desconhecido ou incorreto, o servidor deve retornar o erro "*400 Bad Request*" e encerrar a conexão. (LOMBARDI, 2015).

O *Sec-WebSocket-Key* no lado do cliente é um valor randômico único. A API do JavaScript gera esse valor randômico único automaticamente, isso para navegadores que suportam o protocolo *WebSocket*. O *Sec-WebSocket-Accept* é formado pela derivação do *Sec-WebSocket-Key* que o cliente enviou, e para obtê-lo deve ser concatenado o *Sec-WebSocket-Key* do cliente e a *string* "258EAFA5-E914-47DA-95CA-C5AB0DC85B11", que é fixo e definido pela RFC 6455. Após isso, é gerado o *hash* SHA-1 do resultado da concatenação e retornado o código base64 do *hash*. (IETF, 2011b).

Ao receber uma solicitação de atualização válida com todos os campos necessários, o servidor decide sobre o protocolo aceito e enviará de volta uma resposta HTTP com o código de *status* 101, juntamente com o reconhecimento do *handshake Sec-WebSocket-Accept.* Todos os campos de cabeçalho do protocolo *WebSocket* são apresentados no Quadro 2 (LOMBARDI, 2015).

Quadro 2: Campos do cabeçalho *WebSocket*.

Cabeçalho	**Requerido**	**Valor**
Host	Sim	Campo de cabeçalho que contém a autoridade do servidor.
Upgrade	Sim	*WebSocket*
Connection	Sim	Upgrade (Solicitação para realizar uma atualização para o protocolo *WebSocket*)
Sec-WebSocket-Key	Sim	Campo de cabeçalho com um valor codificado em base64 que, quando decodificado, tem um comprimento de 16 bytes. (Chave gerada automaticamente para verificar o suporte ao protocolo do servidor)
Sec-WebSocket-Version	Sim	13 (Versão do protocolo *WebSocket* usada pelo cliente)
Origin	Não	Este campo de cabeçalho é enviado por todos os clientes do navegador. Uma tentativa de conexão que não inclui este campo de cabeçalho não deve ser interpretada como originária de um cliente do navegador.
Sec-WebSocket-Accept	Sim (servidor)	O servidor envia de volta um reconhecimento que é descrito após a tabela e deve estar presente para que a conexão seja válida.

Continua...

Cabeçalho	Requerido	Valor
Sec-WebSocket-Protocol	Não	Este campo contém uma lista de valores que indicam quais protocolos o cliente gostaria de utilizar, ordenados por preferência.
Sec-WebSocket-Extensions	Não	Este campo contém uma lista de valores que indicam quais extensões o cliente gostaria de usar.

Fonte: Elaborado pelo autor desse trabalho, dados retirados de LOMBARDI, 2015.

A Figura 8 mostra o cabeçalho de solicitação enviado pelo cliente e a Figura 9 mostra o cabeçalho de resposta enviado pelo servidor ao estabelecer a conexão com sucesso (LOMBARDI, 2015).

Figura 8: Cabeçalho de solicitação enviado pelo cliente.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Fonte: IETF, 2011b

Figura 9: Cabeçalho de resposta do servidor *WebSocket* ao estabelecer conexão com sucesso.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Fonte: IETF, 2011b

2.3.3.2 Transferência de dados

Os dados e mensagens são trafegados por meio de uma sequência de quadros. O quadro é uma sintaxe binária que contém as seguintes informações de acordo com a RFC 6455:

- *Fin Bit*: indica se o quadro atual é o último ou se há uma continuação;
- *RSV 1, 2, 3*: indica se há dados para extensões habilitadas;
- *Opcode*: indica se o quadro atual é um quadro de dados ou de comando;
- *Mask*: indica se o quadro possui mascaramento;
- *Payload length*: indica o tamanho da carga útil do quadro;
- *Masking key*: são 4 *bytes* reservados para a chave de mascaramento, caso o quadro possua máscara;
- *Payload data*: indica em *bytes* o conteúdo da mensagem a ser enviada, seja binário ou uma *string* UTF-8.

O primeiro *bit* no quadro é o *Fin Bit* que, se ativado, indica que o quadro corrente é o último quadro da mensagem. Caso contrário, se estiver desativado, a mensagem não está completa e possui mais quadros e para essa situação o *opcode* a ser passado é 0x00. (IETF, 2011b).

Os próximos três *bits* após o *Fin Bit*, RSV1, RSV2 e RSV3 são utilizados para conter dados relacionados a quaisquer extensões habilitadas na conexão e senão houver nenhuma extensão habilitada, os três serão preenchidos com zero. (IETF, 2011b).

O *opcode* é formado por outros quatro *bits* e indica como deve ser interpretado um quadro, se é um quadro de comando ou de dados. Cada quadro possui um *opcode* que identifica o que ele representa de acordo com as definições da RFC 6455. A Tabela 3 informa todos os *opcodes* definidos na RFC 6455. (IETF, 2011b).

Quadro 3: Valores que o *opcode* pode assumir, definidos pela RFC 6455.

Valor Opcode	**Descrição**
0x00	Quadro de Continuação da carga útil do quadro anterior.
0x01	Quadro de Texto; este quadro inclui dados de texto UTF-8.
0x02	Quadro Binário; este quadro inclui dados binários
0x08	Quadro de Fechamento de Conexão; este quadro encerra a conexão.
0x09	Quadro de Ping; este quadro é um ping.
0x0a	Quadro de Pong; este quadro é um pong.

Continua ...

Valor Opcode	Descrição
0x0b-0x0f	Reservado para futuros quadros de controle.

Fonte: Elaborado pelo autor desse trabalho, dados retirados de LOMBARDI, 2015.

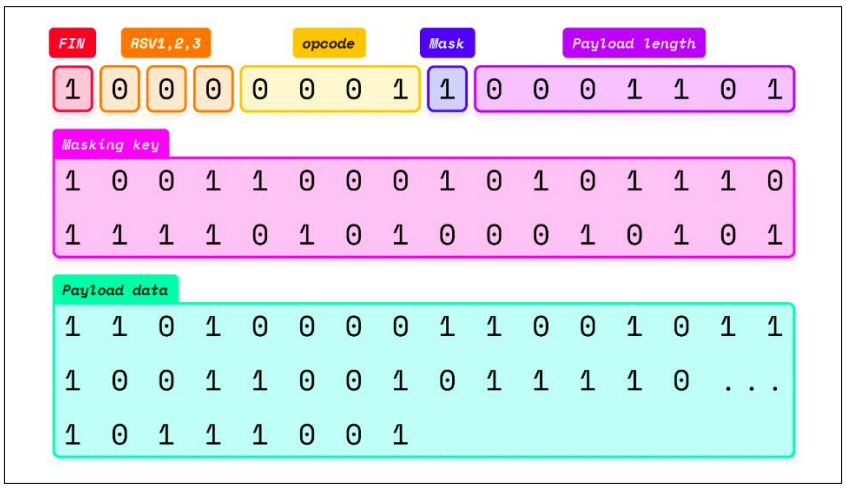
Após o *opcode*, um único *bit Mask* indica se os dados foram mascarados ou não. Na conexão *WebSocket* o mascaramento é a criptografia simétrica dos dados de uma mensagem, que deve ser feito em todas as mensagens do cliente para o servidor, enquanto o servidor não faz o mascaramento dos dados que envia. (IETF, 2011b).

O *Payload length* são os *bits* que indicam a carga útil do quadro em bytes. Os *bits* de *Masking key* representam o comprimento da carga útil, caso a mensagem possua mascaramento, a chave de 4 bytes usada para criptografas os dados. Por fim o *Payload data* indica os bytes que compõe o conteúdo da mensagem. (IETF, 2011b).

Uma mensagem *WebSocket* pode ser composta por vários quadros, e como a comunicação entre o cliente e o servidor é bidirecional, a qualquer momento, qualquer lado pode decidir enviar dados de ida e volta, desde que nenhum quadro de fechamento tenha sido enviado anteriormente por qualquer lado. (LOMBARDI, 2015).

A Figura 10 ilustra o formato de como é um quadro e como estão organizadas as informações:

Figura 10: Representação do quadro de transferência de dados.



Fonte: MAGRINI, 2021

2.3.3.3 Fechamento do *Handshake*

Ao contrário do TCP, no qual as conexões podem ser fechadas a qualquer momento sem aviso prévio, o fechamento do *WebSocket* é um *handshake* de ambos os lados. O fechamento do *handshake* para encerrar uma conexão, se dá pelo envio de um quadro de comando com o *opcode* de 0x08 e pode ser efetuado tanto pelo cliente quanto pelo servidor. (LOMBARDI, 2015).

Se o cliente enviar o quadro de fechamento ele deve ser mascarado, já que o cliente deve mascarar a mensagem. Caso seja o servidor a enviar o quadro de fechamento ele não deve ser mascarado. Além do *opcode*, o quadro de fechamento pode conter um código e uma mensagem que indica o motivo do fechamento. O Quadro 4, definida pela RFC 6455, mostra os códigos de *status* disponíveis para um evento de fechamento do *WebSocket*. (LOMBARDI, 2015).

Quadro 4: Códigos de *status* para um evento de fechamento.

Código	**Significado**	**Descrição**
1000	*Normal closure*	Envie esse código quando sua aplicação tiver sido concluída com sucesso.
1001	*Going away*	Envie este código quando o servidor ou aplicação cliente estiver encerrando ou fechando sem a expectativa de continuar.
1002	*Protocol error*	Envie este código quando a conexão estiver sendo encerrada devido a um erro de protocolo.
1003	*Unsupported data*	Envie este código quando sua aplicação receber uma mensagem de um tipo inesperado que ela não pode manipular
1004	*Reserved*	Não use; isso está reservado conforme o RFC 6455.
1005	*No status rcvd*	Não use; a API usará isso para indicar quando nenhum código válido foi recebido.
1006	*Abnormal closure*	Não use; a API usará isso para indicar que a conexão foi encerrada de forma anormal.
1007	*Invalid frame payload data*	Envie este código se os dados na mensagem recebida não estiverem consistentes com o tipo da mensagem (por exemplo, não estiverem em UTF-8).
1008	*Policy violation*	Este é um código de status genérico que pode ser retornado quando não há mais códigos de status adequados disponíveis.

Continua ... .

Código	Significado	Descrição
1009	*Message too big*	Envie este código quando a mensagem recebida for muito grande para ser processada.
1010	*Mandatory ext.*	Envie este código se você estiver esperando uma extensão do servidor, mas ela não foi retornada no aperto de mão do *WebSocket*.
1011	*Internal error*	Envie este código quando a conexão for encerrada devido a uma condição inesperada.
1012	*Service restart*	Envie este código indicando que o serviço foi reiniciado e que um cliente que se reconectar deve fazê-lo com um atraso randomizado de 5 a 30 segundos.
1013	*Try again later*	Envie este código quando o servidor estiver sobrecarregado e o cliente deverá se conectar a um IP diferente (dado vários alvos) ou reconectar ao mesmo IP quando o usuário tiver realizado uma ação.
1014	*Unassigned*	Não use; este está não atribuído, mas pode ser alterado em futuras revisões.
1015	*TLS handshake*	Não use; isso é enviado quando o aperto de mão TLS falhou.

Fonte: Autoria própria, dados retirados de LOMBARDI, 2015.

2.3.3.4 *Uniform Resource Identifier,* Identificador de Recurso Uniforme (URI)

A especificação RFC 6455 define dois esquemas de URI para o *WebSocket*. A Figura 11 mostra um exemplo desses esquemas.

Figura 11: Esquemas de URI do protocolo *WebSocket*.

```
ws-URI = "ws:" "//" host [ ":" port ] path [ "?" query ]
wss-URI = "wss:" "//" host [ ":" port ] path [ "?" query ]
```

Fonte: IETF, 2011b.

- *Host*: É identificado por um IP literal encapsulado entre colchetes, um endereço IPv4 em formato decimal separado por pontos, ou um nome registrado. (IETF, 2011b).
- *Port*: É designado por um número de porta em decimal, que segue o anfitrião e é delimitado por um único caractere de dois pontos (":"). O

componente de porta é opcional; o valor padrão para "ws" é a porta 80, enquanto o valor padrão para "wss" é a porta 443. (IETF, 2011b).

- *Path*: O caminho é uma parte do URI que ajuda a localizar e identificar recursos dentro de um sistema de nomenclatura de URI. O caminho é terminado pelo primeiro ponto de interrogação ("?") ou sinal de número ("#"), ou pelo final do URI. (IETF, 2011b).
- *Query*: O componente de consulta é uma parte do URI que fornece informações adicionais para identificar o recurso dentro do escopo do esquema de URI e da autoridade de nomeação, se aplicável. O componente de consulta é iniciado pelo primeiro caractere de ponto de interrogação ("?") e terminado por um caractere de número ("#") ou pelo final do URI. (IETF, 2011b).

3 ESTUDO DE CASOS

Para entregar dados e manter sincronismo de usuários em tempo real as empresas Tinder e Dropbox optaram por utilizar o *WebSocket*. No processo de implementação realizado no aplicativo de celular Tinder e no aplicativo de *web* Dropbox Replay, foi documentado quais foram os desafios, vantagens e aprendizados obtidos.

3.1 Como o Tinder entrega seus *matches* e mensagens

O Tinder é um aplicativo de relacionamento para Android e iPhone, que combina pessoas a partir de um "*match*", nome dado pela plataforma para quando ocorre interesse mútuo entre dois usuários. O Tinder permite conhecer pessoas de todos os locais do mundo para possíveis novos relacionamentos.

O aplicativo Tinder, até meados de 2019, entregava suas mensagens e *matches* pesquisando o servidor a cada dois segundos. Nesse intervalo de tempo todos os usuários com o aplicativo ligado, faziam uma solicitação HTTP ao servidor para verificar se havia novas atualizações, na maior parte das vezes era uma consulta desnecessária pois a resposta não tinha nada de novo. Assim foi o funcionamento do aplicativo até então e funcionava bem. (DYANKOV, 2019)

Para essa configuração de funcionamento, de pesquisas periódicas, existem muitas desvantagens, como o consumo desnecessário de dados móveis do aparelho dos usuários, a necessidade de muitos servidores para lidar com a quantidade alta de tráfego vazio e em média, as atualizações reais são retornadas com um atraso de um segundo. Contudo, esse formato é bem confiável e previsível. Segundo Dyankov (2019), gerente sênior de engenharia do Tinder, o objetivo era aumentar a entrega em tempo real sem alterar muito a infraestrutura existente ou sacrificar a confiança, além de melhorar todos esses pontos negativos.

A solução encontrada foi o *WebSocket* como mecanismo de entrega em tempo real. Cada usuário estabelece um *WebSocket* com o servidor e cada processo *WebSocket* multiplexa dezenas de milhares de assinaturas de usuários. (DYANKOV, 2019).

Como resultados, segundo Dyankov, houve uma aceleração na entrega, a latência média de entrega caiu de 1.2 segundos para 300 Milissegundos (ms), uma melhoria de 4 vezes. O tráfego para o serviço de atualização também caiu drasticamente, o que permitiu reduzir os recursos necessários, além de aplicar outros recursos em tempo real, como implementar indicadores de digitação de forma eficiente.

3.2 Como o Dropbox Replay mantém todos sincronizados

O Dropbox Replay é uma ferramenta de colaboração por vídeo que recria a experiência de uma sala de projeção presencial com uma equipe remota e distribuída. Segundo Rogers, líder técnico de vídeo do Dropbox Replay, A ideia era reproduzir a sensação de estar em uma sala de projeção, com *feedback*, cursores e controles de reprodução compartilhados. Uma oportunidade para uma experiência de triagem *online* acessível, mas de alta qualidade, no qual as pessoas possam colaborar em tempo real como fariam pessoalmente, mas virtualmente, com todos podendo ver a mesma coisa e anotar perfeitamente o que estavam vendo em tempo real.

Para essa finalidade foi criado o recurso de Live Review do Dropbox Replay. Qualquer pessoa em uma sessão de Live Review pode pausar, ajustar a velocidade de reprodução ou passar para um quadro diferente a qualquer momento. Isso é ótimo para manter as exibições abertas e colaborativas, mas também significa que todos podem enviar comandos conflitantes ao mesmo tempo. (ROGERS, 2021).

Segundo Rogers (2021), para que o Live Review funcione, tem que haver a garantia de que todos convirjam para o mesmo estado de reprodução em tempo hábil. Quando uma pessoa está pronta para discutir um quadro, ela deve ser capaz de confiar que todos também estão vendo o mesmo quadro.

Em uma sessão de Live Review existem dois tipos de estado de cliente, o estado de cliente único, em que a posição do cursor do *mouse* de alguém ou os desenhos feitos em um quadro de vídeo e o estado de cliente compartilhado, que mantém a reprodução local sincronizada entre todos os clientes em uma sessão de Live Review. (ROGERS, 2021).

Para implementar todas essas funcionalidades o Live Review utiliza *WebSocket*. Quando alguém entra em uma sessão do Live Review, seu cliente abre

uma conexão *WebSocket* com um servidor Dropbox Replay. Sempre que alguém efetua uma ação que altera a posição do vídeo, o cliente envia uma mensagem transmitindo essa alteração ao servidor, que atualiza o estado compartilhado do cliente e depois o envia para todos os demais participantes da sessão. (ROGERS, 2021).

Para Rogers, os *WebSockets* são em grande parte independentes de conteúdo e oferecem mensagens de texto ou binárias que são enquadradas, entregues de maneira confiável e em ordem.

4 CONSTRUÇÃO DE UM SITE DE LEILÕES ONLINE COM WEBSOCKET

Este capítulo tem por finalidade apresentar os métodos e abordagens utilizados na construção e organização do projeto de implementação da aplicação *web* de leilões *online* com lances em tempo real, além de elencar as tecnologias, ferramentas e processos que foram utilizados durante esse processo.

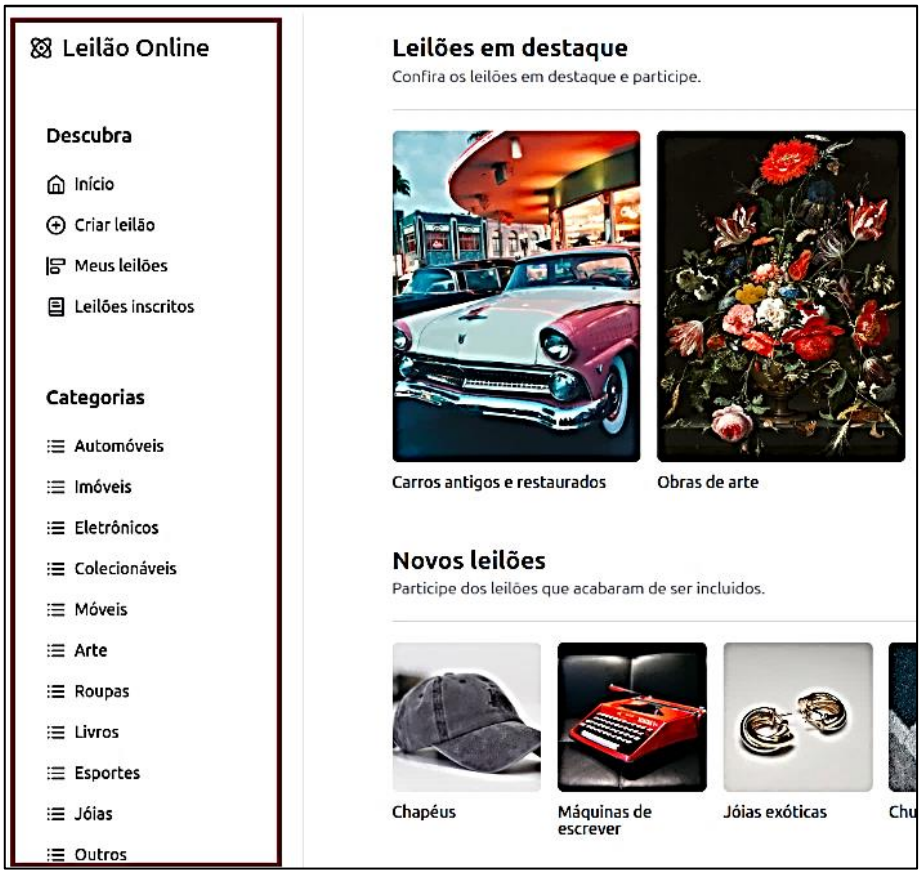
4.1 Aplicação web

A aplicação de leilões *online* objeto desse trabalho foi desenvolvida baseada no modelo cliente-servidor, em que o *frontend* (cliente), envia solicitações ao *backend* (servidor) que processa o pedido e devolve uma resposta. A comunicação é feita em HTTP/HTTPS em todas as telas da aplicação, exceto na tela de lances *online* que usa o *WebSocket*, pois são solicitações em tempo real.

A aplicação tem como propósito a criação e participação em leilões, por parte de qualquer usuário que previamente tenha sido cadastrado no sistema.

A estrutura de páginas *web* da aplicação segue a ideia de *Single Page Application,* Aplicativo de página única (SPA), ou seja, uma única página *web* com o objetivo de fornecer experiência ao usuário similar à de um aplicativo *desktop* e que, é atualizada e alterada de acordo com a interação do usuário. A aplicação *web* possui um menu lateral fixo e por meio dele é possível acessar as demais páginas da aplicação, como mostra a Figura 12.

Figura 12: Menu lateral da aplicação web

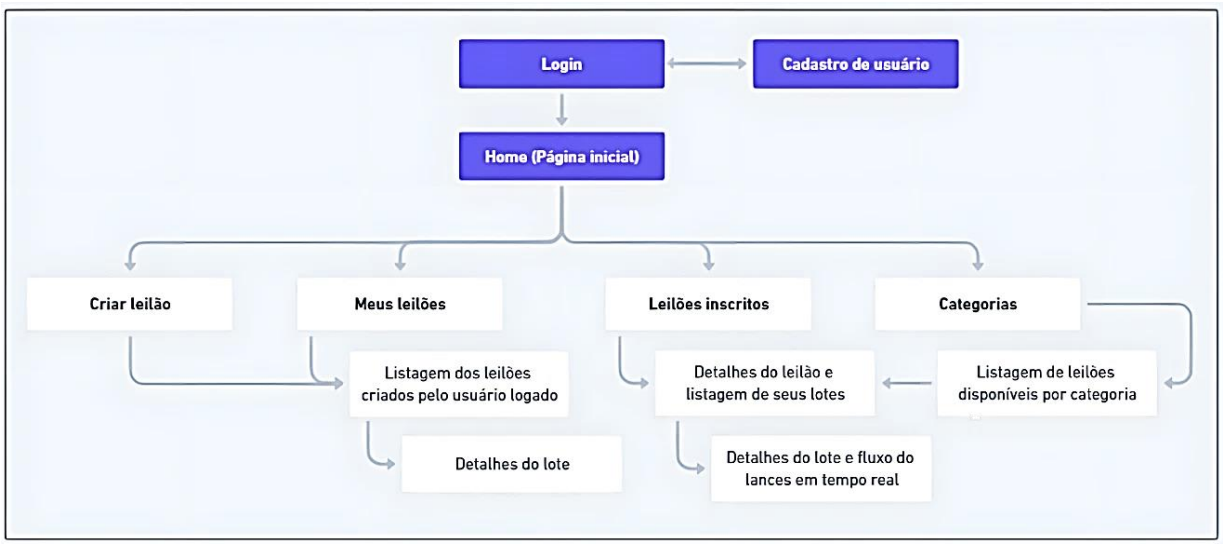


Fonte: Capturada pelo autor desse trabalho

A página inicial é a “Home” que lista os leilões disponíveis para participação. No menu lateral há os *links* para as páginas de “Criação de leilão”, “Meus leilões” e “Leilões inscritos”, além de uma lista com os *links* para leilões separados por categoria.

Ao clicar em algum leilão ocorre o redirecionamento para a página de detalhes do leilão, que é estruturada no modelo Mestre-Detalhe, em que há o cabeçalho com as informações gerais do leilão e uma listagem com os seus respectivos lotes. Ao clicar em um lote, a aplicação *web* é redirecionada para a página de detalhes do lote, que mostra as informações pertinentes ao lote e possui o espaço para efetuar os lances quando a data de abertura do lote for alcançada, assim liberando para que os usuários inscritos realizem seus lances. A Figura 13 corresponde ao mapa de telas da aplicação web.

Figura 13: Mapa de telas da aplicação web



Fonte: Elaborado pelo autor desse trabalho

4.2 Arquitetura da aplicação web

O processo de desenvolvimento foi separado em *frontend* e *backend* com o objetivo de segregar o papel e reponsabilidade de cada camada da aplicação, além de ser possível definir para cada camada as tecnologias e ferramentas necessárias de acordo com a funcionalidade almejada.

4.2.1 Frontend

O *frontend* compreende a parte de interação com o usuário final, ou seja, as páginas da aplicação, no qual é fornecido uma *interface* composta por elementos visuais para facilitar essa interação. Além disso, o *frontend* atua no envio de pedidos para o *backend* para serem processados e devolvidos.

Para a construção do *frontend* foi utilizado a linguagem de programação *Typescript* com a biblioteca *React*, que é útil para criar componentes de código para reutilização. Foram utilizados também: o *HTML* para criar a estrutura das páginas, o *CSS* e o *TailwindCSS* que são tecnologias utilizadas para a criação de estilos das páginas.

4.2.2 Backend

O *backend* atua como servidor, processando as solicitações decorrentes do *frontend*, interagindo com o banco de dados e enviando a resposta ao *frontend* ao finalizar o processamento.

A linguagem de programação utilizada para implementar o servidor foi o *Typescript* que é uma linguagem interpretada que, ao gerar o *build* final é transpilada para *Javascript*. O módulo responsável por possibilitar a execução de código *Typescript/Javascript* no lado do servidor é o Node.js.

As rotas de comunicação do servidor, HTTP e *WebSocket* ficam a cargo da biblioteca *Fastify*. A biblioteca Prisma é responsável pela comunicação e representação do banco de dados e suas tabelas no código.

4.2.3 Application Programming Interface (API)

O agente que torna capaz a comunicação entre *frontend* e *backend*, fazendo com que as solicitações sejam compreensíveis por ambos é a API, que é uma *interface* que estabelece a forma de texto padrão para a troca de mensagens e, por meio da troca de objetos JSON ocorre a comunicação entre *frontend* e *backend*. Isso vale para a comunicação via HTTP e via *WebSocket*.

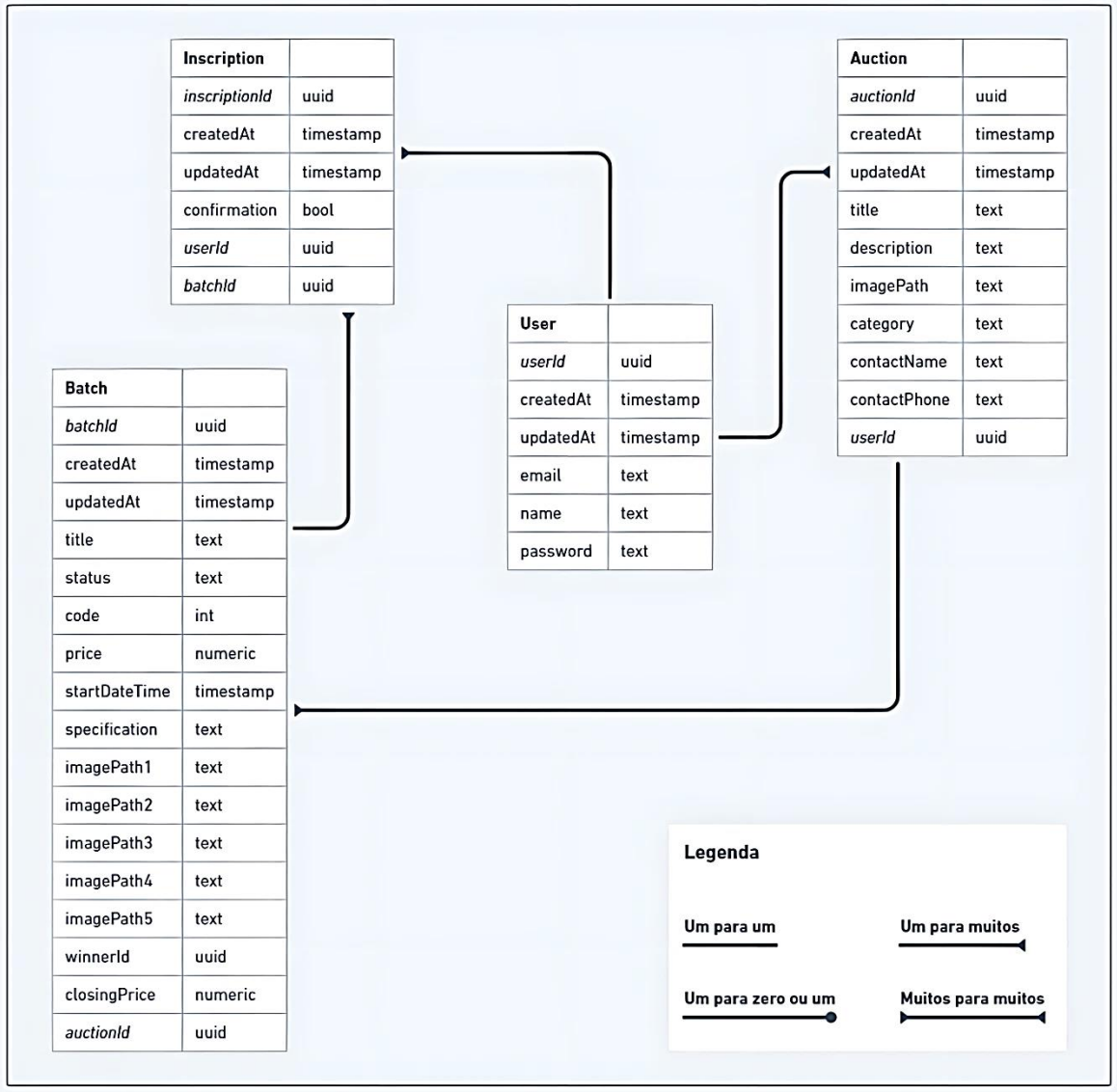
4.2.4 Banco de dados

O banco de dados utilizado foi o PostgreSQL, um banco de dados relacional robusto e de ampla adoção. As tabelas são organizadas da seguinte maneira:

- *User*: possui os dados do usuário cadastrado;
- *Auction*: contém os dados dos leilões criados e se relaciona com a tabela *Batch*;
- *Batch*: contém os dados dos lotes de cada leilão e se relaciona com a tabela *Inscription*;
- *Inscription*: possui os dados de inscrição de um usuário em um determinado lote;

A Figura 14 detalha a estrutura do banco de dados, suas tabelas, atributos e respectivos relacionamentos.

Figura 14: Diagrama da estrutura do banco de dados



Fonte: Elaborada pelo autor desse trabalho.

4.2.5 Docker

A ferramenta Docker foi utilizada para criar e gerenciar o ambiente em que a aplicação vai executar. Com o Docker, configurações, sistema operacional e rede são isolados em um ambiente independente da máquina *host* com a finalidade de

minimizar falhas e garantir a padronização do ambiente de execução da aplicação *web* e do banco de dados.

O ambiente no Docker é definido por um arquivo chamado *Dockerfile* que tem as configurações e definições que o ambiente deve possuir para executar a aplicação *web*. Uma vez criado e executado, o *Dockerfile* gera uma imagem e a partir dessa imagem pode-se criar o contêiner para executar a aplicação *web* e o banco de dados.

4.3 Versionamento do código fonte

O código fonte da aplicação *web, no frontend* e do *backend*, são armazenados em repositórios do *Github* e através da ferramenta *Git* é possível manter o histórico de alterações, correções e adições feitas no código fonte. Isso garante que código não se perca e possibilita acompanhar a evolução do projeto.

4.4 Infraestrutura

A infraestrutura compreende toda a parte do ambiente de hospedagem da aplicação *web,* sua estrutura, forma de disponibilidade e processos de atualização e implementação rápida no ambiente de *cloud*.

4.4.1 Ambiente de desenvolvimento

O desenvolvimento da aplicação foi realizado localmente em uma máquina com as seguintes configurações:

- Processador AMD Ryzen 5 5500, com 6 núcleos;
- Memória RAM de 32GB;
- Armazenamento de 1TB;
- Sistemas operacional Linux Ubuntu.

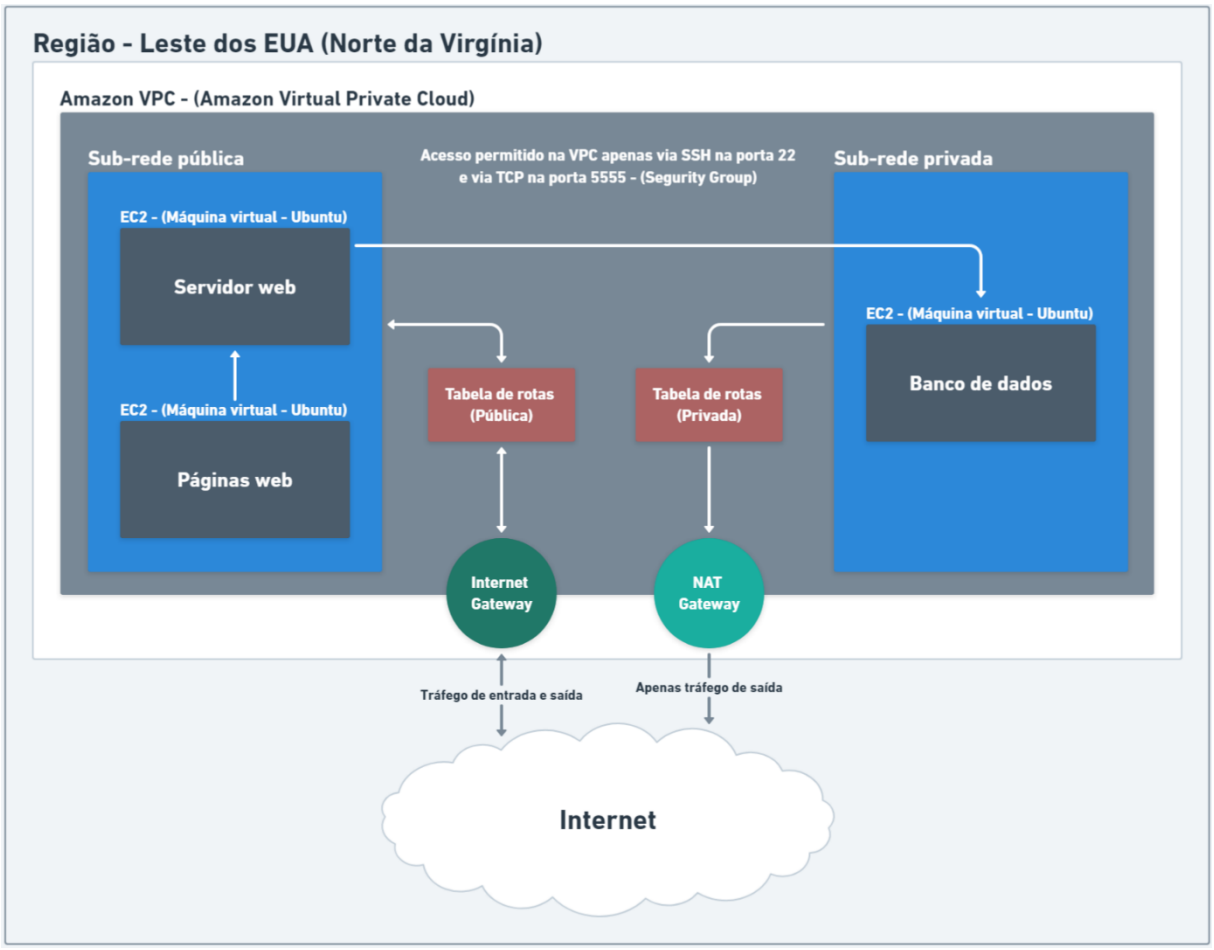
4.4.2 Hospedagem

A hospedagem da aplicação *web* foi feita em um *cloud provider* da Amazon, o *Amazon web Services,* Serviços *web* Amazon (AWS). A Figura 15 mostra a

infraestrutura construída na AWS em detalhes. O ambiente está configurado da seguinte forma:

- Região: os data centers da AWS estão disponíveis por região. A região selecionada foi o Leste dos Estado Unidos no Norte da Virgínia, por motivo de custos menores;
- Rede virtual: após selecionada a região foi criada uma rede virtual, *Virtual Private Cloud*, Nuvem privada virtual (VPC), para conter as máquinas virtuais;
- Sub-redes: dentro da VPC foram criadas duas sub-redes, uma pública e uma privada;
- Máquinas virtuais – Ubuntu: dentro da sub-rede pública foram criadas duas instâncias de *Elastic Compute Cloud*, Nuvem de computação elástica (EC2), que são máquinas virtuais, uma para conter o *frontend* e outra para conter o *backend*. As máquinas virtuais foram configuradas com 1GB de memória e 8GB de armazenamento e com sistema operacional Linux Ubuntu;
- Tabela de rotas: foi configurado uma tabela de rotas para cada sub-rede, ou seja, uma pública e outra privada;
- *Internet* Gateway: usada para permitir conexão da sub-rede pública com a *Internet*;
- NAT Gateway: utilizada para permitir tráfego apenas de saída da sub-rede privada para a *Internet*.

Figura 15: Diagrama da infraestrutura na AWS



Fonte: Elaborado pelo autor desse trabalho.

4.4.3 Disponibilização da aplicação

A aplicação é enviada/disponibilizada na AWS por meio de uma automação no *Github*, o *Github Actions,* que é uma ferramenta que possibilita gerar uma versão final da aplicação, uma *release,* para ser executada no ambiente da AWS. A cada nova alteração no código fonte, que é enviado para o repositório no *Github*, uma nova *release* é gerada e enviada para a AWS. Isso possibilita que que a versão mais atual da aplicação esteja disponível.

4.5 Trechos do código da aplicação cliente-servidor

O cliente e servidor *WebSocket*, responsáveis pelo fluxo de lances, são demonstrados e explicados nessa seção.

4.5.1 Cliente WebSocket

O cliente foi implementado utilizando o objeto *WebSocket* nativo do navegador, que provê uma interface de programação de aplicativos (API), para criar e gerenciar conexões *WebSocket* com um servidor. (MDN, 2023c).

A API do *WebSocket* no navegador é orientada a eventos, que qualificam as ações da conexão, ou seja, toda ação feita em uma conexão *WebSocket*, dispara um evento e o cliente fica “escutado” esses eventos. Os eventos são:

- Evento “*Open*”: Ao concluir a abertura do *handshake* e o servidor responder a solicitação de conexão com sucesso, o evento “*Open*” é disparado e a conexão é estabelecida.
- Evento “*Message*”: Após estabelecer uma conexão, agora é possível trocar mensagens entre cliente e servidor. Quando uma mensagem é recebida do servidor um evento “*Message*” é disparado no cliente.
- Evento “*Error*”: Quando um erro ocorre por qualquer motivo que seja, um evento “*Error*” é disparado.
- Evento “*Close*”: O evento “*Close*” é disparado quando a conexão *WebSocket* é encerrada.

Com cada um dos eventos é possível manipular e fazer tratamentos no código para cada cenário e dessa forma garantir o estado da conexão *WebSocket*. (LOMBARDI, 2015).

Além dos eventos, a API do *WebSocket* no navegador possui os métodos *WebSocket*, que são apenas dois, “*send”* e “*close*”. O método “*send*” é usado para enviar mensagens para o servidor, já o método “*close*” é utilizado para encerrar uma conexão *WebSocket*. (LOMBARDI, 2015).

A Figura 16 mostra a implementação do código do cliente *WebSocket* da aplicação web de leilões.

Figura 16: Trecho do código do cliente *WebSocket*.

```
1.      useEffect(() => {
2.
3.        if (socketRef.current && socketRef.current.readyState <= WebSocket.OPEN) {
4.            console.log('Conexão WebSocket já está aberta.');
5.            return;
6.        }
7.
8.        const token = localStorage.getItem('accessToken');
9.
10.       if (!token) {
11.           console.error('Token de autenticação não encontrado.');
12.           return;
13.       }
14.
15.       const host = import.meta.env.VITE_EC2_IP;
16.       const socket = new WebSocket(`ws://${host}/api/batch/${batchId}/bids`);
17.       socketRef.current = socket;
18.
19.       socket.onopen = () => {
20.           console.log('WebSocket connection opened.');
21.           socket.send(JSON.stringify({ type: 'authentication', token }));
22.       };
23.
24.       socket.onmessage = (event) => {
25.           const data = JSON.parse(event.data);
26.
27.           callback({
28.               value: data.value,
29.               userName: data.userName,
30.               userId: data.userId,
31.               code: data.code,
32.               type: data.type,
33.               message: data.message
34.           });
35.       };
36.
37.       socket.onerror = (error) => {
38.           console.error('Erro na conexão WebSocket:', error);
39.       };
40.
41.       return () => {
42.           if (socket.readyState === WebSocket.OPEN) {
43.               socket.close();
44.               console.log('WebSocket connection closed.');
45.           }
46.       };
47.   }, [batchId]);
```

Fonte: Capturado pelo auto desse trabalho

4.5.2 Servidor WebSocket

O servidor *WebSocket* foi construído utilizando o *framework Fastify* que por meio de uma rota define quais recursos serão acessados pelo cliente *WebSocket* no navegador. No servidor são utilizados os eventos “*message*” para processar as mensagens enviadas do cliente e o evento “*close*” apenas para informar que a

conexão foi encerrada. Os métodos “*send*” e “*close*”, são utilizados para enviar mensagens para o cliente e encerrar uma conexão, respectivamente. A Figura 17 mostra o código do servidor *WebSocket*.

Figura 17: Trecho do código do servidor *WebSocket*.

```
export async function bids(app: FastifyInstance) {
    app.get('/:batchId/bids', { websocket: true }, async (socket, request) => {

        if (batchState[batchId] && batchState[batchId].isOpen) {
            socket.on('message', async (message: string) => {
                try {
                    const data = JSON.parse(message);

                    if (data.type === 'bid') {
                        if (!socket.user) {
                            socket.send(JSON.stringify({ type: 'error', message: 'Usuário não autenticado' }));
                            return;
                        }
                        if (lastValue[batchId] == 0 || !lastValue[batchId] ||
                                        lastValue[batchId] == undefined || lastValue[batchId] == null) {
                            lastValue[batchId] = initialValue;
                        }

                        lastValue[batchId] = data.value;
                        const lance: Lance = {
                            batchId: batchId,
                            userName: socket.user.name,
                            userId: socket.user.id,
                            value: lastValue[batchId],
                            code: Math.floor(Math.random() * 10000),
                            index: bidsByBatchId[batchId] ? bidsByBatchId[batchId].length : 0
                        }

                        if (!bidsByBatchId[batchId]) {bidsByBatchId[batchId] = [];}

                        bidsByBatchId[batchId].push(lance);

                        app.websocketServer.clients.forEach((client: any) => {
                            if (client.batchId === batchId && client.readyState === 1) {
                                const lanceSend = {type: 'bid', ...lance}

                                client.send(JSON.stringify(lanceSend));
                            }
                        });
                    }

                } catch (error) {
                    console.error('Erro ao processar mensagem WebSocket:', error);
                }
            });
        } else {
            socket.send(JSON.stringify({ type: 'error', message: 'Lote fechado para lances' }));
            socket.close();
        }

        socket.on('close', async () => {
            console.log('Conexão WebSocket fechada:', socket.user?.name || 'Anônimo');
            delete socket.batchId;
        });
    });
}

```

Fonte: Capturado pelo autor desse trabalho.

5 RESULTADOS OBTIDOS

Com a disponibilização da aplicação para uso prático, é possível avaliar seu comportamento em um ambiente real e realizar testes detalhados. Este capítulo tem como objetivo expor de maneira abrangente o funcionamento da aplicação *web* de leilões, abordar os problemas enfrentados durante seu desenvolvimento e implementação, bem como validar o método de comunicação via *Websockets*, empregado para a realização de lances em tempo real. Ao longo deste capítulo, serão detalhados os aspectos técnicos da implementação, as dificuldades técnicas e operacionais encontradas, e as soluções adotadas para superá-las.

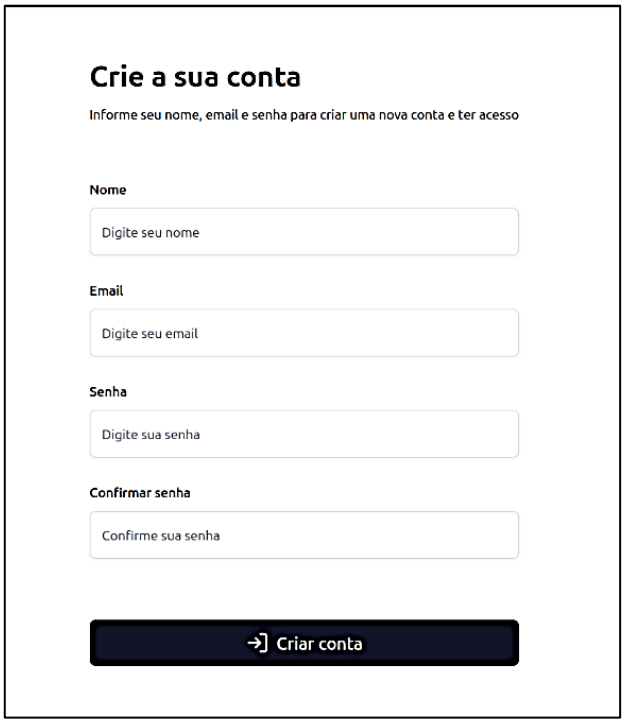
5.1 Funcionamento da aplicação

Para acessar a aplicação de leilões *online* é necessário que haja um cadastro previamente realizado pelo usuário e somente após o preenchimento do cadastro é possível entrar e acessar os leilões disponíveis.

5.1.1 Cadastro e autenticação de usuário

O usuário que deseja acessar a aplicação e participar ou criar leilões deve se cadastrar previamente, informando seus dados, como nome, *e-mail* válido e uma senha forte. A Figura 18 mostra a tela para cadastro.

Figura 18: Tela de cadastro de usuário



Fonte: Capturada pelo autor desse trabalho.

Uma vez cadastrado, uma mensagem de sucesso aparece no topo da tela, do lado direito. Na tela de *login* deve-se informar o *e-mail* e senha cadastrados e, assim acessar os leilões. A Figura 19 mostra a tela de *login*.

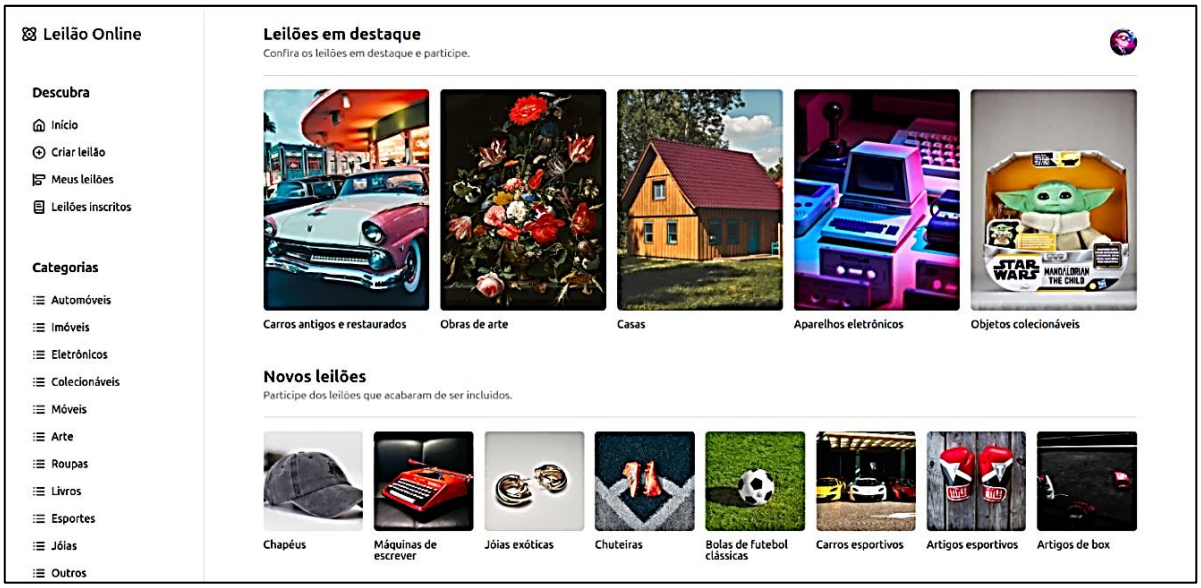
Figura 19: Tela de login

Fonte: Capturada pelo autor desse trabalho.

5.1.2 Criação de leilões

Após efetuar o *login* e acessar a tela inicial “*Home*”, é exibida uma listagem com os leilões disponíveis como mostra a Figura 20.

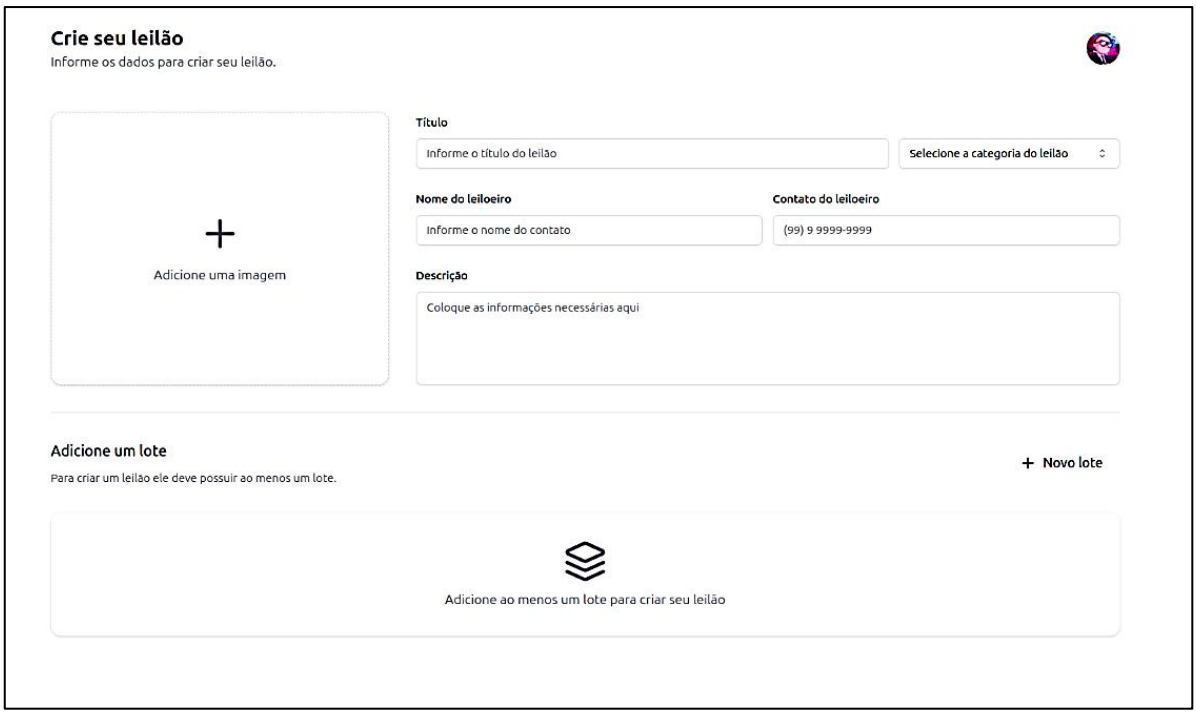
Figura 20: Tela inicial com leilões disponíveis



Fonte: Capturada pelo autor desse trabalho.

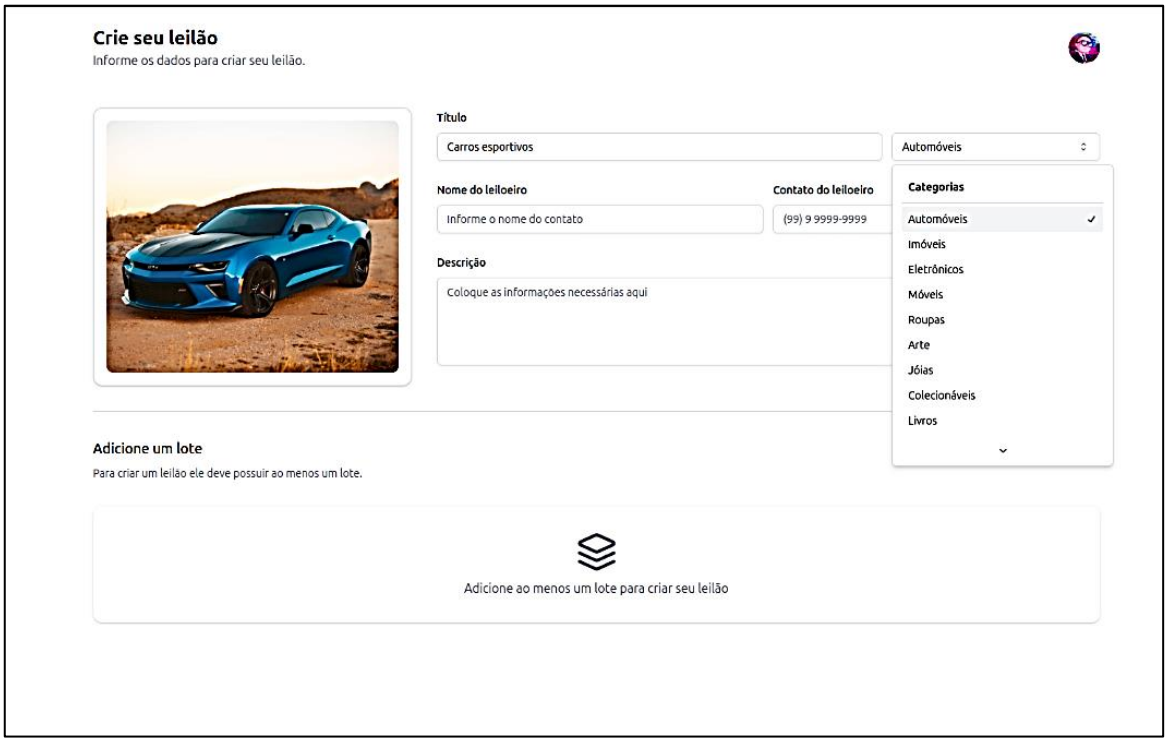
Na tela inicial “*Home*” é exibido um menu lateral, em que é possível navegar até o item “Criar leilão”, que ao ser clicado faz o redirecionamento para a tela de criação de leilões. As Figuras 21 e 22 exibem esse processo.

Figura 21: Tela de criação de leilões



Fonte: Capturada pelo autor desse trabalho.

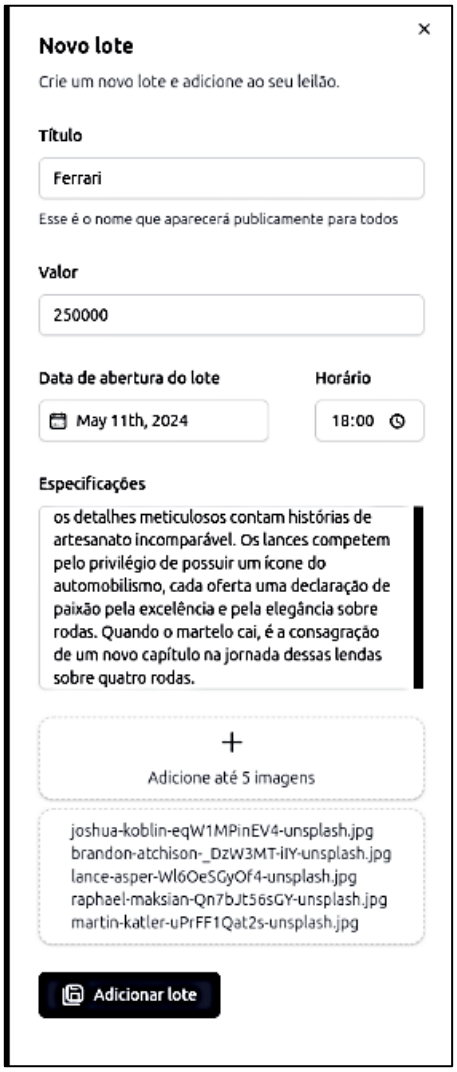
Figura 22: Tela de criação de leilões



Fonte: Elaborado pelo autor do trabalho.

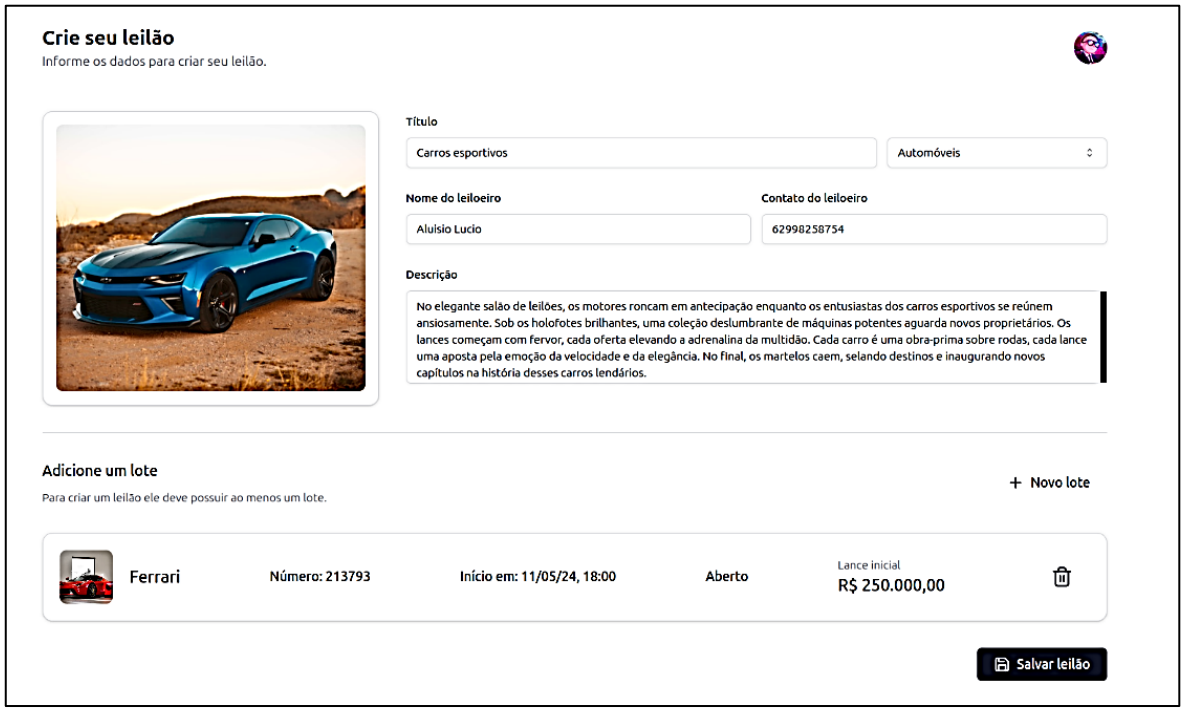
Após o preenchimento das informações do leilão pode-se adicionar os lotes ao leilão. As Figuras 23 e 24 mostram o processo de criação dos lotes.

Figura 23: Formulário de dados do lote na tela de criação de leilões



Fonte: Capturada pelo autor desse trabalho.

Figura 24: Tela de criação de leilões com a lista de lotes



Fonte: Capturada pelo autor desse trabalho.

Após inserir os dados do lote, ele aparece na lista que está na área inferior e contém as informações gerais do leilão, como mostra a Figura 24. Feito esse processo o leilão pode ser salvo e a daí aparece na seção “Meus leilões”. Quando o usuário acessa a seção “Meus leilões”, pode-se visualizar a listagem com todos os leilões criados pelo usuário logado. A Figura 25 mostra a lista dos leilões criados.

Figura 25: Tela de exibição dos leilões criados pelo usuário

Fonte: Capturada pelo autor desse trabalho.

5.1.3 Participação em leilões

Para participar de um leilão é necessário que que ele tenha ao menos um lote e que ele esteja aberto para receber inscrições de usuários previamente cadastrados na aplicação de leilões *online*. Vale ressaltar que um usuário não pode se inscrever em um leilão criado por si mesmo como um participante, pois a aplicação não permite.

O usuário não se inscreve diretamente no leilão, mas em seus lotes. Um lote está aberto para receber inscrições até que a data de abertura não prescreva. A Figura 26 mostra o botão “Inscrever-se”, na tela de detalhes do lote. Após a inscrição ser realizada o lote irá iniciar um período de confirmação de participação, 10 minutos antes de sua data de abertura, para que os usuários inscritos confirmem sua participação. Ao se encerrar o período de confirmação de 10 minutos, o lote de fato abre para receber os lances, dos usuários inscritos e confirmados.

Caso haja algum lote com inscritos e seu prazo de confirmação expire e ninguém tenha confirmado participação o lote é encerrado automaticamente e não abre para receber lances. A Figura 27 mostra a forma de realizar a confirmação por meio do botão “Confirmar participação”, na tela de detalhes do lote.

Figura 26: Tela de detalhes do lote e para inscrição

Fonte: Capturada pelo autor desse trabalho.

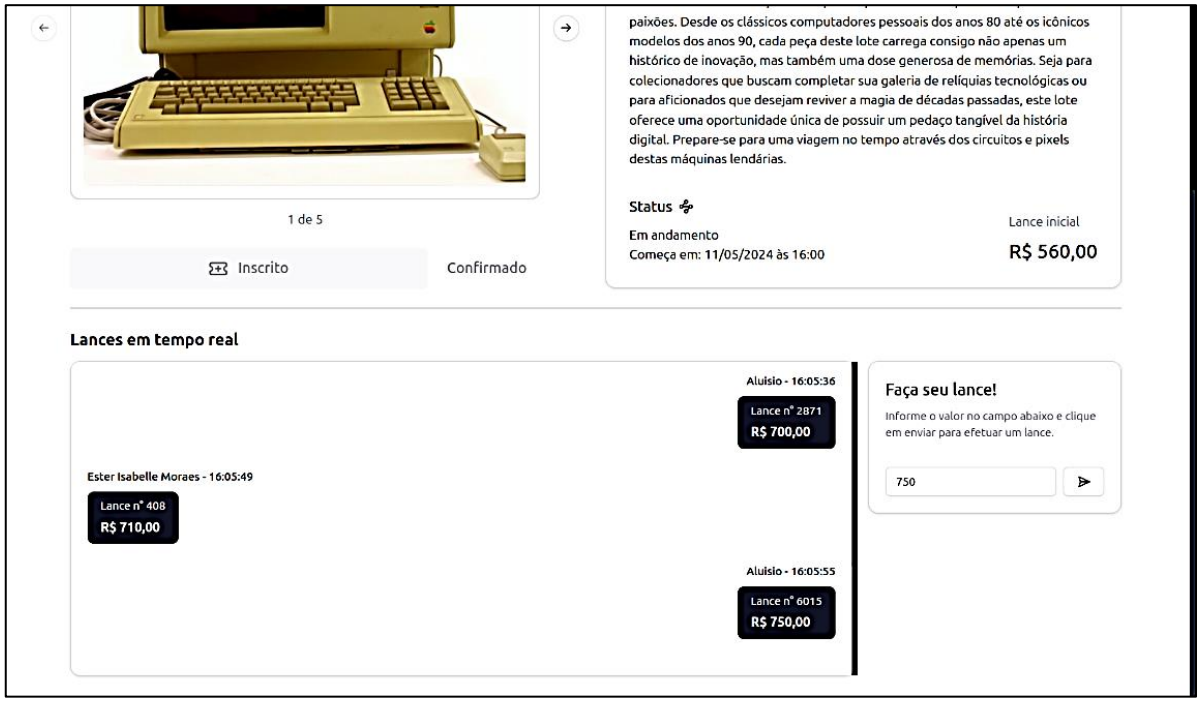
Figura 27: Tela de detalhes do lote e para confirmação de participação

Fonte: Capturada pelo autor desse trabalho.

5.1.4 Efetuar lances em determinado lote de um leilão

Cumprindo-se todas as etapas mencionadas anteriormente e chegando à data de abertura do lote, ele fica aberto por 10 minutos para receber lances. Com o fim dos 10 minutos o vencedor será aquele que tiver dado o maior lance por último e dessa forma o lote é encerrado e o vencedor anunciado para todos os participantes. A Figura 28 é um exemplo de como funciona o fluxo dos lances e a Figura 29 mostra o anúncio do vencedor aos demais participantes, ambos na tela de detalhes do lote.

Figura 28: Tela de detalhes do lote e lances



Fonte: Capturada pelo autor desse trabalho.

Figura 29: Tela de detalhes do lote e anúncio do vencedor

Fonte: Capturada pelo autor desse trabalho.

5.2 Validações e testes

Essa seção tem a finalidade de apresentar os resultados obtidos dos testes realizados para validar o comportamento e o desempenho da aplicação de leilões *online*, com base em um cenário de uso real.

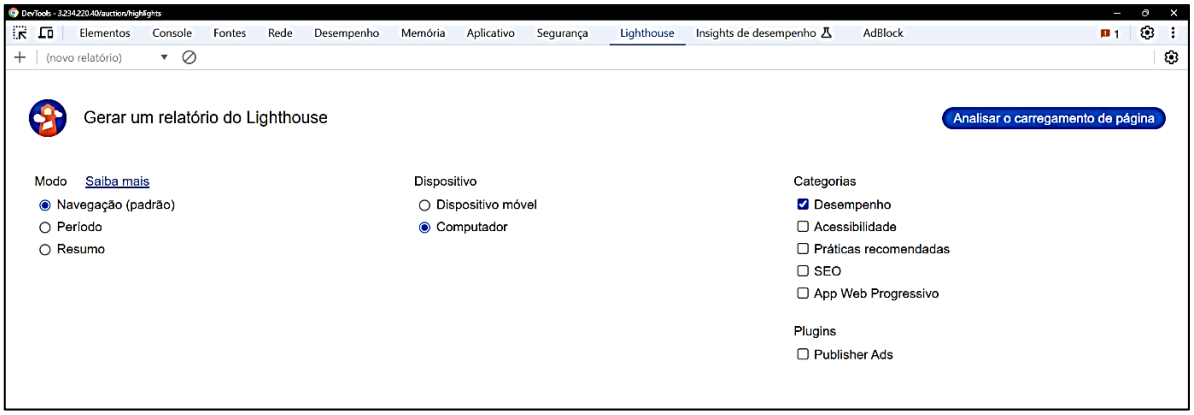
As ferramentas utilizadas para os testes foram a ferramenta de desenvolvedor do navegador Google Chrome (*Google Developer Tools*) e o *Lighthouse*, uma ferramenta de medição de desempenho de aplicações *web,* presente dentro da ferramenta de desenvolvedor do navegador Google Chrome.

5.2.1 Desempenho das páginas web

Para o teste de desempenho, o *Lighthouse* foi configurado na opção de dispositivo, como computador, na navegação padrão e na categoria desempenho. O teste foi executado em algumas páginas da aplicação, dentre elas a página de *login*, página de leilões por categoria e páginas de lances. O *Lighthouse* faz medições relacionadas ao carregamento da página *web* e o tempo de renderização de

elementos em tela e não as relacionadas a conexão, seja ela HTTP ou *WebSocket*. A Figura 30 mostra a configuração feita para o teste no *Lighthouse*.

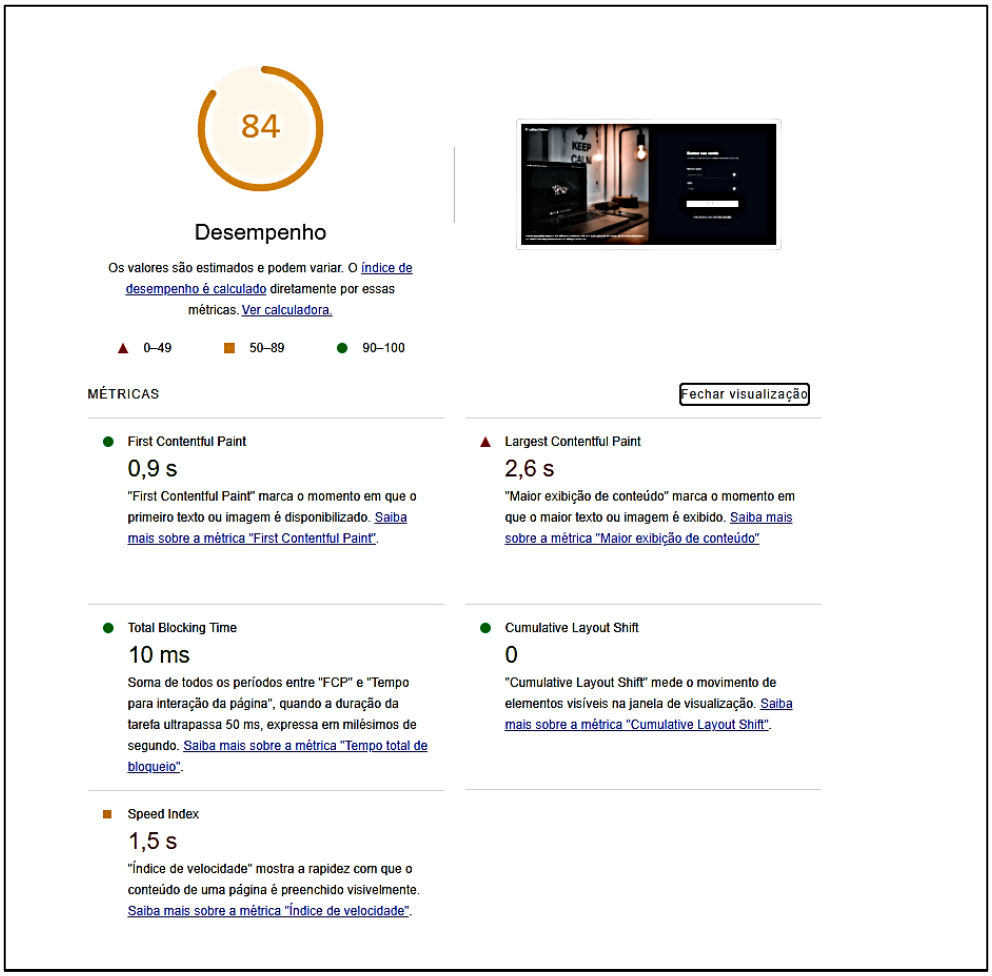
Figura 30: Configuração no *Lighthouse* para teste de desempenho



Fonte: Capturada pelo autor desse trabalho, da ferramenta *Lighthouse.*

Na página de *login* os resultados foram satisfatórios, tendo a pontuação de desempenho mais baixa, no carregamento da imagem presente na página e na velocidade de renderização de elementos na tela. A Figura 31 mostra os resultados.

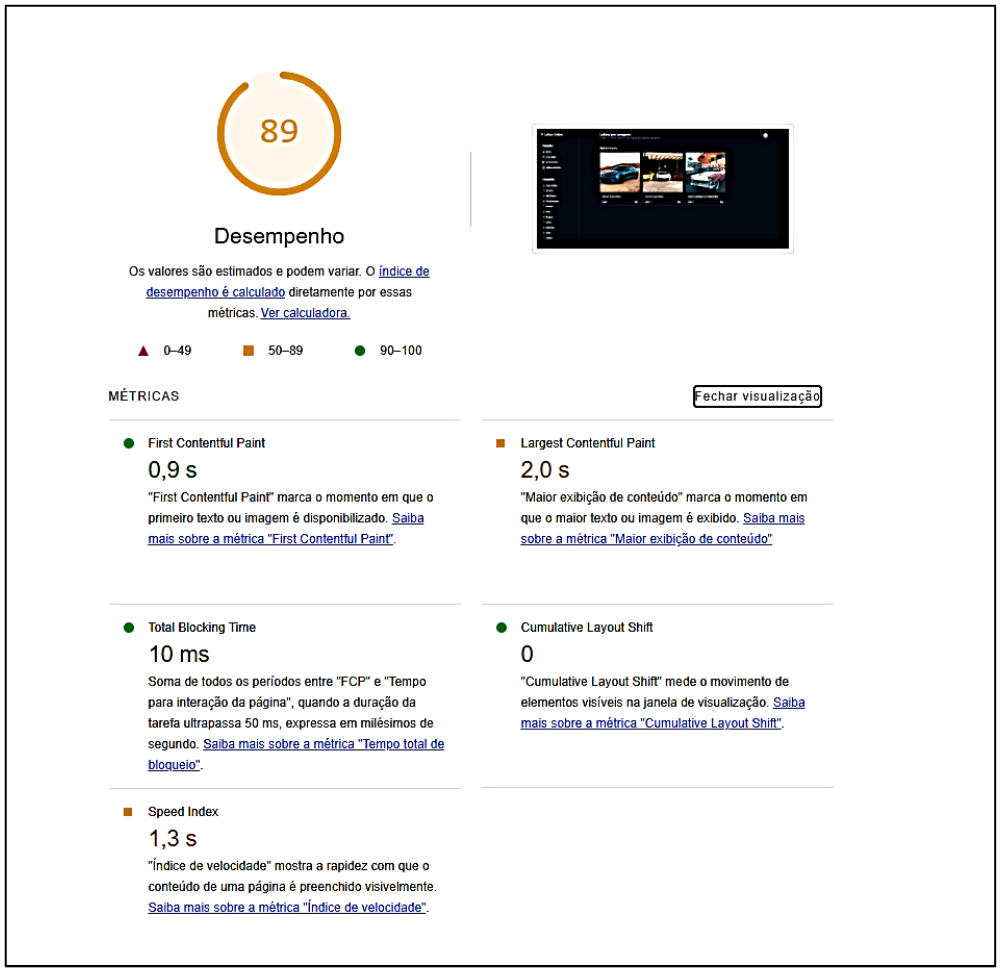
Figura 31: Resultados do teste de desempenho na tela de login



Fonte: Capturada pelo autor desse trabalho, da ferramenta *Lighthouse.*

No teste realizado na página de leilões por categoria, as métricas também foram satisfatórias e as pontuações mais baixas foram obtidos nos mesmos elementos da tela de *login*, o carregamento de imagens, que nessa página demorou um pouco mais, pois há maior quantidade de imagens, que consome maior tempo para renderização em tela. A Figura 32 mostra os resultados obtidos do teste de desempenho da tela de leilões por categoria.

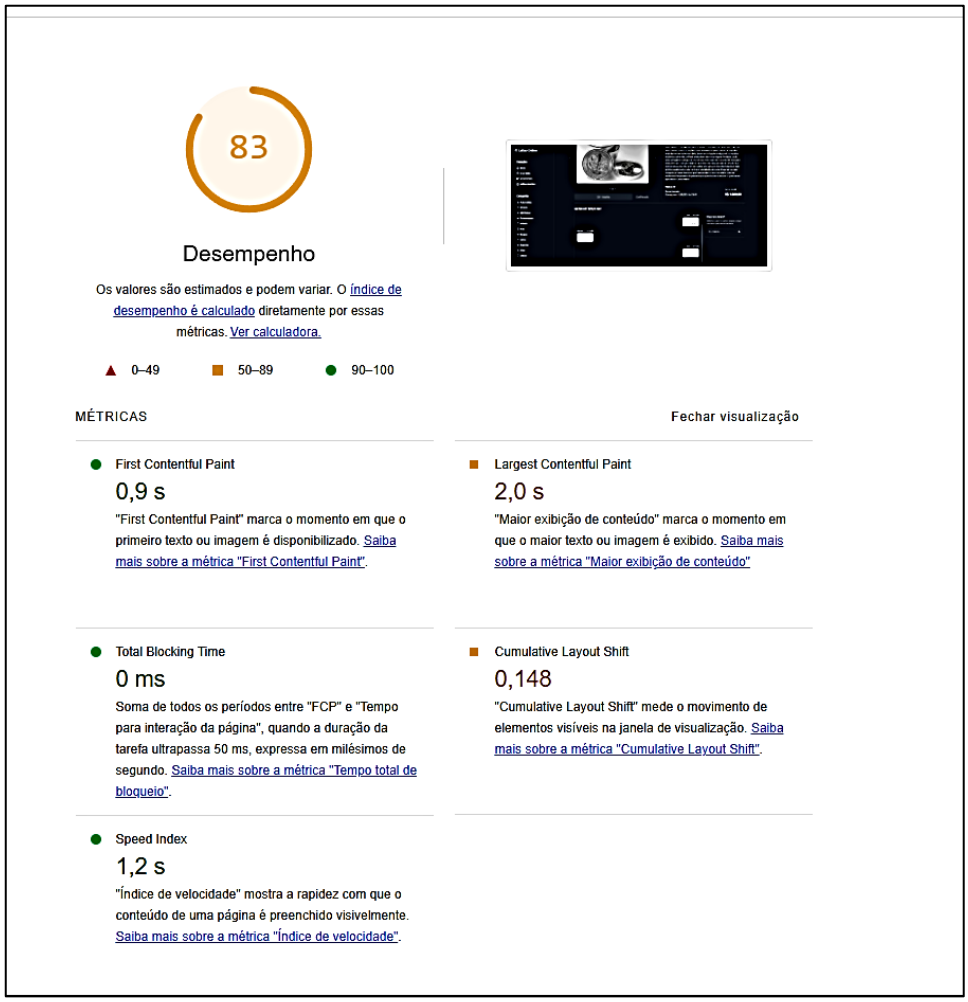
Figura 32: Resultados do teste de desempenho na tela de leilões por categoria



Fonte: Capturada pelo autor desse trabalho, da ferramenta *Lighthouse.*

Na página de lances os resultados foram semelhantes aos das páginas já mencionadas, sendo os pontos fracos, o carregamento de imagens e a velocidade de renderização de componentes em tela. Na Figura 33 estão os resultados do teste de desempenho da tela de lances. Ressalta-se que a tela de detalhes do lote, que é onde ocorre os lances, é a página mais complexa da aplicação, pois envolve a comunicação via *WebSocket*.

Figura 33: Resultados do teste de desempenho da tela de detalhes do lote e lances



Fonte: Capturada pelo autor desse trabalho, da ferramenta *Lighthouse*.

Um fator que deve ser considerado é a região na qual está localizada a infraestrutura de hospedagem da aplicação na AWS, que é o Norte da Virgínia nos Estados Unidos. O fato de ser uma região distante, causa uma maior latência na conexão com o servidor, influenciando no carregamento das imagens e na velocidade de renderização dos elementos na tela. A escolha dessa região, para hospedagem, foi unicamente por seu custo ser inferior ao restante das outras regiões da AWS.

5.2.2 Funcionamento do WebSocket

Com o auxílio da ferramenta de desenvolvedor do Google Chrome, pode-se atestar o correto funcionamento da conexão via *WebSocket*. Na Figura 34 pode-se verificar o tempo de 9,9 minutos, que é período em que a conexão *WebSocket*

permaneceu aberta, ou seja, período em que ocorreram os lances, e em que momento se encerrou.

Figura 34: Tempo de duração da conexão *WebSocket*.

Fonte: Capturada pelo autor desse trabalho, da ferramenta *Google Developer Tools*.

É possível verificar a troca de mensagens realizadas nesse período, em que dois usuários efetuam lances em um lote, como mostra o Quadro 5.

Quadro 5: Lances feitos em um leilão.

Dados	Mensagem	Comprimento	Hora
{"type": "bid", "value": 1700}	Enviada	29	21:45:54.213
{"type": "bid", "userName": "Ana", "value": 1800}	Recebida	165	21:47:38.386
{"type": "bid", "value": 1900}	Enviada	29	21:49:10.317
{"type": "bid", "userName": "Ana", "value": 2000}	Recebida	165	21:50:48.260
{"type": "bid", "value": 2100}	Enviada	29	21:51:17.801
{"type": "bid", "userName": "Ana", "value": 2150}	Recebida	171	21:52:09:251
{"type": "bid", "value": 2200}	Enviada	29	21:52:59.219
{"type": "bid", "userName": "Ana", "value": 2400}	Recebida	171	21:53:40.120
{"type": "bid", "value": 2500}	Enviada	29	21:54:54.911
{"type": "info", "message": "Lote encerrado! Vencedor: Aluisio - R$ 2.500,00"}	Recebida	73	21:56:02.273

Fonte: Elaborado pelo autor desse trabalho com dados extraídos do *Google Developer Tools*.

5.2.3 Comparativo entre HTTP Long Polling e WebSocket

Considerando que existem outras abordagens para lidar com dados em tempo real na *web,* como o SSE e o HTTP *Long Polling* que, antes do WebSocket eram as únicas opções para lidar com informações instantâneas, faz sentido comparar e analisar o comportamento entre o HTTP *Long Polling* e o *WebSocket* no cenário de entrega de lances da aplicação *web* de leilões *online*.

A abordagem de *HTTP Long Polling* foi implementada na aplicação de forma a chegar o mais próximo possível da *performance* da comunicação via *WebSocket* implementada, dessa forma, é necessário realizar duas requisições HTTP ao servidor, uma solicitação *GET* recorrente, que será feita a cada 1 segundo para obter atualizações de novos lances e uma solicitação *POST* que será feita ao servidor com os dados de um lance efetuado pelo usuário.

O Quadro 6 demonstra um comparativo das duas abordagens, evidenciando a latência e a quantidade de conexões que foram necessárias para realizar a entrega e atualização dos lances.

Quadro 6: Comparativo de latência e conexões entre *WebSocket* e *HTTP Long Polling.*

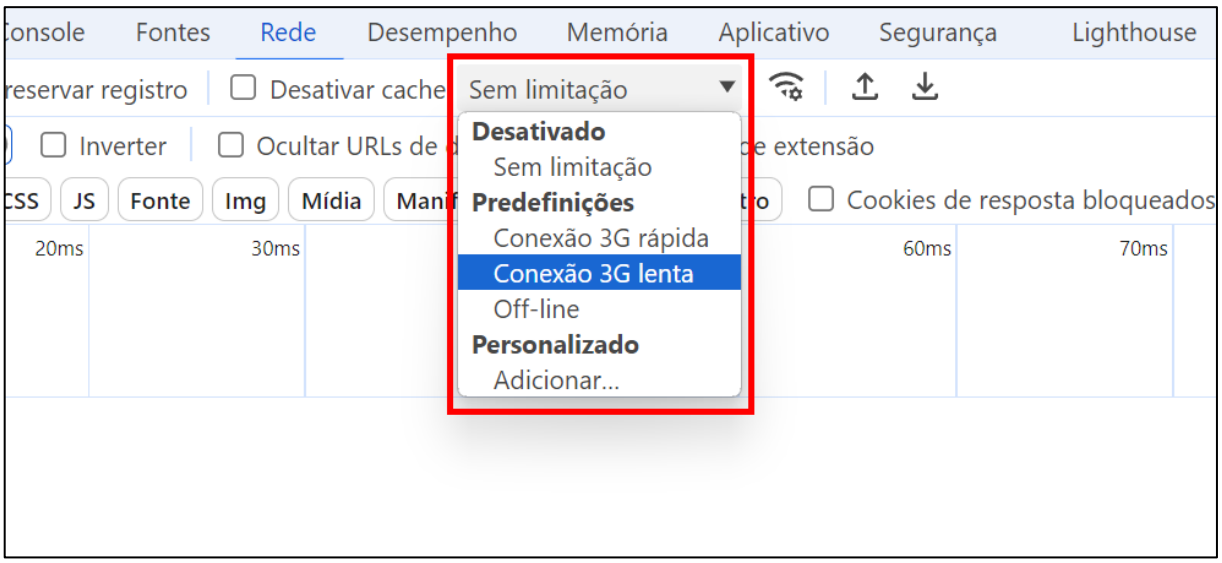
	Participantes	Conexões	Latência mínima	Latência média	Latência máxima
WebSocket	3	3	155 ms	176.8 ms	178 ms
HTTP Long Polling	3	573	316 ms	315 ms	427 ms

Fonte: Elaborado pelo autor desse trabalho com dados extraídos do *Google Developer Tools*.

A diferença é evidente nos tempos de latência e principalmente no número de conexões. Considerando que a cada nova solicitação no *HTTP Long Polling*, uma nova conexão também é criada, esse número se torna bem grande conforme o número de participantes aumentar. Já no *WebSocket* o número de conexões é equivalente ao número de participantes por leilão, ou seja, se houverem 3 participantes, haverá 3 conexões abertas de *WebSocket*. O contraste se acentua ainda mais quando se trata de um cenário com *Internet* de baixa velocidade, em que o *HTTP Long Polling* é muito prejudicado. Utilizando a ferramenta de desenvolvedor do Google Chrome (*Google Developer Tools*) para limitar a velocidade da *Internet,*

mudando para 3G ruim, como mostra a Figura 35, as solicitações ficam extremamente lentas, e os resultados podem ser analisados no Quadro 7.

Figura 35: Configurando *Google Developer Tools* para conexão 3G ruim.



Fonte: Capturada pelo autor desse trabalho, da ferramenta *Google Developer Tools*.

Quadro 7: Comparativo de latência e conexões entre *WebSocket* e *HTTP Long Polling* em *Internet* ruim.

	Participantes	Conexões	Latência mínima	Latência média	Latência máxima
WebSocket	3	3	180 ms	210.33 ms	225 ms
HTTP Long Polling	3	526	2.16 s	2.229 s	2.37 s

Fonte: Elaborado pelo autor desse trabalho com dados extraídos do *Google Developer Tools*.

A internet com velocidade ruim impacta o tempo da troca de mensagens no *WebSocket*, mas é um impacto mínimo, em contrapartida o *HTTP Long Polling* é consideravelmente impactado nesse cenário, demorando segundos para efetuar uma solicitação ao servidor e obter a resposta.

Em termos de fluidez e uniformidade de tempo na entrega dos lances o desempenho *HTTP Long Polling* foi consideravelmente inferior ao *WebSocket*, pois este faz a solicitação a cada 1 segundo, ou seja, a atualização dos lances para usuário pode ocorrer instantaneamente ou demorar um 1 segundo no pior caso, o que não

ocorre no *WebSocket* no qual essa atualização é sempre instantânea, ainda que em um cenário de *Internet* com taxa de transmissão limitada.

Vale ressaltar que na maior parte dos casos a solicitação *GET* do *HTTP Long Polling* retorna vazia, pois nem sempre existem novas atualizações de lances no servidor, diferente do *WebSocket* que só envia uma atualização ao cliente caso um novo lance tenha sido efetuado pelo usuário.

5.3 Problemas enfrentados

Dentre os problemas que surgiram durante a implementação da aplicação *web* de leilões *online*, uma delas foi a parte de implementação da conexão *WebSocket* do lado do servidor. Devido a sua flexibilidade, fica a cargo do desenvolvedor validar todo e qualquer tipo de mensagem que chega, padronizar as mensagens e separá-las por tipo. Além disso, como a implementação do protocolo *WebSocket* no lado do servidor fica a cargo do *Fastify*, houve a necessidade de conhecer como o *framework* abstraiu as funcionalidades do protocolo, para aprender a utilizar de forma correta as funções disponíveis, para abrir e encerrar conexões e enviar mensagens. A solução encontrada para o problema foi trabalhar apenas com o servidor, enviado mensagens menos complexas, uma por vez e observando como cada mensagem era processada.

Outro desafio foi implementar gatilhos no *frontend* para lidar com cada mensagem que retornasse do servidor via *WebSocket*, pois para cada nova mensagem, seja de erro ou aviso, deveria ter uma representação visual dela para que o usuário final pudesse ser informado. A solução para esse problema surgiu de tentativa e erro, até o ponto em que todas as mensagens do servidor foram tratadas, no caso das mensagens de erro e devidamente apresentadas ao usuário, no caso de avisos ou informações sobre os lances.

A configuração da rede virtual na AWS também foi um grande desafio, pois inicialmente houve dificuldades para liberar corretamente as portas que poderiam receber acesso nas instâncias de EC2, que são as máquinas virtuais da AWS. Para contornar esse problema foi necessário realizar estudos de redes de computadores, como criar redes, configurar sub-redes, configurar IP’s e de como a AWS organizava dentro de sua plataforma os recursos de rede.

6 CONSIDERAÇÕES FINAIS

Construir aplicações na *web* com a necessidade de recursos em tempo real, por um certo tempo, foi utilizado o HTTP, que não foi projetado, a princípio, para lidar com esse tipo de tráfego. Por conta disso, o protocolo é limitado, causando aumento de tráfego de rede e alta latência proporcional à quantidade de usuários. A introdução do *WebSocket* foi fundamental para que tarefas em tempo real se tornassem menos custosas e mais eficientes, já que o protocolo foi idealizado para esse fim.

A tarefa de entregar os lances em tempo real, usando *WebSocket*, foi alcançada devido ao extenso e denso conteúdo pesquisado e estudado, o qual foi abordado no referencial teórico. A integração prática do *WebSocket* na aplicação de leilões demonstrou ser uma abordagem eficiente, permitindo a comunicação bidirecional entre servidor e clientes com baixa latência. Por fim, a avaliação de desempenho validou a eficácia da solução, evidenciando que o protocolo é capaz de sustentar a demanda de um ambiente de leilões em tempo real, com transmissão ágil e precisa dos lances.

Os resultados obtidos são coerentes com outros trabalhos semelhantes que exploram a utilização de *WebSocket* em aplicações que exigem comunicação em tempo real. Como visto anteriormente nos estudos de caso, o Tinder obteve ótimos resultados implementando o protocolo em seu aplicativo. Alguns dos resultados foram a aceleração da entrega de mensagens, a diminuição da latência e a redução de recursos alocados na infraestrutura, consequentemente diminuindo custos. Nas validações e testes feitos na aplicação web de leilões online, pode-se observar que foram obtidos alguns resultados semelhantes aos do Tinder, como a diminuição da latência em comparação ao *HTTP Long Polling.*

Um dos principais aprendizados desta pesquisa foi a compreensão detalhada das capacidades e limitações do *WebSocket* no contexto de leilões online. Antes do desenvolvimento deste trabalho, a eficiência específica da tecnologia *WebSocket* para este tipo de aplicação de leilões *online* não era amplamente adotada e com esta pesquisa foi esclarecido como a tecnologia pode ser implementada e otimizada, oferecendo diretrizes práticas para desenvolvedores e empresas que pretendem utilizar *WebSocket* em sistemas de leilões.

Ao proporcionar uma solução eficiente para a comunicação em tempo real em leilões online, esta pesquisa facilita a realização de leilões mais dinâmicos e acessíveis. Isso pode beneficiar tanto compradores quanto vendedores, promovendo um ambiente de mercado mais transparente e competitivo. Além disso, a aplicação prática do *WebSocket* pode ser estendida para outras áreas que exigem comunicação em instantânea, como sistemas financeiros, jogos online e plataformas de colaboração, ampliando ainda mais o impacto positivo desta pesquisa para a área de desenvolvimento de aplicações de software.

Desenvolver uma aplicação web apresenta desafios consideráveis, sendo o fator tempo um dos mais difíceis. São várias etapas envolvidas, desde a criação do *frontend* e *backend*, até a configuração do ambiente de desenvolvimento, escolha da hospedagem na nuvem e a configuração de rede. Cada uma dessas fases representa um desafio significativo para a entrega dentro do prazo estabelecido. No entanto, enfrentar essas dificuldades não apenas gerou aprendizados para o desenvolvimento de softwares robustos e profissionais, mas também fortaleceu a capacidade de adaptação e resiliência.

A comunicação em tempo real proporciona fluidez na troca de dados e é uma excelente opção para reduzir o tráfego de solicitações HTTP recorrentes ao servidor. No entanto, é importante destacar que esse método não se aplica a todas as situações. Em muitos casos, a comunicação em tempo real pode ser excessiva e desnecessária. Em aplicações onde as atualizações de dados não são frequentes ou críticas, como em blogs ou sites de notícias, o uso de comunicação em tempo real pode ser um desperdício. Nestes casos, métodos tradicionais como HTTP padrão pode ser mais eficiente e adequado.

Portanto, a comunicação em tempo real deve ser utilizada apenas quando há uma necessidade real de entrega instantânea de informações, como em aplicações de mensagens instantâneas, como é o caso dos lances na aplicação de leilões *online*, ou outras situações em que a latência mínima é crucial para a funcionalidade e a experiência do usuário.

6.1 Sugestões de trabalhos futuros

Dentre as melhorias que podem ser feitas futuramente, destacam-se as seguintes:

- HTTPS: Adicionar uma camada de segurança ao HTTP;
- Domínio: Registrar um domínio para a aplicação;
- Otimização no carregamento de imagens: Verificar meios de otimizar o carregamento das imagens da aplicação;
- Migração de região na AWS: Migrar aplicação de região na AWS, para uma no Brasil, para diminuir a latência;
- Adicionar outras funcionalidades de tempo real, como indicadores de digitação nos lances e notificações para o usuário sobre abertura de lotes, período de inscrição e período de confirmação de participação;
- Realizar testes de estresse para validar o funcionamento da conexão *WebSocket* em um contexto extremo de utilização, com uma quantidade massiva de usuários com conexões abertas e efetuando lances.

REFERÊNCIAS BIBLIOGRÁICAS

CERN. ***World Wide Web***., <https://info.cern.ch/hypertext/WWW/TheProject.html>, 2013. Acesso em: 07 nov. 2023.

CHOPRA***,*** Varun. ***WebSocket Essentials – Building Apps with HTML5 WebSockets***. 1 ed. Birmingham: *Packt Publishing Ltd*, 2015.

DIACONU, Alex. ***The history of WebSockets***. Ably, 2023. Disponível em: <https://ably.com/topic/websockets-history>. Acesso em: 13 out. 2023.

DYANKOV, Dimitar. ***How Tinder delivers your matches and messages at scale***. *Medium*, 2019. Disponível em: <https://medium.com/tinder/how-tinder-delivers-your-matches-and-messages-at-scale-504049f22ce0>. Acesso em: 14 set. 2023.

GOOGLE WORKSPACE UPDATES BLOG. **Nova categorização de cores no Google Agenda para detalhar como você tem organizado seu tempo***.* <https://workspaceupdates-pt.googleblog.com/2022/08/nova-categorizacao-de-cores-no-google.html>, 2022. Acesso em: 12 nov. 2023.

GRIGORIK, Ilya. ***High Performance Browser Networking***. 1 ed. California: O'Reilly Media, 2013.

HARRISON, Luke. ***20 web design relics of the old Internet.*** *LogRocket*, 2018. Disponível em: <https://blog.logrocket.com/20-web-design-relics-of-the-old-*Internet* -eb3df4ac13e7/>. Acesso em: 27 out. 2023.

IBM. **Transmissões *full* e *half* duplex**. Documentação da IBM, 2023. Disponível em: <https://www.ibm.com/docs/pt-br/aix/7.3?topic=standards-full-half-duplex-transmissions>. Acesso em: 11 nov. 2023.

IETF. ***Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP***. RFC 6202, *Internet Engineering Task Force*, 2011a. Disponível em <https://www.rfc-editor.org/rfc/rfc6202#section-1>. Acesso em: 06 nov. 2023.

IETF. ***The WebSocket Protocol***. RFC 6455, *Internet Engineering Task Force*, 2011b. Disponível em: <https://datatracker.ietf.org/doc/rfc6455>. Acesso em: 10 set. 2023.

LOMBARDI, Andrew. ***WebSocket***. 1 ed. California: O'Reilly Media, 2015.

MAGRINI, Damiano. ***WebSockets Demystified, Part 1: Understanding the Protocol***. *Medium*, 2021. Disponível em: <https://levelup.gitconnected.com/websockets-demystified-part-1-understanding-the-protocol-fccca2ca75eb>. Acesso em: 11 set. 2023.

MARTIN, Eve. ***WebSockets vs Server-Sent Events: Key differences and which to use***. Ably, 2023. Disponível em: <https://ably.com/blog/websockets-vs-sse>. Acesso em: 13 out. 2023.

MOZILLA DEVELOPER NETWORK (MDN). **Primeiros passos - AJAX**. *Mozilla Developer Network*. Disponível em: <https://developer.mozilla.org/pt-BR/docs/conflicting/Web/Guide/AJAX>. Acesso em: 02 nov. 2023a.

MOZILLA DEVELOPER NETWORK (MDN). ***XMLHttpRequest***. *Mozilla Developer Network*. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/API/XMLHttpRequest>. Acesso em: 02 nov. 2023b.

MOZILLA DEVELOPER NETWORK (MDN). ***WebSocket***. *Mozilla Developer Network*. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>. Acesso em: 01 jun. 2023c.

ROGERS, Alan. ***How Dropbox Replay keeps everyone in sync***. *Dropbox.Tech*, 2021. Disponível em: <https://dropbox.tech/application/how-dropbox-replay-keeps-everyone-in-sync>. Acesso em: 14 set. 2023.

STELDT, Ariane van der. ***Migrating Millions of Concurrent Websockets to Envoy***. *Slack Engineering*, 2021. Disponível em: <https://slack.engineering/migrating-millions-of-concurrent-websockets-to-envoy/>. Acesso em: 20 set. 2023.

STICHBURY, Jo. ***The realtime web: evolution of the user experience***. Ably, 2023. Disponível em: <https://ably.com/blog/the-realtime-web-evolution-of-the-user-experience>. Acesso em: 13 out. 2023.

UNIVERSO *ONLINE* (UOL). **UOL Web*site***. [S. l.], [s. d.]. Disponível em: <https://www.uol.com.br/>. Acesso em: 11 nov. 2023.

WAZLAWICK, Raul Sidnei. **Metodologia de Pesquisa para Ciência da Computação**. 2 ed. Rio de Janeiro: Elsevier Editora Ltda, 2014.

WIRESHARK. ***WebSocket***., <https://wiki.wireshark.org/WebSocket>. Publicado em 11 de agosto de 2020. Acesso em: 07 nov. 2023.

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
GABINETE DO REITOR

Av. Universitária, 1069 ● Setor Universitário
Caixa Postal 86 ● CEP 74605-010
Goiânia ● Goiás ● Brasil
Fone: (62) 3946.1000
www.pucgoias.edu.br ● reitoria@pucgoias.edu.br

RESOLUÇÃO n° 038/2020 – CEPE

ANEXO I

APÊNDICE ao TCC

Termo de autorização de publicação de produção acadêmica

O(A) estudante ALUISIO LUCIO DOS SANTOS NETO do Curso de CIÊNCIA DA COMPUTAÇÃO, matrícula 20181002800545, telefone: xxx e-mail 20181002800545@pucgo.edu.br, na qualidade de titular dos direitos autorais, em consonância com a Lei nº 9.610/98 (Lei dos Direitos do autor), autoriza a Pontifícia Universidade Católica de Goiás (PUC Goiás) a disponibilizar o Trabalho de Conclusão de Curso intitulado WEBSOCKET COMO SOLUÇÃO PARA ENTREGA DE LANCES EM TEMPO REAL EM UMA APLICAÇÃO WEB DE LEILÕES, gratuitamente, sem ressarcimento dos direitos autorais, por 5 (cinco) anos, conforme permissões do documento, em meio eletrônico, na rede mundial de computadores, no formato especificado (Texto (PDF); Imagem (GIF ou JPEG); Som (WAVE, MPEG, AIFF, SND); Vídeo (MPEG, MWV, AVI, QT); outros, específicos da área; para fins de leitura e/ou impressão pela internet, a título de divulgação da produção científica gerada nos cursos de graduação da PUC Goiás.

Goiânia, 26 de junho de 2024.

Assinatura do(s) autor(es): Documento assinado digitalmente gov.br ALUISIO LUCIO DOS SANTOS NETO Data: 24/06/2024 17:08:11-0300 Verifique em https://validar.iti.gov.br

Nome completo do autor: ALUISIO LUCIO DOS SANTOS NETO

Assinatura do professor-orientador: Documento assinado digitalmente gov.br ANGELICA DA SILVA NUNES Data: 24/06/2024 16:48:02-0300 Verifique em https://validar.iti.gov.br

Nome completo do professor-orientador: ANGELICA DA SILVA NUNES