

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

Rafael Michels Motta

DESENVOLVIMENTO DE APLICAÇÕES COM WEBSOCKETS

Trabalho de conclusão de curso

Orientador:
Frank Siqueira

Florianópolis, 2012/2

Rafael Michels Motta
DESENVOLVIMENTO DE APLICAÇÕES COM WEBSOCKETS

Trabalho de conclusão de curso submetido à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do Diploma de Graduação em Sistemas de Informação.

Orientador: Prof. Frank Siqueira
Universidade Federal de Santa Catarina
frank@inf.ufsc.br

Banca examinadora

Prof. Ricardo Pereira e Silva
Universidade Federal de Santa Catarina
ricardo@inf.ufsc.br

Prof. Leandro José Komosinski
Universidade Federal de Santa Catarina
leandro@inf.ufsc.br

“Foco é dizer não”
Steve Jobs

Dedicatória

Este trabalho é dedicado aos meus pais, os quais sempre se esforçaram para prover o melhor ensino.

Resumo

A web está vivendo uma nova era, onde tudo está sendo feito em tempo real. As expectativas de quão rápido a internet deve entregar as informações mudou – atrasos de minutos são inaceitáveis. Grandes empresas, como *Twitter*, *Google* e *Facebook* captaram essa necessidade rapidamente, e atualmente já entregam seus dados em tempo real, seja em um simples bate-papo, ou mesmo em notificações de ações em redes sociais. Nesse cenário, este trabalho visa explorar os conceitos e definições das tecnologias envolvidas para se criar aplicações Web em tempo real, com foco na API e protocolo de *WebSockets*. Por fim, pretende-se ainda implementar um serviço utilizando esse recurso.

Palavras chaves: Comunicação na web, HTML5, *WebSockets*.

Abstract

The web is living a new era, where everything is done in real time. The expectations of how fast the Internet must deliver information has changed - minute delays are unacceptable. Large companies, such as Twitter, Facebook and Google have captured this need quickly, and currently deliver their data in real time, whether a simple chat, or even action notifications in social networks. In this scenario, this work explore concepts and definitions of the technologies involved to create real-time Web applications, focusing on API, and WebSockets protocol. Finally, the aim is also to implement a service using this feature.

Key words: Web communication, HTML5, WebSockets.

Sumário

Lista de figuras	9
Lista de tabelas.....	11
Lista de siglas	12
1. Introdução.....	13
1.1. Objetivos e Metas	14
1.1.1 Objetivo Geral.....	14
1.1.2 Objetivos Específicos	14
1.2. Motivação	15
1.3. Metodologia	15
1.4. Estrutura do trabalho	16
2. Desenvolvimento de Aplicações Web	17
2.1 Modelo cliente-servidor.....	18
2.3 Tecnologias de apresentação.....	22
2.4 Programação lado cliente	23
2.5 Protocolo HTTP	23
Tabela 2: Métodos do protocolo HTTP.....	26
3. Aplicações em tempo real na Web.....	27
3.1. Polling	30
3.2. Comet	31
4. HTML5.....	35
4.1. História.....	35
4.2. HTML5 nos dias de hoje.....	37
4.3. Princípios chave	39
4.4. Novas funcionalidades.....	41
4.4.1. Core	41

4.4.2. APIs Javascript.....	41
4.5. Desenvolvimento de aplicações em tempo real com HTML5.....	42
5. WebSockets.....	43
5.1. Compatibilidade.....	44
5.2. Protocolo WebSocket.....	45
5.3. Interface.....	48
5.3.1. Construtor.....	49
5.3.2. Métodos.....	50
5.3.3. Eventos.....	51
6. Projeto.....	53
6.1. Processo de análise.....	53
6.1.2 Análise de requisitos.....	54
6.1.3 Diagrama de casos de uso.....	56
6.1.4 Diagramas de atividade.....	58
6.1.6 Modelagem do banco de dados.....	62
6.2. Desenvolvimento.....	63
6.3. Resultados.....	69
7. Conclusões.....	79
7.1 Avaliação dos Resultados.....	79
7.2 Trabalhos Futuros.....	80
8. Referências bibliográficas.....	81
Apêndice 1 – Artigo.....	82
Apêndice 2 – Código fonte.....	83

Lista de figuras

Figura 1 – Modelo cliente-servidor

Figura 2 – Modelo assíncrono

Figura 3 – Funcionamento do protocolo HTTP

Figura 4 – Exemplo das primeiras páginas com Iframe

Figura 5 – RIAs – Aplicações ricas da internet

Figura 6 – Notificação no Twitter

Figura 7 – Exemplo do uso de Polling

Figura 8 – Streaming

Figura 9 – Diagrama da técnica Long-Polling

Figura 10 – Exemplo de uso de Long-Polling

Figura 11 – Padrão WebSockets de URIs

Figura 12 – Processo de HandShake

Figura 13 – Framing

Figura 14 – Interface da Api WebSockets

Figura 15 – Diagrama de casos de uso

Figura 16 – Diagrama Registro no sistema

Figura 17 – Diagrama Login

Figura 18 – Diagrama alterar dados pessoais

Figura 19 – Criação e edição de um domínio e suas configurações

Figura 20 – Compartilhar navegação

Figura 21 – Visualizar navegação

Figura 22 – Visualizar informações do visitante

Figura 23 – Troca de mensagens

Figura 24 – Criação de uma página fictícia

Figura 25 – Registro

Figura 26 – Entrar

Figura 27 – Layout básico do sistema

Figura 28 – Conta

Figura 29 – Configurações

Figura 30 – Domínio

Figura 31 – Tela da aplicação fictícia

Figura 32 – Tickets

Figura 33 – Compartilhamento

Lista de tabelas

Tabela 1 – Tipos MIME

Tabela 2 – Métodos do protocolo HTTP

Tabela 3 – Agentes de usuário

Tabela 4 – Métodos da API Full Screen

Tabela 5 – Suporte do WebSockets em browsers desktop

Tabela 6 – Suporte do WebSockets em browsers mobile

Tabela 7 – Parâmetros do evento close

Lista de siglas

AJAX – Asynchronous Javascript and XML

API – Application Programming Interface

BLOB – Binary Large Object

CERN – Conseil Européen pour la Recherche Nucléaire

CSS – Cascading Style Sheets

DOM – Document Object Model

ECMA – European Computer Manufacturers Association

JSON – Javascript Object Notation

HTTP – Hypertext Transfer Protocol

HTML – Hypertext Markup Language

MIME – Multipurpose Internet Mail Extensions

RIA – Rich Internet Applications

RF – Requisitos funcionais

RNF – Requisitos não funcionais

SVG – Scalable Vector Graphics

TCP – Transmission Control Protocol

UML – Unified Modeling Language

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

XHTML – eXtensible Hypertext Markup Language

XML – eXtensible Markup Language

W3C – World Wide Web Consortium

WEB – World Wide Web

WHATWG – Web Hypertext Application Technology Working Group

1. Introdução

Por que a Web em tempo real é importante? Vivemos em um mundo em que tudo é feito em tempo real, então, é natural que a web esteja se movendo nesta direção. As expectativas de quão rápido a internet deve entregar as informações mudou – atrasos de minutos são inaceitáveis. Grandes empresas, como *Twitter*, *Google* e *Facebook* captaram essa necessidade rapidamente, e atualmente já entregam seus dados em tempo real, seja em um simples bate-papo, ou mesmo em notificações de ações em redes sociais.

De fato, a web está vivendo uma nova era. Criada originalmente para exibição, organização e compartilhamento de documentos hipertexto, ela passou por muitas fases até chegar no estágio que se encontra hoje. Do simples fornecimento de páginas estáticas, escritas em HTML, surgiu espaço para a criação de linguagens e outros mecanismos que puderam oferecer esse conteúdo de forma dinâmica. Esse modelo, que hoje é tradicionalmente conhecido como HTTP, é baseado em requisições e respostas: um cliente faz um pedido de uma página web, o servidor entrega o conteúdo, e nada mais acontece até que o cliente faça outra requisição. Surge depois o AJAX, técnica que permitiu que a web torna-se muito mais dinâmica, permitindo que a requisição fosse feita de forma assíncrona, sem a necessidade de recarregar da página atual por completo. No entanto, ainda estávamos presos naquele antigo modelo HTTP, não possuindo um padrão para enviar dados no sentido servidor - cliente.

Uma gama de soluções surgiu para resolver esse problema. A mais básica foi a técnica chamada *Polling*, que consistia em fazer requisições em tempos regulares no servidor web utilizando AJAX, a fim de obter alguma nova informação. De fato ela deu a percepção aos usuários de que a aplicação era em tempo real, mas outros problemas acabaram surgindo, geralmente ligados a performance e a sobrecarga dos servidores web, já que utilizavam o tradicional modelo HTTP, que não tinha sido projetado para esse tipo de aplicação. A mais notável foi a *Comet*, um modelo de aplicação baseado no

modelo HTTP, que permitiu que, de fato, dados fossem enviados sem uma requisição explícita do servidor.

Plug-ins de terceiros também foram criados, como *Flash Sockets*. Esses permitiram que, de fato, fosse utilizada a técnica PUSH, isso é, do servidor web enviar dados ao cliente. A ressalva era que esses *plug-ins* tinham que ser instalados por cada usuário em sua máquina, o que não era garantido, principalmente para usuários leigos e em grandes corporações.

Tudo mudou em 2008, quando a W3C anunciou um novo padrão da web, o HTML5. As novidades dessa nova especificação iam desde novas *tags* – que traziam melhorias na semântica e acessibilidade – até um conjunto de APIs especificadas em *Javascript*, mudando totalmente o HTML que estávamos acostumados a lidar. Uma das seções mais interessantes desse novo padrão foi a parte de comunicação, beneficiando principalmente as aplicações em tempo real. A API de *WebSockets* permite agora a comunicação bidirecional por canais *full-duplex* através da porta 80 do protocolo TCP, significando que agora pode ser enviados dados no sentido servidor – cliente e ainda sem os problemas ligados a performance, que tanto afetavam as técnicas que usadas em aplicações em tempo real.

Nesse cenário, serão explorados os conceitos e definições das tecnologias envolvidas para se criar aplicações Web em tempo real, com foco na API e protocolo de *WebSockets*.

1.1. Objetivos e Metas

1.1.1 Objetivo Geral

O objetivo deste trabalho é oferecer um estudo e uma análise sobre a API de *WebSockets*, com a implementação de um serviço utilizando esse recurso.

1.1.2 Objetivos Específicos

1. Estudo teórico sobre o modelo de comunicação da web;

2. Apresentar as principais técnicas para se criar aplicações em tempo real;
3. Analisar a fundamentação teórica sobre a especificação HTML5, com foco na API de *WebSockets*;
4. Levantar os requisitos e ferramentas necessárias para se criar um uma aplicação de compartilhamento de navegação;
5. Implementar a aplicação a partir da análise feita;
6. Apresentar uma análise dos resultados obtidos com a aplicação.

1.2. Motivação

A motivação deste trabalho surgiu a partir da necessidade de oferecer um serviço gratuito, de código aberto, em que se pudesse ter acesso ao navegador Web (*browser*) de um cliente, mas sem a obrigação de se efetuar a instalação de programas de terceiros. São muitas as soluções de compartilhamento de tela no mercado, desde soluções mais simples até soluções mais complexas. Porém, todos esbarram na necessidade de se instalar softwares por parte do compartilhador e do visualizador da tela capturada, o que muitas vezes pode ser complicado em grandes corporações e usuários leigos, bem como atrasar todo o processo já que se leva um tempo para se obter o arquivo de instalação e de fato realizar a instalação do mesmo.

Da mesma forma, esse trabalho visa estimular o estudo e o desenvolvimento de novas tecnologias na web, impulsionadas principalmente pela especificação do HTML5.

Como adicional, promover o uso e a cultura de softwares de código aberto, permitindo que esse código seja empregado e aperfeiçoado tanto no meio acadêmico como também em âmbito comercial.

1.3. Metodologia

A primeira etapa do trabalho consiste no levantamento bibliográfico necessário a fim de se compreender as características e os problemas relacionados ao desenvolvimento de aplicações em tempo real para a Web, buscando dar ênfase às aplicações e técnicas que surgiram ao longo dos

últimos anos, bem como nas novas tecnologias que estão sendo desenvolvidas no presente momento. Com o levantamento bibliográfico, é feito um estudo teórico sobre o tema sobre aplicações em tempo real e todas tecnologias envolvidas no processo para se criar o serviço proposto:

1. Comunicação na web;
2. HTML5;
3. *WebSockets*.

Por fim, é feita uma análise sobre o serviço que será desenvolvido utilizando a API de *WebSockets*, apresentando posteriormente o desenvolvimento e os resultados obtidos.

1.4. Estrutura do trabalho

O capítulo seguinte faz uma introdução ao desenvolvimento de aplicações web, dando foco a parte de comunicação. No terceiro capítulo são apresentados conceitos de aplicações com interação do usuário em tempo real, fazendo uma contextualização com o seu uso nos dias de hoje, bem como apresentando as principais técnicas desenvolvidas ao longo dos anos para se criar esse tipo de aplicação. No quarto, é abordado sobre a especificação HTML5, já que toda API de *WebSockets* está descrita nessa nova especificação. Essa API é bem detalhada no capítulo cinco. No capítulo seis é descrito o desenvolvimento do projeto, com a análise e projeto. No capítulo seguinte é efetuada uma análise sobre os resultados obtidos a partir dos estudos e experimentos realizados. Esse capítulo também destaca as considerações finais sobre o trabalho, bem como sugestões para trabalhos futuros relacionados com a tecnologia de *WebSockets*.

2. Desenvolvimento de Aplicações Web

A *World Wide Web* começou com um programador que teve uma nova ideia de software no grupo de controle e aquisição de dados da Organização Européia para pesquisa nuclear – CERN, localizado em Genebra. Segundo Berners-Lee (1989), o objetivo original do desenvolvimento da web foi tornar mais fácil o compartilhamento de documentos de pesquisas. Tim Berners-Lee propôs que a gerência da CERN adotasse um sistema de informação distribuído baseado em hipertexto, a fim de facilitar o compartilhamento de conhecimento dentro da instituição. O projeto, que inicialmente foi chamado de *Mesh*, convenceu rapidamente todos os gerentes devido a grande perda de informações que acontecia diariamente no CERN. Após um ano de trabalho incessante neste projeto, Berners-Lee finalizou seu projeto e o batizou com um novo nome: *World Wide Web*. Nesse projeto, ele implementou uma série de ferramentas que são usadas ainda hoje:

- O Identificador uniforme de recursos – URI, uma sintaxe para identificar cada documento com um endereço único na web.
- O protocolo de transferência de hipertexto – HTTP, usado para realizar a comunicação entre os dispositivos na web.
- A linguagem de marcação de hipertexto – HTML, para representar documentos na internet.
- O primeiro servidor web, o qual ainda está rodando.
- O primeiro navegador web, o qual Berners-Lee nomeou como *WorldWideWeb*, mas depois acabou mudando para Nexus para não causar confusão com a Web em sim.
- O primeiro editor de HTML, que estava implementado dentro do próprio navegador que ele implementou.

Em agosto de 1991, na primeira página web da história, Berners-Lee escreveu:

“A *WorldWideWeb* é uma grande hipermídia que tem por objetivo dar acesso universal a um grande universo de documentos”.

Daquele momento em diante, a web cresceu em um nível exponencial. Com cinco anos o número de usuários na web já ultrapassava a marca de 40 milhões de usuários (Infonetics Research, 2005). Em um certo momento, o número de usuários dobrava a cada dois meses. O universo de documentos de Berners-Lee era realidade, e estava cada dia se expandido mais e mais.

Cada vez a web evoluiu mais, e as páginas web deixaram de serem apenas páginas estáticas para se comportarem como aplicações. Passou então a ser uma enorme plataforma, com o desenvolvimento de novas linguagens e tecnologias. A demanda pelo desenvolvimento de aplicações web hoje só cresce, e por consequência disso gera uma alta demanda no desenvolvimento e maturação de tecnologias para apoiarem este processo.

2.1 Modelo cliente-servidor

A maioria das aplicações na internet utilizam o modelo de interação Cliente-servidor. Esse modelo descreve a relação em que um cliente realiza uma requisição a um servidor, que responde o pedido.

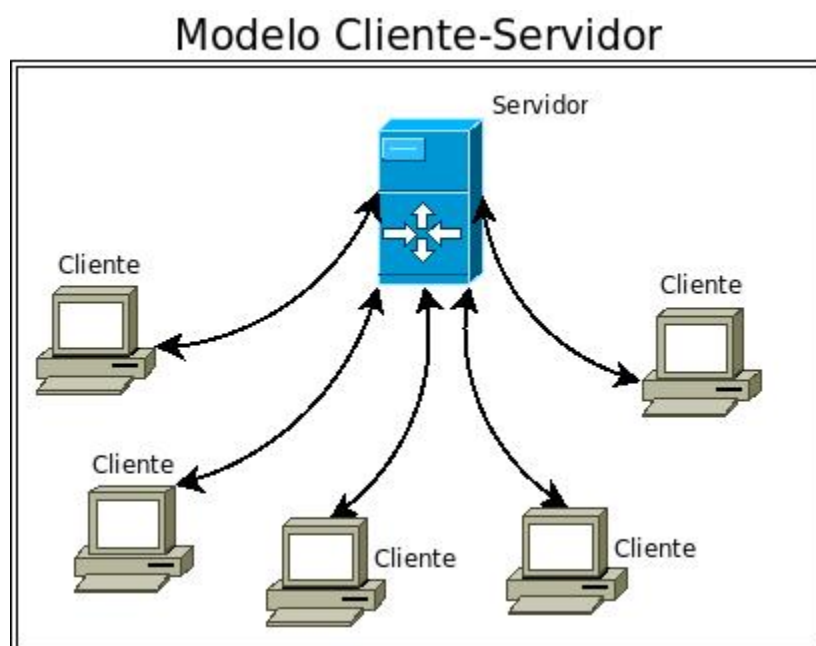


Figura 1: Modelo cliente-servidor (Carvalhana, 2004)

A web foi concebida através desse modelo (Berners-Lee, 1989). O cliente, que representa um navegador de internet comum, interage com um servidor web através do protocolo HTTP – *HyperText Transfer Protocol* – realizando requisições de recursos e dados. Os servidores web, por sua vez, irão analisar o pedido do cliente e retornar uma mensagem com uma série de dados, que podem incluir recursos como documentos HTML ou imagens, ou mesmo uma mensagem de erro. Se necessário, o servidor pode ainda retornar um HTML gerado de forma dinâmica, com dados vindos de um banco de dados, por exemplo (Mendes, 2002).

Uma das grandes vantagens deste modelo é a separação de interesses do cliente e do servidor. Ambos acabam sendo implementados e publicados de forma independente, usando linguagens e tecnologias específicas e com isso é possível que os papéis e responsabilidades de uma aplicação web possam ser distribuídos entre vários computadores que se conhecem e se comunicam através da rede, gerando uma maior facilidade de manutenção. Outro ponto positivo é que todos os dados são armazenados nos servidores, garantindo maiores níveis de segurança, sendo possível ainda oferecer controles de segurança ao acesso de recursos (Litte, 2012).

2.1.1 URLs

Para se acessar qualquer recurso na web através do modelo cliente-servidor, é necessário especificar onde se deseja buscar este recurso. Todos os recursos na web são localizados através da sintaxe URL – *Uniform Resource Locator*. Ela é composta dos seguintes componentes:

- Protocolo: A forma de comunicação que deverá ser usada para acessar o recurso.
- Servidor: Nome da máquina que está fornecendo o recurso
- Domínio: Especifica em que rede se encontra o servidor
- Porta: Porta do servidor web

- Caminho: Localização do recurso dentro do servidor web
- Recurso: Nome do recurso que se deseja obter dentro do servidor web

A URL deve possuir todos esses componentes, já que ela contém apenas o caminho absoluto dentro do servidor.

Para exemplificar, a URL <http://www.ufsc.br> possui os seguintes componentes:

- Protocolo: HTTP
- Servidor: www
- Domínio: ufsc.br
- Porta: Padrão(80)
- Caminho: /
- Recurso: index.php

2.1.2 Tipos MIME

Todo recurso transferido do servidor web para o navegador, seja um documento HTML, imagem ou vídeo, é acompanhado de um cabeçalho que contém o tipo MIME(Turnbull, 2005). Os servidores web não identificam os arquivos com suas extensões, mas apenas associam um arquivo com um tipo MIME específico. Os tipos MIME possuem uma representação padrão, baseando-se no padrão universal MIME – *Multipart Internet Mail Extensions*. A partir da informação do tipo MIME de um documento, o navegador saberá de que forma deve representar para o navegador. Por padrão, o formato do tipo MIME deve seguir a estrutura tipo/subtipo.

A tabela abaixo demonstra exemplos de tipos MIME:

image/jpg	.jpe, .jpg, .jpeg
text/html	.html, .htm, .jsp, .asp, .shtml
text/plain	.txt
x-application/java	.class

Tabela 1: Tipos MIME

2.1.3 Comunicação

A comunicação cliente-servidor é feita a partir do protocolo HTTP (Berners-Lee, 1989), realizando a transferência de recursos dos servidores web aos clientes a partir de requisições feitas por esses últimos.

Tradicionalmente, a comunicação entre o navegador e um servidor web é feita de forma síncrona. Isso significa que após o navegador fazer uma requisição no servidor web, ele deverá aguardar até receber uma resposta. Na prática, o usuário por não poder realizar qualquer interação com o navegador enquanto a resposta não for retornada, o que do ponto de vista da experiência do usuário é bastante ruim. Por isso, há vários casos em que a comunicação assíncrona pode ser mais atraente, visto que um cliente pode realizar uma requisição ao servidor web e continuar o fluxo normal de sua aplicação web. A técnica AJAX é um dos métodos mais comuns em se realizar uma comunicação com o servidor de forma assíncrona. Essa técnica impactou em aplicações web muito mais interativas, justamente por não interromper o fluxo da aplicação durante uma requisição ao servidor.

- **Modelo assíncrono**

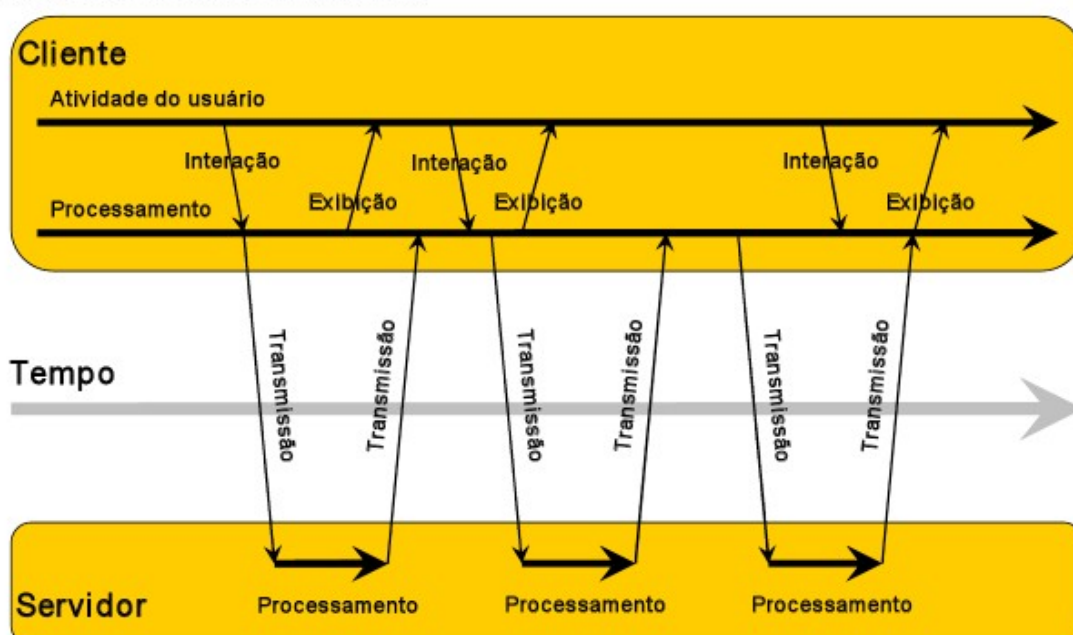


Figura 2: Modelo assíncrono (Paiva, 2006)

2.3 Tecnologias de apresentação

Tecnologias de apresentação são destinadas a formatação e estruturação das páginas web. Atualmente os principais padrões são o HTML e XML. O CSS também surge para auxiliar com a estilização do conteúdo estruturado a partir do HTML ou do XML.

2.3.1 HTML

A linguagem HTML – *HyperText Markup Language* – é uma linguagem de marcação utilizada para marcar documentos na web. Os humanos conseguem identificar facilmente elementos estruturais de uma página, como títulos e parágrafos. Da mesma forma, o navegador deve ser capaz de identificar estes elementos. Para isso, é necessário marcar esses elementos através de *tags*, a fim de criar-se uma semântica para os elementos de um documento da web.

2.3.2 XML

O XML – *eXtensible Markup Language* – é uma especificação de marcação criada pela W3C que consiste em uma metalinguagem, que define uma sintaxe que pode ser usada para se criar novas linguagens de marcação, com *tags* e esquemas personalizados.

2.3.3 CSS

CSS – *Cascading Style Sheets* – é uma linguagem de estilo, capaz de definir a apresentação de documentos escritos em linguagens de marcação, como HTML e XML. O grande benefício do CSS é fazer a separação da camada de estilos da camada de estruturação.

2.4 Programação lado cliente

A programação lado cliente se refere à classe de linguagens de programação executadas no navegador. Atualmente a única linguagem executada de forma nativa nos navegadores é o Javascript, que domina essa classe de linguagens desde 1995(Newswire, 1995). Classificada como uma linguagem de *scripting*, ela é um padrão industrial, especificado pelo ECMA *International*. Ela é fundamental para dar um maior dinamismo às aplicações web, exercendo uma interatividade de acordo com as ações do usuário. Seu uso cresceu principalmente devido à criação da técnica AJAX pela Microsoft em 2000, permitindo aos navegadores realizarem uma comunicação com um servidor web através do protocolo HTTP de forma assíncrona.

2.5 Protocolo HTTP

O protocolo HTTP – *Hypertext Transfer Protocol* – é um protocolo de comunicação da camada de aplicação da web, responsável por definir a estrutura e o modo como as mensagens são trocadas entre cliente e o servidor.

2.5.1 Funcionamento

A partir de uma requisição do cliente através de uma URL, o servidor web fornecerá os recursos através dos protocolos TCP e IP, fazendo com que o cliente receba o documento e saiba representá-lo através do seu tipo MIME. A figura abaixo descreve como as transferências na web ocorrem:

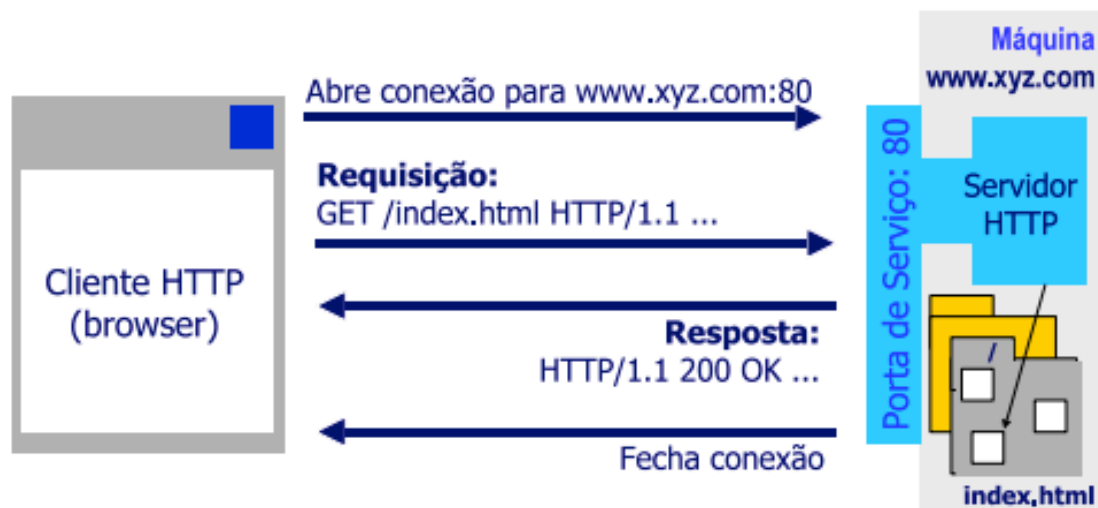


Figura 3: Funcionamento do protocolo HTTP (Pitanga, 2006)

Cada conexão do navegador ao servidor web, possui uma requisição, que é seguida de uma resposta do servidor. A resposta do servidor contém uma linha de status, versão do protocolo, seguido de informações do servidor e conteúdo no corpo da mensagem. Após o envio da resposta, por padrão, a conexão cai e o navegador precisará abrir uma nova conexão caso deseje buscar algo mais(Pitanga, 2006).

O protocolo não memoriza informações sobre o estado entre requisições do navegador ao servidor. Assim, se um servidor receber dez requisições de dez clientes diferentes ao mesmo tempo, ele não saberá distinguir os pedidos, não havendo assim também como uma requisição influenciar na outra. Desta forma, cada cliente deve incluir todas as informações que se consideram relevantes em cada interação com o servidor web. Com os clientes gerenciando o estado de sua aplicação, os servidores web podem suportar um número muito maior de clientes(Evans, 2012).

2.5.2 Mensagem

Toda comunicação entre cliente-servidor é feita a partir de troca de mensagens. O cliente envia uma mensagem de uma requisição e o servidor envia uma mensagem de resposta ao cliente.

Esta mensagem é padronizada conforme definido na RFC 2616, e deve conter uma linha inicial, seguido de uma linha de cabeçalho opcional, uma linha branca e o corpo da mensagem opcional.

2.5.3 Cabeçalho

O cabeçalho é utilizado para transmitir informações adicionais entre cliente e servidor. Tipicamente, contém o nome do servidor, a data e nome e versão do cliente. Dependendo do tipo de requisição, o cabeçalho pode conter ainda dados adicionais, presentes no corpo da mensagem.

2.5.4 Corpo da mensagem

O corpo da mensagem pode estar presente tanto em mensagens de requisição quanto em mensagens de resposta. Quando for uma requisição, o corpo pode conter dados enviados pelo usuário ou um arquivo que está sendo enviado ao servidor – tipicamente em requisições POST. Quando o corpo for de uma mensagem de resposta, ele conterá o recurso que foi requisitado pelo cliente ou uma mensagem de erro caso haja algum problema na obtenção desse recurso. Quando as mensagens possuírem corpo, serão enviadas informações a respeito do recurso, como o tipo MIME e a quantidade de bytes que o corpo da mensagem possui.

2.5.5 Métodos

Segundo Macaneiro (2010), o protocolo HTTP define um conjunto de métodos usados pelos navegadores para enviarem uma requisição ao servidor, que indicam a ação a ser realizada. A versão 1.1 define 8 métodos básicos:

- GET
- HEAD
- POST
- PUT
- DELETE
- TRACE
- OPTIONS
- CONNECT

Um servidor HTTP deve implementar obrigatoriamente os métodos GET e HEAD.

Método	Descrição
GET	O método GET tem a finalidade de solicitar algum recurso do servidor. Esses recursos podem ser um documento HTML, um

	script ou uma imagem, por exemplo.
HEAD	O método HEAD é uma variação do GET. Ele é usado para se obter metadados por meio do cabeçalho da resposta do servidor web, sem a necessidade de recuperar algum recurso.
POST	Envia dados para serem processados no servidor, como dados de um formulário HTML ou mesmo um arquivo a ser enviado para o servidor.
PUT	A entidade enviada na requisição deve ser tratada como uma versão mais recente do recurso a ser atualizado. Um exemplo claro é a atualização de um arquivo mantido no servidor.
DELETE	O servidor deve apagar o recurso identificado pela URI. Por exemplo, queremos apagar um arquivo contido no servidor, basta então realizarmos uma requisição HTTP na URI específica referenciando o método DELETE.
TRACE	Ecoa o pedido ao servidor, de maneira que o cliente possa saber o que os servidores intermediários estão mudando em seu pedido.
OPTIONS	Recupera os métodos HTTP que o servidor aceita.
CONNECT	Serve para uso com um proxy que possa se tornar uma conexão segura.

Tabela 2: Métodos do protocolo HTTP

3. Aplicações em tempo real na Web

Em meados de 1990 a *World Wide Web* estava crescendo muito rapidamente, e estava no caminho para se tornar o meio de distribuição de informação mais dominante (Newswire, 1995). Com o avanço dos navegadores, a web deixou de ser um simples lugar para exibir documentos estáticos, para se comportar como uma plataforma de aplicativos, atingindo mais usuários que qualquer outra plataforma já criada. Inicialmente ela não havia sido projetada para fornecer uma interação rica com o usuário. Entretanto com o passar dos tempos, alguns engenheiros já conseguiam oferecer aplicações com uma melhor experiência de uso. Com o uso dos *iframes* – elemento HTML que exibe o conteúdo de uma página específica dentro da página atual - criava-se a sensação que a comunicação com um servidor era feita de forma assíncrona, já que não se bloqueava toda a aplicação enquanto era realizado a requisição e o processamento no servidor web.

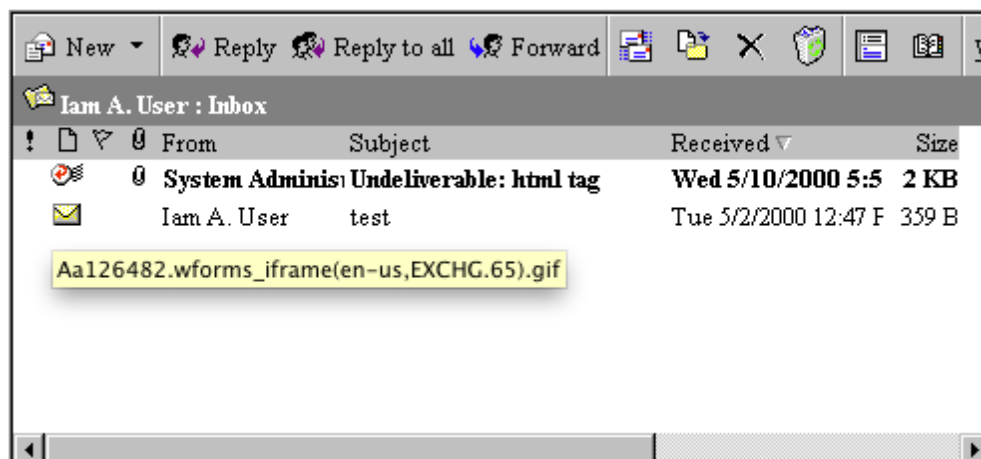


Figura 4: Exemplo das primeiras páginas com Iframe (Fratelli, 2010)

Mesmo com estas soluções bastante rudimentares, a web começou a ficar cada vez mais interativa. Com o surgimento do AJAX - técnica que tornou possível realizar um pedido ao servidor de forma assíncrona, utilizando-se Javascript - a web sofreu uma guinada e passou a ser extremamente dinâmica. A sigla RIA – *Rich Internet Applications* – começou a se popularizar, e a partir

daquele momento as aplicações web começaram a compartilhar uma série de características com as aplicações desenvolvidas no desktop. Todo o processamento da interface passa agora a ser tratado no próprio navegador, permitindo um número muito menor de requisições ao servidor (Fratelli, 2010).



Figura 5: RIAs – Aplicações ricas da internet (Fratelli, 2010)

Em 2004 uma segunda geração de serviços e comunidades na web começou a surgir, tendo como conceito a web como uma plataforma. Essa geração de aplicações foi apelidada de Web 2.0, pela empresa norte-americana *O'Reilly*. Serviços como redes sociais, páginas de distribuição de vídeos, wikis e blogs começaram a se popularizar, e possuíam como característica comum a participação efetiva dos usuários no tráfego de informações (O'Reilly, 2005).

Com uma massa de dados cada vez maior e as aplicações web cada vez mais complexas, devido ao surgimento e aprimoramento de tecnologias, uma demanda pela entrega de informações de forma instantânea começou a surgir, principalmente em aplicações da web 2.0. Implementado inicialmente

pelo *Twitter* para notificar quando um novo *tweet* era efetuado por algum usuário da rede, esse tipo de notificação ficou muito comum na maioria das redes sociais. Dessa forma, as aplicações em tempo real começaram a se popularizar, principalmente devido a esse tipo de aplicação.

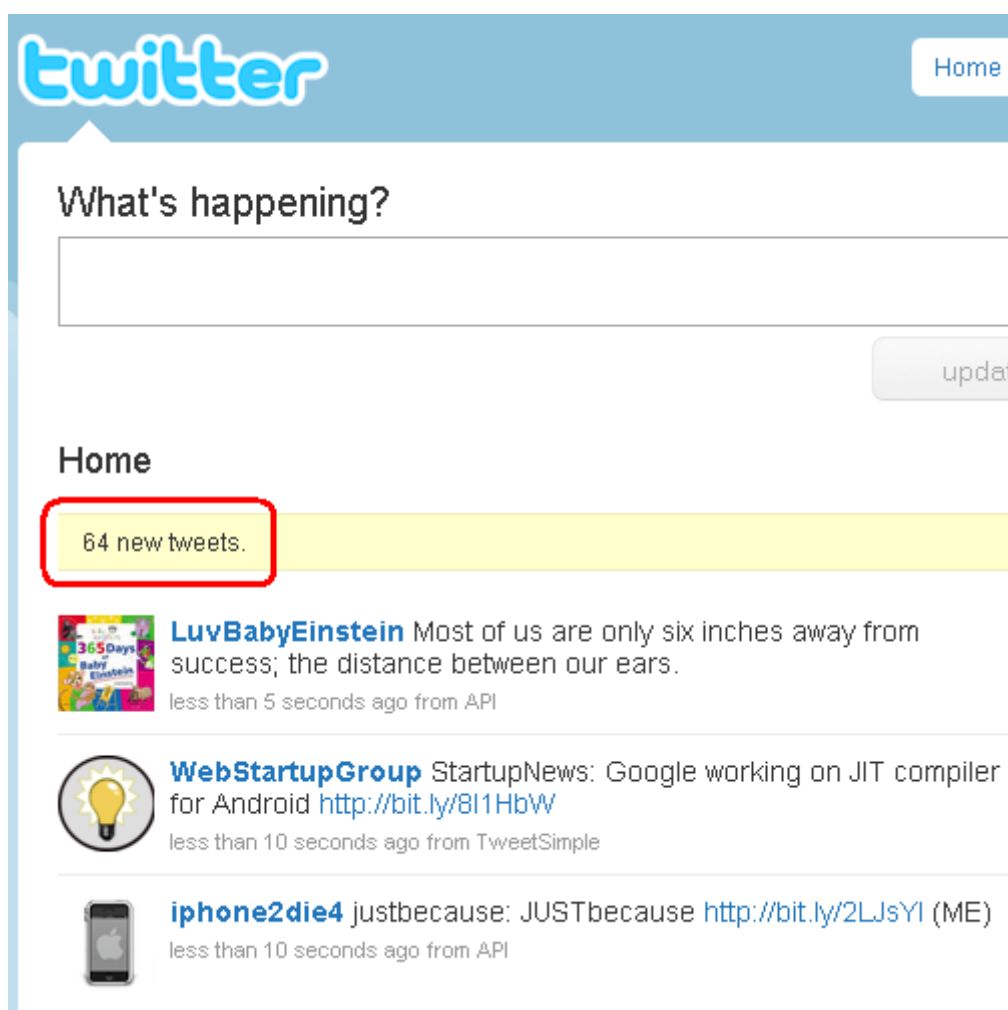


Figura 6: Notificação no Twitter

As aplicações web em tempo real se referem a uma série de tecnologias e práticas que permitem aos usuários receberem notificações assim que os dados são publicados por seus autores, ao invés de realizar requisições em tempos periódicos ao servidor para verificar se há algum dado novo. Esse classe de aplicação também é conhecida como aplicações de tecnologia *push*, já que se baseiam em uma comunicação onde a requisição é iniciada pelo

servidor web, ao invés do tradicional modelo cliente-servidor, que era baseado na requisição de um cliente.

Durante o passar dos anos uma série de tecnologias foram criadas para prover esse tipo de aplicações, que envolveu a técnica *polling* e outras tecnologias do tipo *push*, sendo a mais notável o modelo Comet(Paul, 2007).

3.1. Polling

Segundo Pilgrim (2010), a técnica *Polling* é caracterizada por realizar requisições HTTP em intervalos regulares de tempo a um servidor web. Essa técnica não é boa, e só tende a ser minimizada se conhecer o tempo exato em que um novo dado é fornecido pelo servidor web. A primeira observação dessa técnica é que ela não é considerada uma tecnologia do tipo *push*, já que é o cliente que está realizando a requisição. Além disso, o fato de se realizar conexões no servidor em tempos regulares é um ponto negativo, visto que nem sempre o servidor possuirá novos dados, e logo muitas conexões e requisições podem estar sendo criadas de forma desnecessária.

Com o surgimento do AJAX, essa técnica melhorou bastante, visto que a requisição que antes era feita de forma síncrona passa agora a ser realizada de forma assíncrona, dando uma experiência de usuário muito melhor. No exemplo abaixo é demonstrado o *Polling* utilizando-se AJAX, realizando requisições HTTP no servidor web a cada 100ms com ajuda da biblioteca jQuery:

```
setInterval(function(){
    $.ajax({
        url: '/exemplo',
        success: function(data) {
            // Tratar dado
        }
    });
}, 100);
```

Figura 7: Exemplo do uso de *Polling*

Com as requisições sendo realizadas através do AJAX e com um baixo intervalo de tempo, as aplicações de fato começaram a se comportar como aplicações em tempo real. Entretanto, além do enorme tráfego criado pelo excessivo número de requisições HTTP ao servidor, a obtenção pela resposta poderia demorar mais do que o previsto pelo cliente (100ms no código mostrado na Figura 7), acarretando no pedido de novas requisições sem ao menos ter terminado a primeira delas.

3.2. Comet

Comet é um modelo de aplicações web que permite ao servidor web enviar dados ao navegador, sem este realizar de forma explícita uma requisição. Conhecido também por outros nomes, como *Ajax Push*, *Reverse AJAX*, *Two-way-web*, *HTTP Streaming* e *HTTP Server Push*, este modelo é composto de uma série de técnicas nativas dos navegadores, não fazendo o uso de *plug-ins* de terceiros.

Para este modelo foram criadas duas categorias de implementação: *Streaming* e *Long-Polling*, descritas a seguir.

3.2.1 Streaming

Segundo Bersvendsen(2006), com *Streaming* o navegador envia uma requisição e o servidor envia uma resposta, mas a mantém aberta indefinidamente para que seja continuamente atualizada. Dessa maneira, o servidor pode entregar mensagens futuras, mas nunca sinalizando o fim da resposta. O maior desafio dessa técnica é que os navegadores e os *proxies* não haviam sido projetados para lidar com eventos do servidor, e logo acabaram por se desenvolver vários *hacks* para conseguir esse tipo de interação, cada um com suas vantagens e desvantagens. O maior obstáculo foi que, segundo a especificação HTTP 1.1, um navegador não deve possuir mais de duas conexões abertas simultaneamente com um servidor web ou um

proxy. Logo, mantendo-se uma conexão aberta para ouvir eventos do servidor poderia acabar por bloquear o navegador de enviar novas requisições enquanto se esperava pelo resultado de uma requisição feita anteriormente. Para se contornar isso foi necessário criar um *hostname* diferente para lidar com informações em tempo real, que é um *alias* para o mesmo servidor web.

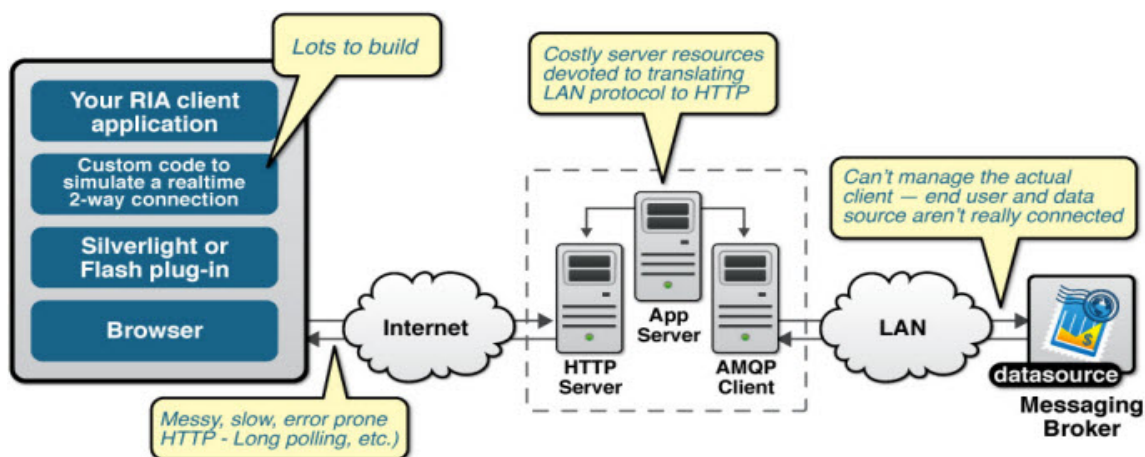


Figura 8: Streaming (Grego, 2009)

O grande problema dessa técnica é que por ela ser encapsulada dentro do protocolo HTTP, *firewall* e servidores *proxy* podem intervir e armazenar a resposta, aumentando o tempo de entrega da resposta. Por isso, muitas aplicações *Comet* acabam usando a técnica *Long-Polling* quando é detectado que a resposta está sendo armazenada.

3.2.1 Long-Polling

Além de eventualmente possuir uma latência na entrega da resposta quando esta for armazenada, nenhuma das implementações *Streaming* trabalham em todos navegadores web. Além disso, é necessário uma implementação complexa para se conseguir enviar eventos a partir do servidor. Dessa forma, muitas aplicações em tempo real optaram por se usar *long-polling*, a qual consiste em uma técnica totalmente baseada no AJAX, que é compatível com a grande maioria de navegadores.

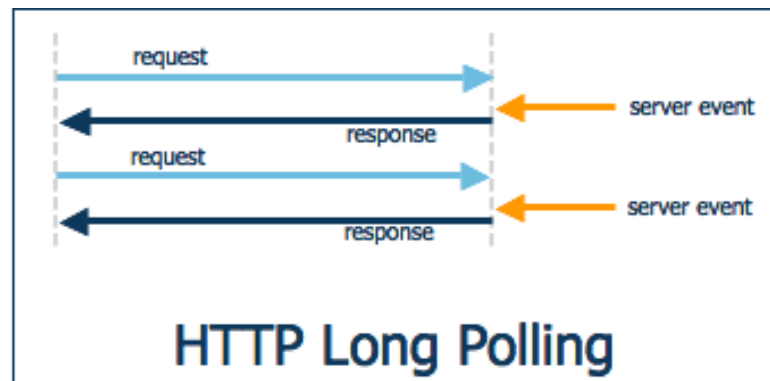


Figura 9: Diagrama da técnica *Long-Polling* (Vital, 2012)

O *long-polling* com AJAX acabou resolvendo os pontos fracos do modelo *Streaming* e da tradicional técnica de *polling*, perguntando ao servidor se há informações novas em um intervalo regular de tempo, mas mantendo a conexão aberta por um período se o servidor não possuir nenhum dado. Durante esse período, era enviado a resposta ao cliente caso algum novo dado fosse disponibilizado. Ao final do processo, o navegador cria e envia outra requisição AJAX, e continua-se nesse processo de forma cíclica.

```
function load() {
    $.ajax({
        url: '/exemplo',
        success: function(data) {
            // Tratar dado
        },
        complete: load,
        timeout: 20000
    });
}
```

Figura 10: Exemplo de uso de *Long-Polling*

Essa técnica acabou reduzindo consideravelmente a latência e o tráfego de rede, mas ao gerar um volume de dados alto ela acaba por não oferecer uma melhora de desempenho em relação ao método *Polling*.

Em todas as técnicas citadas, é usado o protocolo HTTP para se realizar a comunicação. Entretanto, esse protocolo não foi projetado para lidar com esse tipo de aplicação. Um dos fatores que contribuem para isso são os cabeçalhos de resposta, que podem conter uma série de dados que não são necessários, fazendo com que a latência aumente. Outra coisa que merece destaque é que para termos uma conexão *full-duplex*, é necessário mais do que apenas uma conexão do servidor ao cliente. Para simularmos uma conexão *full-duplex* sobre o protocolo HTTP, são usadas duas conexões, o que representa um aumento significativo em termos de complexidade e sobrecarga do servidor.

A dificuldade e os vários problemas envolvidos na criação de aplicações em tempo real estimularam o desenvolvimento de uma API que resolve todos esses problemas. Em 2008 a W3C anunciou a especificação de um novo padrão para web, o HTML5. E um dos itens mais esperados era justamente a API de *WebSockets*, que vinha com o objetivo de se criar aplicações em tempo real de forma fácil, rápido e muito mais escalável se comparado com as técnicas criadas até então.

4. HTML5

No início de 2008, a W3C anunciou a primeira especificação do HTML5. Essa nova especificação inclui grandes alterações, como:

- Novas *tags* para melhoria da semântica;
- Controle embutido de conteúdo multimídia;
- Melhoria na depuração de erros;
- API's *Javascript*.

Esse último, em especial, contém realmente as grandes novidades da nova especificação, e dentro dela há uma seção de comunicação que envolve a API de *WebSockets*, criada para lidar com aplicações em tempo real.

4.1. História

Durante os anos de 1990 a 1995, muito trabalho foi feito em cima do HTML. Até seu quinto ano de vida, a linguagem passou por uma série de revisões e alterações em sua especificação, sendo inicialmente o CERN o primeiro órgão a controlar a especificação, e passando posteriormente para o IETF.

Segundo Pilgrim (2010), a primeira versão do HTML ainda estava pouco madura, e em 1993 surgiu o sucessor do HTML, o HTML 2.0. Apresentado na conferência mundial sobre a web, a *World Wide Web Conference*, essa sucessão veio apenas para corrigir alguns problemas da primeira versão. Nada novo havia sido incluído de fato.

No ano seguinte, mais uma especificação foi criada, e novamente corrigia detalhes da versão 2.0. Essa nova especificação foi denominada 3.2, já que antes de ser lançada, um rascunho do HTML 3.0 havia sido escrito, mas nada havia sido implementado. Além de corrigir detalhes, algumas funcionalidades haviam sido criadas, como tabelas e os *applets*.

Com a versão 2 e 3 sendo lançadas, a linguagem começou a amadurecer, e cada vez mais a ganhar mais apoio de desenvolvedores e fabricantes. Em 1994 foi criada a W3C, um consórcio formado por instituições comerciais e educacionais, que passou a definir os padrões da web. No fim do

ano de 1997, a W3C publicou a nova especificação do HTML 4.0, onde trouxe uma série de melhorias, mudando consideravelmente em relação à especificação anterior. Após dois meses da publicação do HTML, a W3C publicou o XML, uma linguagem de marcação para criação de documentos em geral. A W3C via possibilidades muito maiores com o XML, e no mesmo ano publicou o XHTML, uma reformulação da linguagem HTML, que combinava as marcações do HTML com as regras de XML. A W3C acreditava que a médio prazo o XHTML substituiria o HTML, dessa forma a W3C começou a considerar o HTML uma linguagem morta, alterando seu foco de trabalho para os padrões XML e XHTML, e deixando o HTML de lado. Mesmo com o órgão que regulava os padrões da web considerando o HTML como uma linguagem descontinuada, a maioria da comunidade continuou a servir seus documentos na web nesse formato, e não em XHTML.

O grande impasse foi formalizado em 2004, em um workshop organizado pela própria W3C. Nesse workshop, grandes instituições comerciais estavam presentes, como Opera, Mozilla, Nokia e Apple. Com o objetivo de apresentar uma visão do futuro da web, o W3C deixou claro que o HTML seria descontinuado, cedendo lugar ao XHTML. Assim como a maioria da comunidade, essas empresas ainda acreditavam no HTML, e foi aí que foi necessário apoiar ou se desligar da W3C, e começar a desenvolver uma nova especificação do HTML.

Alguns meses depois surgiu o WHATWG, um pequeno grupo de desenvolvedores de grandes empresas, como Opera e Mozilla, que passaram a trabalhar no desenvolvimento tecnológico de aplicações de hipertexto para web. Mesmo trabalhando apenas por uma lista de email, esse grupo atingiu um nível de maturidade muito grande após 2 anos, e começou a desenvolver uma nova especificação do HTML. Em paralelo a isso, a W3C continuava a trabalhar na nova especificação do XHTML, mas sem muito apoio da maioria das empresas que desenvolviam os navegadores. Com um momento muito melhor, a WHATWG começou a ganhar atenção, e em outubro de 2006, Tim Berners-Lee, o fundador do W3C, se viu obrigado a anunciar a volta do

envolvimento da W3C na especificação do HTML, trabalhando em conjunto com o WHATWG.

Dois anos depois, a primeira especificação do HTML5 surgiu, e um ano após o desenvolvimento do XHTML 2.0 foi parado. Anos após a primeira especificação do HTML5, ela ainda não está totalmente finalizada, e muitos acreditam que ela será revista constantemente, sempre decorrente das necessidades da comunidade.

4.2. HTML5 nos dias de hoje

A especificação do HTML5 que vemos hoje é apenas um rascunho, e portanto não é uma versão final. No fim do ano de 2012 a linguagem passará para o estágio candidata a recomendação, e apenas em 2022 é previsto que ela será a recomendação da Web (Gilbertson, 2008). Mesmo com essa data estando bastante longe dos dias de hoje, isso não quer dizer que não podemos usar HTML5 atualmente. Inclusive, há uma forte crítica da comunidade em relação à imposição desse tipo de data, já que gera um medo aos desenvolvedores para implementarem essa nova especificação em ambiente de produção. Um exemplo claro é o CSS2, que ficou em rascunho durante 10 anos, e só no início de 2012 passou a ser uma especificação final – e nem por isso tivemos que esperar até 2012 para começar a usar a nova versão das folhas de estilo.

Atualmente a especificação já está quase em estágio de candidata a recomendação, e boa partes das funcionalidades já estão bem definidas, e não tão voláteis como eram em 2008. Sendo assim, a maioria dos navegadores atuais já fazem a implementação de boa parte das funcionalidades descritas na especificação. Essa implementação (sem a necessidade de esperar sair a especificação final) só foi possível já que o HTML5 foi construído sobre uma base modular, permitindo que novas funcionalidades fossem adicionadas sem impactar em outras funcionalidades.

Atualmente existe uma corrida para saber qual será o primeiro navegador a implementar um novo recurso proposto pela W3C. Dentro de cada navegador, há um motor, responsável pela renderização e processamento do

código de cada página web (Clary, 2003). Esse motor é conhecido como agente de usuário, e é nele que é feita a implementação de recursos propostos pela W3C.

Motor	Browser
Webkit	Safari e Chrome
Gecko	Firefox e Camino
Trident	IE4 – IE9
Presto	Opera

Tabela 3: Agentes de usuário

Mesmo que alguns recursos sequer possuam uma interface, alguns agentes de usuários fazem uma implementação por sua conta, e por isso é cada vez mais comum vermos alguns recursos sendo utilizados com o prefixo do agente de usuário no início. Um exemplo claro é a recente API de *full screen*, que permite estender para tela cheia elementos da página web ou a própria página web:

	Funcionalidade	
	Habilitar <i>full screen</i>	Desabilitar <i>full screen</i>
Candidato a padrão	.requestFullscreen()	.cancelFullscreen()
Webkit	.webkitRequestFullScren()	.webkitCancelFullscreen()
Gecko	.mozRequestFullScreen	.mozCancelFullScreen()

Tabela 4: Métodos da API *Full Screen* (Kesteren, 2012)

Esse tipo de recurso ainda é pouco recomendável para se usar, mas mesmo estando pouco maduro, já é adotado em ambientes de produção. O Facebook, por exemplo, já adota esse recurso, exibindo as fotos em tela cheia quando o usuário deseja. Mas por ser um recurso muito novo, ainda é necessário utilizar uma técnica proprietária (Flash) para usuários que estiverem utilizando o

Internet Explorer, Opera ou mesmo uma versão mais antiga do Chrome ou do Firefox.

4.3. Princípios chave

Segundo Lubbers (2011), a especificação HTML5 utilizou alguns princípios de projeto durante o seu desenvolvimento, com o objetivo de garantir o sucesso desse novo padrão:

1. Compatibilidade;
2. Utilidade e prioridade;
3. Interoperabilidade;
4. Acesso universal;
5. Paradigma sem *plug-ins*.

4.3.1. Compatibilidade

Um dos principais princípios do HTML5 é manter compatibilidade com todos os documentos web existentes atualmente. Há muitos documentos utilizando a XHTML, HTML4 e até mesmo versões mais antigas, e o HTML5 não pode deixar que esses documentos não sejam mais visualizados, já que já são mais de 20 anos de história do HTML, e há uma quantidade enorme de documentos não escritos em HTML5.

4.3.2. Utilidade

Um ponto bastante interessante da especificação do HTML5 é que ela foi escrita baseada nas necessidades reais da comunidade. Isso significa que os recursos que se encontram na especificação do HTML5 são realmente úteis no desenvolvimento de aplicações web.

Até a conclusão da sua especificação, a W3C ainda aceita sugestões de recursos a serem implementados no HTML5, bastando entrar em contato através das listas de e-mail. Todas as sugestões são analisadas por um grupo, e se for constatada que o recurso é útil, ele pode vir a ser implementado.

4.3.3. Interoperabilidade

É muito importante que todos os sistemas se comuniquem de uma forma transparente entre si. Dessa forma o trabalho com padrões abertos é extremamente importante para não se “quebrar a web”.

4.3.4. Acesso universal

Esse principio está dividido em três conceitos:

- **Acessibilidade:** O HTML5 se preocupa principalmente com usuários com alguma deficiência. A nova especificação inclui um novo padrão, o *ARIA – Accessible Rich Internet Applications*, que são suportadas por leitores de tela, fazendo toda leitura do documento através de atributos nos elementos HTML.
- **Independência de mídia:** Todas as funcionalidades do HTML5 devem funcionar em todos dispositivos e plataformas possíveis.
- **Suporte para todas linguagens:** Por exemplo, há um novo elemento chamado *ruby*, que suporta a notação *ruby*, que é usada no leste da Ásia.

4.3.2. Paradigma sem plug-ins

O HTML5 fornece suporte nativo para uma série de recursos que só seriam possíveis através do uso de *plug-ins* proprietários ou de complexos *hacks*, como é o caso da tecnologia *Comet*. A especificação deixa bem claro que todos recursos deverão ser padrões da web, nativos dos navegadores, sem depender de *plug-ins*. E de fato há muitos motivos que levaram a definir este principio:

- *Plug-ins* podem ser desabilitados ou bloqueados, como é o caso o *flash* no sistema operacional IOS;
- Nem sempre estão instalados em todas máquinas;
- Nem sempre podem ser instalados;
- São proprietários, dependentes de empresas de terceiros;

- São difíceis de se integrar com o resto do documento HTML, já que todo código está em uma camada diferente.

4.4. Novas funcionalidades

O HTML5 deixa de ser uma linguagem simples de marcação para se tornar uma linguagem extremamente poderosa e complexa. A sua estrutura tem mudado bastante, principalmente pelo fato de serem adicionados novos itens regularmente. Mas basicamente ela se divide em duas partes:

1. Core;
2. API's Javascript.

4.4.1. Core

Faz toda especificação de marcação. O HTML5 traz uma série de novas *tags*, com o objetivo de trazer mais semântica aos documentos na web. Muitas das *tags* foram criadas a partir da análise de bilhões de páginas na web, feita pela Google e pela Opera. Um exemplo claro foi a criação da nova *tag* `<header>`, já que foi notado que cerca de 30% dos documentos na web traziam representações com o mesmo significado, mas sem um padrão definido. Com uma semântica melhor dos documentos, o resultado foi páginas mais bem ranqueadas em motores de busca, e também mais fáceis de serem interpretadas por leitores de telas para pessoas cegas.

4.4.2. APIs Javascript

É a parte que realmente chama atenção no HTML5, trazendo recursos bastante interessantes. Javascript foi a linguagem escolhida para se implementar os recursos, já que é a única linguagem cliente que roda nativamente em praticamente todos navegadores, além de ser um padrão de mercado. Dentre os novos recursos destacam-se as API's abaixo:

- Canvas: Permite fazer a representação de desenhos 2D;
- Geolocalização: Fornece a localização do cliente;

- **Áudio e Vídeo:** Permite a representação de elementos multimídia, como áudio e vídeo, da mesma forma que representamos uma imagem;
- **Formulários:** Cria formulários dos mais variados tipos, principalmente para melhorar a acessibilidade para dispositivos móveis e pessoas cegas. Além disso é possível realizar de forma fácil a validação dos valores no lado do cliente, que antes só era possível através do *Javascript*.
- **SVG:** Permite representar arquivos no formato SVG direto nas páginas web.
- **Workers:** Permite criar *threads* no lado do cliente, isso é, no código Javascript.
- **Storage:** Armazena dados no próprio navegador do cliente.
- **Eventos server-side:** Permite ouvir eventos vindos do servidor web.
- **Sockets:** Cria um canal de comunicação bidirecional entre cliente e servidor

4.5. Desenvolvimento de aplicações em tempo real com HTML5

O HTML5 contempla todas funcionalidades necessárias para se construir aplicações em tempo real através da API de WebSockets. Essa API revoluciona a forma como se desenvolve aplicações em tempo real, já que resolve uma série de problemas das técnicas de *push* que existiam até então. Definida na sessão de comunicação da especificação do HTML5, a API de *WebSockets* define um canal de comunicação *full-duplex*, que opera em um socket único através da web (Greco, 2012).

Com esse padrão bem formalizado pela W3C, é possível construir aplicações em tempo real de forma escalável, exigindo uma complexidade muito menor que as técnicas que haviam sido desenvolvidas.

Atualmente esse é uma das especificações mais maduras, e já é amplamente usado em grandes aplicações, em destaque para Gmail, Facebook e Twitter.

5. WebSockets

Historicamente, desenvolver aplicações web que precisem de comunicação bidirecional entre cliente e servidor, era significado de um abuso do protocolo HTTP, devido às incessantes requisições ao servidor web para verificar se havia algum dado novo, além da criação de duas conexões TCP, uma para enviar mensagens ao cliente e outra para receber mensagens (Greco, 2012).

Uma simples solução para esse problema foi usar uma única conexão TCP para trafegar entre ambas direções. É isso basicamente que a especificação WebSockets define. Ela não é uma simples melhora na comunicação do tradicional protocolo HTTP. Ele representa uma revolução na comunicação da web, especialmente para aplicações em tempo real e aplicações baseadas em eventos *server-side*, também conhecidas como aplicações de classe *push*.

A especificação dessa API descreve um novo protocolo para trafegar os dados de maneira bidirecional, entre cliente e servidor, operando em um socket único através da porta 80 (no caso do protocolo HTTP) ou da porta 443 (protocolo HTTPS). Além disso, foi especificada uma interface JavaScript para de fato usar esse protocolo e ouvir os eventos disparados pelo servidor. Com esse novo protocolo, problemas ocorridos nas técnicas de Comet e *Polling*, como o grande tráfego de rede e a alta latência foram totalmente resolvidos. E com a interface JavaScript, agora é possível criar aplicações desse tipo muito mais facilmente, reconhecendo também uma série de eventos padronizados pelo servidor web.

O protocolo WebSocket foi projetado para substituir todas as outras tecnologias existentes de comunicação bidirecional na web que usam o protocolo HTTP como transporte, principalmente para beneficiar a infraestrutura existente. Hoje o protocolo WebSocket ainda é dependente do HTTP, já que para usar o protocolo WebSocket deve-se obrigatoriamente realizar um *handshake* entre cliente e servidor utilizando o protocolo HTTP.

Futuramente isso pode mudar, sendo esse *handshake* feito através de uma porta, sem precisar depender mais do tradicional protocolo HTTP.

O uso desse recurso hoje é grande, e podemos ver em uma série de aplicações que usamos no dia-a-dia. Desde o chat no GMail ou Facebook, notificações em sua rede social preferida, edição em grupo do Google Docs, e até jogos online já fazem o uso da API de WebSockets.

5.1. Compatibilidade

Como WebSockets não opera apenas no cliente, mas também no servidor web, deve-se necessariamente ter a interface da API JavaScript implementada no navegador, bem como o servidor web deve dar suporte ao novo protocolo WebSocket para se fazer o uso desse recurso.

Atualmente a maioria dos navegadores já suporta a API JavaScript. O Chrome foi o pioneiro, e desde a versão 4.0, lançada em 2008 já dá suporte à tecnologia. Dentre os principais navegadores, somente o Opera mini e o navegador padrão do Android não suportam tal tecnologia. No Internet Explorer a tecnologia também só é acessível para a versão 10.0, que ainda não está totalmente concluída(Fyrd, 2012).

Desktop

	IE	Firefox	Chrome	Safari	Opera
Suporte	Sim	Sim	Sim	Sim	Sim
1ª versão a suportar	10.0	4.0	4.0	5.0	11.0
Versão atual	10.0	15.0	21.0	5.2	12.0

Tabela 5: Suporte do WebSockets em *browsers desktop*

Celular

	IOS	Opera Mini	Opera mobile	Android	Chrome Mobile

Suporte	Sim	Não	Sim	Não	Sim
1ª versão a suportar	4.2		11.0		1.0
Versão atual	4.3		12.0		1.0

Tabela 6: Suporte do WebSockets em *browsers mobile*

Dos servidores web, os principais já realizam a implementação, entre eles:

- Apache
- Nginix
- Node
- Tornado
- JBoss
- Jetty
- Tomcat
- GlassFish

5.2. Protocolo WebSocket

O protocolo divide-se nas seguintes partes:

- URIs;
- Handshake;
- Data Framing.

5.2.1. URIs

A especificação prevê um esquema para URIs para se abrir um socket:



Figura 11: Padrão WebSockets de URIs (Hansa, 2010)

Além deste padrão de URI's, é ainda oferecido um método seguro, semelhante ao HTTPS. Para fazer o uso, basta alterar o início da URI para WSS.

5.2.2. Handshake

Segundo Lubbers (2012), para se estabelecer uma conexão WebSocket, é necessário que cliente e servidor iniciem usando o tradicional protocolo HTTP, e então realizem a migração para o protocolo WS. Esse processo é conhecido como *Handshake*.

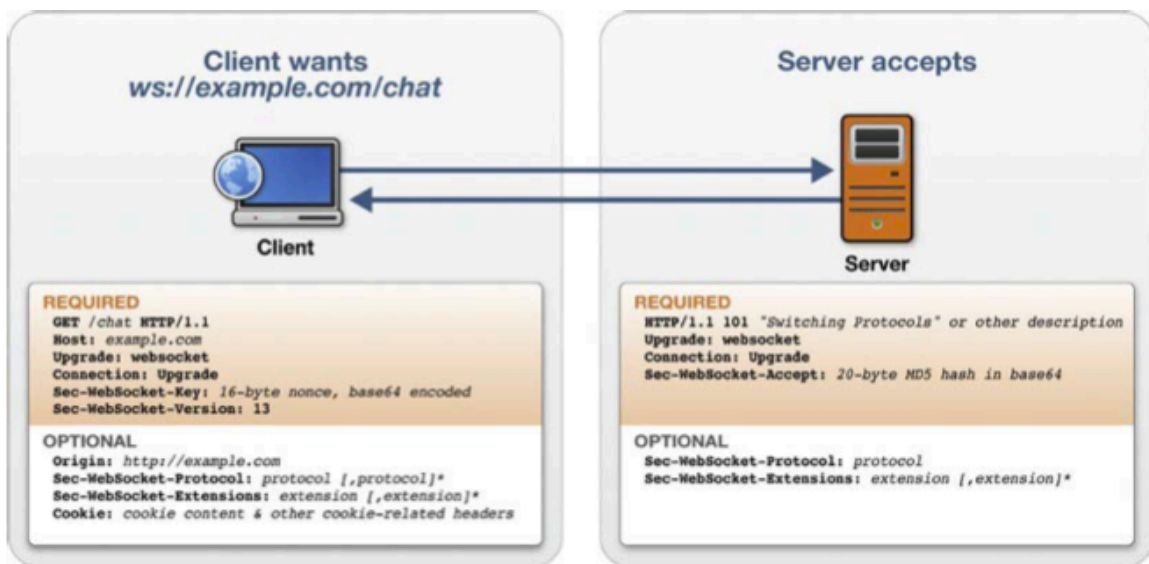


Figura 12: Processo de *HandShake* (Grego, 2009)

Durante o processo, os seguintes passos são executados:

1. Cliente envia um *handshake* ao servidor contendo o seguinte cabeçalho e estrutura:

```
GET /chat HTTP/1.1
Host: example.com
Connection: Upgrade
Sec-WebSocket-Protocol: sample
Upgrade: websocket
```

Sec-WebSocket-Version: 13

Sec-WebSocket-Key: 7cxQRnWs91xJW9TOQLSuvQ==

Origin: http://example.com

2. Servidor lê as informações enviadas pelo cliente e verifica a validade da requisição;
3. Se tudo for válido, o servidor envia um *handshake* para o cliente contendo um cabeçalho com a seguinte estrutura:

HTTP/1.1 101 WebSocket Protocol Handshake

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: 7cxQRnWs91xJW9TOQLSuvQ==

WebSocket-Protocol: sample

4. Cliente recebe do servidor o *handshake* e verifica sua validade;
5. Se tudo tiver ocorrido bem, a conexão estará estabelecida.

5.2.2. Framing

Após feito o *handshake*, o cliente e o servidor podem enviar mensagens a qualquer tempo, de maneira bidirecional. Toda comunicação é transmitida usando uma sequência de quadros (*frames*). Para não causar problemas em dispositivos de rede intermediários (como servidores *proxy*) e também por questões de segurança, todos os dados enviados do cliente ao servidor devem passar pelo processo de mascaramento (*masking*). O processo de mascaramento consiste em inserir uma chave randômica de 32 bits, derivada a partir de uma fonte com uma forte entropia.

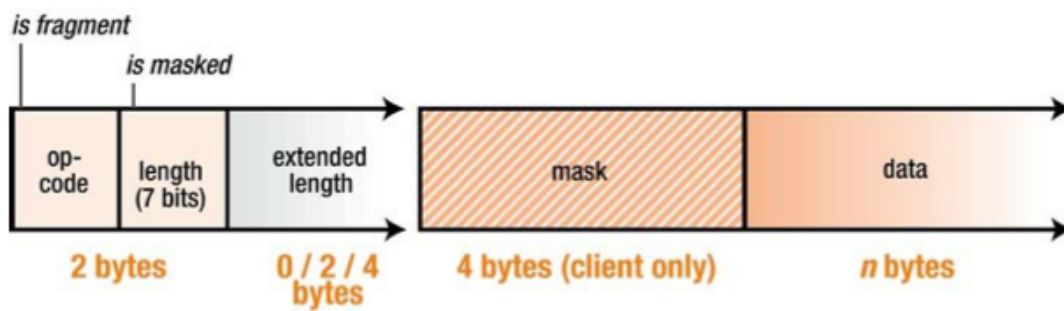


Figura 13: Framing (Grego, 2009)

O servidor deve fechar a conexão caso algum dado não esteja mascarado. Nesse caso, o servidor deve enviar um quadro de fechamento, com código de status 1002 – que significa um erro no protocolo. Pra não ter problemas de codificação, todas mensagens devem estar no formato UTF-8.

5.3. Interface

Além do protocolo WebSocket, a especificação também inclui uma interface para o cliente definida em JavaScript.


```

[Constructor(DOMString url, optional DOMString protocols),
Constructor(DOMString url, optional DOMString[] protocols)]
interface WebSocket : EventTarget {
  readonly attribute DOMString url;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSING = 2;
  const unsigned short CLOSED = 3;
  readonly attribute unsigned short readyState;
  readonly attribute unsigned long bufferedAmount;

  // networking
  [TreatNonCallableAsNull] attribute Function? onopen;
  [TreatNonCallableAsNull] attribute Function? onerror;
  [TreatNonCallableAsNull] attribute Function? onclose;
  readonly attribute DOMString extensions;
  readonly attribute DOMString protocol;
  void close([Clamp] optional unsigned short code, optional DOMString reason);

  // messaging
  [TreatNonCallableAsNull] attribute Function? onmessage;
  attribute DOMString binaryType;
  void send(DOMString data);
  void send(ArrayBuffer data);
  void send(Blob data);
};

```

Figura 14: Interface da Api *WebSockets* (Hickson, 2009)

Percebe-se que a interface é bastante simples, mas oferece o necessário para se abrir, fechar e enviar mensagens, bem como tratar os eventos.

5.3.1. Construtor

Para se abrir uma conexão com um socket basta criar um objeto `WebSocket`, acessível globalmente. Para tal, deve ser enviada como parâmetro a URL do socket, no formato padrão segundo a especificação (Hansa, 2010).

```

websocket = new WebSocket('ws://localhost:8080/echo');

```

Quando se conecta em um WebSocket há ainda a possibilidade de enviar um segundo parâmetro, que pode ser um *String* ou um *array*, que contém os nomes dos sub-protocolos que a aplicação entende e deseja usar para se comunicar.

```
websocket = new WebSocket(url, protocolo);  
websocket = new WebSocket(url,[protocolo1, protocolo2]);
```

5.3.2. Métodos

São oferecidos dois métodos para o cliente:

1. Send;
2. Close.

O método *send* envia um dado ao servidor web.

```
websocket.send(mensagem);
```

O método *close* é disparado quando uma conexão é encerrada.

```
websocket.close();
```

Pode receber 3 parâmetros: Código, razão e se foi fechada por erro.

Atributo	Tipo	Descrição
Código	Long	Código fornecido pelo servidor web
Razão	String	String identificando a

		razão pela qual o servidor fechou a conexão
Fechado por erro	Boolean	Indica se a conexão foi fechada por um erro ou não

Tabela 7: Parâmetros do evento close

5.3.3. Eventos

Segundo Hansa (2010), a API oferece alguns eventos para o tratamento da conexão e para receber as mensagens vindas do servidor:

- Open;
- Message;
- Close;
- Error.

O evento *Open* é disparado quando uma conexão é estabelecida.

```
websocket.onOpen = function() {
  console.log('conexão estabelecida');
}
```

O evento *Message* indica o recebimento de uma mensagem do servidor.

```
websocket.onMessage = function() {
  console.log('Mensagem');
}
```

O evento *Close* disparado quando a conexão é fechada.

```
websocket.onClose = function() {  
  console.log('conexão encerrada');  
}
```

Por fim, o evento *Error* indica o recebimento de uma mensagem de erro do servidor.

```
websocket.onError = function() {  
  console.log('mensagem de erro recebida');  
}
```

6. Projeto

Este capítulo apresenta o desenvolvimento de uma aplicação utilizando a API de *WebSockets*, como cumprimento de um dos objetivos desse trabalho.

6.1. Processo de análise

6.1.1. Descrição

Este trabalho visa desenvolver um sistema de navegação compartilhada, onde um usuário irá compartilhar o conteúdo de uma janela de seu *browser* com outro usuário. A solução será livre de *plug-ins* e registros por parte do visitante, sendo útil principalmente para os administradores de sistema auxiliarem os visitantes no uso do sistema, bem como para reportar *bugs* de forma fácil e rápida.

O grande diferencial do sistema é que ele poderá ser integrado a qualquer aplicação publicada na *web*, independentemente da linguagem utilizada. Um exemplo claro desse tipo de aplicação é o *Google Analytics* – onde basta inserir um código JavaScript em sua página e você terá acesso a informações relevantes sobre os acessos em sua página.

Seguindo estes moldes, será desenvolvida uma aplicação à parte, chamada Observador. Nesta aplicação, administradores de páginas ou sistemas publicados na *web*, com o interesse de observar a navegação de seus visitantes, irão efetuar um registro e seguir alguns procedimentos simples para adicionar esta funcionalidade. Sem a necessidade de realizar qualquer alteração na página ou sistema atual, o Observador não irá trazer qualquer impactos quanto a performance, já que ele estará rodando separadamente em uma infraestrutura dedicada.

Com os procedimentos realizados na página, quando um visitante acessar a página em questão, será criado de forma dinâmica um botão oferecendo ao usuário a capacidade de compartilhar sua navegação com o administrador.

Para que a navegação seja compartilhada será necessário que o visitante clique por vontade própria no botão “Compartilhar navegação”, bem

como de ter um administrador online no sistema Observador. Além disso, este deverá responder positivamente para que a navegação compartilhada seja iniciada. Quando iniciada, será oferecido um *chat* para que os usuários se comuniquem. O administrador ainda terá acesso a informações detalhadas a respeito do sistema utilizado pelo cliente, tais como: plataforma, navegador, agente de usuário, IP e geolocalização, a fim de facilitar, por exemplo, um serviço de suporte. Por questões de segurança e privacidade, o compartilhamento poderá ser cancelado por qualquer um dos usuários a qualquer momento.

6.1.2 Análise de requisitos

Requisitos são uma descrição das necessidades de um software. O objetivo básico da fase de requisitos é identificar e documentar o que é realmente necessário, em uma forma que comunica claramente essa informação ao cliente e aos membros da equipe de desenvolvimento (LARMAN, 2000)

A classificação dos requisitos de software pode ser: funcionais ou não funcionais. Requisitos funcionais (RF) são as funções que o usuário espera que o software faça. O termo função é uma operação genérica a ser realizada pelo sistema, através de comandos dos usuários ou eventos do sistema.

Já os requisitos não funcionais (RNF) são as qualidades do software, como usabilidade, desempenho, tecnologia, etc. A primeira etapa na análise foi a identificação destes requisitos:

Requisitos funcionais:

- RF01: Cadastro junto ao site da aplicação Observador;
- RF02: *Login* no sistema, utilizando email e senha;
- RF03: Cadastro de um domínio a uma conta;
- RF04: Definir configurações de um domínio
- RF05: Alterações dos dados da conta do usuário

- RF05: Geração de um *JavaScript* dinâmico para cada subdomínio cadastrado no sistema administrador;
- RF06: Receber chamadas para visualizar a navegação de um cliente;
- RF07: Criação dinâmica de um botão para o cliente realizar o compartilhamento de tela;
- RF08: Compartilhamento da navegação;
- RF09: Troca de mensagens no *chat* criado durante o compartilhamento;
- RF10: Especificação de informações detalhadas acerca do sistema, *browser* e localização do cliente.

Requisitos não-funcionais:

- RNF01: Utilização da API de *WebSockets* para o envio/recebimento dos dados;
- RNF02: Ser compatível com qualquer sistema operacional e servidor web;
- RNF03: Codificação usando padrões da web: HTML, CSS e JavaScript;
- RNF04: Ser compatível com os *browsers* de mercado;
- RNF04: Codificação da aplicação Observador utilizando a linguagem Node.JS. Esta linguagem nada mais é que a implementação do JavaScript no lado servidor. A escolha para a utilização desta foi motivada pela qualidade das bibliotecas que implementam *WebSockets*, bem como para utilizar a mesma linguagem tanto no lado servidor, como no lado cliente.
- RNF05: Utilização de banco de dados NOSQL utilizando MongoDB;
- RNF06: Ser livre de *plug-ins* e/ou extensões;
- RNF07: Desempenho igual ou superior a 3 quadros por segundo.

6.1.3 Diagrama de casos de uso

Foi utilizado UML como linguagem de modelagem do projeto. Os diagramas de caso de uso tem o objetivo de descrever as principais funcionalidades do sistema do ponto de vista do usuário. Para desenvolvimento da modelagem UML foi utilizado o software *Visual Paradigm* versão 10.0. A figura abaixo representa os casos de uso identificados no sistema a ser desenvolvido:

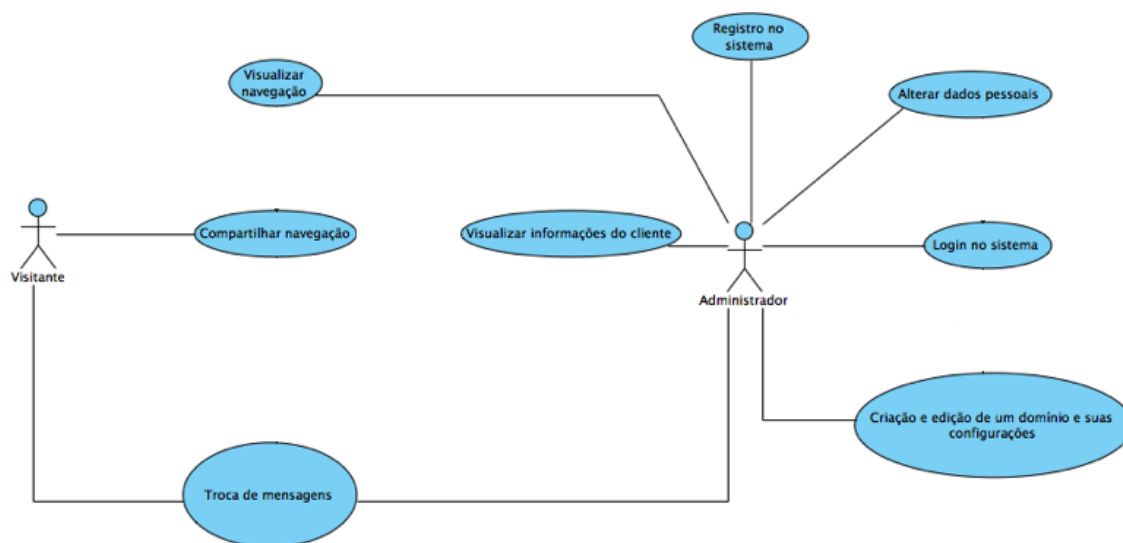


Figura 15: Diagrama de casos de uso

É possível identificar que foram definidos dois atores: Visitante e administrador. O ator visitante representa o usuário que irá visitar uma página e possivelmente compartilhar a sua navegação, enquanto que o ator administrador representa o usuário que irá visualizar a navegação, além de realizar todos os procedimentos necessários para adicionar a funcionalidade de compartilhamento de navegação no sistema. Foram identificados oito casos de uso, detalhados abaixo:

1. Registro no sistema:

Tudo começa por este caso de uso. Um administrador de uma página ou sistema publicado na web deve efetuar um registro no sistema

Observador a fim de adicionar a funcionalidade de compartilhamento de navegação nas páginas que deseja.

2. Login no sistema

Após feito o registro, o administrador deverá efetuar um login junto ao sistema, informando email e senha. Caso as credenciais estiverem corretas, ele terá acesso ao sistema e poderá realizar as operações que deseja.

3. Alterar dados pessoais

Altera dados pessoais da conta, como nome, email e senha.

4. Criação e edição de um domínio e suas configurações

Para adicionar a funcionalidade de compartilhamento de navegação, o usuário deverá indicar primeiramente qual o domínio desta página. Com isso, será necessário efetuar mais alguns procedimentos de configurações para que então seja gerado um código JavaScript, para ser inserido na página com o domínio que foi indicado. Com isso feito, a página já estará apta a oferecer a funcionalidade.

5. Compartilhar navegação

Com as configurações feitas pelo administrador, o visitante ao acessar a página *web* poderá realizar o compartilhamento de sua navegação a partir de um botão criado na lateral da página.

6. Visualizar navegação

Quando um usuário realizar a ação identificada pelo caso de uso número 5(Compartilhar navegação), será exibido ao administrador um pedido de compartilhamento de navegação. Se o administrador optar por aceitar o pedido, o compartilhamento da navegação será iniciado.

7. Visualizar informações do visitante

É importante que o administrador tenha acesso a informações detalhadas sobre o cliente, como plataforma, *browser*, IP e geolocalização, de forma precisa e fácil. Estas deverão estar dispostas na lateral da tela do administrador, abaixo do *chat*.

8. Troca de mensagens

A comunicação entre os usuários também deverá acontecer através de um *chat*, localizado na lateral da tela dos usuários.

6.1.4 Diagramas de atividade

Após a modelagem do diagrama de casos de uso foram feitos os diagramas de atividade. Estes representam uma sequência de ações, suportando nós de decisão e regiões de interrupção.

1. Registro no sistema:

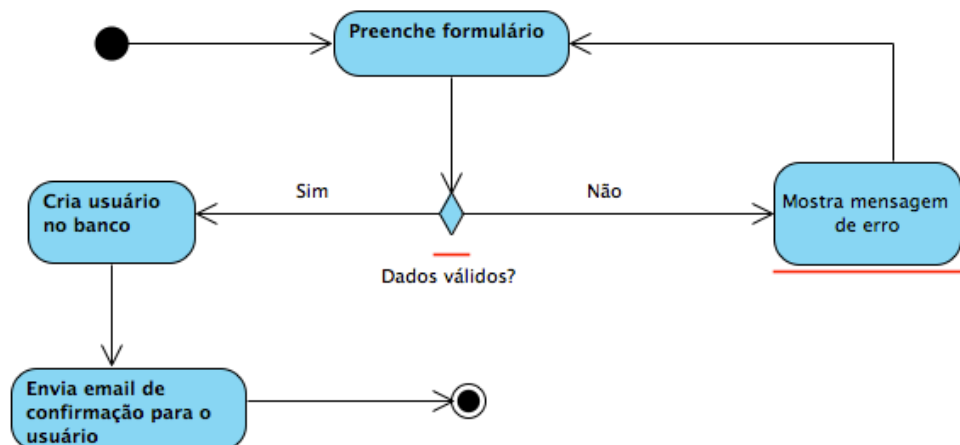


Figura 16: Diagrama Registro no sistema

2. Login

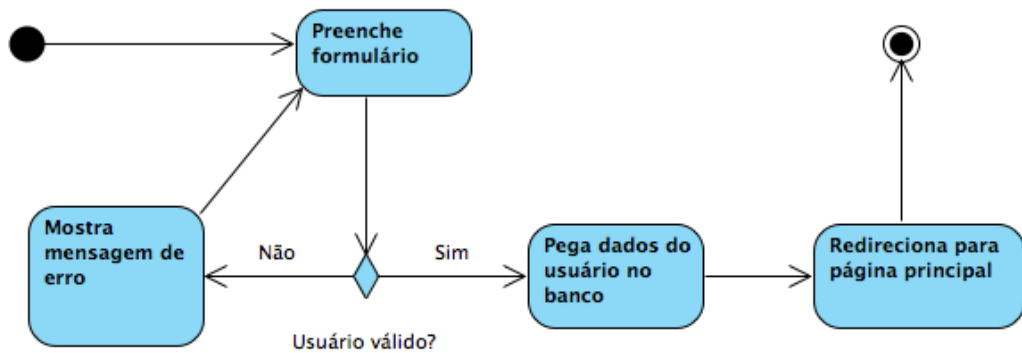


Figura 17: Diagrama Login

3. Alterar dados pessoais

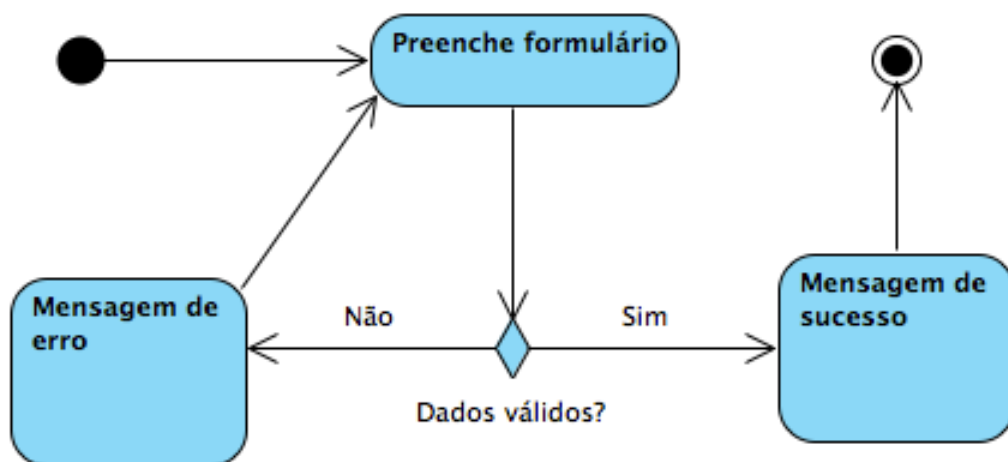


Figura 18: Diagrama alterar dados pessoais

4. Criação e edição de um domínio e suas configurações

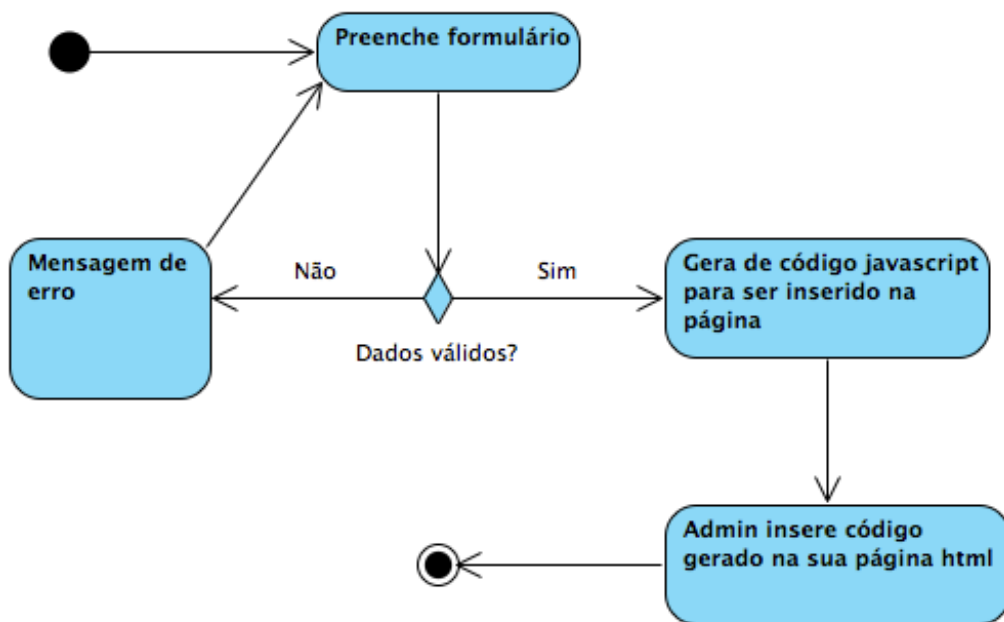


Figura 19: Criação e edição de um domínio e suas configurações

5. Compartilhar navegação

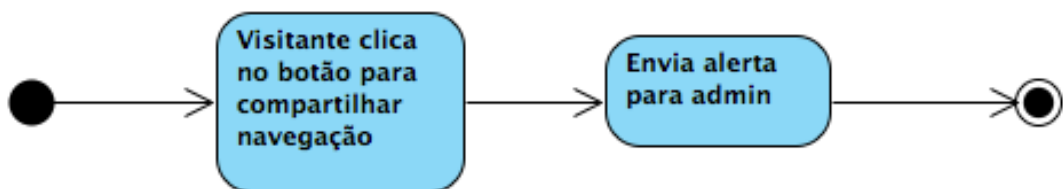


Figura 20: Compartilhar navegação

6. Visualizar navegação

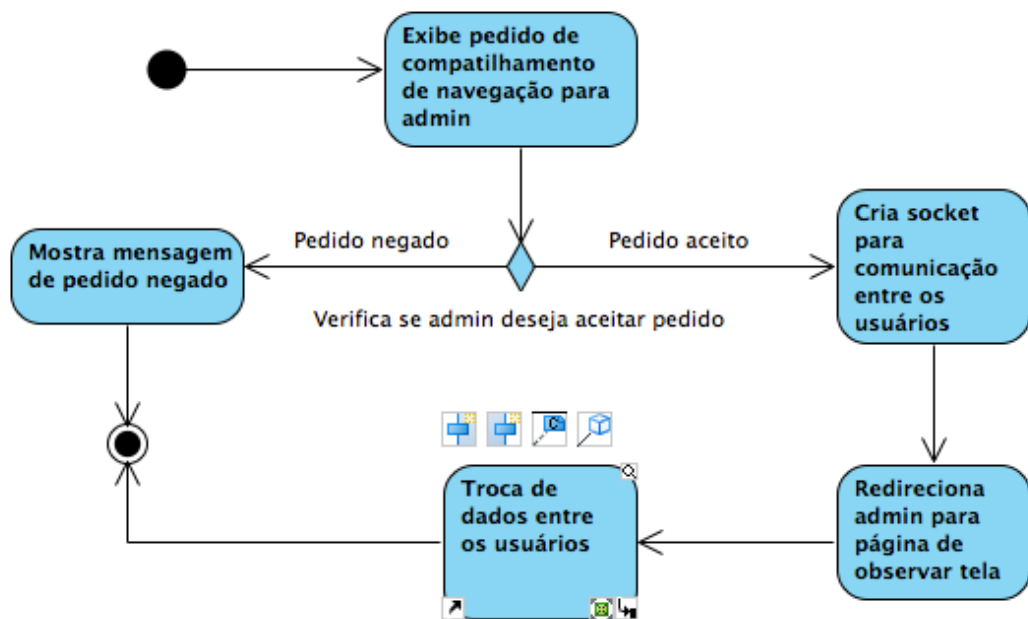


Figura 21: Visualizar navegação

7. Visualizar informações do visitante

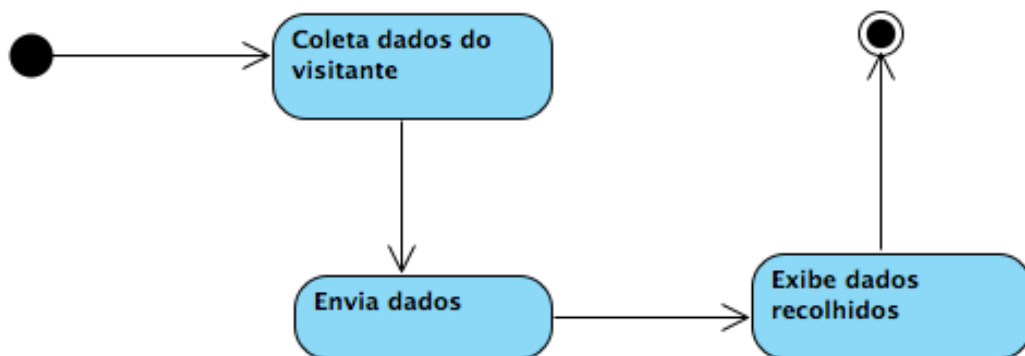


Figura 22: Visualizar informações do visitante

8. Troca de mensagens

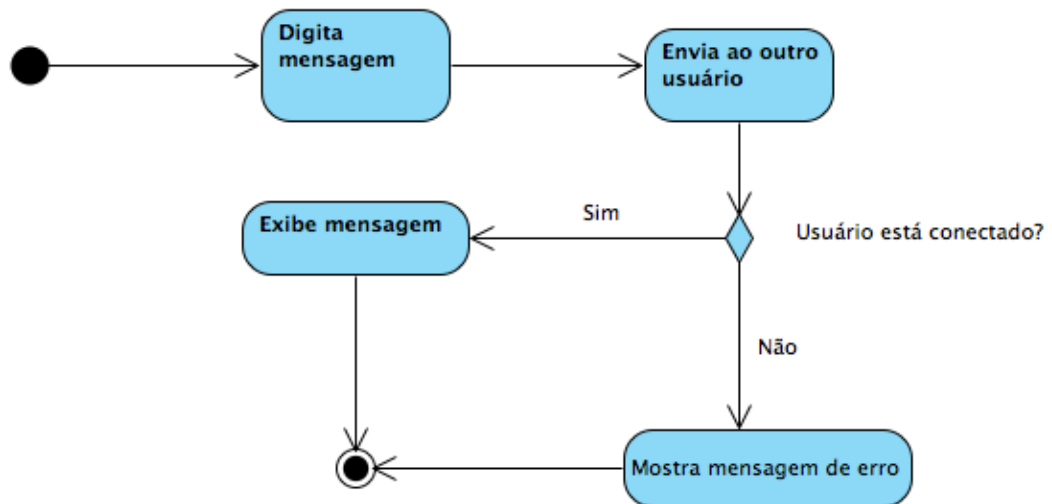


Figura 23: Troca de mensagens

6.1.6 Modelagem do banco de dados

Por questões de simplicidade e familiaridade com a tecnologia, optou-se por utilizar um banco de dados não relacional, o MongoDB. A estrutura do banco criado é bastante simples, já que a maioria das interações da aplicação são em tempo real, sem haver a necessidade de serializar os dados para um acesso posterior. Basicamente o banco de dados só lida com o acesso dos usuários no sistema, além de configurações acerca de um domínio a ser adicionado pelo usuário em sua conta. Na modelagem do banco NOSQL, foram usados apenas 2 modelos:

- User:

```

{
  "activated" : { type: Boolean, default: true },
  "domains" : [
    type: mongoose.SchemaTypes.ObjectId
    ref: Domain
  ],
  "createAt" : { type: Date, default: Date.now },
  "email" : { type: String, required: true },
  "firstName" : { type: String, required: true }
  "lastName" : { type: String },
  "password" : { type: String, required: true },
  "role" : { type: Number, default: 1 , required: true }
}
  
```

- Domain:

```
{ "name" : { type: String, required: true },
  "url" : { type: String, required: true },
  "activated" : { type: Boolean, default: true },
  "buttonPosition" : { type: String, default: 'left'},
  "sidebarPosition" : { type: String, default: 'left'},
  "chat" : { type: Boolean, default: true },
  "config" : { type: Boolean, default: true }
}
```

6.2. Desenvolvimento

6.2.1. Ambiente

Esta breve seção tem como objetivo apresentar o ambiente de desenvolvimento que foi utilizado durante todo o projeto.

Para realização deste projeto foi utilizada uma máquina com sistema operacional MacOS 10.7, sendo necessário instalar os seguintes softwares:

- Node.js versão 0.8, rodando sobre a porta 3000;
- Framework Express.js para o rápido desenvolvimento da aplicação web;
- Biblioteca Socket.io para prover a funcionalidade de Web Sockets em todos os browsers, de forma padrão;
- *Engine de templates* Jade;
- MongoDB como banco de dados, sobre a porta 27017, utilizando o driver *Mongoose* para se comunicar com a aplicação.
- *Redis Server* para gerenciamento das sessões, sobre a porta 6379, utilizando o driver *Connect Redis* para se comunicar com a aplicação.

6.2.2 A Aplicação

A primeira etapa do desenvolvimento foi tornar possível o compartilhamento da navegação entre os usuários. Foi definido então que o compartilhamento da navegação estaria focado apenas no conteúdo de uma aba do navegador. Dessa forma, a complexidade de desenvolvimento era menor, e não seria necessário realizar o uso de *plug-ins* ou de extensões dos navegadores.

Para tornar o compartilhamento da navegação, não se pode esquecer dos requisitos não-funcionais, principalmente os listados abaixo:

- Desempenho;
- Funcionalidade *Cross-Browser*;
- Livre de *plug-ins*.

O primeiro requisito, desempenho, com certeza é um dos mais importantes, já que a aplicação estará sendo construída a partir de uma tecnologia que visa criar aplicações que se comuniquem em tempo real, logo não deve existir um atraso muito grande no envio/recebimento das informações. Outro ponto importante foi que a aplicação deverá funcionar em qualquer *browser* atual, independente de ser para dispositivos móveis ou *desktop*, com ou sem suporte a API Web Sockets. Inicialmente isso poderia ser um grande problema, entretanto a biblioteca Socket.io, escrita na linguagem Node.js resolve exatamente esse problema. A primeira vantagem de se utilizar essa biblioteca é que ela esconde as diferenças de implementações de Web Sockets entre os navegadores. Dessa forma, foi possível desenvolver um código único, sem a necessidade de se escrever um código específico para cada browser. Além disso, a biblioteca identifica se o navegador possui ou não suporte à tecnologia de WebSockets, e em caso negativo utiliza outras tecnologias de comunicação (como *Long Polling* ou até mesmo *Flash Sockets*), tudo isso sem o desenvolvedor se preocupar. O último ponto, que já foi discutido anteriormente, é que a aplicação deve ser desenvolvida a partir de tecnologias padronizadas na web, sem o uso de *plug-ins*.

Partindo dos requisitos não funcionais, a primeira etapa do trabalho foi tornar possível o compartilhamento da navegação entre dois usuários. Durante esta etapa, alguns métodos foram considerados para criar essa funcionalidade com sucesso:

- **Envio de um *print-screen* da tela**

O primeiro método consistiu basicamente em se retirar um *print-screen* da tela do visitante, transformar para binário e enviar ao outro usuário em intervalos regulares. Este método foi com certeza o mais simples

encontrado, e ao mesmo tempo bastante fácil de ser implementado. A maior dificuldade seria realizar o *print-screen*, mas que foi resolvido facilmente utilizando-se a biblioteca *HTML2Canvas*, que basicamente realiza uma cópia de toda árvore HTML, e a partir desta cria uma imagem utilizando a API HTML5 Canvas. Com a imagem codificada em base64, foi necessário realizar o envio ao cliente. Mesmo sendo um método livre de *plug-ins* e funcionando em todos os navegadores, o desempenho deixou muito a desejar, já que o tráfego dos dados gerado na rede é muito alto. Além disso, todas as operações utilizadas na biblioteca *HTML2Canvas* consomem muitos recursos computacionais, ocasionando desta forma problemas de desempenho na máquina do cliente. Em testes realizados localmente, foi alcançado a marca de 1 quadro por segundo, número que tende a diminuir mais ainda se inserido em um ambiente de produção. Desta forma, este método foi totalmente desconsiderado visando os requisitos não-funcionais destacados anteriormente.

- **Uso da API Mutation Observers**

Outro método encontrado foi fazer o uso da *API Mutation Observers*, que também faz parte da especificação HTML5. Esta API é bastante interessante, mas ao mesmo tempo muito pouco conhecida. A funcionalidade básica, segundo a especificação, é oferecer aos desenvolvedores uma maneira de tratar alterações no DOM. Dessa forma, quando um elemento da árvore HTML sofrer alguma alteração, é possível tratar essa alteração com um evento, da mesma forma quando tratamos um evento de mouse ou teclado, por exemplo. A ideia seria enviar uma cópia da árvore HTML entre os clientes através de Web Sockets, e toda vez que fosse notado alguma alteração na árvore seria enviado apenas o que foi alterado. Essa solução provavelmente seria a ideal, já que ela iria ter uma performance muito boa, e uma complexidade de implementação não muito grande. O grande problema é que a *API Mutation Observer* ainda é pouco madura, não estando

implementada em nenhum *browser* no momento, o que torna inviável o desenvolvimento da aplicação usando esta técnica. É importante destacar que já existe uma extensão do Google Chrome que torna possível fazer o uso dessa API, entretanto, como foi frisado anteriormente, queremos que essa aplicação seja *cross-browsers* e sem a instalação de qualquer *plug-in* ou extensão.

- **Envio da árvore HTML**

O último método encontrado é uma mescla dos anteriores, mas com uma performance satisfatória, funcional em todos os *browsers* e sem a necessidade instalação de extensões. Ele consiste no envio de toda a árvore HTML em um objeto binário (BLOB), através de WebSockets. Por conseguir satisfazer todas as necessidades desejadas de forma simples e eficiente, este foi o método escolhido. O trabalho de se desenvolver esse método consistiu basicamente em realizar uma cópia do DOM. Felizmente, foi encontrado um código criado por um engenheiro do Google para realizar esta cópia do DOM. A partir da cópia do DOM, foi necessário enviá-lo através de WebSockets e então renderizar o HTML dentro de um *iframe*. Com isso, o funcionamento básico da aplicação já estava encaminhado.

A partir da lógica básica da aplicação criada, a próxima etapa consistiu em organizar a aplicação utilizando o *framework* Express. O *framework* Express é bastante simples, e foi inspirado em outro *Framework*, o Sinatra, escrito para aplicações em Ruby. O uso do Express consistiu principalmente facilitar todas as interações HTTP. A seguinte estrutura foi usada para construir a aplicação:

- App
 - Controllers
 - Models
 - Views
 - Helpers
- Config

- Environments
- Routes.js
- Environment.js
- Log
- Node_modules
- Public
 - Images
 - Javascripts
 - Stylesheets
- Server.js
- Package.json

O *framework* Express permite a criação utilizando-se o modelo MVC, o que nos permite classificar o código em três subconjuntos: *Model*, *View* e *Controller*. Nos tópicos a seguir apresenta-se as implementações:

- **Model**
 - Os modelos representam as instâncias dos esquemas de dados definidos no MongoDB. Como visto no capítulo 6.1.4, os objetos são bastante simples, e são representados apenas pelas classes de modelo *User* e *Domain*.
 - **User**: Representa o usuário;
 - **Domain**: Representa o domínio.
- **View**
 - Views representam as interfaces da aplicação. Para codificação delas, foi utilizado a *engine* de *templates* Jade, criando as *views* com mais facilidade e rapidez.
 - **Includes**
 - **Layout.jade**: Representa o layout básico da aplicação.
 - **Head.jade**: Define as propriedades dentro do elemento head.
 - **Navbar.jade** Barra de navegação principal.

- **404.jade:** Renderiza requisições que não possuem um mapeamento no sistema
 - **500.jade:** Renderiza requisições que o sistema não pode compreender com sucesso.
- **Observer**
 - **Index.jade:** Representa a tela onde o administrador irá visualizar a navegação de um visitante.
- **Tickets**
 - **Index.jade:** Exibe os pedidos de compartilhamento de navegação feitos pelos visitantes.
- **Settings**
 - **Index.jade:** Permite realizar configurações específicas de compartilhamento de um site.
- **Account**
 - **Index.jade:** Permite alterar configurações acerca da conta do usuário
- **Signin**
 - **Index.jade:** Tela para o usuário entrar no sistema.
- **Signup**
 - **Index.jade:** Tela para os usuários se registrarem no sistema.
- **Controller**
 - Os controladores são responsáveis por receber requisições e intermediar as relações entre o banco de dados e a interface do usuário.
 - **Application.js:** Provê um conjunto básico de funcionalidades, principalmente para controle de sessão dos usuários.
 - **Observer.js:** É o principal controlador do sistema. Ele lida com toda a comunicação entre um visitante e um administrador.
 - **Tickets.js:** Trata os pedidos de compartilhamento da navegação feita pelos visitantes e exibe seus pedidos na *view*.

- **Settings.js:** Exibe a tela para o usuário cadastrar / editar algum domínio.
- **Account.js:** Serve a tela para o usuário realizar alterações em sua conta.
- **Signin.js:** Esse controlador lida com as operações de *login* e *logout* no sistema.
- **Signup.js:** Trata requisições para o usuário se registrar no sistema.

A aplicação ainda conta com alguns *helpers* para ajudar no seu funcionamento pleno. A pasta *conf* possui uma série de configurações, como mapeamento de rotas do sistema, configurações de ambiente e banco de dados. O pasta *public* possui arquivos estáticos, como folhas de estilo, imagens e arquivos JavaScript.

6.3. Resultados

Esta seção visa apresentar os resultados obtidos com o desenvolvimento dessa aplicação. Os resultados serão apresentados no formato de capturas de tela, as quais representam as funcionalidades do sistema, descritas previamente através dos diagramas de casos uso. Para tornar a compreensão mais fácil, serão descritos todos os passos possíveis de utilização da aplicação, desde o registro até o compartilhamento da navegação.

O primeiro passo para testar a aplicação consistiu na criação de uma página web qualquer, para que fosse possível adicionar a funcionalidade de compartilhamento da navegação posteriormente.

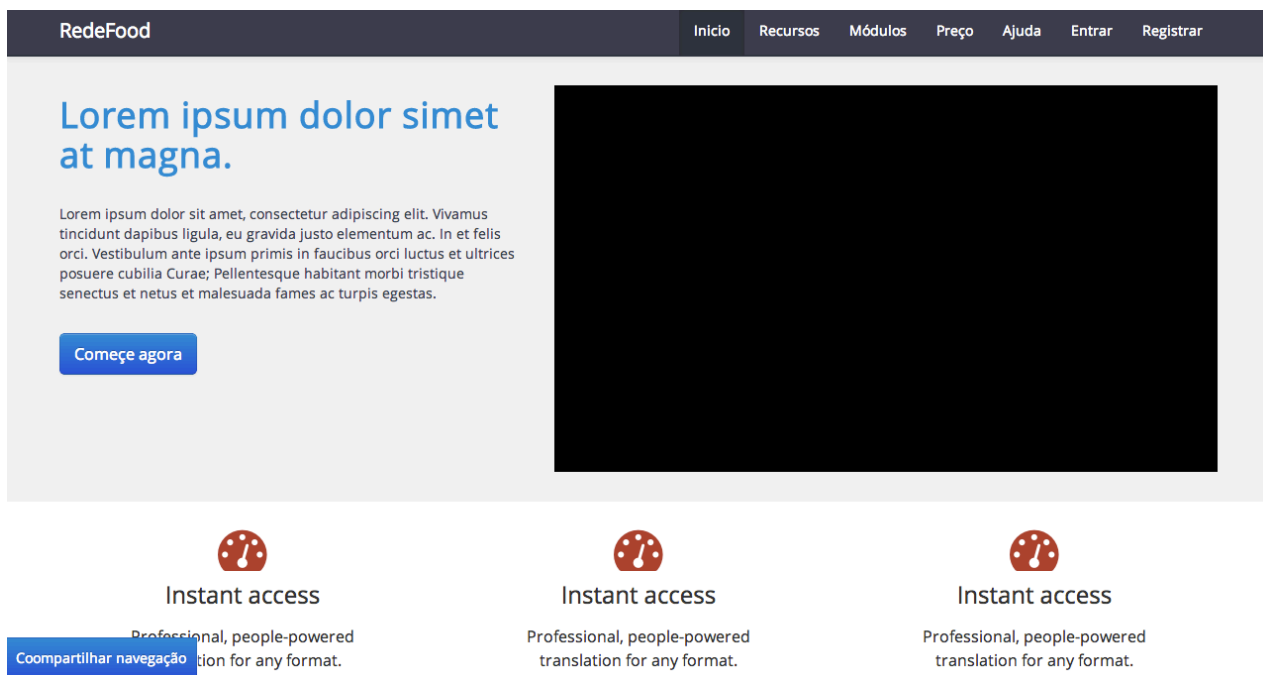
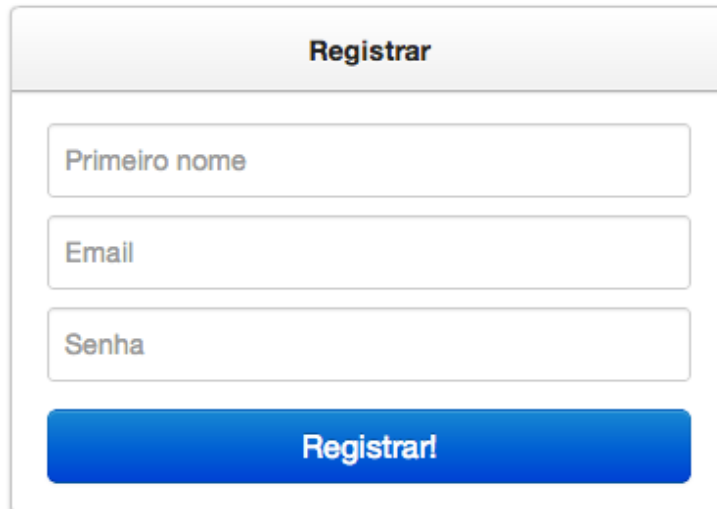


Figura 24: Criação de uma página fictícia

Após o desenvolvimento desta foi inicializado um servidor Apache a fim de publicar a aplicação. Com a aplicação de teste rodando sobre a porta 8000, o próximo passo consistiu no registro do usuário junto ao sistema Observador. O registro é feito em uma página bastante simples, bastando informar nome, email e senha.

Observador



O formulário de registro, intitulado "Registrar", contém três campos de entrada: "Primeiro nome", "Email" e "Senha". Abaixo dos campos, há um botão azul com o texto "Registrar".

Figura 25: Registro

Após o preenchimento dos dados é enviada uma requisição HTTP via método POST ao controlador *Signup*. Este vai verificar se os dados estão corretos e então criar um novo registro no banco representado pelo *schema User*. Com o registro efetuado com sucesso, o usuário será redirecionado a página de *login* para realizar a autenticação no sistema.

Observador



The image shows a login form titled "Entrar" (Login). It features two input fields: one for "Email" and one for a password, represented by six dots. Below the password field is a blue button labeled "Entrar".

Esqueceu sua senha? [Clique aqui para resetar](#)

Figura 26: Entrar

A página de *login* apresenta dois campos(email e senha) para que o usuário faça autenticação através do controlador *Signin*.

Com os dados fornecidos corretamente, uma sessão será criada através do Redis server, um mini-banco avançado para armazenamento de dados no formato chave-valor. Foi optado fazer o uso do Redis server ao invés do controle de sessão em memória para otimizar ao máximo o desempenho da aplicação, consumindo o mínimo de recursos computacionais.

Com uma sessão criada, o administrador terá acesso ao painel administrativo. O layout básico da aplicação consiste em uma barra superior com o logo da aplicação e 2 ações: Conta e o botão de sair. Na barra lateral estão as principais ações do sistema: Tickets e configurações.

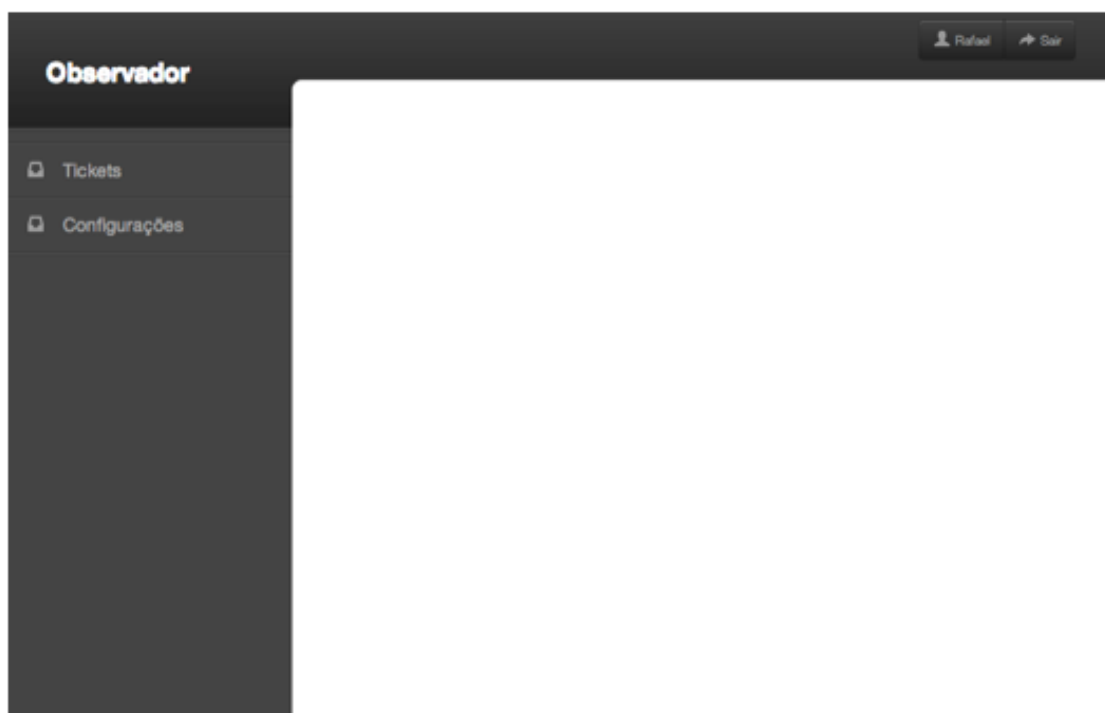


Figura 27: Layout básico do sistema

Entrando na primeira ação citada (Conta), será disponibilizada uma tela para o usuário realizar alterações em sua conta. Nesta tela o usuário poderá realizar alterações pontuais, como seu nome, email e senha.

Conta

Dados pessoais

Primeiro nome

Último nome

Email

Endereço de email

Resetar sua senha

Senha atual

Nova senha

Confirme a nova senha

Figura 28: Conta

Para adicionar a funcionalidade de compartilhamento de navegação na página web que foi construída anteriormente, primeiramente é necessário associar um novo domínio à conta do usuário. Para isto é necessário entrar na tela de configurações e clicar no botão “Novo domínio”.

Configurações

Configurações			
Nome	URL	Ativo	Ações
Teste TCC	localhost:8000	✓	✎ ✕

Figura 29: Configurações

Nesta tela o administrador deverá fornecer informações a respeito do domínio, como nome, URL, posição dos elementos na tela e se quer ativar ou

não o chat e exibir as configurações do cliente. Com o domínio criado, será exibido no fim da página um bloco de código. É este bloco de código que o administrador deverá adicionar em sua página para concluir todo o processo.

Dados básicos

Nome

Dominio

Ativo

Exibir Exibir chat
 Exibir configurações do cliente

Posição sidebar

Posição do botão de compartilhamento

Código

```
<script type="text/javascript">
(function() {
  var po = document.createElement("script"); po.type = "text/javascript"; po.async = true;
  po.src = "https://localhost:3000/observer.js";
  var s = document.getElementsByTagName("script")[0]; s.parentNode.insertBefore(po, s);
})();
</script>
```

Figura 30: Dominio

Com o código inserido antes do fechamento da *tag* <body>, os visitantes que acessarem a página irão visualizar o botão “Compartilhar navegação” no canto esquerdo da sua tela.

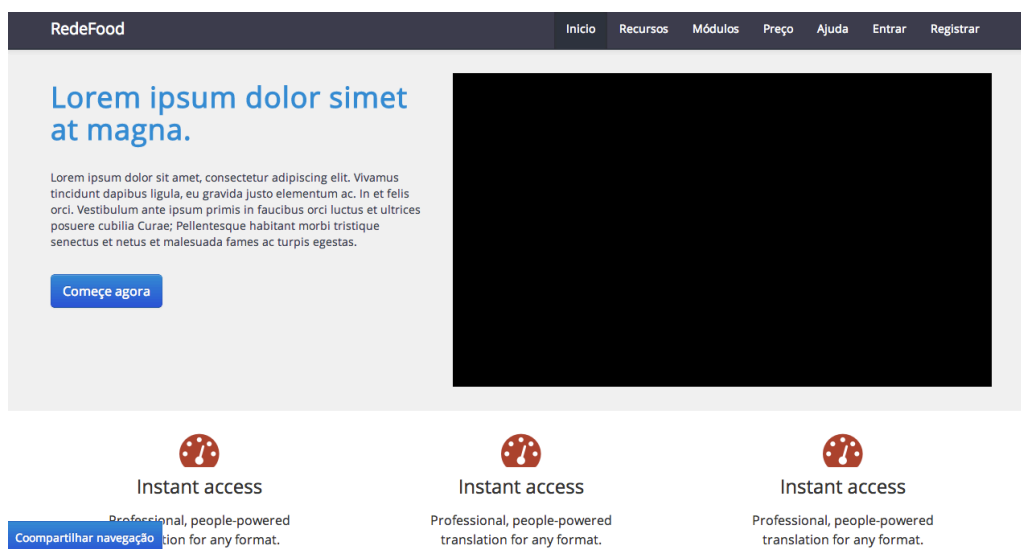


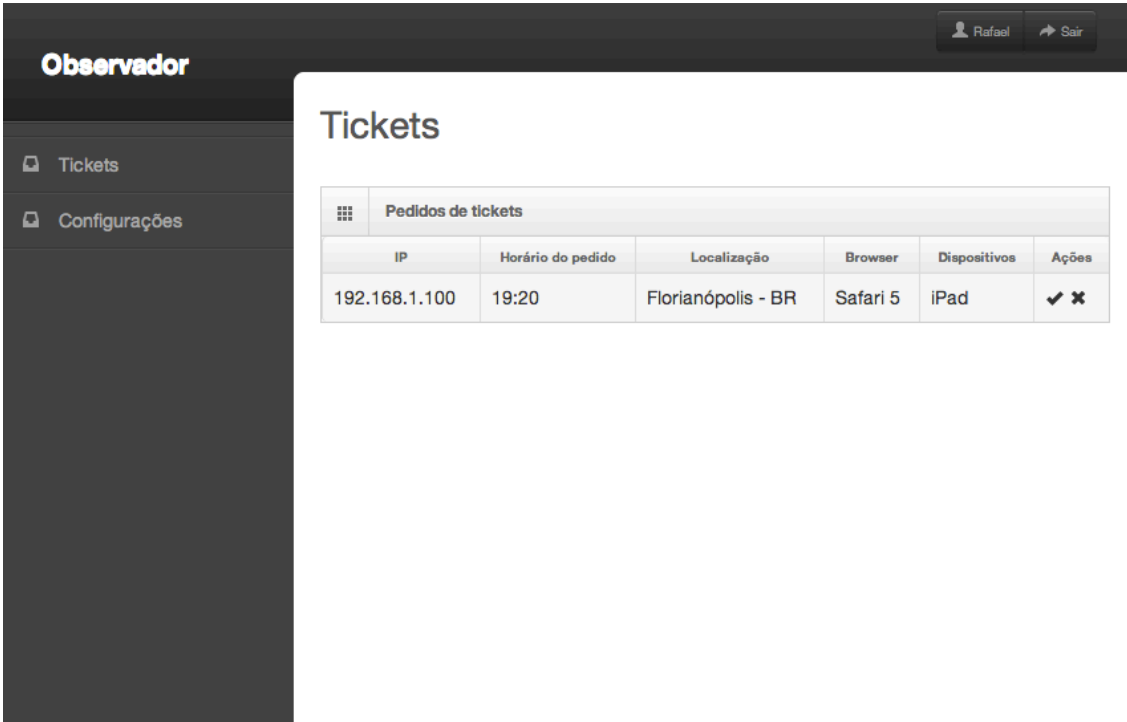
Figura 31: Tela da aplicação fictícia

Este botão é criado de forma dinâmica a partir do código JavaScript inserido na página. A partir do clique neste botão, é iniciado o uso da API de *WebSockets*. Utilizando a API *Socket.io*, descrita anteriormente, o cliente se conecta ao Socket da aplicação Observador:

```
var socket = io.connect('http://localhost:3000');  
socket.emit(newTicket, { data: window.Observer.params });
```

Para conectar basta indicar o endereço do socket criado. Neste caso, toda nossa aplicação está rodando localmente a partir da porta 3000. Após ser estabelecido a conexão é enviado uma mensagem a aplicação Observador. Uma das vantagens de se utilizar a biblioteca *Socket.io* é que ela permite a criação de eventos próprios, tornando a troca de dados entre cliente e servidor muito mais fácil. Caso estivesse sendo usado a implementação padrão de *WebSockets*, a única forma de comunicação entre as aplicações(cliente e servidor) seria a partir do evento *onMessage*. Nesse trecho de código acima, foi criado o evento *newTicket*, enviando como parâmetros detalhes internos de configuração da aplicação Observador(como ID), a fim de notificar o pedido de compartilhamento de navegação para o usuário correto. Com o pedido feito, o

administrador deverá ser notificado. A tela *tickets* exibe todos os pedidos de compartilhamento de navegação feito pelos visitantes.



The screenshot shows a web interface for an 'Observador' (Observer) application. The top navigation bar includes the user name 'Rafael' and a 'Sair' (Logout) button. A sidebar on the left contains 'Tickets' and 'Configurações' (Settings). The main content area is titled 'Tickets' and displays a table of 'Pedidos de tickets' (Ticket Requests).

IP	Horário do pedido	Localização	Browser	Dispositivos	Ações
192.168.1.100	19:20	Florianópolis - BR	Safari 5	iPad	✓ ✕

Figura 32: Tickets

Na imagem acima é possível perceber que há um pedido de compartilhamento de tela aberto, realizado às 19:20, criado pelo visitante com IP 192.168.1.100, geolocalizado em Florianópolis, através do navegador Safari 5 em um Ipad. Na última coluna da tabela ainda é possível identificar que há 2 ações que o administrador pode tomar: aceitar ou rejeitar o pedido. Essa tela não apresenta qualquer interação com banco, realizando a comunicação através de um socket aberto pelo visitante.

A partir de uma resposta positiva, o administrador será redirecionado para outra tela para observar a navegação do visitante. Essa tela apresenta diversas interações através de WebSockets, entre o controlador Observer e o *browser* do visitante. Na imagem abaixo é possível visualizar o resultado final. Na lateral da tela é apresentada uma área onde os usuários poderão se

comunicar. Embaixo, são exibidos detalhes do sistema do cliente. Na direita é exibida a tela que o visitante está compartilhando.

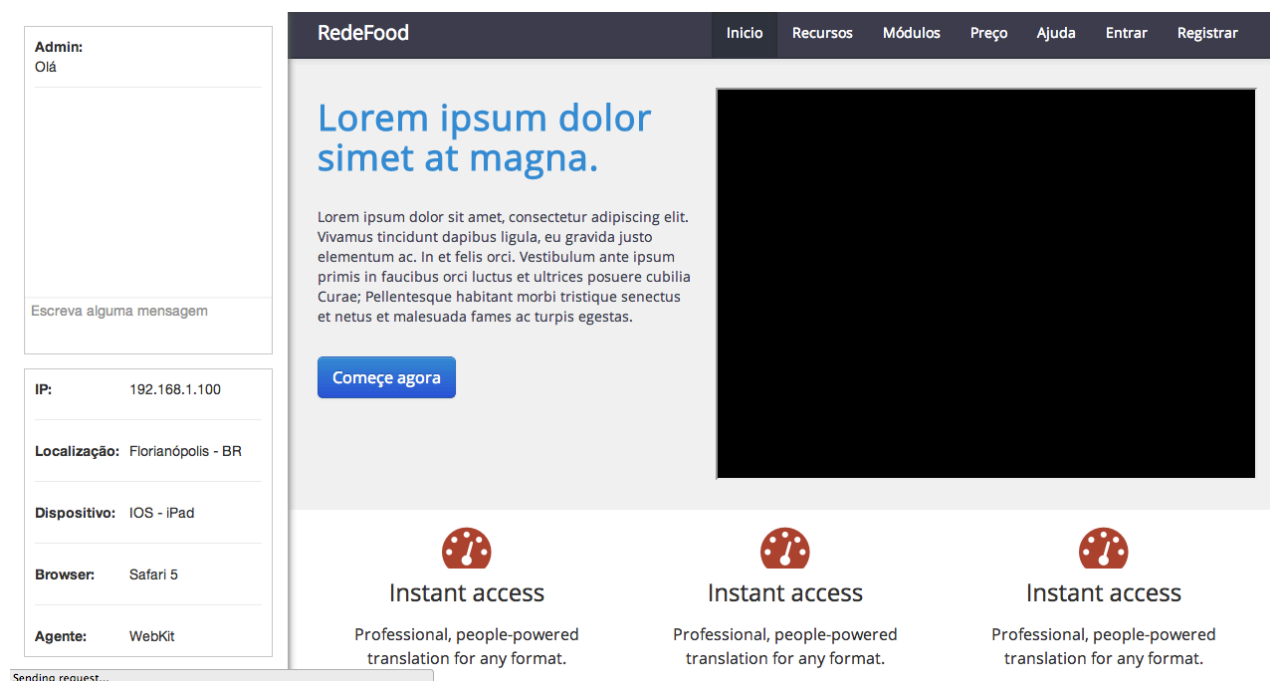


Figura 33: Compartilhamento

Para o desenvolvimento desta aplicação a taxa de atualização da tela é feita a cada 333ms, ou seja, 3 quadros por segundo – número que pode ser aumentado facilmente com algumas otimizações a partir dos dados trafegados entre os usuários. A lógica desta tela segue o método que foi abordado no capítulo 6.2.2. A partir de um socket, o cliente irá realizar uma cópia de toda sua árvore HTML e criar um BLOB a partir desta. Esses dados serão enviados via Web Sockets para o administrador. Com os dados recebidos só será necessário criar um *Iframe* e inserir toda árvore HTML dentro, para que se tenha acesso à tela do cliente.

7. Conclusões

A finalidade desse trabalho foi realizar um estudo sobre a tecnologia de comunicação em tempo real WebSockets, além de desenvolver uma aplicação para observar a navegação de usuários utilizando essa tecnologia. O que mais dificultou a execução da aplicação foi o desempenho, já que o volume de dados trafegado era bastante grande, e não foi possível realizar grandes otimizações com os recursos oferecidos atualmente sem realizar o uso de extensões ou plug-ins. Entretanto, de uma forma geral, todos os objetivos foram concluídos com sucesso.

7.1 Avaliação dos Resultados

Na primeira etapa desse trabalho foi feito um levantamento de informações onde foi possível perceber que a comunicação na web vem mudando muito nos últimos anos, em especial para aplicações sociais, que compartilham um volume muito grande de dados e necessitam que estes dados sejam exibidos em tempo real. Em aplicações como estas, a API de WebSockets se demonstra muito útil, já que consegue consumir menos recursos computacionais e ao mesmo tempo tendo uma latência muito menor se comparado a outros métodos de comunicação.

Foi possível perceber ainda, que, mesmo a especificação da API e o protocolo de WebSockets não estejam 100% concluídos, eles já se encontram em um estágio bastante maduro, sendo utilizado em aplicações importantes, como o Facebook e Twitter. Esse uso deve-se principalmente pela maioria dos navegadores e servidores web já terem implementado todos recursos previstos pela especificação, além de que, a própria W3C, órgão responsável por definir a especificação, já ter dado um parecer que esta especificação provavelmente não mudará mais, mesmo estando em estágio de rascunho.

O sistema desenvolvido, intitulado de observador, visou reproduzir um sistema de compartilhamento de navegação. O principal objetivo deste experimento foi avaliar o funcionamento da API de WebSockets. A aplicação foi desenvolvida com sucesso, e conclui-se que ela já está bastante madura ao

ponto de se utiliza-la em ambiente de produção, embora, como já dito acima, a W3C ainda não recomende fazer seu uso, pois a API ainda encontra-se em estágio de desenvolvimento, podendo ter eventuais alterações em breve.

7.2 Trabalhos Futuros

Com o final do desenvolvimento do trabalho, mesmo tendo atingido os objetivos propostos, é possível perceber que ainda existem melhorias que podem ser feitas para enriquecer o projeto. Dentre elas, pode-se citar:

- Melhorias no desempenho, através da *API Mutation Observers*, que em breve deverá ser implementada nos principais navegadores do mercado;
- Permitir que o administrador possa ter controle da navegação do cliente, através de eventos do mouse, teclado e rolagem (*scroll*);
- Permitir que se crie uma conta administradora com mais agentes – hoje apenas um administrador pode interagir com seus visitantes;
- Registrar a navegação do visitante no banco de dados, para que se possa ter acesso à navegação de um visitante em outro momento.

8. Referências bibliográficas

LUBBERS , Peter. Pro HTML5 Programming. Segunda edição. Cidade desconhecida. Apress, 2011.

Pilgrim , Mark. HTML5 Up and Running. Primeira edição. Cidade desconhecida. O'reilly, 2010.

Hansa , Umar. Start using HTML5 WebSockets today. Disponível em <<http://net.tutsplus.com/tutorials/javascript-ajax/start-using-html5-websockets-today/>>. Acesso em 23 de junho de 2012.

Bersvendsen, Arve. Event Streaming to Web Browsers. Disponível em <<http://my.opera.com/WebApplications/blog/show.dml/438711> />. Acesso em 23 de junho de 2012.

Gilbertson, Scott. Html5 won't be ready until 2022. Disponível em <http://www.webmonkey.com/2008/09/html_5_won_t_be_ready_until_2022dot_yes__2022dot/ />. Acesso em 16 de novembro de 2012.

Clairy, Bob. Browser detection and cross browser support. Disponível em <https://developer.mozilla.org/en-US/docs/Browser_Detection_and_Cross_Browser_Support>. Acesso em 10 de outubro de 2012.

Kesteren, Anne. Fullscreen. Disponível em <<https://dvcs.w3.org/hg/fullscreen/raw-file/tip/Overview.html>>. Acesso em 15 de outubro de 2012

PILGRIM , Mark. Dive into HTML5. Disponível em <<http://diveintohtml5.com.br/semantics.html#divingin>>. Acesso em 23 de junho de 2012.

SYNODINOS, Dio. HTML5 Web Sockets vs. Comet and AJAX. Disponível em <<http://www.infoq.com/news/2008/12/websockets-vs-comet-ajax>>. Acesso em 23 de junho de 2012.

LUBBERS, Petters. How HTML5 WebSockets Interact With Proxy Servers. Disponível em <<http://www.infoq.com/articles/Web-Sockets-Proxy-Servers>>. Acesso em 23 de junho de 2012.

GRECO, Frank. HTML5 WebSockets: A quantum Leap in Scalability for the Web. <Disponível em <http://www.websocket.org/quantum.html>>. Acesso em 23 de junho de 2012.

LITTE, Mark. WebSockets versus REST. Disponível em <<http://www.infoq.com/news/2012/02/websockets-rest>>. Acesso em 23 de junho de 2012.

EVANS, Nathan. WebSockets versus REST Fight. Disponível em <<http://nbevans.wordpress.com/2011/12/16/websockets-versus-rest-fight>>. Acesso em 23 de junho de 2012.

DRYSDALE, Brad. WebSockets: The web Communication Revolution. Disponível em <<http://www.infoq.com/presentations/WebSockets-The-Web-Communication-Revolution>>. Acesso em 23 de junho de 2012.

Apêndice 1– Artigo

Desenvolvimento de aplicações com Websockets

Rafael Michels Motta¹

¹UFSC – Universidade Federal de Santa Catarina
INE – Departamento de Informática e Estatística
Florianópolis(SC), Brasil

rafaelmotta021@inf.ufsc.br

Abstract. *The web is living a new era, where everything is done in real time. The expectations of how fast the Internet must deliver information has changed - minute delays are unacceptable. Large companies, such as Twitter, Facebook and Google have captured this need quickly, and currently deliver their data in real time, whether a simple chat, or even action notifications in social networks. In this scenario, this work explore concepts and definitions of the technologies involved to create real-time Web applications, focusing on API, and WebSockets protocol. Finally, the aim is also to implement a service using this feature.*

Key words: *Web communication, HTML5, WebSockets.*

Resumo. *A web está vivendo uma nova era, onde tudo está sendo feito em tempo real. As expectativas de quão rápido a internet deve entregar as informações mudou – atrasos de minutos são inaceitáveis. Grandes empresas, como Twitter, Google e Facebook captaram essa necessidade rapidamente, e atualmente já entregam seus dados em tempo real, seja em um simples bate-papo, ou mesmo em notificações de ações em redes sociais. Nesse cenário, este trabalho visa explorar os conceitos e definições das tecnologias envolvidas para se criar aplicações Web em tempo real, com foco na API e protocolo de WebSockets. Por fim, pretende-se ainda implementar um serviço utilizando esse recurso.*

Palavras chaves: *Comunicação na web, HTML5, WebSockets.*

1. Introdução

Por que a Web em tempo real é importante? Vivemos em um mundo em que tudo é feito em tempo real, então, é natural que a web esteja se movendo nesta direção. As expectativas de quão rápido a internet deve entregar as informações mudou – atrasos de minutos são inaceitáveis. Grandes empresas, como *Twitter*, *Google* e *Facebook* captaram essa necessidade rapidamente, e atualmente já entregam seus dados em tempo real, seja em um simples bate-papo, ou mesmo em notificações de ações em redes sociais.

De fato, a web está vivendo uma nova era. Criada originalmente para exibição, organização e compartilhamento de documentos hipertexto, ela passou por muitas fases até chegar no estágio que se encontra hoje. Do simples fornecimento de páginas estáticas, escritas em HTML, surgiu espaço para a criação de linguagens e outros

mecanismos que puderam oferecer esse conteúdo de forma dinâmica. Esse modelo, que hoje é tradicionalmente conhecido como HTTP, é baseado em requisições e respostas: um cliente faz um pedido de uma página web, o servidor entrega o conteúdo, e nada mais acontece até que o cliente faça outra requisição. Surge depois o AJAX, técnica que permitiu que a web torna-se muito mais dinâmica, permitindo que a requisição fosse feita de forma assíncrona, sem a necessidade de recarregar da página atual por completo. No entanto, ainda estávamos presos naquele antigo modelo HTTP, não possuindo um padrão para enviar dados no sentido servidor - cliente.

Uma gama de soluções surgiu para resolver esse problema. A mais básica foi a técnica chamada *Polling*, que consistia em fazer requisições em tempos regulares no servidor web utilizando AJAX, a fim de obter alguma nova informação. De fato ela deu a percepção aos usuários de que a aplicação era em tempo real, mas outros problemas acabaram surgindo, geralmente ligados a performance e a sobrecarga dos servidores web, já que utilizavam o tradicional modelo HTTP, que não tinha sido projetado para esse tipo de aplicação. A mais notável foi a *Comet*, um modelo de aplicação baseado no modelo HTTP, que permitiu que, de fato, dados fossem enviados sem uma requisição explícita do servidor.

Plug-ins de terceiros também foram criados, como *Flash Sockets*. Esses permitiram que, de fato, fosse utilizada a técnica PUSH, isso é, do servidor web enviar dados ao cliente. A ressalva era que esses *plug-ins* tinham que ser instalados por cada usuário em sua máquina, o que não era garantido, principalmente para usuários leigos e em grandes corporações.

Tudo mudou em 2008, quando a W3C anunciou um novo padrão da web, o HTML5. As novidades dessa nova especificação iam desde novas *tags* – que traziam melhorias na semântica e acessibilidade – até um conjunto de APIs especificadas em *Javascript*, mudando totalmente o HTML que estávamos acostumados a lidar. Uma das seções mais interessantes desse novo padrão foi a parte de comunicação, beneficiando principalmente as aplicações em tempo real. A API de *WebSockets* permite agora a comunicação bidirecional por canais *full-duplex* através da porta 80 do protocolo TCP, significando que agora pode ser enviados dados no sentido servidor – cliente e ainda sem os problemas ligados a performance, que tanto afetavam as técnicas que usadas em aplicações em tempo real.

Nesse cenário, serão explorados os conceitos e definições das tecnologias envolvidas para se criar aplicações Web em tempo real, com foco na API e protocolo de *WebSockets*.

2. Desenvolvimento de aplicações web

A *World Wide Web* começou com um programador que teve uma nova ideia de software no grupo de controle e aquisição de dados da Organização Européia para pesquisa nuclear – CERN, localizado em Genebra. Segundo Berners-Lee (1989), o objetivo original do desenvolvimento da web foi tornar mais fácil o compartilhamento de documentos de pesquisas. Tim Berners-Lee propôs que a gerência da CERN adotasse um sistema de informação distribuído baseado em hipertexto, a fim de facilitar o compartilhamento de conhecimento dentro da instituição. O projeto, que inicialmente foi chamado de *Mesh*, convenceu rapidamente todos os gerentes devido a grande perda de informações que acontecia diariamente no CERN. Após um ano de trabalho incessante neste projeto, Berners-Lee finalizou seu projeto e o batizou com um novo

nome: *World Wide Web*. Nesse projeto, ele implementou uma série de ferramentas que são usadas ainda hoje:

- O Identificador uniforme de recursos – URI, uma sintaxe para identificar cada documento com um endereço único na web.
- O protocolo de transferência de hipertexto – HTTP, usado para realizar a comunicação entre os dispositivos na web.
- A linguagem de marcação de hipertexto – HTML, para representar documentos na internet.
- O primeiro servidor web, o qual ainda está rodando.
- O primeiro navegador web, o qual Berners-Lee nomeou como *WorldWideWeb*, mas depois acabou mudando para Nexus para não causar confusão com a Web em sim.
- O primeiro editor de HTML, que estava implementado dentro do próprio navegador que ele implementou.

Em agosto de 1991, na primeira página web da história, Berners-Lee escreveu:

“A *WorldWideWeb* é uma grande hipermídia que tem por objetivo dar acesso universal a um grande universo de documentos”.

Daquele momento em diante, a web cresceu em um nível exponencial. Com cinco anos o número de usuários na web já ultrapassava a marca de 40 milhões de usuários (Infonetics Research, 2005). Em um certo momento, o número de usuários dobrava a cada dois meses. O universo de documentos de Berners-Lee era realidade, e estava cada dia se expandido mais e mais.

Cada vez a web evoluiu mais, e as páginas web deixaram de serem apenas páginas estáticas para se comportarem como aplicações. Passou então a ser uma enorme plataforma, com o desenvolvimento de novas linguagens e tecnologias. A demanda pelo desenvolvimento de aplicações web hoje só cresce, e por consequência disso gera uma alta demanda no desenvolvimento e maturação de tecnologias para apoiarem este processo.

3. Aplicações em tempo real na web

Em meados de 1990 a *World Wide Web* estava crescendo muito rapidamente, e estava no caminho para se tornar o meio de distribuição de informação mais dominante (Newswire, 1995). Com o avanço dos navegadores, a web deixou de ser um simples lugar para exibir documentos estáticos, para se comportar como uma plataforma de aplicativos, atingindo mais usuários que qualquer outra plataforma já criada. Inicialmente ela não havia sido projetada para fornecer uma interação rica com o usuário. Entretanto com o passar dos tempos, alguns engenheiros já conseguiam oferecer aplicações com uma melhor experiência de uso. Com o uso dos *iframes* – elemento HTML que exibe o conteúdo de uma página específica dentro da página atual - criava-se a sensação que a comunicação com um servidor era feita de forma assíncrona, já que não se bloqueava toda a aplicação enquanto era realizado a requisição e o processamento no servidor web.

Mesmo com estas soluções bastante rudimentares, a web começou a ficar cada vez mais interativa. Com o surgimento do AJAX - técnica que tornou possível realizar um pedido ao servidor de forma assíncrona, utilizando-se Javascript - a web sofreu uma guinada e passou a ser extremamente dinâmica. A sigla RIA – *Rich Internet Applications* – começou a se popularizar, e a partir daquele momento as aplicações web começaram a compartilhar uma série de características com as aplicações desenvolvidas no desktop. Todo o processamento da interface passa agora a ser tratado no próprio navegador, permitindo um número muito menor de requisições ao servidor (Fratelli, 2010).

Em 2004 uma segunda geração de serviços e comunidades na web começou a surgir, tendo como conceito a web como uma plataforma. Essa geração de aplicações foi apelidada de Web 2.0, pela empresa norte-americana *O'Reilly*. Serviços como redes sociais, páginas de distribuição de vídeos, wikis e blogs começaram a se popularizar, e possuíam como característica comum a participação efetiva dos usuários no tráfego de informações(O'Reilly, 2005).

Com uma massa de dados cada vez maior e as aplicações web cada vez mais complexas, devido ao surgimento e aprimoramento de tecnologias, uma demanda pela entrega de informações de forma instantânea começou a surgir, principalmente em aplicações da web 2.0. Implementado inicialmente pelo *Twitter* para notificar quando um novo *tweet* era efetuado por algum usuário da rede, esse tipo de notificação ficou muito comum na maioria das redes sociais. Dessa forma, as aplicações em tempo real começaram a se popularizar, principalmente devido a esse tipo de aplicação.

As aplicações web em tempo real se referem a uma série de tecnologias e práticas que permitem aos usuários receberem notificações assim que os dados são publicados por seus autores, ao invés de realizar requisições em tempos periódicos ao servidor para verificar se há algum dado novo. Esse classe de aplicação também é conhecida como aplicações de tecnologia *push*, já que se baseiam em uma comunicação onde a requisição é iniciada pelo servidor web, ao invés do tradicional modelo cliente-servidor, que era baseado na requisição de um cliente.

Durante o passar dos anos uma série de tecnologias foram criadas para prover esse tipo de aplicações, que envolveu a técnica *polling* e outras tecnologias do tipo *push*, sendo a mais notável o modelo Comet(Paul, 2007).

4. HTML5

No início de 2008, a W3C anunciou a primeira especificação do HTML5. Essa nova especificação inclui grandes alterações, como:

- Novas *tags* para melhoria da semântica;
- Controle embutido de conteúdo multimídia;
- Melhoria na depuração de erros;
- API's *Javascript*.

Esse último, em especial, contém realmente as grandes novidades da nova especificação, e dentro dela há uma seção de comunicação que envolve a API de *WebSockets*, criada para lidar com aplicações em tempo real.

5. WebSockets

Historicamente, desenvolver aplicações web que precisem de comunicação bidirecional entre cliente e servidor, era significado de um abuso do protocolo HTTP, devido às incessantes requisições ao servidor web para verificar se havia algum dado novo, além da criação de duas conexões TCP, uma para enviar mensagens ao cliente e outra para receber mensagens (Greco, 2012).

Uma simples solução para esse problema foi usar uma única conexão TCP para trafegar entre ambas direções. É isso basicamente que a especificação WebSockets define. Ela não é uma simples melhora na comunicação do tradicional protocolo HTTP. Ele representa uma revolução na comunicação da web, especialmente para aplicações em tempo real e aplicações baseadas em eventos *server-side*, também conhecidas como aplicações de classe *push*.

A especificação dessa API descreve um novo protocolo para trafegar os dados de maneira bidirecional, entre cliente e servidor, operando em um socket único através da porta 80 (no caso do protocolo HTTP) ou da porta 443 (protocolo HTTPS). Além disso, foi especificada uma interface JavaScript para de fato usar esse protocolo e ouvir os eventos disparados pelo servidor. Com esse novo protocolo, problemas ocorridos nas técnicas de Comet e *Polling*, como o grande tráfego de rede e a alta latência foram totalmente resolvidos. E com a interface JavaScript, agora é possível criar aplicações desse tipo muito mais facilmente, reconhecendo também uma série de eventos padronizados pelo servidor web.

O protocolo WebSocket foi projetado para substituir todas as outras tecnologias existentes de comunicação bidirecional na web que usam o protocolo HTTP como transporte, principalmente para beneficiar a infraestrutura existente. Hoje o protocolo WebSocket ainda é dependente do HTTP, já que para usar o protocolo WebSocket deve-se obrigatoriamente realizar um *handshake* entre cliente e servidor utilizando o protocolo HTTP. Futuramente isso pode mudar, sendo esse *handshake* feito através de uma porta, sem precisar depender mais do tradicional protocolo HTTP.

O uso desse recurso hoje é grande, e podemos ver em uma série de aplicações que usamos no dia-a-dia. Desde o chat no Gmail ou Facebook, notificações em sua rede social preferida, edição em grupo do Google Docs, e até jogos online já fazem o uso da API de WebSockets.

6. Projeto

Este trabalho visa desenvolver um sistema de navegação compartilhada, onde um usuário irá compartilhar o conteúdo de uma janela de seu *browser* com outro usuário. A solução será livre de *plug-ins* e registros por parte do visitante, sendo útil principalmente para os administradores de sistema auxiliarem os visitantes no uso do sistema, bem como para reportar *bugs* de forma fácil e rápida.

O grande diferencial do sistema é que ele poderá ser integrado a qualquer aplicação publicada na *web*, independentemente da linguagem utilizada. Um exemplo claro desse tipo de aplicação é o *Google Analytics* – onde basta inserir um código JavaScript em sua página e você terá acesso a informações relevantes sobre os acessos em sua página.

Seguindo estes moldes, será desenvolvida uma aplicação à parte, chamada Observador. Nesta aplicação, administradores de páginas ou sistemas publicados na *web*, com o interesse de observar a navegação de seus visitantes, irão efetuar um

registro e seguir alguns procedimentos simples para adicionar esta funcionalidade. Sem a necessidade de realizar qualquer alteração na página ou sistema atual, o Observador não irá trazer qualquer impactos quanto a performance, já que ele estará rodando separadamente em uma infraestrutura dedicada.

Com os procedimentos realizados na página, quando um visitante acessar a página em questão, será criado de forma dinâmica um botão oferecendo ao usuário a capacidade de compartilhar sua navegação com o administrador.

Para que a navegação seja compartilhada será necessário que o visitante clique por vontade própria no botão “Compartilhar navegação”, bem como de ter um administrador online no sistema Observador. Além disso, este deverá responder positivamente para que a navegação compartilhada seja iniciada. Quando iniciada, será oferecido um *chat* para que os usuários se comuniquem. O administrador ainda terá acesso a informações detalhadas a respeito do sistema utilizado pelo cliente, tais como: plataforma, navegador, agente de usuário, IP e geolocalização, a fim de facilitar, por exemplo, um serviço de suporte. Por questões de segurança e privacidade, o compartilhamento poderá ser cancelado por qualquer um dos usuários a qualquer momento.

7. Conclusões

A finalidade desse trabalho foi realizar um estudo sobre a tecnologia de comunicação em tempo real WebSockets, além de desenvolver uma aplicação para observar a navegação de usuários utilizando essa tecnologia. O que mais dificultou a execução da aplicação foi o desempenho, já que o volume de dados trafegado era bastante grande, e não foi possível realizar grandes otimizações com os recursos oferecidos atualmente sem realizar o uso de extensões ou plug-ins. Entretanto, de uma forma geral, todos os objetivos foram concluídos com sucesso.

8. Referências

LUBBERS , Peter. Pro HTML5 Programming. Segunda edição. Cidade desconhecida. Apress, 2011.

Pilgrim , Mark. HTML5 Up and Running. Primeira edição. Cidade desconhecida. O'reilly, 2010.

Hansa , Umar. Start using HTML5 WebSockets today. Disponível em <<http://net.tutsplus.com/tutorials/javascript-ajax/start-using-html5-websockets-today/>>. Acesso em 23 de junho de 2012.

Bersvendsen, Arve. Event Streaming to Web Browsers. Disponível em <<http://my.opera.com/WebApplications/blog/show.dml/438711> />. Acesso em 23 de junho de 2012.

Gilbertson, Scott. Html5 won't be ready until 2022. Disponível em <http://www.webmonkey.com/2008/09/html_5_won_t_be_ready_until_2022dot_yes__2022dot/>. Acesso em 16 de novembro de 2012.

Clairy, Bob. Browser detection and cross browser support. Disponível em <https://developer.mozilla.org/en-US/docs/Browser_Detection_and_Cross_Browser_Support>. Acesso em 10 de outubro de 2012.

Kesteren, Anne. Fullscreen. Disponível em <<https://dvcs.w3.org/hg/fullscreen/raw-file/tip/Overview.html>>. Acesso em 15 de outubro de 2012

PILGRIM, Mark. Dive into HTML5. Disponível em <<http://diveintohtml5.com.br/semantics.html#divingin>>. Acesso em 23 de junho de 2012.

SYNODINOS, Dio. HTML5 Web Sockets vs. Comet and AJAX. Disponível em <<http://www.infoq.com/news/2008/12/websockets-vs-comet-ajax>>. Acesso em 23 de junho de 2012.

LUBBERS, Petters. How HTML5 WebSockets Interact With Proxy Servers. Disponível em <<http://www.infoq.com/articles/Web-Sockets-Proxy-Servers>>. Acesso em 23 de junho de 2012.

GRECO, Frank. HTML5 WebSockets: A quantum Leap in Scalability for the Web. <Disponível em <http://www.websocket.org/quantum.html>>. Acesso em 23 de junho de 2012.

LITTE, Mark. WebSockets versus REST. Disponível em <<http://www.infoq.com/news/2012/02/websockets-rest>>. Acesso em 23 de junho de 2012.

EVANS, Nathan. WebSockets versus REST Fight. Disponível em <<http://nbevans.wordpress.com/2011/12/16/websockets-versus-rest-fight>>. Acesso em 23 de junho de 2012.

DRYSDALE, Brad. WebSockets: The web Communication Revolution. Disponível em <<http://www.infoq.com/presentations/WebSockets-The-Web-Communication-Revolution>>. Acesso em 23 de junho de 2012.

Apêndice 2 – Código fonte

Controllers

Account_Controller.coffee

```
load 'application'
```

```
before use 'isUserLogged'
```

```
action 'index', ->
```

```
  render title: t('account.title')
```

```
action 'create', ->
```

```
  User.findByIdAndUpdate(request.session.user._id, { firstName:
request.body.firstName, lastName: request.body.lastName, email:
request.body.email }, (err, user) ->
```

```
    if not err
```

```
      request.session.user = user
```

```
      flash 'info', 'Usuário atualizado com sucesso'
```

```
      return redirect path_to.account()
```

```
    )
```

```
if request.body.currentPassword
```

```
  if request.body.currentPassword isnt request.session.user.password
```

```
    flash 'error', 'Senha atual não corresponde a digitada'
```

```
    return redirect path_to.account()
```

```
  else
```

```
    if request.body.newPassword isnt request.body.password
```

```
      flash 'error', 'Senhas não conferem'
```

```
      return redirect path_to.account()
```

```
    else
```

```
      if request.body.password.length < 2
```

```
        flash 'error', 'Senha deve ter pelo menos 2 caracteres'
```

```
return redirect path_to.account()
```

Application_controller.coffee

```
before 'protect from forgery', ->
```

```
  protectFromForgery '09fcdda644a6978e42d5698679188aa9d67c76d5'
```

```
publish 'isUserLogged', ->
```

```
  if not request.session.user
```

```
    flash 'redirect', request.originalUrl
```

```
    redirect path_to.signin()
```

```
  else
```

```
    next()
```

```
dashboard_controller.coffee
```

```
load 'application'
```

```
before use 'isUserLogged'
```

```
action 'index', ->
```

```
  render title: t('dashboard.title')
```

```
observer_controller.coffee
```

```
load 'application'
```

```
before use 'isUserLogged'
```

```
action 'index', ->
```

```
app.io.sockets.on "connection", (socket) ->
```

```
  socket.on "screen", (data) ->
```

```
    socket.emit "updateScreen", {data: data.base64}
```

```
layout(false)
```

```
render title: t('observer.title')
```

signin_controller.coffee

```
load 'application'
```

```
action 'index', ->
```

```
  if request.session.user
    return redirect path_to.dashboard()
```

```
  layout false
  render title: t('signin.title')
```

```
action 'logout', ->
```

```
  request.session.destroy()
  redirect path_to.signin()
```

```
action 'create', ->
```

```
  User.findOne({email: request.body.email, password:
request.body.password}).populate('companies').exec (err, user) ->
```

```
    if user
      request.session.user = user
      redirect path_to.dashboard()
    else
      flash 'error', 'Usuário ou senha incorretos'
      redirect path_to.signin()
```

Views

Account/index.ejs

```
<article id="account-page">
  <div class="row-fluid">
    <div class="span9">
      <% form_for(account, {action: path_to.account(), method: 'POST',
class:'form-horizontal'}, function (form) { %>
        <% flash = request.flash('error').pop(); if (flash) { %>
          <div class="alert alert-error">
            <a class="close" data-dismiss="alert">x</a>
```

```

    <%- flash %>
</div>
<% }; %>
<% info = request.flash('info').pop(); if (info) { %>
<div class="alert alert-info">
    <a class="close" data-dismiss="alert">x</a>
    <%- info %>
</div>
<% }; %>
<fieldset>
    <legend>
        <%= t('account.personalData') %>
    </legend>
    <div class="control-group">
        <label class="control-label"> <%= t('account.firstName')
%></label>
        <div class="controls">
            <input type="text" class="input-large" name="firstName"
placeholder=" <%= t('account.firstName') %>" value="<%=
request.session.user.firstName %>">
        </div>
    </div>
    <div class="control-group">
        <label class="control-label"><%= t('account.lastName') %></label>
        <div class="controls">
            <input type="text" class="input-large" name="lastName"
placeholder="<%= t('account.lastName') %>" value="<%=
request.session.user.lastName %>">
        </div>
    </div>
</fieldset>
<fieldset id="email">
    <legend>
        <%= t('account.email') %>

```

```

</legend>
<div class="control-group">
  <label class="control-label"><%= t('account.emailAddress')
%></label>
  <div class="controls">
    <input type="email" class="input-large" name="email"
placeholder="<%= t('account.emailAddress') %>" value="<%=
request.session.user.email %>">
  </div>
</div>
</fieldset>
<fieldset>
  <legend>
    <%= t('account.resetPassword') %>
  </legend>
  <div class="control-group password-check">
    <label for="password_old_1" class="control-label"><%=
t('account.currentPassword') %></label>
    <div class="controls">
      <input class="input-large" name="currentPassword" size="16"
type="password" id="current-password" placeholder="<%=
t('account.currentPassword') %>">
    </div>
  </div>
  <div class="control-group">
    <label for="password_new_1" class="control-label"><%=
t('account.newPassword') %></label>
    <div class="controls">
      <input class="input-large" name="newPassword" size="16"
type="password" id="new-password-1" placeholder="<%=
t('account.newPassword') %>">
    </div>
  </div>
  <div class="control-group">

```

```

        <label for="password_new_2" class="control-label"><%=
t('account.confirmPassword') %></label>
        <div class="controls">
            <input class="input-large" name="password" size="16"
type="password" id="new-password-2" placeholder="<%=
t('account.confirmPassword') %>">
        </div>
    </div>
</fieldset>
<fieldset id="support">
    <div class="form-actions">
        <button class="btn btn-primary" type="submit">
            <%= t('account.save') %>
        </button>
    </div>
</fieldset>
<% });%>
</div>
</div>
</article>

```

Dashboard/index.ejs

```

<div id="dashboard-page">
    <div class="row-fluid">
        <div class="span12 center" style="text-align: center;">
            <ul class="stat-boxes">
                <li>
                    <div class="right">
                        <strong>36094</strong>
                        Visitas
                    </div>
                </li>
                <li>
                    <div class="left peity_bar_neutral">

```

```

        <span>20,15,18,14,10,9,9,9</span>0%
    </div>
    <div class="right">
        <strong>1433</strong>
        Usuários ativos
    </div>
</li>
<li>
    <div class="left peity_bar_bad">
        <span>3,5,9,7,12,20,10</span>-50%
    </div>
    <div class="right">
        <strong>8650</strong>
        Orders
    </div>
</li>
<li>
    <div class="left peity_line_good">
        <span>12,6,9,23,14,10,17</span>+70%
    </div>
    <div class="right">
        <strong>8650</strong>
        Orders
    </div>
</li>
</ul>
</div>
<div class="row-fluid">
    <div class="span12">
        <div class="widget-box">
            <div class="widget-title">
                <span class="icon"><i class="icon-signal"></i></span><h5>Site
Statistics</h5>

```



```

    <div class="buttons">
      <a href="#" class="btn btn-mini"><i class="icon-refresh"></i>
Update stats</a>
    </div>
  </div>
  <div class="widget-content">
    <div class="row-fluid">
      <div class="span4">
        <ul class="site-stats">
          <li>
            <i class="icon-
user"></i><strong>1433</strong><small>Total Users</small>
          </li>
          <li>
            <i class="icon-arrow-
right"></i><strong>16</strong><small>New Users (last week)</small>
          </li>
          <li class="divider"></li>
          <li>
            <i class="icon-shopping-
cart"></i><strong>259</strong><small>Total Shop Items</small>
          </li>
          <li>
            <i class="icon-
tag"></i><strong>8650</strong><small>Total Orders</small>
          </li>
          <li>
            <i class="icon-
repeat"></i><strong>29</strong><small>Pending Orders</small>
          </li>
        </ul>
      </div>
      <div class="span8">
        <div class="chart"></div>

```

```
        </div>
      </div>
    </div>
  </div>
</div>
</div>
</div>
</div>
</div>
</div>
</div>
```

Application/index.ejs

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title><%= title %></title>
    <%= stylesheet_link_tag('bootstrap', 'bootstrap-responsive', 'unicorn',
'unicorn.grey', 'style') %>
    <%= csrf_meta_tag() %>
  </head>
  <body>
    <div id="header">
      <h1><a href="./dashboard.html">Unicorn Admin</a></h1>
    </div>
    <div id="user-nav" class="navbar navbar-inverse">
      <ul class="nav btn-group">
        <li class="btn btn-inverse" >
          <a title="" href="/account"><i class="icon icon-user"></i> <span
class="text"><%= request.session.user.firstName %></span></a>
        </li>
        <li class="btn btn-inverse">
          <a title="" href="/logout"><i class="icon icon-share-alt"></i> <span
class="text"><%= t('logout') %></span></a>
        </li>
      </ul>
    </div>
    <div id="sidebar">
      <ul>
```

```

        <li>
            <a href="/dashboard"><i class="icon icon-home"></i><span><%=
t('dashboard.title') %></span></a>
        </li>
        <li>
            <a href="/tickets"><i class="icon icon-inbox"></i> <span><%=
t('tickets.title') %></span></a>
        </li>
        <li>
            <a href="/settings"><i class="icon icon-inbox"></i> <span><%=
t('settings.title') %></span></a>
        </li>
    </ul>
</div>
<div id="content">
    <div id="content-header">
        <h1><%= t(request.res.info.controller + ".title") %></h1>
    </div>
    <div class="container-fluid">
        <%- body %>
    </div>
</div>
</body>
</html>

```

Observer/index.ejs

```

<!DOCTYPE html>
<html lang="en">
    <head>
        <title><%= title %></title>
        <%- stylesheet_link_tag('bootstrap', 'bootstrap-responsive', 'style') %>
        <script src="http://localhost:8888/socket.io/socket.io.js"></script>

```

```

    <%-
javascript_include_tag('http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.
min.js', 'bootstrap', 'rails', 'application') %>
    <%- csrf_meta_tag() %>
</head>
<body>
  <article id="observer-page">
    <div></div>
    <aside>
      <dl class="dl-horizontal">
        <dt>Description lists</dt>
        <dd>A description list is perfect for defining terms.</dd>
        <dt>Euismod</dt>
        <dd>Vestibulum id ligula porta felis euismod semper eget lacinia
odio sem nec elit.</dd>
        <dd>Donec id elit non mi porta gravida at eget metus.</dd>
        <dt>Malesuada porta</dt>
        <dd>Etiam porta sem malesuada magna mollis euismod.</dd>
        <dt>Felis euismod semper eget lacinia</dt>
        <dd>Fusce dapibus, tellus ac cursus commodo, tortor mauris
condimentum nibh, ut fermentum massa justo sit amet risus.</dd>
      </dl>
    </aside>
  </article>
</body>
</html>
Settings/index.ejs
<article id="settings-page">
  <div class="row-fluid">
    <div class="span12">
      <div class="widget-box">
        <div class="widget-title">
          <span class="icon"> <i class="icon-th"></i> </span>
          <h5>Configurações </h5>

```

```

</div>
<div class="widget-content nopadding">
  <table class="table table-bordered table-striped">
    <thead>
      <tr>
        <th>Nome</th>
        <th>URL</th>
        <th>Ativo</th>
        <th>Ações</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td> Teste TCC </td>
        <td> localhost:8000 </td>
        <td><i class="icon-ok"></i></td>
        <td><a href="javascript:void(0);" class="icon-pencil"
rel="tooltip" data-original-title="Aceitar"></a><a href="javascript:void(0);"
class="icon-remove" rel="tooltip" data-original-title="Aceitar"></a></td>
      </tr>
    </tbody>
  </table>
</div>
</div>
</div>
</div>
</article>

```

Settings/show.ejs

```

<article id="settings-page">
  <div class="row-fluid">
    <div class="span9">
      <% form_for(account, {action: path_to.account(), method: 'POST',
class:'form-horizontal'}, function (form) { %>

```

```

<% flash = request.flash('error').pop(); if (flash) { %>
<div class="alert alert-error">
  <a class="close" data-dismiss="alert">×</a>
  <%- flash %>
</div>
<% }; %>
<% info = request.flash('info').pop(); if (info) { %>
<div class="alert alert-info">
  <a class="close" data-dismiss="alert">×</a>
  <%- info %>
</div>
<% }; %>
<fieldset>
  <legend>
    Dados básicos
  </legend>
  <div class="control-group">
    <label class="control-label">Nome</label>
    <div class="controls">
      <input type="text" class="input-large" name="firstName"
value="Teste TCC">
    </div>
  </div>
  <div class="control-group">
    <label class="control-label">Dominio</label>
    <div class="controls">
      <input type="text" class="input-large" name="firstName"
value="localhost:8000">
    </div>
  </div>
  <div class="control-group">
    <label class="control-label">Ativo</label>
    <div class="controls">

```

```

        <label class="checkbox"><input type="checkbox"
checked=""></label>
    </div>
</div>
<br />
<div class="control-group">
    <label class="control-label">Exibir</label>
    <div class="controls">
        <label class="checkbox"><input type="checkbox"
value="">Exibir chat </label>
        <label class="checkbox"><input type="checkbox"
value="">Exibir configurações do cliente </label>
    </div>
</div>
<div class="control-group">
    <label class="control-label">Posição sidebar</label>
    <div class="controls">
        <select>
            <option>Lateral esquerda</option>
        </select>
    </div>
</div>
<div class="control-group">
    <label class="control-label">Posição do botão de
compartilhamento</label>
    <div class="controls">
        <select>
            <option>Lateral direita</option>
        </select>
    </div>
</div>
<br />
<legend>Código</legend>
<div class="control-group" style="border-bottom: none;">

```

```

    <pre>
    &lt;script type="text/javascript"&gt;
    (function() {
    var po = document.createElement("script"); po.type = "text/javascript";
    po.async = true;
    po.src = "https://localhost:3000/observer.js";
    var s = document.getElementsByTagName("script")[0];
    s.parentNode.insertBefore(po, s);
    })();
    &lt;/script&gt;
    </pre>

```

Cole esse código abaixo em seu site para adicionar a funcionalidade

```

    </div>
    <Br />
    </fieldset>
    <fieldset id="support">
    <div class="form-actions">
    <button class="btn btn-primary" type="submit">
    <%= t('account.save') %>
    </button>
    </div>
    </fieldset>
    <% });%>
    </div>
    </div>
    </article>

```

Signin/index.ejs

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <title><%= title %></title>
    <%- stylesheet_link_tag('bootstrap', 'bootstrap-responsive', 'style') %>

```



```

    <%- csrf_meta_tag() %>
</head>
<body>
  <article id="login-page">
    <h1><%= t('company.name') %></h1>
    <section id="login-box">
      <h1><%= t('signin.title') %></h1>
      <% form_for(signin, {action: path_to.signin(), method: 'POST'},
function (form) { %>
        <% flash = request.flash('error').pop(); if (flash) { %>
          <div class="alert alert-error">
            <a class="close" data-dismiss="alert">x</a>
            <%- flash %>
          </div>
          <% }; %>
          <ul class="unstyled">
            <li class="field row-fluid">
              <input name="email" class="span12"
value="rafaelmotta021@gmail.com" placeholder="<%= t('signin.email') %>"
type="email" required="required">
            </li>
            <li class="field row-fluid">
              <input name="password" class="span12" value="123456"
placeholder="<%= t('signin.password') %>" type="password"
required="required">
            </li>
            <li class="actions">
              <button class="btn btn-primary btn-block btn-large"
type="submit">
                <%= t('signin.title') %>
              </button>
            </li>
          </ul>
        <% });%>

```

```

</section>
<div id="login-link">
  <%= t('signin.forgotPass') %>?
  <a href="/forgot_password?method=login" class="utility"><%=
t('signin.resetClickHere') %></a>
</div>
</article>
</body>
</html>

```

Tickets/index.ejs

```

<article id="tickets-page">

<div class="row-fluid">
  <div class="span12">
    <div class="widget-box">
      <div class="widget-title">
        <span class="icon"> <i class="icon-th"></i> </span>
        <h5>Static table</h5>
      </div>
      <div class="widget-content nopadding">
        <table class="table table-bordered table-striped">
          <thead>
            <tr>
              <th>IP</th>
              <th>Inicio</th>
              <th>Duração</th>
              <th>Localização</th>
              <th>Browser</th>
              <th>Dispositivos</th>
              <th>Ações</th>
            </tr>
          </thead>
          <tbody>
            <tr>

```

```

        <td><%= ip %></td>
        <td><%= begin %></td>
        <td><%= duration %></td>
        <td><%= location %></td>
        <td><%= browser %></td>
        <td><%= device %></td>
        <td>
            <a href="javascript:void(0);" class="icon-ok" rel="tooltip"
data-original-title="Aceitar"></a>
            <a href="javascript:void(0);" class="icon-remove"
rel="tooltip" data-original-title="Aceitar"></a>
        </td>
    </tr>
</tbody>
</table>
</div>
</div>
</div>
</div>

```

```
</article>
```

Config

Development.coffee

```

app.configure 'development', ->
  app.enable 'log actions'
  app.enable 'env info'
  app.disable 'view cache'
  app.disable 'model cache'
  app.disable 'eval cache'
  app.use require('express').errorHandler dumpExceptions: true, showStack:
true

```

production.coffee

```

app.configure 'production', ->

```

```
app.enable 'view cache'  
app.enable 'model cache'  
app.enable 'eval cache'  
app.enable 'merge javascripts'  
app.enable 'merge stylesheets'  
app.disable 'assets timestamps'  
app.use require('express').errorHandler()  
app.enable 'quiet'
```

test.coffee

```
app.configure 'test', ->  
  app.use require('express').errorHandler dumpExceptions: true, showStack:  
true  
  app.settings.quiet = true  
  app.enable 'view cache'  
  app.enable 'model cache'  
  app.enable 'eval cache'
```

en.yml

en:

logout: "Sair"

account:

title: "Conta"

personalData: "Dados pessoais"

firstName: "Primerio nome"

lastName: "Último nome"

email: "Email"

emailAddress: "Endereço de email"

resetPassword: "Reseta sua senha"

currentPassword: "Senha atual"

newPassword: "Nova senha"

confirmPassword: "Confirme a nova senha"

save: "Salvar"

company:

name: "Observador"

signin:

title: "Entrar"

email: "Email"

password: "Senha"

forgotPass: "Esqueceu sua senha"

resetClickHere: "Clique aqui para resetar"

dashboard:

title: "Dashboard"

tickets:

title: "Tickets"

settings:

title: "Configurações"

observer:

title: "Observar"

database.yml

development:

driver: "mysql"

host: "localhost"

port: "3306"

driver: "mysql"

database: "observer"

username: "root"

password: ""

test:

```
driver: "memory"
```

environment.coffee

```
express = require 'express'
```

app.configure ->

```
  cwd = process.cwd()
```

```
  app.set 'view engine', 'ejs'
```

```
  app.set 'view options', complexNames: true
```

```
  app.enable 'coffee'
```

```
  app.use express.static(cwd + '/public', maxAge: 86400000)
```

```
  app.use express.bodyParser()
```

```
  app.use express.cookieParser 'secret'
```

```
  app.use express.session secret: 'secret'
```

```
  app.use express.methodOverride()
```

```
  app.use app.router
```

routes.coffee

```
exports.routes = (map)->
```

```
  map.resources 'observer'
```

```
  map.resources 'account'
```

```
  map.resources 'settings'
```

```
  map.resources 'tickets'
```

```
  map.resources 'dashboard'
```

```
  map.resources 'signin'
```

```
  map.get '/logout', 'signin#logout'
```

DB

Schema.coffee

customSchema =>

```
mongoose = require 'mongoose'  
db = mongoose.createConnection 'localhost', 'observer'
```

SCHEMAS

```
userSchema = new mongoose.Schema  
  firstName: { type: String, required: true }  
  lastName:  { type: String }  
  email:    { type: String, required: true }  
  password: { type: String, required: true }  
  role:     { type: Number, default: 1 , required: true }  
  activated: { type: Boolean, default: true }  
  photo:    { type: String }  
  createdAt: { type: Date, default: Date.now }  
  lastLogin: { type: Date }  
  companies: [  
    type: mongoose.SchemaTypes.ObjectId  
    ref: 'Company'  
  ]
```

```
companySchema = new mongoose.Schema  
  name:    { type: String, required: true }  
  url:     { type: String, required: true }  
  settings:  
    observeQuiet: { type: Boolean, default: true }  
    openTicket:   { type: Boolean, default: true }  
    enableChat:   { type: Boolean, default: true }  
    enableShareEvents: { type: Boolean, default: true }
```

MODELS

```
User = db.model 'User', userSchema
```

```
Company = db.model 'Company', companySchema
```

```
module.exports['User'] = User
```

```
module.exports['Company'] = Company
```

```
Observer = describe 'Observer', () ->
```

Server.coffee

```
#!/usr/bin/env coffee
```

```
app = module.exports = require('railway').createServer()
```

```
app.io = require('socket.io').listen(app)
```

```
app.socket =
```

```
if not module.parent
```

```
  port = process.env.PORT or 3000
```

```
  app.listen port
```

```
  console.log "Node server listening on port %d within %s environment", port,
```

```
app.settings.env
```

client.js

```
#!/usr/bin/env node
```

```
var WebSocketServer = require('ws').Server;
```

```
var http = require('http');
```

```
var url = require('url');
```

```
var fs = require('fs');
```

```
var args = { /* defaults */
```

```
  port: '3000'
```



```

};

/* Parse command line options */
var pattern = /^--(.*)?(?:=(.*))?$/;
process.argv.forEach(function(value) {
  var match = pattern.exec(value);
  if (match) {
    args[match[1]] = match[2] ? match[2] : true;
  }
});

var port = parseInt(args.port, 10);

console.log("Usage: ./server.js [--port=8080]");

var connections = {};

// ws is the fastest websocket lib - http://einaros.github.com/ws/
var wsServer = new WebSocketServer({port: port});

wsServer.on('connection', function(ws) {

  ws.id = Date.now(); // Assign unique id to this ws connection.
  connections[ws.id] = ws;

  console.log((new Date()) + ' Connection accepted: ' + ws.id);

  ws.on('message', function(message, flags) {
    console.log('Received ' + (flags.binary ? 'binary' : '') + ' message: ' +
      message.length + ' bytes.');
```

```

    broadcast(message, this, flags);
  });

  ws.on('close', function() {

```

```

    console.log((new Date()) + " Peer " + this.id + " disconnected.");
    delete connections[this.id];
  });
});

// Broadcasts a message to all connected sockets except for the sender.
function broadcast(message, fromWs, flags) {
  for (var id in connections) {
    if (id !== fromWs.id) {
      connections[id].send(message, {
        binary: flags.binary ? true : false,
        mask: false
      });
    }
  }
}

```

screencapture.js

```

(function(exports) {
function urlsToAbsolute(nodeList) {
  if (!nodeList.length) {
    return [];
  }

  var attrName = 'href';
  if (nodeList[0].__proto__ === HTMLImageElement.prototype ||
    nodeList[0].__proto__ === HTMLScriptElement.prototype) {
    attrName = 'src';
  }

  nodeList = [].map.call(nodeList, function(el, i) {
    var attr = el.getAttribute(attrName);
    // If no src/href is present, disregard.

```

```

if (!attr) {
    return;
}

var absURL = /^(https?|data):/i.test(attr);
if (absURL) {
    return el;
} else {
    // Set the src/href attribute to an absolute version.
    // if (attr.indexOf('/') != 0) { // src="images/test.jpg"
    //     el.setAttribute(attrName, document.location.origin +
document.location.pathname + attr);
    // } else if (attr.match(/^VV/)) { // src="//static.server/test.jpg"
    //     el.setAttribute(attrName, document.location.protocol + attr);
    // } else {
    //     el.setAttribute(attrName, document.location.origin + attr);
    // }

    // Set the src/href attribute to an absolute version. Accessing
    // el['src']/el['href'], the browser will stringify an absolute URL, but
    // we still need to explicitly set the attribute on the duplicate.
    el.setAttribute(attrName, el[attrName]);
    return el;
}
});
return nodeList;
}

// TODO: current limitation is css background images are not included.
function screenshotPage() {
    // 1. Rewrite current doc's imgs, css, and script URLs to be absolute before
    // we duplicate. This ensures no broken links when viewing the duplicate.
    urlsToAbsolute(document.images);
    urlsToAbsolute(document.querySelectorAll("link[rel='stylesheet']"));
}

```

```

//urlsToAbsolute(document.scripts);

// 2. Duplicate entire document.
var screenshot = document.documentElement.cloneNode(true);

// Use <base> to make anchors and other relative links absolute.
var b = document.createElement('base');
b.href = document.location.protocol + '//' + location.host;
var head = screenshot.querySelector('head');
head.insertBefore(b, head.firstChild);

// 3. Screenshot should be readonly, no scrolling, and no selections.
screenshot.style.pointerEvents = 'none';
screenshot.style.overflow = 'hidden';
screenshot.style.webkitUserSelect = 'none';
screenshot.style.mozUserSelect = 'none';
screenshot.style.msUserSelect = 'none';
screenshot.style.oUserSelect = 'none';
screenshot.style.userSelect = 'none';

// 4. Preserve current x,y scroll position of this page. See addOnPageLoad_().
screenshot.dataset.scrollX = window.scrollX;
screenshot.dataset.scrollY = window.scrollY;

// 4.5. When the screenshot loads (e.g. as ablob URL, as iframe.src, etc.),
// scroll it to the same location of this page. Do this by appending a
// window.onDOMContentLoaded listener which pulls out the saved scrollX/Y
// state from the DOM.
var script = document.createElement('script');
script.textContent = '(' + addOnPageLoad_.toString() + ')();'; // self calling.
screenshot.querySelector('body').appendChild(script);

// 5. Create a new .html file from the cloned content.
var blob = new Blob([screenshot.outerHTML], {type: 'text/html'});

```

```

return blob;
}

// NOTE: Not to be invoked directly. When the screenshot loads, it should scroll
// to the same x,y location of this page.
function addOnPageLoad_() {
  window.addEventListener('DOMContentLoaded', function(e) {
    var scrollX = document.documentElement.dataset.scrollX || 0;
    var scrollY = document.documentElement.dataset.scrollY || 0;
    window.scrollTo(scrollX, scrollY);
  });
}

exports.screenshotPage = screenshotPage;

})(window);

//window.URL = window.URL || window.webkitURL;
//window.open(window.URL.createObjectURL(screenshotPage()));

```

index.html

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="chrome=1">

    <link href="/public/stylesheets/bootstrap.min.css" media="screen"
rel="stylesheet" type="text/css" />
    <link href="/public/stylesheets/bootstrap-responsive.min.css"
media="screen" rel="stylesheet" type="text/css" />
    <link href="/public/stylesheets/app.css" media="screen" rel="stylesheet"
type="text/css" />

```

```

<script
src="//ajax.googleapis.com/ajax/libs/jquery/1.8.2/jquery.min.js"></script>
<script>
        window.jQuery || document.write('<script
src="js/jquery.min.js"></script>')
</script>
<script src="/public/javascripts/bootstrap.min.js"></script>
</head>
<body>
<div id="content">
<section>
<div id="frames-container"></div>
</section>
<aside>
<ul class="unstyled">
<li>
<b>Admin</b>
<p>Olá</p>
</li>
<textarea placeholder="Escreva alguma mensagem"></textarea>
</ul>
<dl class="dl-horizontal">
<dt>IP:</dt>
<dd>192.168.1.100</dd>
<hr />
<dt>Localização:</dt>
<dd>Florianópolis - BR</dd>
<hr />
<dt>Dispositivo:</dt>
<dd>IOS - iPad</dd>
<hr />
<dt>Browser:</dt>
<dd>Safari 5</dd>

```

```
<hr />
<dt>Agente:</dt>
<dd>WebKit</dd>
</dl>
</aside>
</div>
```

```
<iframe id="screenshot0" hidden></iframe>
<iframe id="screenshot1" hidden></iframe>
<script src="screencapture.js"></script>
<script src="/public/javascripts/ws.js"></script>
<script>
```

```
var framesContainer = document.querySelector('#frames-container');
var currentFrameIdx = 0;
var playbackIntervalId = null;
var ws = connect();
```

```
ws.onmessage = function(e) {
  // Shouldn't have to create a new blob, but e.data isn't preserving type.
  var blob = new Blob([e.data], {type: 'text/html'});
```

```
  var iframe = document.createElement('iframe');
  iframe.src = window.URL.createObjectURL(blob);
  iframe.hidden = true;
  iframe.onload = renderFrame;
```

```
  // Force the iframe content to load by appending to the DOM.
  framesContainer.appendChild(iframe);
};
```

```
function renderFrame() {
  if (framesContainer.children.length) {
    var frame = framesContainer.children[currentFrameIdx];
```

```

if(!frame) {
    return;
}

if (currentFrameIdx > 0) {
    var prevFrame = frame.previousElementSibling;
    prevFrame.hidden = true;
    window.URL.revokeObjectURL(prevFrame.src);
}

frame.hidden = false;

currentFrameIdx++;
}
}

function playback(interval) {
    clearInterval(playbackIntervalId);

    if (!framesContainer.children.length) {
        return;
    }

    var i = 0;
    playbackIntervalId = setInterval(function() {
        var iframe = framesContainer.children[i];
        if (i > 0) {
            framesContainer.children[i - 1].hidden = true;
        } else if (i == 0) {
            framesContainer.children[framesContainer.children.length - 1].hidden =
true;
        }
        iframe.hidden = false;
    }, interval);
}

```



```
    i++;
    i %= framesContainer.children.length;
  }, interval);
}
```

```
//setInterval(renderFrame, 1000);
</script>
  </body>
</html>
```

app.js

```
window.URL = window.URL || window.webkitURL;
```

```
var WS_HOST = 'localhost:3000';
```

```
var ws = null;
```

```
var REFRESH_EVERY = 1000; // ms
```

```
function connect() {
```

```
  ws = new WebSocket('ws://' + WS_HOST, ['dumby-protocol']);
```

```
  ws.binaryType = 'blob';
```

```
  ws.onopen = function(e) {
```

```
    console.log('WebSocket connection OPEN');
```

```
  };
```

```
  ws.onclose = function(e) {
```

```
    console.log('WebSocket connection CLOSED');
```

```
  };
```

```
  ws.onerror = function(e) {
```

```
    console.log('WebSocket connection ERROR', e);
```

```
  };
```

```
  return ws;
```

```
}
```

```
function send() {  
  setInterval(function() {  
    if (ws.bufferedAmount == 0) {  
      ws.send(screenshotPage());  
    }  
  }, REFRESH_EVERY);  
}
```

cliente_app.js

```
var framesContainer = document.querySelector('#frames-container');  
var currentFrameIdx = 0;  
var playbackIntervalId = null;  
var ws = connect();
```

```
ws.onmessage = function(e) {  
  // Shouldn't have to create a new blob, but e.data isn't preserving type.  
  var blob = new Blob([e.data], {type: 'text/html'});  
  
  var iframe = document.createElement('iframe');  
  iframe.src = window.URL.createObjectURL(blob);  
  iframe.hidden = true;  
  iframe.onload = renderFrame;  
  
  // Force the iframe content to load by appending to the DOM.  
  framesContainer.appendChild(iframe);  
};
```

```
function renderFrame() {  
  if (framesContainer.children.length) {  
    var frame = framesContainer.children[currentFrameIdx];  
  
    if(!frame) {
```

```

    return;
}

if (currentFrameIdx > 0) {
    var prevFrame = frame.previousElementSibling;
    prevFrame.hidden = true;
    window.URL.revokeObjectURL(prevFrame.src);
}

frame.hidden = false;

currentFrameIdx++;
}
}

function playback(interval) {
    clearInterval(playbackIntervalId);

    if (!framesContainer.children.length) {
        return;
    }

    var i = 0;
    playbackIntervalId = setInterval(function() {
        var iframe = framesContainer.children[i];
        if (i > 0) {
            framesContainer.children[i - 1].hidden = true;
        } else if (i == 0) {
            framesContainer.children[framesContainer.children.length - 1].hidden =
true;
        }
        iframe.hidden = false;

        i++;
    }, interval);
}

```

```
        i %= framesContainer.children.length;  
    }, interval);  
}
```