

Comunicação em
tempo real com WebRTC
Veja como integrar sua aplicação
com o browser sem plugins

Edição 09



JAVASCRIPT + OO

Obtenha uma
programação mais eficaz



Django e Python

Crie aplicações web rápidas com
o Django e extensões

Socket.IO + jQuery

Crie um chat web em tempo
real estilo MIRC

MVP

R\$ 1.000.000,00
INVESTIDOS EM CONTEÚDO
NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO
NA SUA CARREIRA...

E MOSTRE AO MERCADO
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R\$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR
MAIS DO MERCADO!



 **DEVMEDIA**

EXPEDIENTE

Editor

Diogo Souza (diogosouzac@gmail.com)

Consultor Técnico

Daniella Costa (daniella.devmedia@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



DIOGO SOUZA

diogosouzac@gmail.com

Analista de Sistemas Java na Indra Company e já trabalhou em empresas como Instituto Atlântico e Ebix L.A. É instrutor Android, palestrante em eventos sobre Java e o mundo mobile e consultor DevMedia. Conhecimentos e experiências em diversas linguagens e ferramentas de programação e manipulação de dados, bem como metodologias úteis no desenvolvimento de Sistemas diversificados.

Sumário

Conceúdo sobre Boas Práticas, Artigo no estilo Solução Completa

04 – Comunicação em tempo real com WebRTC

[Júlio Sampaio]

Conceúdo sobre Boas Práticas, Artigo no estilo Curso

16 – Como criar um Blog com Django e Python - Parte 1

[Júlio Sampaio]

Conceúdo sobre Boas Práticas, Artigo no estilo Solução Completa

27 – Orientação a objetos com JavaScript

[Willian Carvalho]

Conceúdo sobre Boas Práticas

36 – Como criar um chat com Node.js

[Rafael Milleo]

Comunicação em tempo real com WebRTC

Veja como integrar seus projetos web com recursos dos browsers sem a necessidade de plugins ou extensões

Até o presente momento, várias foram as tentativas de criar plataformas híbridas e abstratas o suficiente para permitir a comunicação, em tempo real, de aplicativos entre os diversos tipos de dispositivos diferentes que existem. Alguns esforços recentes que exploram tal conceito mais focado no universo mobile e IoT (*Internet of Things*, ou Internet das Coisas), tais como o Tizen ou o próprio Android, trazem consigo a possibilidade de desenvolver voltado para uma gama de hardwares dos mais diversos tipos, tamanhos e finalidades (TVs, smartphones, tablets, wearables – relógios inteligentes, óculos, etc.). Mas todo esse aparato de tecnologia envolve uma complexidade considerável, principalmente se levarmos em conta a quantidade de fabricantes dos dispositivos, suas próprias especificações, os modelos de hardware e firmware, e, sobretudo, a forma como a web se comunica com tudo isso, uma vez que ela será o ponto de partida para essa tão sonhada padronização.

Tudo na web funciona (e deve) de forma padronizada. As especificações do W3C definem os rumos que uma dada tecnologia segue nesse universo, bem como sua morte em detrimento do nascimento de uma melhor. Ao mesmo tempo, empresas como Google, Mozilla, Facebook, Twitter, dentre outras, influenciam diretamente em tais rumos através do lançamento de novas features para os seus browsers, da submissão de novas especificações para o W3C, da criação de novos frameworks front-end (a saber, AngularJS, Bootstrap, etc.), bem como da centralização de suas comunidades de desenvolvimento.

Agora, longe das realidades do mundo móvel, imagine uma realidade onde a sua TV, seu smartphone, sua geladeira e seu computador possam se comunicar sem todas essas barreiras, em tempo real, em uma só plataforma. Imagine poder compartilhar todo tipo de dado, vídeos, chats, mensagerias entre esses dispositivos de

Fique por dentro

Uma das maiores dificuldades da maioria dos desenvolvedores web é ter de lidar com tantos protocolos, especificações e regras distintas para cada fabricante de hardware, software e middleware lançados no mercado. Muitas vezes a inclusão de drivers, web services, softwares de meio campo, dentre outras opções para integrar as tecnologias, se torna a única opção viável, mas neste artigo veremos que o WebRTC veio para quebrar esse paradigma. Ao final deste você estará apto a criar suas aplicações e fazer uso total do poder dos browsers modernos.

forma integrada e rápida. Pode parecer muito futurista, mas já temos uma tecnologia que tenta abraçar esse ideal: o WebRTC (contração de *Web Real Time Communication*, ou Comunicação Web em Tempo Real).

O conceito de RTC já é mais antigo e tentou abraçar por muito tempo um dos maiores desafios para a web: a comunicação humana via voz e vídeo. A ideia é que ele passe a ser tão natural nas aplicações web quanto digitar um texto qualquer em um campo de input no browser. Sem isso, estamos presos às soluções separadas de terceiros que tendem a dividir cada vez mais o processo. Historicamente, o desenvolvimento desse tipo de tecnologia sempre foi custoso e complexo, requerendo tecnologias de áudio e vídeo licenciadas ou desenvolvidas dentro das próprias empresas. Entretanto, talvez o maior desafio seja integrar a tecnologia RTC com o conteúdo já existente, dados e serviços, uma vez que eles consomem muito tempo para serem submetidos a tal adaptação, principalmente no universo web.

Desde os seus primeiros passos, quando o Google comprou a empresa GIPS, a qual foi responsável pela criação de vários componentes requeridos pelo RTC, o WebRTC tem agora vários padrões abertos para comunicação em tempo real, plugins gratuitos de vídeo, áudio e comunicação de dados, além de estar presente por padrão nos browsers mais usados (Chrome, Firefox, Opera e Edge). Essa complexidade já abraça alguns casos reais, a saber:

- Muitos serviços atualmente já usam RTC, porém necessitam downloads, apps nativas ou plugins. É o caso do Skype, Facebook (que faz uso do Skype) e Google Hangouts (que faz uso do plugin Google Talk);
- Efetuar o download, instalação e atualização de plugins pode ser complexo, não intuitivo e complicado;
- Plugins podem ser difíceis de depurar, efetuar deploy, encontrar soluções para eventuais erros, testar e manter (além de poderem exigir licenciamento ou integração com tecnologias mais complexas e caras). Além disso, o próprio fato de se tratar de um plugin gera desconfiança na maioria dos usuários leigos, que não querem instalar nada de terceiros no browser, sob pena de se tratar de algum conteúdo malicioso.

Além disso, o WebRTC é usado em vários apps mobile como WhatsApp, Facebook Messenger, appear.in e algumas plataformas como a TokBox. A própria Microsoft incluir as APIs de Stream e MediaCapture no seu mais recente browser, o Edge.

Meu primeiro WebRTC

O WebRTC implementa basicamente e três APIs: MediaStream, RTCPeerConnection e RTCDATAChannel.

MediaStream

A API de MediaStream é nada mais que streams de mídia sincronizados que está presente nos browsers Chrome, Opera, Firefox e Edge. Por exemplo, uma stream de dados tirada diretamente de uma câmera ou de um microfone tem implicitamente faixas de áudio e vídeo sincronizadas (*synchronized*). Não confunda as faixas de MediaStream com o elemento da HTML5 `<track>`, que é usado para outra finalidade. A melhor forma de entender o que é uma MediaStream é testando-o via exemplos base disponibilizados no site do projeto hospedado no GitHub. Para isso:

- Abra a URL do demo de MediaStream (vide seção **Links**) no Google Chrome ou Opera;
- Abra a ferramenta de desenvolvedor *Console* (use o atalho F12 para isso);
- Inspecione a variável **stream** que está em escopo global (você deverá ver uma tela parecida com a ilustrada na **Figura 1**).

```

stream
  ↳ MediaStream {}
    active: true
    ended: false
    id: "DEKcNCABccPsTkgH3t6D8vcM3Z047Ddew9g"
    label: "DEKcNCABccPsTkgH3t6D8vcM3Z047Ddew9g"
    onactive: null
    onaddtrack: null
    onended: function ()
    oninactive: null
    onremovetrack: null
    __proto__: MediaStream
  
```

Figura 1. Variável **stream** inspecionada no *Console*

Cada MediaStream tem um input (que corresponde ao MediaStream gerado pelo método `navigator.getUserMedia()`), e um output (que deve ser passado sempre ao elemento de vídeo ou para um objeto do tipo `RTCPeerConnection`).

O método `getUserMedia()`, por sua vez, recebe três parâmetros, a saber:

- Um objeto de constraint;
- Uma função de callback de sucesso que, se chamada, receberá um MediaStream como parâmetro;
- Uma função de callback de falha que, se chamada, receberá um objeto de erro como parâmetro.

Cada MediaStream também tem uma label, no exemplo em questão de valor “`DEKcNCABccPsTkgH3t6D8vcM3Z047Ddew9g`”. Além disso, um vetor de faixas de MediaStream é retornado sempre que uma chamada aos métodos `getAudioTracks()` e `getVideoTracks()` for efetuada. Contudo, a função também serve como um nó de entrada de dados para a API de Web Audio (veja na **Listagem 1** um exemplo básico disso).

Listagem 1. Exemplo básico de acesso à Web Audio API.

```

function recuperarStream(stream) {
  window.AudioContext = window.AudioContext || window.webkitAudioContext;
  var contextoAudio = new AudioContext();

  // Cria um AudioNode para o stream
  var fonteMidia = contextoAudio.createFonteMidia(stream);

  fonteMidia.connect(contextoAudio.destination);
}

navigator.getUserMedia({audio:true}, recuperarStream);
  
```

RTCPeerConnection

Este é o componente WebRTC responsável por manipular a comunicação estável e eficiente do streaming de dados entre peers (pares). Veja na **Figura 2** um diagrama completo sobre a arquitetura do WebRTC, na qual podemos ver o papel do RTCPeerConnection. Note que as partes marcadas em verde são as mais complexas da arquitetura e, portanto, requerem mais atenção.

Analizando sob uma perspectiva do JavaScript, a principal coisa a se entender a partir deste diagrama é que o RTCPeerConnection protege os desenvolvedores web das inúmeras complexidades que se escondem por baixo da sua arquitetura. Os codecs e protocolos usados pelo WebRTC efetuam uma enorme quantidade de trabalho para tornar a comunicação em tempo real possível, mesmo em redes não confiáveis:

RTCDATAChannel

Além de áudio e vídeo, o WebRTC suporta comunicação em tempo real com outros tipos de dados. A API do RTCDATAChannel possibilita o câmbio peer-to-peer de dados arbitrários, com baixa latência e alta taxa de transferência. Existem inúmeros e potenciais casos de uso para essa API, incluindo:

Comunicação em tempo real com WebRTC

- Jogos;
- Aplicações desktop remotas;
- Chats em tempo real;
- Transferência de arquivos;
- Redes descentralizadas.

A API tem várias features para fazer a maioria das conexões via RTCPeerConnection, além de habilitar uma comunicação flexível peer-to-peer. Na **Listagem 2** podemos conferir um exemplo básico da sintaxe desse tipo de recurso que se assemelha em muito aos WebSockets, com um método *send()* e um evento de *message*.

Como a comunicação ocorre diretamente entre os browsers, os RTCDDataChannels podem ser mais rápidos que os WebSockets. Esse recurso está disponível para os browsers Chrome, Opera e Firefox.

Listagem 2. Exemplo básico de uso dos RTCDDataChannels.

```
var pc = new webkitRTCPeerConnection(servers,
{optional: [{RtpDataChannels: true}]});

pc.ondatachannel = function(event) {
  canalRecebido = event.channel;
  canalRecebido.onmessage = function(event){
    document.querySelector("div#receive").innerHTML = event.data;
  };
};

canalEnvio = pc.createDataChannel("sendDataChannel", {reliable: false});

document.querySelector("button#send").onclick = function (){
  var dados = document.querySelector("textareatextarea#send").value;
  canalEnvio.send(dados);
};
```

Constraints

As *constraints* (restrições) foram implementadas em todos os principais browsers como recursos que permitem ser usados para configurar valores para resolução de vídeo como resultado de chamadas às funções *getUserMedia()* e *addStream()*. O maior objetivo desses recursos é permitir implementar o suporte a outras constraints já comuns no universo front-end, tais como o aspecto de ratio (bordas redondas), *facing mode* (poder usar a câmera de frente ou trás), frame rate (computação da quantidade de frames por quadro), largura e altura, etc.

Captura de tela e tabs

Através do WebRTC também é possível, exclusivamente no Chrome, compartilhar um recurso de vídeo ao vivo de uma única tab do navegador ou do desktop inteiro via APIs *chrome.tabCapture* e *chrome.desktopCapture*. A extensão que possibilita o acesso ao desktop pode ser encontrada na página oficial do framework no GitHub (vide seção **Links**).

Também é possível usar as capturas de telas como MediaStream no Chrome usando a constraint experimental *chromeMediaSource*. Para isso, entretanto, é necessário lembrar que esse tipo de captura faz uso de HTTPS e deve ser usado apenas em ambientes de desenvolvimento.

Acessando a câmera com *getUserMedia*

Para exemplificar o que vimos até agora, vamos desenvolver como primeiro exemplo um teste simples de acesso ao dispositivo de câmera do seu computador (seja ela acoplada ao notebook ou um dispositivo real). Após, faremos uso da mesma API para gravar o vídeo da câmera e salvá-lo via JavaScript.

Para iniciar precisamos, portanto, de algum pacote que disponibilize recurso de servir arquivos HTML estáticos nos browsers, isso

porque o Chrome (ou qualquer outro browser que suporte WebRTC) não nos permite usar *getUserMedia* em páginas servidas via *file://* (URL padrão para arquivos estáticos salvos localmente, que é exatamente o que acontece quando estamos desenvolvendo localmente). Você pode usar a tecnologia que se sentir mais à vontade: Java Web, PHP, Python para ambientes server side, ou Node.js se preferir manter tudo no front-end, que é o que faremos nos exemplos deste artigo. Efetue o download do arquivo de instalação do Node.js (seção **Links**), dê duplo clique no mesmo e siga todos os passos até o fim sem alterar nenhuma opção padrão nas wizards de instalação. Se por acaso não lembrar se já instalou o Node.js anteriormente no seu sistema operacional, basta executar o seguinte comando no prompt de comando:

```
node --version
```

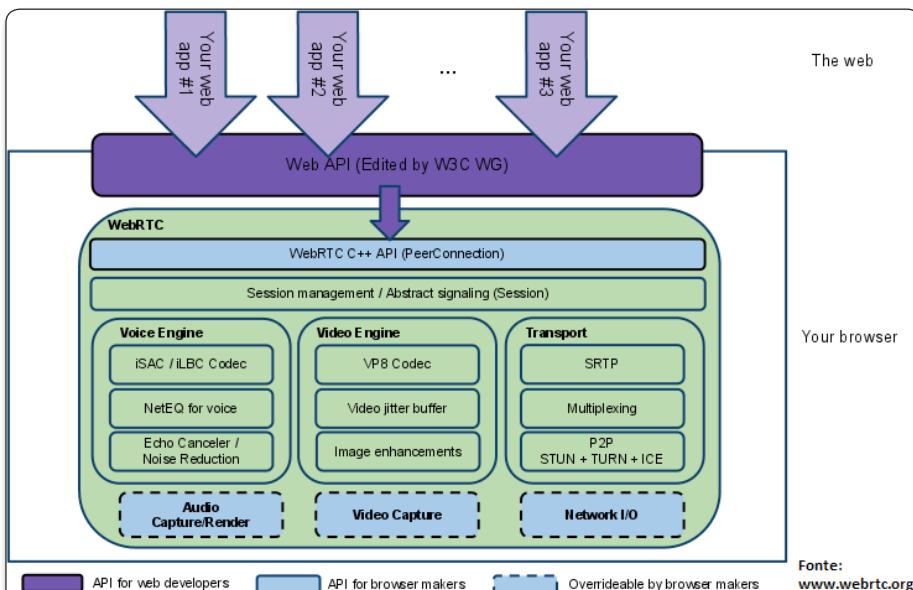


Figura 2. Arquitetura interna do WebRTC

No momento de escrita deste artigo a versão mais recente era a v4.2.0. Verifique se qualquer outro valor é impresso e, caso ele siga o padrão mostrado, seu Node.js já se encontra instalado. Se por ventura a versão estiver muito desatualizada é aconselhável baixar o instalador mais recente e efetuar os passos de atualização (todos default, também). Nessa última opção, em especial, quando da execução do exe, aguarde até que ele faça a verificação de ambiente e espaço em disco, isso pode levar algum tempo.

Após finalizado todo o processo, fecha o cmd, abra-o novamente e redigite o comando para verificar se a versão foi instalada/atualizada, dependendo do seu caso. Ao fazer a instalação do Node.js, você automaticamente baixou também o npm (*Node Package Manager*), o gerenciador de pacotes do Node.js que se encarrega de controlar o acervo de dependências de pacotes que Node.js provê por padrão e disponibiliza para os seus projetos. Para verificar se o npm foi instalado corretamente, digite o seguinte comando no terminal de comandos:

```
npm --version
```

A versão correspondente é a 2.14.7. É importante salientar que o npm não pode ser instalado separadamente ao Node.js, portanto, siga sempre os passos focando no Node.js primeiro.

Além disso, também precisaremos instalar o pacote “node-static” do Node.js para gerenciar os arquivos estáticos HTML/Javascript que criaremos e servi-los no browser via localhost. Para isso, vá até o terminal e digite o seguinte comando:

```
npm install -g node-static
```

Ele será responsável por efetuar a instalação do pacote de forma global (por isso a flag -g), ou seja, para todos os usuários do computador. Aguarde até que finalize e após digite o seguinte comando para verificar se tudo ocorreu bem:

```
static -v
```

O texto “node-static 0.7.7” deverá aparecer, a versão mais recente até o momento. Pronto, estamos prontos para começar a desenvolver com o getUserMedia. Para iniciar com o primeiro exemplo precisamos antes criar uma nova página HTML. Podemos usar qualquer editor de texto de sua preferência (aqui faremos uso do Notepad++ para fins de simplicidade). Adicione o conteúdo apresentado na **Listagem 3** no novo arquivo de nome index.html, apenas para testar o node-static por enquanto. Esse conteúdo servirá somente para verificarmos o correto funcionamento do pacote que servirá as páginas via HTTP. Veja que também adicionamos a tag <video> no fim do documento para que possamos exibir a câmera na mesma em detrimento da autorização que nosso código solicitará. Para testar o node-static, navegue até o diretório onde criou nossa página index.html e digite o seguinte comando:

```
static
```

Listagem 3. Conteúdo HTML inicial da página de index.html.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<meta http-equiv="X-UA-Compatible" content="chrome=1"/>
<title>Record a getUserMedia() Session</title>
<style>
body {
background-color: #000;
color: white;
}
a[download] {
text-transform: uppercase;
font-size: 11px;
font-weight: bold;
}
h4 {
padding: 15px;
background: black;
color: white;
margin: 10px 0 10px 0;
border-radius: 100px 100px 0;
letter-spacing: 1px;
font-weight: 300;
}
section > div {
text-align: center;
display: inline-block;
margin: 0 15px;
min-width: 400px;
}
#video-preview {
height: 300px;
}
button.recording {
color: darkred;
border-color: red;
}
section {
margin-top: 2em;
}
h2 {
text-align: center;
}
</style>
</head>
<body>
<h2>Acessando <code>getUserMedia()</code> via WebRTC.</h2>
<div id="video-container">
<video id="camera-stream" width="500" autoplay/>
</div>
</body>
</html>
```

Você deverá ver algo semelhante ao que temos na **Figura 3**, com a mensagem ‘serving “.” at <http://127.0.0.1:8080>’ sendo impressa. Essa mensagem diz que todos os arquivos presentes neste atual diretório estão também disponibilizados na respectiva URL (que também pode ser substituída por <http://localhost:8080>) e porta.

Agora você só precisa acessar no browser o endereço que ele gerou no log do Console e acrescentar o nome da página ao mesmo: <http://localhost:8080/index.html> e o resultado será igual ao da **Figura 4**.

Comunicação em tempo real com WebRTC

Após isso, para recuperarmos o objeto de getUserMedia, precisamos criar uma tag <script> logo após a tag <body> e incluir o conteúdo da **Listagem 4** à mesma.

Nota

O elemento de vídeo não apareceu ainda na tela em vista de não termos nenhum arquivo associado. O leitor pode adicionar ainda o atributo "controls" à tag para adicionar os controles de pausa e visualização do vídeo em tela cheia.



Figura 3. Mensagem de retorno no cmd para o comando static



Figura 4. Tela index.html acessada via localhost

Listagem 4. Código da função JavaScript que recupera getUserMedia.

```
<script>
window.onload = function() {
    // Normaliza as várias versões de getUserMedia de acordo com os fabricantes.
    navigator.getUserMedia = (navigator.getUserMedia ||
        navigator.webkit GetUserMedia ||
        navigator.mozGetUserMedia ||
        navigator.msGetUserMedia);

    }
</script>
```

A função em si não surtirá em nenhum efeito visível direto na página, uma vez que ela é responsável somente por normalizar as diferentes versões de código que recupera o getUserMedia em detrimento dos muitos browsers existentes.

Agora precisamos instanciar a função passando para a mesma três parâmetros, a saber:

- **constraints:** esse parâmetro representa um objeto que especifica que media stream você gostaria de acessar. Por exemplo, para recuperar ambos áudio e vídeo, você deve usar: {video: true, audio: true}.
- **successCallback:** função que será chamada se a media stream for carregada com sucesso. A função passará um objeto do tipo LocalMediaStream.

- **errorCallback:** parâmetro opcional, representa a função que será chamada caso a media stream não seja carregada.

Logo, para solicitar o acesso à câmera, modifique o seu código da tag <script> para o representado na **Listagem 5**. Dê uma olhada nos comentários inseridos dentro do código.

Listagem 5. Conteúdo final do JavaScript que solicita acesso à câmera.

```
<script>
window.onload = function() {
    // Normaliza as várias versões de getUserMedia de acordo com os fabricantes.
    navigator.getUserMedia = (navigator.getUserMedia ||
        navigator.webkit GetUserMedia ||
        navigator.mozGetUserMedia ||
        navigator.msGetUserMedia);

    // Checa se o browser suporta getUserMedia.
    // Caso não, mostra um alert, senão continua.
    if(navigator.getUserMedia) {
        // Requesita a câmera.
        navigator.getUserMedia(
            // Constraints
            {
                video: true
            },
            // Função de Callback de Sucesso
            function(localMediaStream) {
                // Código aqui...
            },
            // Função de Callback de Erro
            function(err) {
                // Loga o erro no console.
                console.log('O erro aconteceu quando tentamos acessar
                    getUserMedia:' + err);
            }
        );
    } else {
        alert('Desculpe, seu browser não suporta getUserMedia');
    }
}
</script>
```

Perceba que não estamos fazendo nada na segunda função de sucesso, mas você pode acessar as propriedades do objeto localMediaStream via debug no Console e inspecioná-las. Caso algum erro aconteça, na última função estamos logando tudo no Console JavaScript, então você poderá acompanhar a execução e seus resultados.

Para testar, salve o arquivo e recarregue a página no browser. Você verá o resultado semelhante ao da **Figura 5**, onde temos a exibição do diálogo solicitando a permissão para desbloquear o recurso de câmera. Note também que no prompt cmd que você usou para servir as conexões estáticas, as alterações nos arquivos surtem efeito imediato no formato de mensagens de log, indicando o que mudou, portanto, só precisamos dar um refresh na página e tudo será recarregado e atualizado.

Após isso, você verá um símbolo circular vermelho piscando na aba do browser, indicando que a câmera está ligada.

O último passo é lançar a imagem de vídeo capturada direto na tag <video> HTML. Para isso, precisamos acessar o objeto LocalMediaStream retornado como parâmetro da segunda função (de sucesso). Portanto, modifique o conteúdo do seu JavaScript da segunda função para o exibido na **Listagem 6**. Em seguida salve novamente o arquivo e recarregue a página conforme a **Figura 6**. Tudo que a listagem faz é recuperar o elemento referente à tag <video> e associar o localMediaStream retornado a um objeto URL.

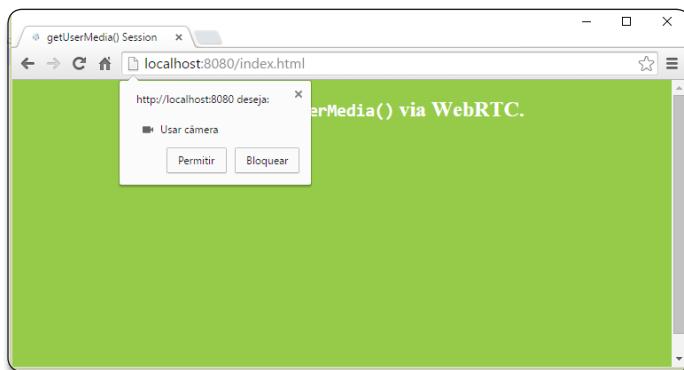


Figura 5. Tela index.html com câmera sendo requisitada

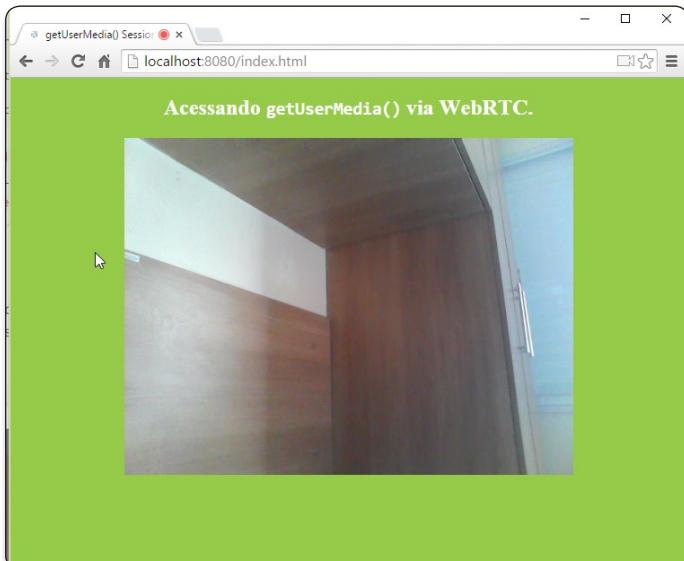


Figura 6. Tela index.html com imagem da câmera exibida na tag <video>.

Listagem 6. Função de sucesso para exibição da câmera na tag <video>.

```
// Função de Callback de Sucesso
function(localMediaStream) {
    // Recupera a referência ao elemento de vídeo na página.
    var id_video = document.getElementById('camera-stream');

    // Cria um objeto URL para o stream de vídeo e o usa
    // para configurar o atributo source do vídeo.
    id_video.src = window.URL.createObjectURL(localMediaStream);
},
```

Adicionando efeitos

Você pode ainda brincar com esse recurso através da inclusão de código CSS no meio. Os filtros do CSS3 permitem que você crie facilmente efeitos para os seus vídeos, como sépia, blur, grayscale, dentre outros famosos (inclusive usados por apps como Instagram, por exemplo). Tudo isso é possível através do atributo CSS `-webkit-filter`, que pode ser adicionado diretamente no elemento de vídeo da página.

Para tanto, adicione a seguinte regra dentro da sua tag <style> na mesma página (ou crie um arquivo .css separado e importe-o no cabeçalho, caso prefira):

```
#camera-stream {
    -webkit-filter: sepia(1);
}
```

Agora recarregue a página e pronto, você verá o resultado. Você pode inclusive, dinamizar isso via JavaScript e combobox HTML, mudando o efeito CSS em detrimento da opção selecionada na mesma. Por exemplo, modifique o conteúdo do seu CSS (tag <style>) dentro da página para o representado na **Listagem 7**, bem como o conteúdo HTML para o exibido na **Listagem 8**.

Listagem 7. Código CSS para efeitos de filtro.

```
...
#video-container {
    text-align: center;
}
.sepia {
    -webkit-filter: sepia(1);
}
.blur {
    -webkit-filter: blur(3px);
}
.grayscale {
    -webkit-filter: grayscale(1);
}
.saturate {
    -webkit-filter: saturate(3);
}
</style>
```

Listagem 8. Código HTML para conter a nova combobox de filtros.

```
<body>
<h2>Acessando <code>getUserMedia()</code> via WebRTC.</h2>
<div id="video-container">
    <select id='efecto' onchange='selectChange()'>
        <option value='1'>Sepia</option>
        <option value='2'>Blur</option>
        <option value='3'>Grayscale</option>
        <option value='4'>Saturate</option>
    </select>
    <br><br>
    <video id="camera-stream" width="500" autoplay/>
</div>
</body>
```

Comunicação em tempo real com WebRTC

Perceba que estamos apenas incluindo um novo elemento HTML, o `<select>` que conterá as opções de filtros CSS. Também precisamos de um `id` para recuperar o elemento via JavaScript, bem como setar a função `selectChange()` no evento `onchange` do elemento, que será responsável por executar o código JavaScript de mudança da respectiva classe CSS associada.

Listagem 9. Código JavaScript para lidar com nova combobox de filtros.

```
function selectChange() {
    efeito = document.getElementById('efeito');
    camera_stream = document.getElementById('camera-stream');
    switch(efeito.options[efeito.selectedIndex].value) {
        case '1':
            camera_stream.className = 'sepia';
            break;
        case '2':
            camera_stream.className = 'blur';
            break;
        case '3':
            camera_stream.className = 'grayscale';
            break;
        case '4':
            camera_stream.className = 'saturate';
            break;
    }
}
```

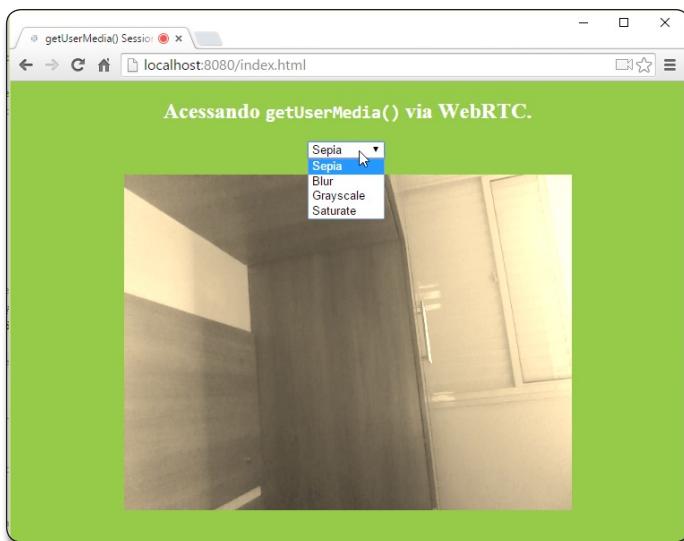


Figura 7. Tela com combobox e filtro de sépia aplicado



Figura 8. Tela para gravar em vídeo imagem da webcam

No conteúdo CSS, por sua vez, criamos apenas efeitos através da propriedade `-webkit-filter` citada, referentes a cada opção disponível no CSS3. Para fazer o exemplo funcionar, basta que incluamos a função JavaScript referenciada na **Listagem 9** dentro da nossa tag `<script>`.

Certifique-se de inserir o seu conteúdo fora da função de `.onload` do objeto `window`. O que ela faz é basicamente recuperar os elementos de efeito e da tag `<video>`, verificar qual foi selecionado através de um `switch-case` e modificar o valor de cada propriedade `className` nos mesmos.

Após isso, salve tudo e reexecute no browser. O resultado pode ser visto na **Figura 7**.

Gravando vídeos da câmera

Através do exemplo anterior, conseguimos ver não só o poder dos WebRTCs para transferir o vídeo recebido da webcam para a tag da HTML `<video>` como também as diversas formas de habilitar tal recurso. Mas não é só isso, também podemos salvar os vídeos, reproduzi-los, liberar opções de download, dentre outros recursos que iremos implementar a seguir.

Para isso, vamos criar uma nova página HTML e acrescentar à mesma o conteúdo da **Listagem 10**.

Perceba que além das classes CSS usuais para incutir efeitos e design na página, temos agora três novos botões: o primeiro (de id "câmera-me") se encarregará de disparar a solicitação da câmera e exibir a imagem no nosso primeiro elemento de vídeo na HTML. O atributo `autoplay` diz que o vídeo deverá executar automaticamente assim que encontrado uma fonte válida; já o segundo botão se encarrega de iniciar a gravação (id "record-me"); e o último trata de parar a mesma quando o usuário tiver finalizado o processo (id "stop-me"). O resultado dessa página será igual ao da **Figura 8**.

Agora, ainda dentro da tag `<body>`, criemos uma nova tag `<script>` para iniciar a implementação do código JavaScript que irá dinamizar todo o processo. Acrescente como código inicial o conteúdo da **Listagem 11**.

O código das linhas 1 a 14 trata de recuperar os objetos base de URL, animação e `getUserMedia` que já vimos como declarar antes. Das linhas 16 a 22 recuperamos os demais objetos canvas e criamos algumas variáveis que nos auxiliarão no restante da implementação. Na linha 24 criamos a função que vai mapear os seletores (processo semelhante ao que o jQuery faz) para não ter de usar a função `getElementById()` sempre que quisermos recuperar um elemento do DOM.

Na linha 35 declaramos a função `turnOnCamera()` que, além de executar o mesmo código do exemplo anterior em relação à chamada à função `getUserMedia()`, também redimensiona as dimensões padrão do vídeo (320x240) e exibe um vídeo de um filme online caso algum erro aconteça ou o usuário bloqueeie o acesso à câmera no momento que for solicitada.

Listagem 10. Código HTML da nova página de teste.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<meta http-equiv="X-UA-Compatible" content="chrome=1"/>
<title>Gravando um sessão getUserMedia()</title>
<style>
body, button {
    font-family: Segoe UI;
}
button {
    padding: 10px;
}
a[download] {
    text-transform: uppercase;
    font-size: 11px;
    font-weight: bold;
}
h4 {
    padding: 15px;
    background: #8cc53e;
    color: white;
    margin: 10px 0 10px 0;
    border-radius: 100px 0 100px 0;
    letter-spacing: 1px;
    font-weight: 300;
}
section > div {
    text-align: center;
    display: inline-block;
    margin: 0 15px;
    min-width: 400px;
}
#video-preview {
    height: 300px;
}

}
button.recording {
    color: darkred;
    border-color: red;
}
section {
    margin-top: 2em;
}
h2 {
    text-align: center;
}
</style>
</head>
<body>
<h2>Gravando vídeo .webm do <code>getUserMedia()</code>.</h2>

<section>
<div style="float:left;">
    <button id="camera-me">1. Ligar a câmera</button>
    <h4>
        <code>getUserMedia()</code> Camera</h4>
        <video autoplay/>
    </div>
    <div id="video-preview">
        <button id="record-me" disabled>2. Gravar
        </button>
        <button id="stop-me" disabled>■</button>
        <span id="elapced-time"/>
        <h4>gravando .webm (sem áudio)</h4>
    </div>
</section>
<script src="whammy.min.js"></script>
</body>
</html>
```

Listagem 11. Código JavaScript da nova página de teste.

```
01 (function(exports) {
02     exports.URL = exports.URL || exports.webkitURL;
03
04     exports.requestAnimationFrame = exports.requestAnimationFrame ||
05         exports.webkitRequestAnimationFrame || exports.mozRequestAnimationFrame ||
06         exports.msRequestAnimationFrame || exports.oRequestAnimationFrame;
07
08     exports.cancelAnimationFrame = exports.cancelAnimationFrame ||
09         exports.webkitCancelAnimationFrame || exports.mozCancelAnimationFrame ||
10         exports.msCancelAnimationFrame || exports.oCancelAnimationFrame;
11
12     navigator.getUserMedia = navigator.getUserMedia ||
13         navigator.webkitGetUserMedia || navigator.mozGetUserMedia ||
14         navigator.msGetUserMedia;
15
16     var ORIGINAL_DOC_TITLE = document.title;
17     var video = $('#video');
18     var canvas = document.createElement('canvas');
19     var rafId = null;
20     var startTime = null;
21     var endTime = null;
22     var frames = [];
23
24     function $(selector) {
25         return document.querySelector(selector) || null;
26     }
27
28     function toggleActivateRecordButton() {
29         var b = $('#record-me');
30         b.textContent = b.disabled ? 'Gravar' : 'Gravando...';
31         b.classList.toggle('recording');
32         b.disabled = !b.disabled;
33     }
34
35     function turnOnCamera(e) {
36         e.target.disabled = true;
37         $('#record-me').disabled = false;
38
39         video.controls = false;
40
41         var finishVideoSetup_ = function() {
42             // Nota: video.onloadedmetadata não dispara no Chrome quando usa
43             // getUserMedia
44             // então temos de usar setTimeout. Veja em crbug.com/110938.
45             setTimeout(function() {
46                 video.width = 320;//video.clientWidth;
47                 video.height = 240;// video.clientHeight;
48                 // Canvas é 1/2 para performance. Caso contrário, o getImageData() será péssimo
49                 canvas.width = video.width;
50                 canvas.height = video.height;
51             }, 1000);
52
53             navigator.getUserMedia({
54                 video: true,
55                 audio: true
56             }, function(stream) {
57                 video.src = window.URL.createObjectURL(stream);
58                 finishVideoSetup_();
59             }, function(e) {
60                 alert('Ops, você cancelou a câmera... :( Então, teremos de rodar um
61                     vídeo no lugar...!');
62                 video.src = 'http://video.webmfiles.org/big-buck-bunny_trailer.webm';
63                 finishVideoSetup_();
64             });
65         };
66     }
67
68     function finishVideoSetup_(e) {
69         if (e) {
70             alert('Ops, ocorreu um erro ao tentar gravar o vídeo...:(');
71         } else {
72             $('#stop-me').disabled = false;
73             $('#record-me').disabled = true;
74         }
75     }
76
77     turnOnCamera();
78
79     $('#record-me').addEventListener('click', toggleActivateRecordButton);
80
81     $('#stop-me').addEventListener('click', function() {
82         if (video.src) {
83             video.src = '';
84         }
85     });
86
87     $('#elapced-time').text(new Date().toLocaleTimeString());
88
89     var start = new Date();
90
91     $('#start').text(start.toLocaleTimeString());
92
93     $('#stop').text('');
94
95     $('#stop').addEventListener('click', function() {
96         var end = new Date();
97         var duration = end - start;
98         var durationString = duration.toLocaleString();
99         $('#stop').text(durationString);
100    
```

Comunicação em tempo real com WebRTC

Agora precisamos incluir as funções que serão responsáveis por gravar e salvar o vídeo de fato. Portanto, acrescente o conteúdo da **Listagem 12** ao fim do da anterior.

A função `record()` se encarrega de recuperar o elemento de Canvas, recuperar a data atual, contar os segundos (frames) que

a gravação está levando e chamar o método `drawImage()` para atualizar o vídeo em tempo real. Note também que na linha 16 estamos modificando o valor do título do documento para receber também a duração da gravação. No fim, salvamos cada URL dentro de um vetor de frames, para não precisar repetir todo o processo nas futuras tentativas.

A função `stop()` trata de parar a gravação, recuperar a hora em que isso aconteceu, mudar o título da página e chamar a função `embedVideoPreview()` (linha 38) que, por sua vez, cria um novo elemento de vídeo no DOM, seta os atributos de dimensão, `autoplay` e `controls`, e associa o vídeo capturado salvo no vetor ao mesmo, exibindo-o. No fim da listagem temos a função `initEvents()` que associa cada um dos elementos de botão da página às suas respectivas funções. Agora é só testar, o resultado pode ser visto na **Figura 9**.

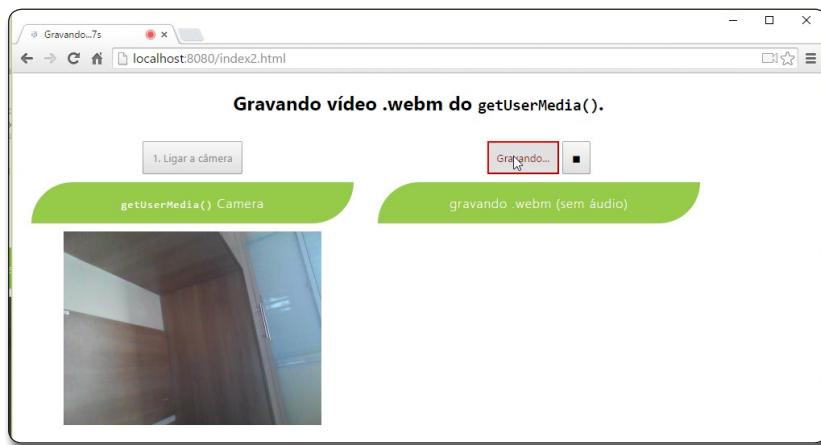


Figura 9. Tela final de gravação do vídeo da webcam

Controle de sessão, network e mídias

O WebRTC faz uso do `RTCPeerConnection` para se comunicar com dados de streaming entre browsers,

Listagem 12. Código JavaScript da nova página de teste - Parte 2.

```
01 function record() {
02     var elapsedTime = $('#elapsed-time');
03     var ctx = canvas.getContext('2d');
04     var CANVAS_HEIGHT = canvas.height;
05     var CANVAS_WIDTH = canvas.width;
06
07     frames = []; // limpa os frames remanescentes
08     startTime = Date.now();
09
10    toggleActivateRecordButton();
11    $('#stop-me').disabled = false;
12
13    function drawVideoFrame_(time) {
14        rafId = requestAnimationFrame(drawVideoFrame_);
15        ctx.drawImage(video, 0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);
16        document.title = 'Gravando...' + Math.round((Date.now() - startTime) / 1000) + 's';
17        var url = canvas.toDataURL('image/webp', 1);
18        frames.push(url);
19    };
20
21    rafId = requestAnimationFrame(drawVideoFrame_);
22 };
23
24 function stop() {
25    cancelAnimationFrame(rafId);
26    endTime = Date.now();
27    $('#stop-me').disabled = true;
28    document.title = ORIGINAL_DOC_TITLE;
29
30    toggleActivateRecordButton();
31
32    console.log('frames capturados:' + frames.length + '=>' +
33      ((endTime - startTime) / 1000) + 's video');
34
35    embedVideoPreview();
36 };
37
38 function embedVideoPreview(opt_url) {
39    var url = opt_url || null;
40    var video = $('#video-preview video') || null;
41
42    var downloadLink = $('#video-preview a[download]') || null;
43
44    if (!video) {
45        video = document.createElement('video');
46        video.autoplay = true;
47        video.controls = true;
48        video.loop = true;
49        video.style.width = canvas.width + 'px';
50        video.style.height = canvas.height + 'px';
51        $('#video-preview').appendChild(video);
52
53        downloadLink = document.createElement('a');
54        downloadLink.download = 'capture.webm';
55        downloadLink.textContent = '[ download video ]';
56        downloadLink.title = 'Download your .webm video';
57        var p = document.createElement('p');
58        p.appendChild(downloadLink);
59
60        $('#video-preview').appendChild(p);
61    } else {
62        window.URL.revokeObjectURL(video.src);
63    }
64
65    if (!url) {
66        var webmBlob = Whammy.fromImageArray(frames, 1000 / 60);
67        url = window.URL.createObjectURL(webmBlob);
68    }
69    video.src = url;
70    downloadLink.href = url;
71
72    function initEvents() {
73        $('#camera-me').addEventListener('click', turnOnCamera);
74        $('#record-me').addEventListener('click', record);
75        $('#stop-me').addEventListener('click', stop);
76    }
77    initEvents();
78    exports.$ = $;
79 })(window);
```

mas também precisa de um mecanismo para coordenar essa comunicação e enviar mensagens de controle, um processo chamado de *signaling* (sinalização). Os métodos e protocolos de sinalização não são especificados pelo WebRTC, uma vez que não pertencem à API.

Em vez disso, os desenvolvedores de aplicativos WebRTC podem escolher o protocolo de mensageria de sua preferência, como o SIP ou XMPP, e qualquer canal de comunicação duplex (via dupla). Esse processo é usado para trocar três tipos de informações:

- Mensagens de controle de sessão: para inicializar ou fechar a comunicação e reportar erros;
- Configurações de rede: para o mundo exterior, em detrimento das informações de IP e porta;
- Mídias: que tipos de codecs e resoluções podem ser manipuladas pelo browser e quais browsers eles desejam se comunicar.

Por exemplo, imagine que o Pedro deseja se comunicar com a Mônica. Vejamos na **Listagem 13** um pequeno exemplo que mostra o processo de sinalização em ação. O código assume a existência de alguma mecanismos de sinalização, criado no método `criarCanalSinalizacao()`. Note também que no Chrome e Opera, o `RTCPeerConnection` está atualmente prefixado.

Inicialmente, Pedro e Mônica trocam informações de rede:

1. Pedro cria um objeto do tipo `RTCPeerConnection` com um handler `onicecandidate` atrelado ao mesmo;
2. O handler é executado quando a rede de ambos os candidatos se torna disponível;
3. Pedro lança um candidate serializado a Mônica, através de um canal de sinalização qualquer: WebSocket ou outro mecanismo;

4. Quando Mônica recebe a mensagem de Pedro, ela chama o método `addIceCandidate()` para adicionar um candidato ao peer remoto de descrições.

Transmitindo texto via `RTCDataChannel`

É possível fazer muitos tipos de implementação usando o `RTCDataChannel`, dentre elas o uso de protocolos de transferência de arquivos, texto, mídias, assim como quaisquer recursos de mensageria como web chats, dentre outros.

Vejamos como fazer uso do mesmo através da criação de um exemplo simples de página com três botões: um botão de *Iniciar* que se encarrega de iniciar a comunicação com o objeto do WebRTC e habilita as duas textareas que criaremos para receber o texto; um botão de *Enviar* que faz o envio efetivo via API; e um botão *Parar* que se encarrega de encerrar o canal de comunicação.

O conteúdo HTML dessa nova página se encontra na **Listagem 14**. Crie uma nova página HTML e adicione-o à mesma.

A nova página, além de trazer o convencional CSS customizado, também apresenta os três botões citados, bem como as duas textareas e o import de três arquivos de JavaScript auxiliares baixados diretamente da API no GitHub, que você pode importar diretamente do arquivo de fontes deste artigo no topo da página. O visual da página pode ser conferido na **Figura 10**.

Para testar, basta clicar no botão Iniciar, digitar o conteúdo e clicar em Enviar. O conteúdo digitado irá aparecer na segunda textarea. De todo o código JavaScript responsável por fazer o exemplo funcionar, selecionamos as três funções mais importantes (**Listagem 15**) do arquivo `main.js`.

A primeira função da listagem (`createConnection()`) se encarrega de abrir a conexão com o objeto `RTCPeerConnection` passando

Listagem 13. Código de exemplo de comunicação usando sinalização.

```
var canalSinalizacao = criarCanalSinalizacao();
var pc;
var configuracao = ...;

// execute start(true) para iniciar a chamada
function start(isChamador) {
  pc = new RTCPeerConnection(configuracao);

  // envia qualquer candidatos para a outra pilha
  pc.onicecandidate = function (evt) {
    canalSinalizacao.send(JSON.stringify({ "candidate": evt.candidate }));
  };

  // assim que o stream remoto chegar, mostra-o no elemento de video remoto
  pc.onaddstream = function (evt) {
    remoteView.src = URL.createObjectURL(evt.stream);
  };

  // recupera o stream local, exibe-o no elemento de video e envia-o
  navigator.getUserMedia({ "audio": true, "video": true }, function (stream) {
    selfView.src = URL.createObjectURL(stream);
    pc.addStream(stream);
  });
}

if (isChamador)
  pc.createOffer(recuperaDescricao);
else
  pc.createAnswer(pc.remoteDescription, recuperarDescricao);

function recuperarDescricao(desc) {
  pc.setLocalDescription(desc);
  canalSinalizacao.send(JSON.stringify({ "sdp": desc }));
}

canalSinalizacao.onmessage = function (evt) {
  if (!pc)
    start(false);

  var sinal = JSON.parse(evt.data);
  if (sinal.sdp)
    pc.setRemoteDescription(new RTCSessionDescription(sinal.sdp));
  else
    pc.addIceCandidate(new RTCIceCandidate(sinal.candidate));
};
```

Comunicação em tempo real com WebRTC



Figura 10. Exemplo de uso do RTCDataChannel para transferir texto

as constraints definidas para o exemplo que foram criadas como variáveis globais. Note que o processo segue sempre os mesmos passos: primeiro abrimos uma conexão direta com o peer, depois criamos um novo DataChannel (linha 10) que, por sua vez, se encarregará de enviar o conteúdo. Em seguida, mapeamos nas linhas 14 e 15 as funções que serão executadas quando o canal for aberto e fechado, respectivamente.

Finalmente, criamos uma mesma instância desse objeto, porém remota, para se encarregar de lidar com mensagens vindas de um servidor, como no nosso caso.

Na segunda função, sendData(), temos o código necessário para enviar os dados, que se resume a uma linha e uma função - send() na linha 30. Por último, na função closeDataChannels() criamos o código que se encarrega de varrer cada um dos objetos de canal/conexão e chamar suas funções close(), além de anular/desabilitar as referências aos mesmos e desabilitar os botões

Listagem 14. Código HTML da nova página que usa o RTCDataChannel.

```
<!DOCTYPE html>
<html>
<head>

<meta charset="utf-8">
<meta name="description" content="WebRTC code samples">
<meta name="viewport" content="width=device-width, user-scalable=yes,
initial-scale=1, maximum-scale=1">
<meta name="mobile-web-app-capable" content="yes">
<meta id="theme-color" name="theme-color" content="#ffffff">

<base target="_blank">

<title>Envio de texto WebRTC</title>

<style>
body {
    background-color: #8cc53e;
    color: white;
}
body, button {
    font-family: Segoe UI;
}
button {
    padding: 10px;
}
a[download] {
    text-transform: uppercase;
    font-size: 11px;
    font-weight: bold;
}
h4 {
    padding: 15px;
    background: #8cc53e;
    color: white;
    margin: 10px 0 10px 0;
    border-radius: 100px 0 100px 0;
    letter-spacing: 1px;
    font-weight: 300;
}
section > div {
    display: inline-block;
    margin: 0 15px;
    min-width: 400px;
}

#video-preview {
    height: 300px;
}
button.recording {
    color: darkred;
    border-color: red;
}
section {
    margin-top: 2em;
}
</style>
</head>

<body>

<div id="container">

<h1>WebRTC exemplo de envio de texto</h1>

<div id="buttons">
<button id="startButton">Iniciar</button>
<button id="sendButton" disabled>Enviar</button>
<button id="closeButton" disabled>Parar</button>
</div>

<div id="sendReceive">
<div id="send">
<h2>Enviar</h2>
<textarea id="dataChannelSend" disabled placeholder="Pressione Iniciar,
digite o texto e pressione Enviar." rows="9" cols="50"></textarea>
</div>
<div id="receive">
<h2>Receber</h2>
<textarea id="dataChannelReceive" disabled rows="9" cols="50"></textarea>
</div>
</div>

<script src="adapter.js"></script>
<script src="common.js"></script>
<script src="main.js"></script>

</body>
</html>
```

Listagem 15. Três principais funções para o exemplo do RTCDataChannel.

```
01 function createConnection() {
02   dataChannelSend.placeholder = '';
03   var servers = null;
04   pcConstraint = null;
05   dataConstraint = null;
06   trace('Using SCTP based data channels');
07   window.localConnection = localConnection = new RTCPeerConnection(
08     servers, pcConstraint);
09   trace('Created local peer connection object localConnection');
10   sendChannel = localConnection.createDataChannel
11     ('sendDataChannel', dataConstraint);
12   trace('Created send data channel');
13   localConnection.onicecandidate = iceCallback1;
14   sendChannel.onopen = onSendChannelStateChange;
15   sendChannel.onclose = onSendChannelStateChange;
16
17   window.remoteConnection = remoteConnection =
18     new RTCPeerConnection(servers, pcConstraint);
19   trace('Created remote peer connection object remoteConnection');
20   remoteConnection.onicecandidate = iceCallback2;
21   remoteConnection.ondatachannel = receiveChannelCallback;
22
23   localConnection.createOffer(gotDescription1, onCreateSessionDescriptionError);
24   startButton.disabled = true;
25   closeButton.disabled = false;
26 }
27
28 function sendData() {
29   var data = dataChannelSend.value;
30   sendChannel.send(data);
31   trace('Sent Data:' + data);
32 }
33
34 function closeDataChannels() {
35   trace('Closing data channels');
36   sendChannel.close();
37   trace('Closed data channel with label:' + sendChannel.label);
38   receiveChannel.close();
39   trace('Closed data channel with label:' + receiveChannel.label);
40   localConnection.close();
41   remoteConnection.close();
42   localConnection = null;
43   remoteConnection = null;
44   trace('Closed peer connections');
45   startButton.disabled = false;
46   sendButton.disabled = true;
47   closeButton.disabled = true;
48   dataChannelSend.value = '';
49   dataChannelReceive.value = '';
50   dataChannelSend.disabled = true;
51   disableSendButton();
52   enableStartButton();
53 }
```

Estes foram exemplos singelos que puderam expressar, em suma, todo o poder dos WebRTCs. Seus recursos basicamente focam na criação e comunicação entre dispositivos, visando uma facilitação desse trabalho no que se refere à abstração de hardware, protocolos, telefonia, produção de áudio/vídeo, entre outros. Muito se espera das futuras versões do WebRTC e o que a comunidade tem aprontado para essa tecnologia promissora, principalmente considerando-se o poder que ela poder ter se associada à HTML5. Agora é com você, se aprofunde na documentação oficial (seção **Links**), faça mais alguns testes, amplie o leque de dispositivos, migre seu código para outros browsers e veja como eles proveem suporte ao WebRTC. Bons estudos!

Autor**Fabrício Hissao Kawata**

fabricio.kawata@bol.com.br

Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos.

**Links:****Página oficial do WebRTC.**

<http://www.webrtc.org/>

Página de exemplo do MediaStream.

<https://webrtc.github.io/samples/src/content/getusermedia/gum/>

Arquitetura do WebRTC.

<http://www.webrtc.org/architecture>

Página de download do Node.js

<https://nodejs.org/download/>

Como criar um Blog com Django e Python - Parte 1

Aprenda a criar aplicações web fáceis e rápidas com o Django e suas extensões

ESTE ARTIGO FAZ PARTE DE UM CURSO

Um framework web nada mais é que um conjunto de componentes integrados, e até certo ponto independentes, que nos ajudam a desenvolver aplicações web de forma mais rápida e facilitada. Quando falamos no universo front-end, inúmeras são as opções: desde as baseadas no JavaScript em si, tais como jQuery, AngularJS, CoffeScript, etc.; passando pelo universo CSS/CSS3 como o Sass e o Less; até chegar no imprescindível HTML/HTML5, como o Bootstrap, Thymeleaf, dentre outros. Isso sem citar o universo web móvel que traria mais exemplos famosos como o jQuery Mobile, Ionic e Framework7.

Independentemente da solução escolhida, é certo que não conseguimos mais viver sem os benefícios dos frameworks. Alguns deles também precisam de uma pinçada back-end, na maioria das vezes baseada em Node.js, o que é uma ótima opção, considerando-se as vantagens de usá-lo em conjunto com suas soluções afins. Quando falamos de comunicação com o servidor, recursos como autenticação, serviços, painel de gerenciamento, formulários, upload/download de arquivos, etc. que são funcionalidades amplamente presentes, precisam ser cuidados e solucionados.

O Django é um framework web de aplicações, totalmente gratuito e escrito em Python. Ele fornece uma stack pronta e completa para solucionar problemas de autenticação, controle de administração, site maps, feeds RSS, além de implementar recursos de segurança nativa.

Fique por dentro

Este artigo é útil por explorar as principais facetas de um dos frameworks web mais usados na atualidade. Baseado em Python e estruturas web comuns (HTML5, CSS, JavaScript, etc.), o Django é flexível, rápido e extremamente fácil de usar. Em poucos passos conseguiremos configurar seu ambiente, utilizar a interface de linha de comando para criar projetos, customizá-los e desenvolver suas primeiras aplicações usando o Django. Aqui trataremos de entender o framework e criar o esqueleto inicial do microblog com o nosso template, integrações, publicação, servidor, etc.

vamente, sem que você tenha que se preocupar em configurá-los. É fácil de usar, de configurar e, sobretudo, de manter.

Para os objetivos deste artigo é necessário que você tenha conhecimento prévios, ao menos básicos, sobre a linguagem Python, uma vez que a usaremos como base para o restante da implementação. Trataremos de criar uma aplicação de microblog, com administração, tela de posts, gerenciamento de comentários, dentre outras funcionalidades comuns a esse tipo de aplicação, de forma semelhante ao de blogs famosos como o Blogspot. Tentaremos explorar ao máximo os principais recursos do framework, expondo seus detalhes de configuração, boas práticas, e regras de codificação.

Configurando o ambiente

Antes de começar a instalação, é interessante que o leitor tire um tempinho para fazer um tour pelo site oficial do framework (seção [Links](#)), lá você encontrará alguns resumos rápidos sobre o que é o Django e suas principais features.

Basicamente, precisamos instalar duas coisas no nosso sistema operacional: o Python em si, que no Windows não vem com um

binário adicionado por padrão e, portanto, precisa ser baixado e configurado; e o Django, que será instalado via interface de linha de comando. Portanto, acesse a página de downloads do Python (seção **Links**) e baixe a versão correspondente ao seu sistema operacional (tem uma para cada tipo diferente, certifique-se também de selecionar o arquivo correto para a versão do seu SO: 32/64 bits). No momento de escrita deste artigo estávamos na versão 3.5.0, mas pode ficar à vontade para baixar uma versão mais recente, caso já exista. Execute o .exe e, na primeira tela que aparecer, clique em *Install Now* (certifique-se de manter checada a checkbox “*Install launcher for all users (recommended)*”). Essa ação também precisará de suas permissões de administrador do SO, aguarde até que o processo todo termine.

Por padrão, a instalação descompactará os arquivos do Python no diretório *C:\Program Files\Python3.5*, é aconselhável que não modifique isso (na primeira tela ele mostra o diretório, caso não seja de seu agrado, modifique-o via opção de customização). Para testar se tudo ocorreu com sucesso, abra o prompt de comando do seu SO e digite o comando *python --version*. Caso o comando não seja reconhecido pelo sistema, então durante a instalação não foram adicionados os diretórios do Python à respectiva variável de ambiente. Para corrigir isso, acesse o menu *Windows* do seu computador e digite “variáveis de ambiente” e clique na opção que aparecer. Na seção “Variáveis do sistema” clique em Novo... e configure os valores tal como mostrado na **Figura 1**. O conteúdo da caixa de texto “Valor da variável” é *;D:\Program files\Python3.5;D:\Program files\Python3.5\Scripts* e corresponde ao diretório dos binários onde você instalou o Python.

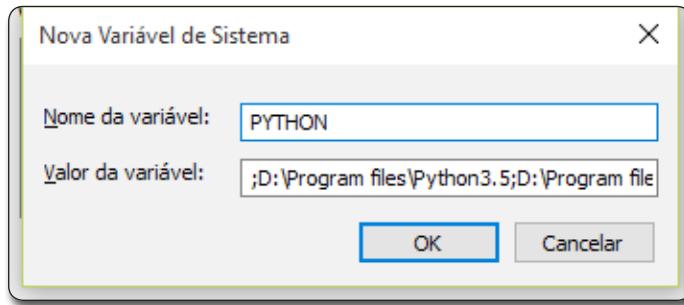


Figura 1. Configurando variável de ambiente do PYTHON

Clique em *OK*, e, em seguida, procure pela variável *Path* e clique em *Editar...*. Adicione ao final do campo o valor *%PYTHON%* e clique novamente em *OK* três vezes. Agora, abra novamente o prompt cmd e redigite o comando. Você deve ver a versão da linguagem sendo exibida.

Virtual Environments

Antes de fazermos a instalação do Django, precisamos nos assegurar de que teremos como gerenciar as diferentes versões do Python em nosso sistema operacional sem confundir ou danificar nenhuma outra. Esse tipo de prática se chama *virtualenvs* (*virtual environments*, ou ambientes virtuais), e permite que diferentes versões do Python e seus pacotes associados possam existir em

harmonia no mesmo lugar, ao mesmo tempo em que isola os setups de projetos Python/Django uns dos outros. O conceito é semelhante aos workspace que vemos em outras linguagens como o Java, por exemplo.

Tudo que precisamos fazer é selecionar o diretório onde desejamos criar o nosso virtualenv, portanto navegue até o mesmo via cmd e execute o seguinte comando na pasta:

```
python -m venv microblog
```

Após o término, este comando irá criar a estrutura inicial do projeto com algumas pastas de bibliotecas e scripts necessários para trabalhar com o Python no seu virtualenv. Porém, para iniciar os trabalhos no mesmo é preciso ativá-lo, portanto execute o seguinte comando:

```
microblog\Scripts\activate
```

Isso ativará uma espécie de subconsole, que você poderá usar para efetuar quaisquer comandos como se estivesse no cmd padrão. O nome do seu virtualenv também vem por padrão pré-adicionado ao endereço de máquina acessado.

Agora sim podemos instalar o Django. Para isso, faremos uso do pip que é uma ferramenta gerenciadora de pacotes e extensões do Python, dos quais o Django faz parte. Para verificar se o pip foi corretamente instalado junto com o Python, digite o comando *pip --version* e verá o resultado (versão 7.1.2 no momento de escrita deste artigo). Para finalizar, execute o seguinte comando (com dois ==):

```
pip install Django==1.8.5
```

Essa é a versão mais recente do Django até o momento. Consulte o site oficial para verificar isso. Aguarde até que o utilitário faça o download de todas as dependências e pronto, você está com o Django instalado. Se tudo correr bem, você terá uma tela semelhante à da **Figura 2**.

```
(microblog) D:\tests\microblog_django>pip --version
pip 7.1.2 from d:\tests\microblog_django\microblog\lib\site-packages (python 3.5)

(microblog) D:\tests\microblog_django>pip install Django==1.8.5
Collecting Django==1.8.5
  Downloading Django-1.8.5-py2.py3-none-any.whl (6.2MB)
    100% [########################################] 6.2MB 45kB/s
Installing collected packages: Django
  Successfully installed Django-1.8.5

(microblog) D:\tests\microblog_django>
```

Figura 2. Tela de instalação bem-sucedida do Django

Nota

Se, durante a instalação, você receber alguma mensagem de erro, verifique se o caminho de diretório onde criou o seu projeto contém algum espaço em branco ou caracteres especiais. Caso sim, mova o mesmo para um que atenda a essas exigências.

Também é interessante que o leitor opte pelo editor de código fonte de sua preferência, de acordo com sua experiência em Python. Para os fins deste tutorial, faremos uso do Notepad++ que provê suporte completo ao Python, mas pode usar quaisquer outros, como o Sublime, Gedit, Atom ou inclusive as IDEs mais robustas como o Eclipse (e seu plugin pyDev) ou JetBrains PyCharm. Ou ainda, se preferir escapar de toda essa configuração desktop, o leitor pode configurar seus projetos via projeto *pythonanywhere.com* que fornece todos os recursos para trabalhar com Python totalmente online.

Primeiro projeto Django

Criar um projeto no Django, basicamente, significa dizer que vamos executar alguns scripts fornecidos pelo mesmo que irão nos darão a estrutura inicial para o projeto. Trata-se apenas de um monte de diretórios e arquivos que precisaremos usar mais tarde.

Em relação aos nomes de arquivos/diretórios, é interessante que não renomeie os arquivos que estamos prestes a criar. Também é importante que você não os move de local. O Django por si só já mantém uma certa estrutura e organização para ser capaz de encontrar coisas importantes mais à frente. Portanto, execute o seguinte comando no cmd:

```
django-admin startproject django_project .
```

Não esqueça de pôr o ponto no fim do comando, pois ele serve para indicar que o comando encerrou e que o projeto deve ser criado na raiz do diretório atual. Note também que o comando django-admin trata-se de um script presente nos binários de instalação que se encarregará de criar as estruturas de diretórios e arquivos necessárias para o projeto base do Django. Ao final, sua pasta microblog deve estar estruturada de forma semelhante à exibida na **Listagem 1**.

Listagem 1. Estrutura de arquivos/diretórios gerada pelo Django.

```
microblog_django
├── manage.py
├── microblog
└── django_project
    ├── settings.py
    ├── urls.py
    ├── wsgi.py
    └── __init__.py
```

Dos arquivos de extensão .py do Python, temos:

- **manage.py**: se encarrega de ajudar com o gerenciamento do site. Através dele estaremos aptos a iniciar um servidor web no computador sem ter de instalar nada mais;
- **settings.py**: contém a configuração do website;
- **urls.py**: contém uma lista de patterns que serão usados pelo *urlresolver*;

Os demais não são importantes para os nossos objetivos, logo não precisamos nos preocupar com eles.

Mudando Time Zone

Agora que já temos o projeto criado precisamos fazer algumas configurações específicas do *time zone* (fuso horário) do Brasil. Para isso, abra o arquivo *blog_site/settings.py* com o seu editor de código e procure a linha que contém a chave `TIME_ZONE`. Altere seu conteúdo para “America/Sao_Paulo” e salve o arquivo. Esse valor é correspondente à constante que representa a maioria dos estados brasileiros.

Além disso, também precisamos criar um caminho de diretório para os arquivos estáticos (CSS, JS, imagens, etc.) que precisarmos usar no projeto. Para tanto, vá até o fim do arquivo e adicione a seguinte linha de código:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

Isso será responsável por rotear as requisições a esses tipos de arquivos diretamente para essa pasta.

Configurando o banco de dados

Ao se trabalhar com Python você pode usar as configurações do banco de dados que desejar, porém, para abstrair os passos de configuração e instalação de bancos proprietários, optaremos por usar o que já vem por padrão com a plataforma: o sqlite3. Esse banco é extremamente leve e fácil de usar e está presente em vários ambientes como os browsers, Android, iOS, etc.

Ainda dentro do *settings.py* é possível observar o trecho de código que configura o acesso à engine de banco de dados do sqlite3, tal como vemos na **Listagem 2**.

Listagem 2. Código que configura o acesso ao banco de dados.

```
# Database
# https://docs.djangoproject.com/en/1.8/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Entretanto, para que o banco propriamente dito seja criado, precisamos executar o seguinte comando no prompt, diretamente no diretório *microblog_django* (onde está contido o arquivo *manage.py*):

```
python manage.py migrate
```

Se tudo ocorrer bem, veremos um log no console semelhante ao que temos na **Listagem 3**. É preciso que todas as configurações recebam um *OK* no final, assim teremos certeza que tudo deu certo.

Pronto, isso é tudo que precisamos para criar o nosso banco. Agora só precisamos iniciar o servidor para verificar se a nossa aplicação está funcional. Ainda no mesmo diretório raiz (que

contém o `manage.py`), vamos iniciar o servidor através do seguinte comando:

```
python manage.py runserver
```

Listagem 3. Log de saída para migração do banco de dados.

```
(microblog) D:\tests\microblog_django>python manage.py migrate
Operations to perform:
  Synchronize unmigrated apps: messages, staticfiles
  Apply all migrations: auth, admin, sessions, contenttypes
  Synchronizing apps without migrations:
    Creating tables...
      Running deferred SQL...
    Installing custom SQL...
    Running migrations:
      Rendering model states... DONE
      Applying contenttypes.0001_initial... OK
      Applying auth.0001_initial... OK
      Applying admin.0001_initial... OK
      Applying contenttypes.0002_remove_content_type_name... OK
      Applying auth.0002.Alter_permission_name_max_length... OK
      Applying auth.0003.Alter_user_email_max_length... OK
      Applying auth.0004.Alter_user_username_opts... OK
      Applying auth.0005.Alter_user_last_login_null... OK
      Applying auth.0006_require_contenttypes_0002... OK
      Applying sessions.0001_initial... OK
```

Se uma mensagem como “*Starting development server at http://127.0.0.1:8000/*” aparecer no console, então significa que o servidor iniciou com sucesso. Após fazer isso, a sua janela do prompt se torna o terminal do servidor, o que te impede de executar comandos Python para o projeto. Para resolver isso, basta abrir uma nova janela cmd e entrar no seu virtualenv novamente, mantendo as duas janelas ao mesmo tempo. Para parar o servidor, basta usar o atalho Ctrl + C. Certifique-se também que algum outro programa (talvez servidores) não esteja usando a porta 8000, uma vez que ela é usada por padrão para estabelecer a comunicação HTTP e dará erro caso esteja em uso.

Agora, para testar o site, basta digitar o endereço `http://127.0.0.1:8000/` no browser e dar um enter. A página deverá se equiparar à que temos na **Figura 3**.

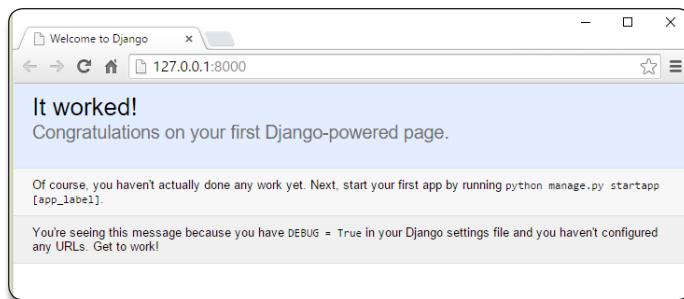


Figura 3. Tela inicial do projeto Django

Trabalhando com modelos

O Django faz amplo uso dos conceitos de orientação a objetos para manipular seus objetos e classes. A esse conjunto de premissas damos o nome de Modelos do Django. Um modelo no

Django é um tipo especial de objeto (muito semelhante ao que manipulamos em outras linguagens server side), pois ele é salvo no banco de dados. Um banco de dados para o Django é apenas uma coleção de dados. Este é o lugar onde salvaremos informações sobre nossos usuários, posts do blog, etc. Para o desenvolvimento do blog em questão, faremos uso do SQLite como banco de dados padrão, conforme vimos antes.

Para lidar com os modelos, precisaremos de um arquivo `models.py` que se encarregará de gerenciá-los. Porém, não vamos criá-lo manualmente, mas sim via um novo comando. O Python também disponibiliza um comando `startapp`, semelhante ao `startproject` que vimos, que traz consigo uma estrutura mais detalhada de arquivos `.py` como testes, views, etc. Portanto, dentro da pasta raiz, execute o seguinte comando:

```
python manage.py startapp django_app
```

Após isso, você verá que um novo diretório foi criado, tal como representado na **Figura 4**.

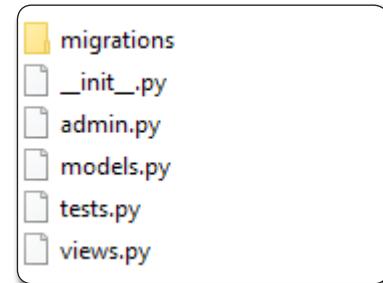


Figura 4. Estrutura de arquivos e diretórios gerada pelo comando `startapp`

Dentre os arquivos gerados, destacam-se:

- Outro `__init__.py`, servindo ao mesmo propósito que o criado anteriormente;
- `Models.py`, para armazenar os modelos que citamos de dados, bem como as entidades e relacionamentos entre elas;
- `Tests.py`, para armazenar uma série de funções para testar o código unitariamente;
- `Views.py`, para armazenar uma série de funções para recebem requisições dos clientes e retornam respostas;
- E `admin.py` onde você pode registrar seus modelos para que possa se beneficiar da maquinaria do Django para criar uma interface de administração padrão para você.

`Views.py` e `models.py` são os dois arquivos que você irá usar para quaisquer aplicações, e fazem parte do principal padrão de projeto arquitetural empregado pelo Django, ou seja, o padrão *Model-View-Template*.

Porém, antes que comece com a criação de seus próprios modelos e views, primeiro você deve informar ao seu projeto Django sobre a existência do seu novo aplicativo. Para fazer isso, precisamos modificar o arquivo `settings.py`, contido dentro de diretório de configuração do seu projeto.

Como criar um Blog com Django e Python - Parte 1

Abra o arquivo e encontre a tupla INSTALLED_APPS. Adicione o aplicativo django_app que acabamos de criar no fim da tupla, o que deve, então, parecer com o do exemplo exibido na **Listagem 4**.

Verifique que o Django carregou o seu novo aplicativo através da execução do servidor de desenvolvimento novamente. Se o servidor iniciar sem erros, então sua aplicação foi subida com sucesso e você estará pronto para avançar para a próxima etapa.

Listagem 4. Configurando settings.py para receber nova app.

```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'django_app',  
)
```

Criando uma View

Com a aplicação devidamente criada, vamos focar agora nas views. Como primeira view, vamos enviar apenas um simples texto de volta para o cliente como resposta a uma requisição. Portanto, abra o seu arquivo views.py, localizado dentro da nova aplicação criada, e remova o comentário “#Create your views here” (a propósito, comentários no Python começam assim) deixando o arquivo totalmente vazio. Em seguida, inclua o conteúdo da **Listagem 5** ao mesmo.

Listagem 5. Conteúdo da página views.py.

```
from django.shortcuts import render  
from django.http import HttpResponseRedirect  
  
def index(request):  
    return HttpResponseRedirect("<h1>Olá mundo, Django!</h1>")
```

A primeira coisa que temos de fazer é importar o objeto HttpResponseRedirect diretamente do módulo django.http, haja vista a necessidade de enviar um texto como resposta ao request. Todas as views do seu projeto devem ficar obrigatoriamente dentro deste arquivo, e cada uma deve ser declarada através da palavra reservada *def*, precedida do nome da view. Até o momento temos somente a view de nome index, mas você pode criar quantas quiser uma abaixo da outra, em forma de série.

Cada view, por sua vez, recebe pelo menos um argumento: um objeto HttpResponseRedirect, que também pertence ao mesmo módulo django.http. Por questões de convenção, normalmente o chamamos de *request*, mas você pode pôr o nome que desejar. Além disso, cada view retorna um objeto HttpResponseRedirect. Um objeto desse tipo simples assume a forma de uma string que representa o conteúdo da página que desejamos enviar de volta para o cliente que requisitou essa view.

Com a view criada, estamos a meio caminho andado de exibi-la finalmente ao cliente. Porém, antes que isso aconteça, precisamos

antes mapear a mesma a uma URL, para que seja entendida pelo sistema interno de navegação do Django.

Mapeando URLs

Ainda dentro do diretório da aplicação django_app, precisamos agora criar um novo arquivo chamado urls.py que nos permitirá mapear as URLs (por exemplo www.meuappdjango.com/django_app/) para views específicas. Após criado, adicione ao mesmo o conteúdo exposto na **Listagem 6**.

Listagem 6. Conteúdo da página urls.py.

```
from django.conf.urls import patterns, url  
from django_app import views  
  
urlpatterns = patterns(''  
    url(r'^$', views.index, name='index')
```

Este é praticamente todo o código que precisamos para importar o controle de URLs mapping do Django no projeto, atrelado ao import das views da aplicação django_app, o que nos fornece acesso completo às mesmas.

Para criar os mapeamentos, usamos as enuplas, mais conhecidas como n-tuplo ou tuplas, que nada mais são que sequências ordenadas de n elementos recursivos. Para o Django é obrigatório o nome urlpatterns, cujas tuplas contém uma série de chamadas à função django.conf.urls.url(). No exemplo, estamos usando a função url() apenas uma vez, o que significa que apenas um mapeamento foi definido, e assim sucessivamente. O primeiro parâmetro a ser passado à mesma função é uma expressão regular (no caso ^\$), que verifique se o texto casa com uma string vazia. Em suma, essa expressão define que todas as URLs fornecidas pelo usuário que casem com esse padrão estarão associadas à views.index(). A view será passado um objeto HttpRequest como parâmetro, contendo informações sobre a requisição do usuário para o servidor. Também fazemos uso do parâmetro opcional *name* da função url(), usando a string ‘index’ como valor associado. Esse parâmetro, por sua vez, serve para diferenciar um mapeamento de outro, uma vez que é inteiramente plausível que duas expressões de mapeamentos de URLs distintas possam acabar chamando a mesma view.

Você provavelmente tenha percebido que dentro do nosso “projeto” criado anteriormente no Django, já exista um arquivo urls.py, então por que dois? Tecnicamente, você pode colocar todas as URLs do seu projeto dentro desse arquivo. Entretanto, isso é considerado uma má prática, uma vez que aumenta o acoplamento em suas aplicações individuais. Um arquivo urls.py separado para cada aplicação permite que você possa configurar suas URLs de forma individual também. Com um baixo acoplamento, fica mais fácil migrar suas aplicações para futuras integrações com outros projetos.

Para corrigir isso, vamos abrir o arquivo urls.py localizado dentro do nosso projeto blog_site e atualize a tupla urlpatterns exibida conforme mostrado no exemplo da **Listagem 7**.

Observe que agora mapeamos tudo de forma centralizada e focada na aplicação django_app. O mapeamento adicionado agora busca por strings de URLs que casem com o padrão ^django_app. Quando o padrão é encontrado, o método include() entra na história recebendo o restante da string e direcionando-a à view correspondente.

Para testar, inicie novamente o servidor e acesse a URL http://127.0.0.1:8000/django_app/ e você verá a tela da **Figura 5**.

Listagem 7. Novo conteúdo da tupla urlpatterns.

```
"""blog_site URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/1.8/topics/http/urls/
Examples:
Function views
    1. Add an import: from my_app import views
    2. Add a URL to urlpatterns: url(r'^$', views.home, name='home')
Class-based views
    1. Add an import: from other_app.views import Home
    2. Add a URL to urlpatterns: url(r'^$', Home.as_view(), name='home')
Including another URLconf
    1. Add an import: from blog import urls as blog_urls
    2. Add a URL to urlpatterns: url(r'^blog/', include(blog_urls))
"""

from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^ django_app/', include('django_app.urls')),
]
```



Figura 5. Tela inicial do app criado no Django

Em suma, quando criamos um projeto com o Django, é o comando startproject que usaremos para isso. Um projeto é a soma de várias aplicações, mapeamentos de URLs, workflows, etc. E dentro de cada aplicação você fará quantas views forem precisas, assim como seus mapeamentos a cada URL.

Trabalhando com templates

Até o momento, juntamos algumas peças para criar as páginas no Django. Quando se trata de aplicações web, suas páginas

geralmente estão cheias de estruturas repetitivas, isto é, pedaços do HTML estrutural que estão presentes em todas ou grande parte da navegação do site. Exemplo disso são os famosos cabeçalho e rodapé, bem como notas de navegação, dentre outros. O Django fornece um recurso de templates para facilitar esse tipo de implementação aos desenvolvedores web, além de já separar a lógica de aplicação dos conceitos de apresentação.

Para fazer uso dos templates, é necessário antes que configuremos um diretório específico para isso, onde os mesmos deverão ser armazenados. No nosso projeto, crie uma nova pasta chamada templates dentro do projeto. E dentro deste, crie mais uma chamada Django, dessa forma o diretório poderá ser acessado pela URL relativa .../django_project/templates/django e é essa que usaremos para associar às nossas páginas.

Além disso, precisamos dizer ao Django que esse diretório existe e que ele está associado à casa dos templates do projeto. Portanto, vá novamente até o arquivo settings.py e adicione uma nova tupla TEMPLATE_DIRS ao mesmo, logo após a tupla TEMPLATES. Configure-a de acordo com o seu caminho absoluto, tal como na **Listagem 8**.

Listagem 8. Configurando templates no Django – settings.py.

```
TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [],
    'APP_DIRS': True,
    'OPTIONS': {
        'context_processors': [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
},
]

TEMPLATE_DIRS = ('D:/tests/django/workspace/django_project/django_project/templates/')
```

Note que somos obrigados nessa configuração a informar o caminho absoluto do diretório de templates que criamos, logo se você estiver trabalhando em um time cada um com diferentes computadores e o código sincronizado, isso pode ser um grande problema, uma vez que URL é estática. Ou seja, temos diferentes usuários o que implica diretamente em diferentes caminhos para o diretório de workspace. Uma saída seria adicionar um template diferente para cada diretório em cada setup diferente, mas isso definitivamente não é uma boa prática porque deixaria a configuração hard-coded. A solução então seria usar Paths Dinâmicos. Vejamos um pouco mais sobre isso.

Paths Dinâmicos

Para solucionar o problema de caminhos estáticos no Python devemos fazer uso de algumas funções nativas que gerenciarão os

Como criar um Blog com Django e Python - Parte 1

paths para nós automaticamente. Dessa forma, podemos garantir que o caminho absoluto possa ser recuperado independentemente de onde nosso projeto estiver localizado, tornado nosso código o mais portável possível.

A partir do Django 1.7, o arquivo `settings.py` contém agora uma variável chamada `BASE_DIR`. Ela armazena o caminho do diretório onde o módulo `settings.py` do projeto está guardado. Ele pode ser obtido usando um atributo especial do Python, o `__file__`, que é configurado com o caminho absoluto do seu módulo `settings`. Após retornar o caminho, uma chamada à função `os.path.dirname()` provê a referência para o caminho absoluto do diretório. Em seguida, se chamarmos essa função novamente, ela remove uma camada, e podemos navegar por entre elas, como se estivéssemos usando o comando `cd` do cmd.

Vejamos como funciona. Crie uma nova variável no `settings.py` chamada `TEMPLATE_PATH` e adicione o seguinte conteúdo à mesma:

```
TEMPLATE_PATH = os.path.join(BASE_DIR, 'templates')
```

Perceba que fizemos uso da função `os.path.join()` para juntar o conteúdo da variável `BASE_DIR` à string '`templates`'. Dessa forma, podemos agora modificar o caminho *hard-coded* que criamos antes pelo seguinte:

```
TEMPLATE_DIRS = (
    TEMPLATE_PATH,
)
```

Adicionando o template

Após tudo configurado, crie um novo arquivo chamado `index.html` (sim, o exemplo com a view estava usando código HTML estático e *hard-coded*, mas na vida real usamos as páginas comuns) e o coloque dentro do diretório `template/django`. Adicione o conteúdo da [Listagem 9](#) ao mesmo.

Listagem 9. Código HTML da página de index.

```
01 <!DOCTYPE html>
02 <html>
03
04   <head>
05     <title>Django Index</title>
06   </head>
07
08   <body>
09     <h2>Django diz pra você:</h2>
10     olá fulano! <strong>{{ msgnegrito }}</strong><br />
11     <a href="/django_app/sobre/">Sobre</a><br />
12   </body>
13
14 </html>
```

Veja que temos apenas um HTML comum, exceto pelo uso de uma estrutura nova na linha 10: `{{ msgnegrito }}`. Trata-se de uma variável de template do Django (elas sempre devem vir nesse

formato, com as chaves duplas) e será usada mais à frente para substituir o conteúdo HTML final.

Para usar esse template precisamos reconfigurar a view `index()` que criamos antes. Em vez de exibir uma simples mensagem, modificaremos seu código para redirecionar o fluxo de navegação diretamente para o nosso template. Por isso foi importante mantermos o conteúdo do `import render` no arquivo de `views.py` quando criamos o nosso código. Vá até esse arquivo e altere a view `index` para a demonstrada na [Listagem 10](#).

Listagem 10. Configurando `views.py` para redirect.

```
from django.shortcuts import render
from django.http import HttpResponseRedirect

def index(request):
    # Constrói um dicionário para passar ao template seu contexto.
    # Note que a chave 'msgnegrito' representa o {{ msgnegrito }} no template!
    dicionario_contexto = {'msgnegrito': "Testando fonte em negrito..."}

    # Retorna uma resposta renderizada para enviar ao cliente.
    # Fazemos uso da função de atalho para facilitar tudo.
    # Note que o primeiro parâmetro é o template que desejamos usar.
    return render(request, 'django/index.html', dicionario_contexto)
```

Primeiro, construímos um dicionário de pares de chave-valor que queremos usar junto com o template, então chamamos a função utilitária `render()`, que, por sua vez, se responsabiliza por receber como entrada o `request` do usuário, o nome do arquivo de template, e o dicionário de contexto. Essa função receberá esses dados e os juntará ao template para produzir uma página HTML final completa. Esta, então, é retornada e despachada para o browser web do usuário.

Quando um arquivo de template é carregado no sistema de templating do Django, um contexto de template é criado. Em outras palavras, um contexto de template é essencialmente um dicionário do Python que mapeia os nomes das variáveis dos templates com variáveis do Python. No exemplo que criamos há pouco, incluímos uma variável chamada `msgnegrito` que, em contrapartida, será mapeada para a variável `{{ msgnegrito }}` do nosso arquivo HTML de template.

Agora, reinicie o servidor e execute a URL `http://127.0.0.1:8000/django_app/` novamente no browser e você verá a tela da [Figura 6](#).



Figura 6. Tela com template criado no Django

Inserindo elementos estáticos

Quando falamos em elementos estáticos estamos nos referindo a tudo que um website pode usar de arquivo não dinâmico, como imagens, CSS, arquivos de script, vídeos, arquivos Flash, dentre outros. No Django, tais arquivos são servidos de uma forma um tanto quanto diferente das páginas web convencionais, já que não são disponibilizados via requests simples como temos no HTML comum.

Assim como para os templates, também precisamos criar um diretório específico para esse tipo de mídia. Portanto, crie um novo diretório chamada `static` dentro do seu projeto e, dentro deste, um novo chamado `img`. Em seguida, ponha uma imagem qualquer dentro desse diretório para fazermos alguns testes. Com o diretório estático criado, precisamos informar ao Django sobre o mesmo, da mesma forma que fizemos com o diretório de templates. Vá até o `settings.py` e atualize duas variáveis, `STATIC_URL` e `STATICFILES_DIRS`, tal como fizemos na **Listagem 11**.

Listagem 11. Configurando variáveis de diretório estático.

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.8/howto/static-files/

STATIC_PATH = os.path.join(os.path.dirname(__file__), 'static')

STATIC_URL = '/static/'

STATICFILES_DIRS = (
    STATIC_PATH,
)
```

A primeira variável representa a URL base que as aplicações do Django usarão para encontrar os arquivos de mídia estáticos quando o servidor estiver em execução. Por exemplo, ao rodar um servidor Django com a variável `STATIC_URL` configurada para `/static/` tal como no exemplo que criamos, as mídias estáticas estarão disponíveis no endereço `http://127.0.0.1:8000/static/`.

Enquanto a variável `STATIC_URL` define a URL de acesso às mídias via servidor web, a variável `STATICFILES_DIRS` permite especificar o local do diretório recentemente criado `static`. Logo, da mesma forma que a variável `TEMPLATE_DIRS` exige caminhos absolutos em sua configuração, também é com essa nova variável.

Para testar, basta acessar a imagem via URL `http://127.0.0.1:8000/static/images/logo.png` e o resultado será semelhante ao da **Figura 7**.

Agora que temos nosso sistema de redirecionamento automático de URLs para arquivos estáticos, podemos começar a adicioná-los em nossos templates, afinal é lá que precisaremos deles. Portanto, abra novamente o arquivo `index.html` e modifique seu conteúdo tal como temos na **Listagem 12**.

Antes de fazer uso das mídias estáticas, precisamos informar ao Django que queremos fazer isso. É o código da linha 3 o responsável por importar os arquivos estáticos na página e os fazer disponíveis para uso posterior. Não esqueça sempre das chaves {} para abrir e fechar as tags de template do Django.



Figura 7. Acessando imagem estática via URL /static/

Listagem 12. Novo conteúdo da página de template.

```
01 <!DOCTYPE html>
02
03 {% load staticfiles %} <!-- Nova linha que carrega os arquivos estáticos -->
04
05 <html>
06
07   <head>
08     <title>Django Index</title>
09   </head>
10
11   <body>
12     <h2>Django diz pra você:</h2>
13     Olá fulano! <strong>{{ msgnegrito }}</strong><br />
14     <a href="/django_app/sobre/">Sobre</a><br />
15     
        <!-- Nova imagem recuperando arquivo direto de diretório estático -->
16   </body>
17
18 </html>
```

Em seguida, configuraremos uma nova tag `` que receberá em seu atributo `src` o conteúdo da variável `STATIC_URL` combinado com a imagem `logo.png` em si. O resultado final a nível de HTML na página será algo como:

```
 <!-- Nova imagem recuperando arquivo direto de diretório estático -->
```

Basta reiniciar o servidor e rodar a página `django_app` novamente e você verá o resultado expresso na **Figura 8**.

Também podemos usar para importar quaisquer outros tipos de arquivos, como arquivos JavaScript e CSS, por exemplo.

Suponha que temos uma página HTML de formulário com três campos: nome, endereço e CPF. Essas três informações devem ser validadas por um script JavaScript comum, mas precisamos fazer o processo todo via Django API. Como primeiro passo, vamos criar um arquivo de cabeçalho que conterá o mesmo conteúdo do `index`, com a logo, um menu e uma mensagem de boas-vindas, apenas para vermos como funciona o import de templates uns nos outros no Django. Nomeie-o como `cabecalho.html` e adicione o conteúdo da **Listagem 13** no mesmo.

Como criar um Blog com Django e Python - Parte 1

Listagem 13. Conteúdo do arquivo de template de cabeçalho.

```
01 <!DOCTYPE html>
02
03 {% load staticfiles %} <!-- Nova linha que carrega os arquivos estáticos -->
04
05 <html>
06
07   <head>
08     <title>Django Form Exemplo</title>
09     <style>
10       body {
11         font-family: Segoe UI;
12       }
13       .logo {
14         float: left;
15       }
16       .logo img {
17         width: 250px;
18         padding: 20px;
19     }
20     .menu {
21       float: right;
22       padding: 40px;
23       font-size: 15pt;
24     }
25     .menu a {
26       margin-left: 15px;
27       margin-right: 15px;
28   }
29   .bemvindo {
30     clear: both;
31     padding: 0 20px;
32   }
33
34 </style>
35 </head>
36
37 <body>
38   <div class='logo'>
39     
40   </div>
41
42   <div class='menu'>
43     <a href="/django_app/sobre/">Sobre</a>
44     <a href="/django_app/sair/">Sair</a>
45   </div>
46
47   <div class='bemvindo'>
48     <h2>Bem vindo ao nosso Site Django Exemplo!</h2>
49   </div>
50
51 </body>
52
53 </html>
```



Figura 8. Visualizando conteúdo estático nas páginas da Django app

Veja que apenas reestruturamos o conteúdo do arquivo de index. O resultado pode ser visualizado na **Figura 9**: repare que o estilo da página foi todo definido dentro do arquivo de template, na tag `<style>`, o que não é uma boa prática. Para resolver isso vamos criar um arquivo `cabecalho.css` dentro da pasta `static/css` do projeto (a pasta `css` também deve ser criada). Extraia o conteúdo dessa tag para o novo arquivo e substitua a mesma pelo seguinte trecho de código (import do arquivo físico de CSS):

```
<link rel='stylesheet' href='{% static "css/cabecalho.css" %}' />
```

O resultado será o mesmo visualizado na figura.

Agora que já entendemos o mecanismo de import estático, bem como sua comunicação com os diferentes tipos de diretórios do projeto, podemos prosseguir modificando o conteúdo da página de template `index.html`. Altere-a para o que temos na **Listagem 14**.



Figura 9. Tela de cabeçalho da aplicação

Veja que logo no início temos a referência para um novo arquivo de CSS (você deve criá-lo também e adicionar o conteúdo da **Listagem 15**). Além disso, para que não tenhamos de duplicar o código do nosso template de cabeçalho em todas as páginas, basta que usemos o comando `include` do Django (linha 13) referenciando em seguida o nome do arquivo de template (lembre-se que configuramos o nível de acesso aos templates somente até a pasta `templates`, portanto se faz necessário referenciar a pasta `django` diretamente no comando). O restante da listagem é HTML de formulário simples. Você pode verificar o resultado na **Figura 10**.

Migrando layout para o Bootstrap

O Bootstrap é um framework de estruturação de HTML e estilo via templates. É extremamente usado na comunidade e também é possível acoplá-lo aos nossos projetos Django. Basta que para isso, importemos os scripts (que também incluem o jQuery) e

Listagem 14. Novo conteúdo da página index.html.

```
01 <!DOCTYPE html>
02
03 [% load staticfiles %] <!-- Nova linha que carrega os arquivos estáticos -->
04
05 <html>
06
07   <head>
08     <title>Django Form Exemplo</title>
09     <link rel='stylesheet' href='[% static "css/form.css" %]' />
10   </head>
11
12   <body>
13     {% include "django\cabecalho.html" %}
14
15   <div>
16     <fieldset>
17       <legend>Formulário de Pessoa</legend>
18       <form>
19         <table cellpadding='5'>
20           <tr>
21             <td>Nome:</td>
22             <td><input type='text' name='nome' id='nome' /></td>
23           </tr>
24           <tr>
25             <td>Endereço:</td>
26             <td><input type='text' name='endereco' id='endereco' /></td>
27           </tr>
28           <tr>
29             <td>CPF:</td>
30             <td><input type='text' name='cpf' id='cpf' /></td>
31           </tr>
32           <tr>
33             <td colspan='2'><input type='submit' value='Enviar' /></td>
34           </tr>
35         </table>
36       </form>
37     </fieldset>
38   </div>
39 </body>
40
41 </html>
```

arquivos CSS correspondentes à versão mais recente e criemos nosso HTML em detrimento deles.

Para transcrever nosso template para o framework, o primeiro passo é o cabeçalho. Vamos, portanto, alterar o código da página para o conteúdo na **Listagem 16**.



Figura 10. Tela final de formulário com import de cabeçalho

Listagem 15. Arquivo de código CSS da página de index.

```
fieldset {
  padding: 30px;
  margin: 0 auto;
  width: 50%;
}
input[type='text'] {
  width: 650px;
}
input[type='submit'] {
  font-family: Segoe UI;
  padding: 15px;
  width: 150px;
}
```

Perceba que nas linhas 10 e 11 estamos importando dois novos arquivos de CSS referentes ao core do Bootstrap minificado e ao estilo de telas de dashboard. No fim do arquivo vêm os arquivos JavaScript referentes ao jQuery e Bootstrap, respectivamente. É importante que o leitor se atenha a essa boa prática: sempre importar os CSSs no topo e os JSs no fim, para melhorar a performance de carregamento da página no browser.

Listagem 16. Arquivo HTML do template de cabeçalho com Bootstrap.

```
01 <!DOCTYPE html>
02
03 [% load staticfiles %] <!-- Nova linha que carrega os arquivos estáticos -->
04
05 <html>
06
07   <head>
08     <title>Django Form Exemplo</title>
09     <link rel='stylesheet' href='[% static "css/cabecalho.css" %]' />
10    <link href="http://getbootstrap.com/dist/css/bootstrap.min.css" rel="stylesheet">
11    <link href="http://getbootstrap.com/examples/dashboard/dashboard.css" rel="stylesheet">
12   </head>
13
14   <body>
15     <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
16       <div class="container-fluid">
17         <div class="navbar-header">
18           <a class="navbar-brand" href="[% url 'index' %]">Home</a>
19           <a class="navbar-brand" href="django/sobre/">Sobre</a>
20         </div>
21         <div class="navbar-collapse collapse">
22           <ul class="nav navbar-nav navbar-right">
23             <li><a class="navbar-brand" href="django/sair/">Sair</a></li>
24           </ul>
25         </div>
26       </div>
27     </div>
28     <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/
29       jquery.min.js"></script>
30     <script src="http://getbootstrap.com/dist/js/bootstrap.min.js"></script>
31   </body>
32 </html>
```

Como criar um Blog com Django e Python - Parte 1

Na linha 18 também temos o uso da expressão `{% url 'index' %}` que basicamente importa a URL de home da aplicação. O restante da implementação traz apenas classes do Bootstrap que você pode se inteirar melhor no site oficial (vide seção **Links**).

Agora é hora de modificar o template da página index para receber as alterações, inclusive formatando um formulário com o design do Bootstrap. Para isso altere o conteúdo da index.html para o representado na **Listagem 17**.

Nessa página mantemos os mesmos scripts Django de include, agora com a estrutura HTML de elementos e classes CSS correspondente ao Bootstrap. O resultado pode ser conferido na **Figura 11**.

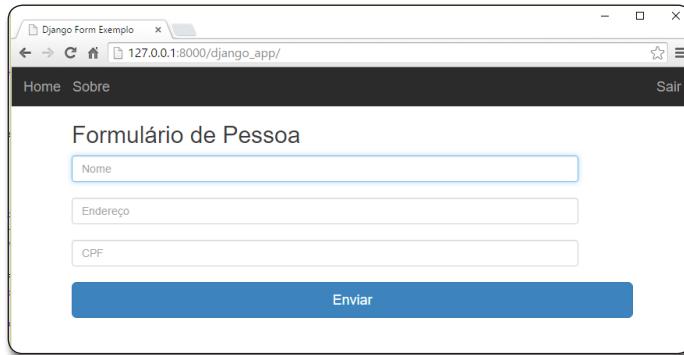


Figura 11. Tela final de formulário com estilo do Bootstrap

Veja como a simples inclusão do framework através de poucas linhas de código já trouxe uma diferença considerável para o projeto. E esse “esqueleto” que iremos usar para finalizar a nossa aplicação de blog. Até o momento você já aprendeu a criar novos projetos, usar a interface de linha de comando para virtualizar seu desenvolvimento, criar suas páginas, navegar entre elas, mapear URLs, importar plugins e outros frameworks, bem como gerenciar todo o conteúdo estático de suas aplicações Django. Esses são passos importantes no entendimento desse framework.

Agora precisamos aprender a como dinamizar todo esse conteúdo, através do uso efetivo dos modelos Django, sincronizar com o Python e o servidor e o nosso blog estará pronto. Também precisaremos criar uma tela de administração e veremos como fazer isso facilmente com as estruturas já fornecidas pelo próprio Django. Até a próxima e bons estudos!

Listagem 17. Código da página de index com o Bootstrap.

```
01 <!DOCTYPE html> {% load staticfiles %}
02
03 <html>
04
05 <head>
06   <title>Django Form Exemplo</title>
07   <link rel='stylesheet' href='{% static "css/form.css" %}' />
08 </head>
09
10 <body>
11   {% include "django\cabecalho.html" %}
12
13   <div class="container">
14
15     <form class="form-signin">
16       <h2 class="form-signin-heading">Formulário de Pessoa</h2>
17       <label for="nome" class="sr-only">Nome:</label>
18       <input type="text" id="nome" class="form-control" placeholder="Nome" required autofocus>
19       <br>
20       <label for="endereco" class="sr-only">Endereço:</label>
21       <input type="text" id="endereco" class="form-control" placeholder="Endereço" required>
22       <br>
23       <label for="cpf" class="sr-only">CPF:</label>
24       <input type="text" id="cpf" class="form-control" placeholder="CPF" required>
25       <br>
26       <button class="btn btn-lg btn-primary btn-block" type="submit">
27         Enviar
28       </button>
29     </form>
30   </div>
31
32 </html>
```

Autor



Júlio Sampaio

É analista de sistema e entusiasta da área de Tecnologia da Informação. Atualmente é consultor na empresa Visagio, trabalhando em projetos de desenvolvimento de sistemas estratégicos, é também instrutor JAVA. Possui conhecimentos e experiência em áreas como Engenharia de Software e Gerenciamento de Projetos, tem também interesse por tecnologias relacionadas ao front-end web.



Links:

Página oficial do Django Project.

<https://www.djangoproject.com>

Página de download do Python.

<https://www.python.org/downloads/>

Página oficial do Bootstrap.

<http://getbootstrap.com/>

Orientação a objetos com JavaScript

Conceitos avançados sobre a linguagem número um do front-end para uma programação mais eficaz

A orientação a objetos é uma velha conhecida dos desenvolvedores e está presente na grande maioria das linguagens de programação, tais como PHP, Java e JavaScript. Conhecer o conceito de orientação a objetos está comumente associado à sua linguagem de programação do dia a dia, ou à primeira linguagem em que você a utilizou.

Uma vez que se conhece esse conceito e se passa a aprender uma nova linguagem também orientada a objetos, os conhecimentos adquiridos previamente facilitam a transição para a nova linguagem, bem como carregamos conosco o conhecimento de instruções condicionais, laços e variáveis. No entanto, quando se trata de orientação a objetos em JavaScript, nem sempre a transição e compreensão são tão simples quanto gostaríamos.

Primeiro, porque a sua característica dinâmica e fraca-mente tipada permitem mais liberdade na construção de estruturas de dados, objetos e valores das variáveis, ou seja, não há uma restrição que force o desenvolvedor a seguir um certo padrão de programação. Essa liberdade exige maior atenção e, principalmente, disciplina do desenvolvedor.

Outro fator marcante é que, diferente da maioria das linguagens, o JavaScript é uma linguagem orientada a objetos baseada em **protótipos** e não em classes como as linguagens mais populares.

A **programação baseada em protótipos** é um estilo de programação orientada a objetos no qual o comportamento de reuso (também conhecido como herança) é aplicado através da clonagem de objetos existentes, que servem como protótipos.

Usemos como exemplo um **objeto animal**, que possui todas as características e comportamentos de um animal, tais como tamanho, peso, andar e comer. Com a clonagem desse objeto poderíamos dar origem ao **objeto mamífero**, que herdaria todas as características e comportamentos do objeto animal e agregaria suas próprias características particulares, tais como olhos, patas, amamentar e mamar.

Fique por dentro

A orientação a objetos é uma das partes fundamentais do JavaScript que, por ser uma linguagem dinâmica, acaba oferecendo muitas maneiras de se programar. Devido a esse dinamismo, as vezes acabamos não a utilizando da forma mais adequada (ainda que não exista uma forma certa ou errada). Portanto, nesse artigo entenderemos o que acontece por trás dos bastidores, além de ter um maior domínio sobre a linguagem e como utilizar alguns de seus conceitos mais avançados de forma mais adequada.

A partir daí cada **cachorro**, por exemplo, seria um novo clone do objeto **mamífero**.

Para entender bem como funciona a orientação a objetos no JavaScript, você precisa dominar, além dos **protótipos**, a função **call()** e compreender porque as **funções são consideradas "cidadãs de primeira classe"** no JavaScript.

É claro que se dominar outros aspectos da linguagem poderá tirar vantagem da combinação desses três pontos base com outros recursos, explorando ainda mais o potencial do JavaScript.

A importância das funções

Por ser uma linguagem de script feita para ser simples, o JavaScript faz uso das funções para diferentes propósitos, desde o seu uso tradicional até construtores de protótipos.

Um aspecto fundamental da linguagem é a utilização de funções para diversos propósitos, que vão muito além de mera funções que recebem argumentos e executam um trecho de código. Por terem essa característica “multiuso” as funções são chamadas de **cidadãos de primeira classe** da linguagem JavaScript.

Vejamos alguns exemplos:

- Funções são usadas para definir um protótipo e serem instanciadas posteriormente através da palavra-chave *new*, como veremos mais adiante.
- Funções podem ser auto executáveis, podendo utilizar valores locais, porém dentro de um contexto volátil, como podemos ver em IIFE (*Immediately Invoked Function Expressions*).

- Funções podem ser passadas como argumentos para outras funções.
- Em JavaScript funções são objetos, portanto, possuem outras funções e atributos, como por exemplo `toString()`, `call()`, `apply()`, `length`, `name`.
- Funções podem ser declaradas e executadas dentro de outras funções.

Apenas com os exemplos citados é possível perceber que, diferentemente de outras linguagens, as funções desempenham um papel bastante especial no JavaScript e é por essa razão que devemos dar a devida atenção a elas. E para nos aprofundar ainda mais nesse universo vamos falar um pouco mais sobre a importante função `call()`.

A função `call()`

A função `call` de uma função (lembre-se: em JavaScript as funções possuem funções) oferece um novo objeto “`this`” como seu objeto principal, dando-lhe assim um novo contexto. Quando chamamos a função `call()` de uma determinada função estamos dizendo que ela deve utilizar o contexto de outro objeto `this`, ao invés de usar o seu objeto original, do qual ela faz parte ou que foi instanciado a partir de um protótipo. Em outras palavras, é como se tivéssemos um **objeto A** e um **objeto B**, cada qual com suas propriedades e métodos.

Digamos então que o objeto **A** possui os atributos **valorBase**, **taxa** e **desconto** e um método chamado **calcular()**, que faz um cálculo envolvendo os três atributos do mesmo. Para chamarmos esse método usariamos **A.calcular()**, porém o **objeto B** possui os mesmos três atributos e com os mesmos nomes daqueles do **objeto A**, mas não possui o método **calcular()**.

Com a função `call()` aplicada ao método **calcular** do **objeto A** é possível executá-la utilizando o contexto do **objeto B**. Ou seja, a partir da instância do objeto **A** podemos executar o método **calcular** e “pegar emprestado” os valores do objeto **B** para realizar o cálculo, como é demonstrado na **Listagem 1**.

Listagem 1. Exemplo de uso da função `call`.

```
var A = {
    valorBase: 100,
    taxa: 10,
    desconto: 8,
    calcular: function() {
        return this.valorBase + (this.taxa - this.desconto);
    }
};

var B = {
    valorBase: 120,
    taxa: 25,
    desconto: 0
};

console.log(A.calcular());
console.log(A.calcular.call(B));
```

A função `call` recebe como primeiro argumento o novo objeto, cujo contexto será utilizado pela função, mas o número de argumentos original da função. Por exemplo, se a função **calcular** recebesse um argumento **multa**, a chamada com a função `call` ficaria da seguinte forma:

```
A.calcular.call(B, valorDaMulta);
```

Há vários cenários em que o uso da função `call` pode ser benéfico ao seu código, como propagar o reuso de código e servir na aplicação de *design patterns*, já que o JavaScript não possui um conceito explícito de interfaces e classes abstratas.

Nota

Outra função interessante é `apply()`, que faz a mesma coisa que a função `call`. A diferença entre elas é que a função `call` recebe o novo objeto `this` como primeiro argumento e um número variável de argumentos na sequência. Já a função `apply` recebe o novo objeto `this` como primeiro argumento e um array de argumentos como segundo argumento.

Por essa característica de aceitar uma quantidade desconhecida de argumentos, a função `apply` é bastante útil na criação de frameworks.

O papel da função `call` costuma ser bem claro e de fácil compreensão. Porém, além de ter que possuir um bom entendimento de objetos em JavaScript, o desenvolvedor precisa também conhecer os **protótipos**. E é justamente sobre eles que falaremos a seguir.

Protótipos

De acordo com a definição do MDN (*Mozilla Developer Network*), a programação baseada em protótipos é um estilo de programação orientada a objetos que dispensa a utilização de classes. Ao invés de utilizar a herança tradicional de linguagens baseadas em classes, a programação orientada a protótipos utiliza o **reaproveitamento de comportamentos**, determinado pela **extensão de objetos** e seus protótipos.

A extensão de objetos nada mais é do que a cópia de comportamentos de objetos já existentes para um novo objeto, cujos comportamentos também poderão ser copiados, e assim por diante. Essa cópia contínua de comportamentos é chamada de **“prototype chain”**, algo que poderíamos traduzir como **hierarquia de protótipos**.

À primeira vista pode parecer confuso entender por completo qual é a definição da programação baseada em protótipos. Portanto, para facilitar a compreensão veremos a seguir um exemplo de um protótipo.

Estrutura básica de um objeto JavaScript

Quando queremos definir um novo protótipo **criamos uma função**, que será posteriormente instanciada através da palavra reservada **“new”**. Na **Listagem 2** podemos ver um exemplo de definição de um protótipo **User** e a sua instanciação para um objeto **user1**.

Listagem 2. Estrutura de um protótipo e a instanciação para um objeto.

```
01. function User (id, name) {  
02.   this.id = id;  
03.   this.name = name;  
04. }  
05.  
06. User.prototype.getId = function () {  
07.   return this.id;  
08. };  
09.  
10. User.prototype.getName = function () {  
11.   return this.name;  
12. };  
13.  
14. var user1 = new User(1, 'Bruno');
```

No começo do código temos como construtor o próprio corpo da função, que recebe dois argumentos: **id** e **name**. Esses argumentos são atribuídos aos atributos **this.id** e **this.name** respectivamente.

Bem como nas linguagens orientadas a objeto tradicionais, a palavra-chave “*this*” é uma referência ao novo objeto recém instanciado. No nosso caso, *this* é uma referência à uma instância do protótipo User. No construtor do protótipo User, **this.id** e **this.name** são criados e atribuídos em tempo de execução do construtor. Ou seja, esses atributos farão parte do objeto **user1**, ainda que não estejam definidos no objeto *prototype* de User. Isso quer dizer que esses atributos não fazem parte da estrutura do protótipo, já que não foram explicitamente declarados.

Para que os atributos **this.id** e **this.name** pudessem fazer parte do protótipo User teriam que ter sido declarados da seguinte maneira, bem como as funções **getId** e **getName**:

```
User.prototype.id = null;  
User.prototype.name = null;
```

Em nosso código os atributos *id* e *name* foram inicializados apenas dentro do construtor propositalmente para demonstrar que eles não farão parte do protótipo, já que eles começam a fazer parte do objeto apenas no momento em que o construtor é chamado.

Não há necessariamente uma recomendação de boas práticas para essa situação, a menos que se deseje estender o protótipo no futuro e querer fazer uso desses atributos.

Na demonstração anterior estamos inicializando os valores dos atributos *id* e *name* com *null*, mas poderíamos ter utilizado quaisquer valores, já que o JavaScript é uma linguagem de tipagem fraca e não exige que atribuirmos tipos específicos de valores. No entanto, recomenda-se inicializar as variáveis com valores “vazios” de acordo com o tipo que se deseja utilizar para os atributos para manter uma maior legibilidade do código, bem como mantê-lo melhor estruturado.

Por exemplo, para o atributo *id* subentende-se que é um valor numérico, logo este poderia ter sido inicializado com 0 e o atributo *name* poderia ter sido inicializado com uma string vazia “”. Com exceção desses dois atributos, cada membro do **protótipo User** é adicionado ao objeto **prototype**, como é o caso das funções **getId** e **getName**.

O **prototype** é um objeto especial criado automaticamente quando declaramos uma função. Qualquer atributo que criamos a partir dele fará parte da estrutura de dados do objeto. Além disso, podemos adicionar qualquer coisa como membro de um protótipo como strings, números, objetos e funções.

O objeto **prototype** já possui um conjunto de propriedades e métodos que foram copiados automaticamente do **protótipo Object**, que é o protótipo base no topo da hierarquia do JavaScript. Em outras palavras, todo protótipo criado será copiado do objeto *Object*.

Na linha 14 instanciamos o protótipo User. Instanciar seria uma palavra mais apropriada se estivéssemos trabalhando com a orientação a objetos tradicional e é muito comum nos confundirmos, já que em JavaScript também usamos o operador “*new*” para “instanciar” um protótipo. Na verdade, quando usamos o operador “*new*”, estamos dizendo ao interpretador do JavaScript para criar uma cópia do protótipo e inicializar tudo o que o construtor estiver executando, mantendo a estrutura criada para o seu protótipo.

O motivo pelo qual o JavaScript utiliza o operador *new* para inicializar objetos é que na época de sua criação adotou-se uma sintaxe de linguagem que fosse familiar para os programadores da época. E esse é um dos motivos da sintaxe ser similar a C, C++ e Java.

Uma vez que instanciamos ou copiamos as características do protótipo User para o objeto **user1** e passamos os argumentos de inicialização ao construtor podemos finalmente trabalhar com o objeto.

Como já é de se esperar, podemos acessar os valores *id* e *name* através dos métodos **user1.getId()** e **user1.getName()**, ainda que os valores **user1.id** e **user1.name** não sejam privados e possam ser acessados diretamente. Mas isso é uma discussão que teremos mais adiante.

Prototype Chain

A instância de um objeto não passa de uma cópia sucessiva de objetos e seus comportamentos, ao qual damos o nome também de **prototype chain**. Uma maneira de visualizar essa herança de comportamentos pode ser observada através das ferramentas do desenvolvedor do Google Chrome.

Abra o Google Chrome e pressione F12 para ter acesso ao console e cole todo o conteúdo da **Listagem 1**, onde temos a definição de User e uma instância **user1**. Logo em seguida, exiba o conteúdo do objeto **user1** utilizando o comando **console.log**, como mostrado a seguir:

```
console.log(user1);
```

Após pressionar as setas que ocultam as heranças do objeto veremos algo semelhante ao mostrado na **Figura 1**.

Note que os atributos **id** e **name** estão imediatamente abaixo de **user1**, pois foram definidos em tempo de execução do construtor ao invés de serem previamente declarados no objeto *prototype* de User. Já os métodos **getId** e **getName**, acompanhados do método

`constructor`, são representados em um segundo nível hierárquico pela palavra `_proto_`.

Esse objeto `_proto_` é criado pelo browser para propósitos de visualização da hierarquia e não se recomenda utilizá-lo em seus programas, pois ele está marcado para ser removido em versões futuras do JavaScript.

Ainda no mesmo nível dos métodos `getId` e `getName` podemos ver mais um objeto `_proto_`, cujo valor é um objeto `Object`. Este objeto é o que está no topo da hierarquia dos protótipos do JavaScript e será sempre herdado, não importa o que façamos.

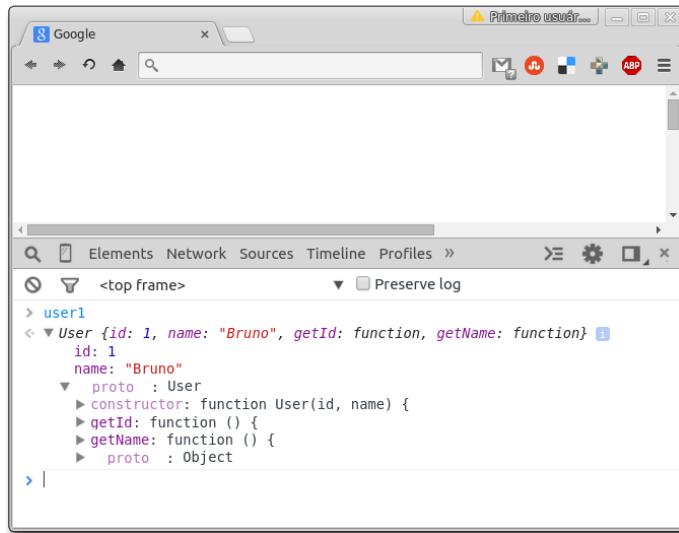


Figura 1. Demonstrando a herança de protótipos do objeto user1.

Herança de protótipos com a função call()

Agora que sabemos utilizar os protótipos para instanciar objetos e vimos como as funções podem, entre outras coisas, ser utilizadas como construtores de inicialização dos objetos, vamos aprender como usar a função `call()` para herdar comportamentos de um protótipo existente para um novo protótipo.

Já vimos como utilizar a função `call()` para reutilizar funções já implementadas, emprestando-as para um novo objeto e fazendo com que suas lógicas sejam aplicadas com os valores do novo objeto configurado.

O que faremos a seguir é muito semelhante, mas dessa vez, aplicando a função `call` em dois cenários um pouco diferentes:

1. Para executar o construtor do protótipo herdado no momento em que o novo protótipo for chamado;
2. Para sobrestrar métodos do protótipo herdado, complementando esses métodos com lógica adicional e reutilizando o conteúdo do método original;

Para demonstrarmos a herança criaremos um protótipo complementar ao protótipo `User`, chamado `AccountUser`. Este novo protótipo irá herdar o protótipo `User` aplicando as seguintes modificações:

1. Ele aproveitará os atributos `id` e `name`, bem como os métodos `getId` e `getName`;

2. Irá incluir os atributos `login` e `password`, já que estamos falando de um protótipo de conta de usuário e não mais um simples usuário;

3. Irá sobrestrar o método `getId` fazendo com que retorne o valor do protótipo original, concatenado ao nome do usuário.

O conteúdo do protótipo `AccountUser` pode ser analisado na [Listagem 3](#).

Listagem 3. Estrutura do protótipo `AccountUser`, que estende o protótipo `User`.

```
01. function AccountUser(id, name, login, password) {  
02.   User.call(this, id, name);  
03.   this.login = login;  
04.   this.password = password;  
05. }  
06.  
07. AccountUser.prototype = new User();  
08.  
09. AccountUser.prototype.getLogin = function () {  
10.   return this.login;  
11. };  
12.  
13. AccountUser.prototype.getPassword = function () {  
14.   return this.password;  
15. };  
16.  
17. AccountUser.prototype.getId = function () {  
18.   return User.prototype.getId.call(this) + this.name;  
19. };  
20.  
21. var accountUser = new AccountUser(2, 'Alex', 'aruiz', 'abcd1234');  
22. console.log(accountUser);  
23. console.log(accountUser.getId());
```

Como podemos notar há algumas diferenças entre o protótipo `User` e o novo `AccountUser`. A primeira está na linha 1, onde adicionamos os argumentos `login` e `password` ao construtor. Na linha 2 vemos a primeira utilização da função `call()`, onde chamamos o construtor do protótipo `User` e passamos como argumentos o objeto `this`, que é uma referência à uma instância de `AccountUser`, e os argumentos `id` e `name`, já conhecidos (e esperados) pelo construtor de `User`.

Como a função `call` funciona para funções, não utilizamos o operador `new` para referenciarmos o construtor de `User`. Nas linhas 3 e 4 atribuímos os valores dos novos atributos `login` e `password` da mesma maneira como fizemos com os atributos `id` e `name` no protótipo `User`, já que esses valores são novos e não faria diferença se os passássemos como argumento para o construtor de `User`.

Na linha 7 temos algo novo: estamos copiando todo o conteúdo do protótipo `User` para o protótipo `AccountUser`. É nessa linha que estamos herdando os atributos e comportamentos do protótipo `User`. Essa etapa, somada à utilização da função `call` para executar o construtor de `User` dentro do construtor `AccountUser`, caracteriza uma herança bem-sucedida em JavaScript.

Note que estamos utilizando o operador `new` para atribuir o protótipo de `User` à variável `prototype` de `AccountUser`. Isso é necessário pois, se simplesmente copiarmos um objeto para o

outro estaremos apenas criando uma referência entre eles e não atribuindo um novo valor. Ou seja, se fizermos algo assim:

```
AccountUser.prototype = User.prototype
```

Toda vez que alterarmos algo no objeto `prototype` de `User` ou `AccountUser` a mudança ocorrerá em ambos, pois são o mesmo objeto com duas referências diferentes. Quando utilizamos o operador `new` uma cópia do protótipo original é criada e atribuída à variável `prototype` de `AccountUser`, garantindo assim que seu valor será exclusivo para aquele protótipo e que não seja uma mera referência ao objeto de origem.

Para visualizar melhor o que acontece na linha 7 imagine que algo muito semelhante com o demonstrado na Listagem 4 aconteceu, ainda que não seja exatamente dessa maneira na prática.

Listagem 4. Demonstração de como funciona a cópia (herança) de protótipos.

```
1. AccountUser.prototype = {  
2.   getId: function() {  
3.     return this.id;  
4.   },  
5.   getName: function() {  
6.     return this.name;  
7.   }  
8.};
```

Nas linhas 9 a 15 criamos os métodos `getLogin` e `getPassword`, que retornam respectivamente os valores de `login` e `password`, da mesma maneira como fizemos com `id` e `name`. Das linhas 17 a 19 usamos mais uma vez a função `call` para sobrecarregar o método `getId`, porém de uma maneira um pouco diferente das que vimos até agora.

Como a ideia é reutilizar o conteúdo do método `getId` do protótipo `User` precisamos seguir a mesma estrutura de dados, já que este método será instanciado posteriormente. Por isso, mantemos a estrutura completa do método reaproveitado, passando o nome do protótipo, seguido do objeto `prototype` e, finalmente, o atributo (que é uma função) `getId`.

Adicionamos então a função `call` da função `getId`, passando o argumento `this`, que aponta para o novo protótipo `AccountUser`. No final concatenamos ao retorno do `id` ao valor do atributo `name`, formando então o novo valor do método `getId` para o protótipo `AccountUser`. Na linha 21 criamos uma instância do novo protótipo `AccountUser` e na linha 22 exibimos o seu conteúdo.

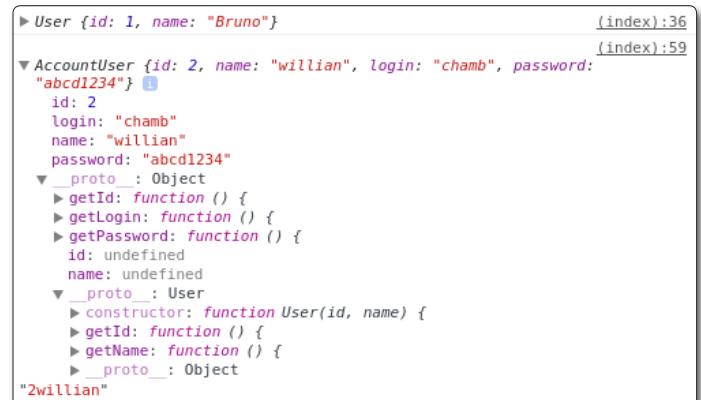
Finalmente, na linha 23 mostramos como fica o resultado do método `getId` após as alterações da sobrecarga deste método.

Se visualizarmos a estrutura do objeto `accountUser` que foi instanciado do protótipo `AccountUser` nas ferramentas de desenvolvedor do Google Chrome teremos algo parecido com o exibido na Figura 2.

Modificadores de acesso

Na orientação a objetos os modificadores de acesso são palavras chave que definem o nível de acesso a atributos, métodos e classes, para facilitar no encapsulamento de componentes,

ou seja, permitem ao desenvolvedor declarar se um membro é privado (cujo acesso é limitado à própria classe da qual faz parte) ou público (do qual o acesso pode ser feito por outros objetos de qual for a classe).



```
► User {id: 1, name: "Bruno"} (index):36  
  ↴  
  ► AccountUser {id: 2, name: "willian", login: "chamb", password: "abcd1234"} (index):59  
    ↴  
    id: 2  
    login: "chamb"  
    name: "willian"  
    password: "abcd1234"  
    ► __proto__: Object  
      ► getId: function () {  
      ► getLogin: function () {  
      ► getPassword: function () {  
        id: undefined  
        name: undefined  
        ► __proto__: User  
          ► constructor: function User(id, name) {  
            ► getId: function () {  
              ► getName: function () {  
                ► __proto__: Object  
                "2willian"
```

Figura 2. Demonstrando a herança de protótipos do objeto `AccountUser`

O encapsulamento é normalmente utilizado para proteger dados de uma classe (tornando-os privados), permitindo o acesso através de métodos públicos, comumente conhecidos como *setters* e *getters*. Essa proteção das informações garante que dados importantes da classe serão manipulados de forma correta, sem comprometer a estrutura dos mesmos.

Um exemplo onde a proteção dos dados evitaria problemas futuros no processamento das informações é o método `getId` que sobrecrevemos na Listagem 3, quando alteramos o seu retorno, adicionando a informação do nome do usuário, conforme podemos ver na linha 18. Note que o método sobreescrito retorna o valor original do método `getId` do protótipo pai `User`, concatenando seu valor com o nome do usuário, atribuído à variável `this.name`.

Essa variável é pública e pode ser alterada por qualquer componente externo que tenha acesso ao objeto do protótipo `AccountUser`, pondo em risco a integridade dos dados. Seria conveniente então que o nome do usuário, bem como o `id`, pudesse ser acessado apenas pelo próprio protótipo. Assim, o desenvolvedor que consumir esse objeto terá acesso ao novo `id` apenas através do método público `getId`.

Nota

Além dos modificadores privados e públicos, algumas linguagens possuem outros modificadores, tais como o `protected`, como é o caso do Java. Este modificador é usado em linguagens orientadas a objeto que possuem pacotes (packages) e permite o acesso ao membro pelo pacote e subclasses no qual ele faz parte.

Protegendo membros do objeto com IIFEs e closures

Já vimos a importância de se proteger membros de um protótipo e sua importância. O problema é que em JavaScript não existem modificadores de acesso como o `private` e `public`, mas então, como podemos atingir esse recurso?

Podemos fazer uso das IIFEs, ou funções auto executáveis para criar um escopo com variáveis e métodos locais, expondo publicamente apenas o que nos é relevante. Isso pode ser atingido com uma função auto executável que retorna o protótipo, porém mantendo os atributos desejáveis visíveis somente no escopo da IIFE.

Para facilitar a compreensão vamos modificar os protótipos User e AccountUser para ficar em conformidade com o modelo de encapsulamento, conforme pode ser visto nas **Listagens 5 e 6**.

Listagem 5. Versão aprimorada do protótipo User.

```
01. var User = (function() {
02.   var _id, _name;
03.
04.   function User (id, name) {
05.     _id = id;
06.     _name = name;
07.   }
08.
09.   User.prototype.getId = function () {
10.     return _id;
11.   };
12.
13.   User.prototype.getName = function () {
14.     return _name;
15.   };
16.
17.   return User;
18. })();
19.
20. var user1 = new User(1, 'Bruno');
21. console.log(user1.name); //retornará undefined
22. console.log(user1.getName()); //retornará o valor da variável privada _name
```

Listagem 6. Versão aprimorada do protótipo AccountUser.

```
01. var AccountUser = (function() {
02.
03.   //atributos privados, visíveis somente no escopo da IIFE
04.   var _login, _password;
05.
06.   function AccountUser (id, name, login, password) {
07.     User.call(this, id, name);
08.     _login = login;
09.     _password = password;
10.   }
11.
12.   AccountUser.prototype = new User();
13.
14.   AccountUser.prototype.getLogin = function () {
15.     return _login;
16.   };
17.
18.   AccountUser.prototype.getPassword = function () {
19.     return _password;
20.   };
21.
22.   AccountUser.prototype.getId = function () {
23.     return User.prototype.getId.call(this) + this.getName();
24.   };
25.
26.   return AccountUser;
27.
28. })();
29.
30. var accountUser = new AccountUser(2, 'Alex', 'aruiz', 'abcd1234');
31. console.log(accountUser);
32. console.log(accountUser.getId());
```

Nessa versão aprimorada do protótipo User, definimos o protótipo dentro de uma IIFE, a qual retorna o resultado da definição, visto na linha 17, para uma variável User declarada na linha 1.

Como toda função, a IIFE possui o seu escopo local, ou seja, todas as variáveis e funções declaradas dentro de si podem ser visualizadas apenas dentro de seu corpo, explicitando apenas o retorno da mesma, que no exemplo anterior expõe a definição do protótipo User.

Além da declaração local (ou privada, se preferir) do protótipo User, a IIFE também declara as variáveis `_id` e `_name` na linha 2. Essas variáveis substituirão os atributos, até então públicos, `this.id` e `this.name` da declaração antiga do protótipo User.

Nota

Por questões de convenção utilizamos um underscore `_` no começo do nome dos atributos para identificar que esses serão privados, já que não há uma maneira oficial de identificá-los em JavaScript.

Adotar essa convenção é opcional, pois facilita a leitura do código e identificação dos membros e seus níveis de acesso.

Para contemplar as novas mudanças o construtor foi alterado de forma a setar os valores de `_id` e `_name`, já que abolimos os atributos públicos `this.id` e `this.name`, como pode ser visto das linhas 4 a 7. Também foram comprometidos os métodos `getId` e `getName`, que sofreram uma alteração semelhante ao construtor, agora retornando respectivamente `_id` e `_name`, como mostrado nas linhas 9 a 11 e 13 a 15.

Note que agora a única forma de setar os valores do `id` e `name` é através do construtor, e os valores podem apenas ser lidos, através dos métodos `getId` e `getName`, impedindo assim que os mesmos sejam manipulados após a instância. Na linha 20 instanciamos um objeto de User, da mesma maneira como foi feito na versão original. Já na linha 21 tentamos acessar diretamente o valor `name`, sem utilizar o método público `getName`, porém, sem sucesso, já que esse atributo não mais é acessível.

Finalmente, na linha 22 acessamos o atributo `name` da forma correta, através do método `getName`, que é público e possui acesso direto à variável `_name`, já que ambos são membros do mesmo escopo local dentro da IIFE. Ao comportamento de se proteger valores locais no escopo de uma função e expor apenas os valores que queremos deixar públicos dá-se o nome de *closures*.

Na **Listagem 6** podemos ver como ficou a versão aprimorada do protótipo `AccountUser`, onde fazemos uso da IIFE, assim como fizemos com o protótipo `User`. De maneira geral, não há muitas diferenças na nova implementação de `AccountUser` em relação ao que vimos na listagem anterior. Porém, por se tratar de uma herança entre protótipos, há alguns pontos que precisamos prestar atenção:

- No construtor, note que antes do aprimoramento que fizemos no código, os atributos herdados de `User` eram públicos, ou seja, podiam ser facilmente acessados através da referência `this`, (`this.id` e `this.name`), pois ambos estavam declarados como parte do

protótipo *prototype* de User. Já nessa nova versão, como protegemos os valores *_id* e *_name* dentro da IIFE criada para o protótipo User, esses valores não podem mais ser acessados pelo protótipo estendido AccountUser.

• Na linha 22, onde temos a função *getId* que já havia sido alterada na primeira versão de AccountUser, foi preciso realizar uma mudança. Assim como mencionamos no item anterior, devido ao isolamento dos valores *id* e *name* pelo protótipo User e consequentemente por não termos mais acesso a esses atributos pelo protótipo AccountUser, foi necessário utilizar o método público *getName*, responsável por retornar o valor do nome do usuário.

Com o isolamento dos atributos dos protótipos conseguimos simular algo próximo de um modificador de acesso, mantendo então esses atributos privados.

No entanto, vale lembrar que diferentemente de uma programação orientada a objetos mais tradicional, ao se utilizar essa técnica, uma subclasse (ou no nosso caso, um subprotótipo) não pode acessar diretamente os atributos privados de sua “superclasse”, sendo necessário disponibilizar um método público para que possa então ter acesso a esses atributos.

Closures

Além de isolar atributos é possível também proteger outros elementos, tais como objetos e funções, tornando a aplicação menos propensa a erros. Isso se dá porque os *closures* dificultam o acesso a dados, que deveriam ter um acesso do tipo somente leitura, tais como dados de configuração da aplicação.

Para facilitar a compreensão imagine uma aplicação cujas informações de diretórios são compartilhadas por todo o sistema e, por isso, não podem ser alteradas para evitar erros. A **Listagem 7** demonstra a proteção e acesso a esses diretórios.

No exemplo queremos que o conteúdo do objeto *directories* possa ser lido sem riscos de ser alterado por ninguém, já que possui informações de configuração do sistema. Na linha 1 foi declarada uma variável *Config* que receberá o retorno da função auto executável.

Das linhas 3 a 9, já dentro do corpo da função auto executável, declaramos um objeto *directories*, que possui um mapa de chaves/valor contendo os diretórios que o sistema precisará acessar no decorrer de sua execução. Note que, por ser declarado dentro de uma função, sua visibilidade é local para aquela função, ou seja, o objeto *directories* não pode ser acessado fora do escopo dessa função. Ainda assim, ele pode ser acessado por outros elementos que compõem o mesmo escopo, como é o caso da função privada *_getList*. Essa função declarada entre as linhas 12 e 20 itera as chaves do objeto *directories* e monta um array com essas chaves, retornando uma lista de chaves conhecidas. A ideia é permitir ao usuário ter acesso a todas as chaves existentes na configuração do sistema.

Já a função privada *_getValue*, declarada nas linhas de 23 a 28, recebe uma chave como argumento, verifica se é uma chave válida do objeto *directories* e retorna o valor dessa chave e, caso

seja uma chave inexistente, retorna *null*. É através dessa função que o desenvolvedor pode acessar os valores do objeto *directories*. Porém, há algo errado: tanto o objeto *directories* quanto as funções só podem ser acessados no escopo da IIFE, logo ninguém fora desse escopo pode ler quaisquer valores.

Listagem 7. Usando closures na prática.

```
01. var Config = (function() {
02.   //dicionário *privado* de diretórios da aplicação
03.   var _directories = {
04.     images:'./resources/images',
05.     scripts:'./resources/js',
06.     libs:'./resources/libs',
07.     styles:'./resources/css',
08.     static_content:'static'
09.   };
10.
11. //função *privada* que retorna uma lista de chaves dos diretórios disponíveis
12. function _getList() {
13.   var keyList = [],
14.     key;
15.
16.   for(key in _directories) {
17.     keyList.push(key);
18.   }
19.   return keyList;
20. }
21.
22. //função *privada* que retorna o valor de um diretório a partir de sua chave
23. function _getValue(key) {
24.   if(key in _directories) {
25.     return _directories[key];
26.   }
27.   return null;
28. }
29.
30. //objeto público que permite a leitura de diretórios
31. return {
32.   getList:_getList,
33.   getValue:_getValue
34. };
35.)();
36.
37. var firstKey = Config.getList()[0];
38. console.log(Config.getValue(firstKey));
```

É preciso expor as funções de listagem de chaves e valores, de maneira a preservar e proteger o objeto *directories*, que deve ser somente para leitura. Para tal, começando na linha 31 até a 34, criamos um objeto que é o retorno da nossa função auto executável, cujo valor será atribuído à variável *Config* declarada na primeira linha da **Listagem 7**.

Esse objeto retornado possui dois atributos: *getList* e *getValue*, dos quais recebem consecutivamente uma referência das funções privadas *_getList* e *_getValue*. Ou seja, através do retorno da função auto executável conseguimos tornar público somente aquilo que quisermos expor ao desenvolvedor.

Na linha 37 declaramos uma variável *firstKey*, que recebe como valor a primeira chave de configuração dos diretórios, através do método *Config.getList()[0]*. Este, por sua vez, retorna uma lista com todas as chaves e é por isso que adicionamos o índice 0 à chamada

Orientação a objetos com JavaScript

da função, fazendo com que o valor de `firstKey` seja `images`. Agora que descobrimos o nome da primeira chave, podemos imprimir o seu valor, através da função `Config.getValue(firstKey)`, como é demonstrado na linha 38.

As closures também são ótimas estruturas para a meta-programação (conceito aplicado a programas que lidam com dados ou lógicas de outros programas, agilizando o desenvolvimento em alguns casos, ao evitar que todo o código seja escrito manualmente). Por exemplo, imagine que você tenha de manipular algum código que mapeie as teclas do teclado do seu computador, mas você não deseja ter que, mais uma vez, pesquisar a relação de *key codes* na internet para seguir com a implementação. Uma técnica interessante é usar um mapa de keys (chaves), tal como vemos na **Listagem 8**.

Dessa forma, quando precisarmos fazer qualquer checagem a nível de JavaScript, só precisamos verificar a tecla em específico no nosso mapa (**Listagem 9**).

Listagem 8. Exemplo de mapa de chaves com closures.

```
var KeyMap = {  
    "Backspace":8,  
    "Tab":9,  
    "Ctrl":17,  
    "Alt":18,  
    "Delete":46  
};
```

Listagem 9. Exemplo de uso do mapa de chaves com closures.

```
var texto = document.getElementById('meuInputTexto');  
texto.onkeypress = function(e) {  
    var codigo = e.keyCode || e.which // código usado para pegar a tecla pressionada  
    if (codigo === KeyMap.Tab) {  
        alert(texto.value);  
    }  
}
```

Veja como o conteúdo em formato “chave”:valor rapidamente se transforma em uma espécie de *enumeration* para o JavaScript, simplificando em muito o nosso trabalho. Esse exemplo, contudo, é relativamente simples, podemos fazer uso da meta-programação e closures para criar uma solução ainda melhor. Usando o nosso objeto `KeyMap` podemos gerar algumas funções muito úteis, como a que temos representada na **Listagem 10**.

As closures são recursos poderosos em detrimento da possibilidade de capturar variáveis locais e associações de parâmetros da função em que estão definidos. No exemplo, o loop gera uma função `is` para cada chave no `KeyMap` e nossa função `input.onkeypress` pode se tornar um pouco mais legível, como podemos ver na **Listagem 11**.

Apesar de todo o código, a closure que estamos de fato interessados é que está interna à função principal, funcionando como uma função anônima:

```
return function(ev) {  
    var codigo = ev.keyCode || ev.which;  
    return codigo === comparador;
```

Listagem 10. Função criada a partir do mapa de chaves com closures.

```
for (var chave in KeyMap) {  
    // acessamos o objeto com o acessor do vetor para configurar o nome da função  
    // dinâmica  
    KeyMap["is" + chave] = (function(comparador) {  
        return function(ev) {  
            var codigo = ev.keyCode || ev.which;  
            return codigo === comparador;  
        }  
    })(KeyMap[chave]);  
}
```

Listagem 11. Exemplo de uso da função `is` criada a partir das closures.

```
var input = document.getElementById('meuInputTexto');  
input.onkeypress = function(e) {  
    if(KeyMap.isCtrl(e)) {  
        alert(input.value);  
    }  
}
```

Lembre-se, as funções são executadas com o escopo que foi usado quando elas foram definidas. O parâmetro `comparador` é associado ao valor `KeyMap` que foi usado durante a iteração do loop, assim como nossa closure aninhada é capaz de capturá-lo. Dessa forma, o código nos permite configurar a variável `codigo` sempre que precisamos checar qualquer key code, porém através de funções convenientes e muito mais legíveis.

Com todo esse conhecimento acerca das closures, é mais que evidente que elas são relativamente fáceis de implementar e são vitais para o JavaScript. Vejamos agora alguns exemplos onde elas se atrelam a tipos nativos, a começar pelas funções dos objetos, como o tipo `Function` (**Listagem 12**).

Listagem 12. Exemplo de closure com prototypes.

```
Function.prototype.cached = function() {  
    var siMesmo = this, // "this" se refere à função original  
        temp = {}; // nosso local, de armazenamento de cache  
    return function(args) {  
        if (args in temp) return temp[args];  
        return temp[args] = siMesmo(args);  
    };  
};
```

Essa pequena função permite a cada outra função criar uma versão cacheada de si mesma. Você pode observar no código que a função retorna ela mesma, logo esse tipo de implementação pode muito bem ser aplicado a exemplos como podemos ver na **Listagem 13**.

Para testar o exemplo em questão, assim como os demais desenvolvidos até então, caso não deseje salvar o conteúdo em arquivos físicos JavaScript, você pode usar o site jsfiddle, que permite a execução online de conteúdo HTML, CSS e JavaScript (vide seção **Links**).

```
Math.cos = Math.cos.cached();
console.log(Math.cos(2)); // Resultado: -0.4161468365471424
console.log(Math.cos(2)); // Resultado: -0.4161468365471424 Mesmo valor mas dessa vez buscado direto do cache
```

Listagem 13. Exemplo de uso da função cached.

```
Function.prototype.cached = function() {
  var siMesmo = this, // "this" se refere à função original
      temp = {} // nosso local, de armazenamento de cache
  return function(args) {
    if (args in temp) return temp[args];
    return temp[args] = siMesmo(args);
  };
}
```

Ainda sobre o código, observe os recursos das closures que estão sendo usados. Temos uma variável local *cache* que é mantida privada e protegida do mundo exterior. Isso nos privará de ter o nosso cache invalidado.

Além disso tudo, as closures também são amplamente usadas em várias ferramentas de terceiros, bem como frameworks JavaScript famosos, como o jQuery, por exemplo. Imagine os cenários onde somos privados de usar o famoso operador de fábrica do jQuery \$ (como no universo WordPress, por exemplo) e precisamos usá-lo da mesma forma de sempre (sem precisar sobrescrever pelo jQuery), seja porque está migrando todo o fonte de outro projeto ou por qualquer outra razão.

Uma solução comum a esse problema é o uso do *jQuery.noConflict*, uma função nativa que permite definir qual operador será usado sem conflitos, porém também podemos usar closures para permitir funções de dentro a ter acesso ao nosso parâmetro de binding \$, tal como vemos na **Listagem 14**.

Listagem 14. Usando closures para resolver problema de binding com jQuery.

```
(function($){
  $(document).ready(function(){
    // código comum...
  });
})(jQuery);
```

Agora que entendemos em detalhes como funciona a orientação a objetos em JavaScript, faça algumas experiências, como por exemplo, realizar heranças em mais níveis, ou seja, crie um

protótipo Base (ex.: ProdutoBase) e crie um subprotótipo a partir dele. Uma vez que você tiver este novo protótipo, tente criar protótipos mais específicos a partir dele, criando uma hierarquia maior de protótipos (*prototype chain*).

Com os exemplos acima, pode-se notar o poder da função *call*. Não há restrições para o seu uso, ou seja, você pode estender funções de protótipos que nem sequer são o protótipo base do novo protótipo. Isso quer dizer que você pode “pegar emprestado” praticamente qualquer função de qualquer protótipo. Este caso pode ser caracterizado com uma herança múltipla, por exemplo, já que podemos reutilizar funções de qualquer lugar, mesmo que estejamos estendendo um único protótipo, ou até mesmo nenhum.

Você pode inicializar atributos dos protótipos no momento de sua definição, mas isso pode gerar problemas se você estiver usando tipos mais complexos como *arrays* e objetos. Isso ocorre porque esses tipos não serão duplicados de protótipo para protótipo e se você os alterar a qualquer momento, os protótipos pais serão afetados também. Nesse caso, sugere-se que se inicialize os valores dentro dos construtores, para evitar dores de cabeça futuras.

A orientação a objetos aplicada ao JavaScript nos permite tirar proveito de seu aspecto dinâmico para, digamos, burlar regras da mesma, comumente aplicadas a linguagens estáticas, obtendo assim um resultado tão bom quanto os atingidos nas linguagens estruturadas. Um exemplo disso é a herança múltipla através da função *call*.

Autor



Willian Carvalho

o.chambs@gmail.com

Programador desde 2000, trabalha com desenvolvimento para web e já passou por ASP, PHP e principalmente Java. Formado em Tecnologia da Informação pela FASP, atualmente é front-end engineer na Nagra, onde trabalha com JavaScript em aplicações para TV digital.



Links:

Programação orientada a protótipos do MDN

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

Site do JSFiddle

<https://jsfiddle.net/>

Como criar um chat com Node.js

Crie aplicações de comunicação em tempo real com Node.js

Emuito comum encontrarmos na internet artigos explicando a criação de sistemas de chats, mas a maioria faz uso de uma linguagem de servidor, algumas linhas de AJAX e um banco de dados, fora as inúmeras linhas de código. Normalmente esse tipo de projeto exige uma série de conhecimentos em linguagens de back-end, front-end e um pouco de AJAX.

Com a criação da plataforma Node.js, foi possível que surgissem também servidores web nos mais diversos protocolos de comunicação (HTTP, HTTPS, FTP, dentre vários outros) e dentre um deles, está o WebSocket, um protocolo de comunicação suportado por browsers exatamente com o propósito de estabelecer entre o navegador e o servidor uma comunicação bidirecional e em tempo real, possibilitando uma troca de mensagens mais ágil sem o refresh de página e um tempo de espera demorado.

Isso não lembra um pouco o AJAX? Infelizmente o WebSocket não é suportado por todas as versões dos browsers do mercado como o AJAX, mas as últimas versões do Internet Explorer, Firefox, Safari, Opera e Google Chrome já o suportam. No caso de navegadores mais antigos, seria necessário programar em protocolos similares como um plano B. Ao mesmo tempo, se desenvolvêssemos uma aplicação deste tipo puramente com o Node.js teríamos um trabalho muito maior.

Visando atingir a todos esses problemas, foi criado o módulo do Node.js, que cuida de todos os protocolos de transporte que podem servir de plano B para os navegadores mais antigos. Que seriam os seguintes (nesta ordem): Adobe Flash Socket, AJAX long polling, AJAX multipart streaming, Forever iframe e o JSONP Polling. Opção é o que não falta, então o próprio módulo fica a cargo de realizar a comunicação com o servidor pelo protocolo de transporte que lhe for mais conveniente.

Portanto, neste artigo vamos criar um projeto para conhecemos o básico do Socket.IO e o que ele pode nos oferecer, sem o uso de banco de dados, AJAX e alguma outra linguagem back-end, usando apenas JavaScript, o Node.js e o jQuery.

Fique por dentro

Aplicações de mensageria em tempo real são aplicações cada vez mais comuns nas realidades das empresas e suas aplicações web. Inúmeras são as soluções de mercado que proveem esse tipo de recurso, mas a maioria delas são complexas, envolvem tecnologias server side, bem como bancos de dados e outras configurações mais. Neste artigo você verá como criar um web chat completo usando apenas jQuery, Socket.IO e suas abstrações dos WebSockets, no estilo UOL ou o antigo MIRC.

Nosso Projeto

Antes de entrarmos nos detalhes técnicos, vamos entender o projeto que realmente vamos desenvolver no decorrer deste artigo. Trata-se de um aplicativo de salas de chat simples, igual ao antigo e popular serviço do MIRC e as salas de bate-papo dos antigos provedores de internet.

Os usuários primeiro irão se deparar com uma tela para inserir o apelido, assim que for inserido um válido (que não tenha nenhum outro usuário com o mesmo nome) será apresentada a tela de chat, com um campo de mensagem, um botão enviar, um painel onde aparecerão todas as mensagens e uma lista com o nome de cada um dos usuários.

Mas para não ficarmos dando rodeios em linhas intermináveis de instruções de código sem sentido, vamos realizar o projeto de forma iterativa, começando com uma aplicação bastante simples e funcional. Conforme as etapas forem completadas, vamos adicionar mais alguma funcionalidade, para assim termos um aprendizado onde acompanhamos o passo a passo do projeto já fazendo os testes de tudo.

Um pouco sobre Socket.io

Este módulo do Node.js traz uma forma de conexão direta do browser do cliente com o servidor de aplicação. A biblioteca funciona através de eventos, ou seja, o servidor ou o cliente irão disparar eventos para que haja respostas de uma das partes, veja uma exemplificação na **Figura 1**.

De certa forma, vamos usar dois métodos muitos básicos do módulo, que são o *emit* e o *on*. Um serve para efetuar a emissão do evento e o outro para receber a resposta do mesmo. Cada um dos lados da aplicação, portanto, terão a biblioteca Socket.IO adicionada.

Além de permitir a troca direta de mensagens entre dois dispositivos, o Socket.IO também permite o broadcast de mensagens, o envio de um evento a todos os outros usuários conectados. O broadcast pode ser tanto do cliente quanto do servidor, conforme demonstrado na **Figura 2**.

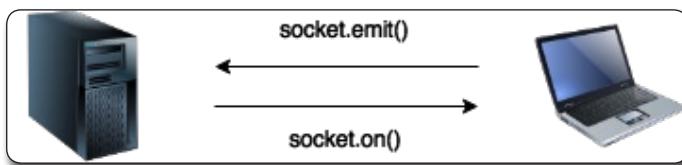


Figura 1. Troca de mensagens entre cliente e servidor

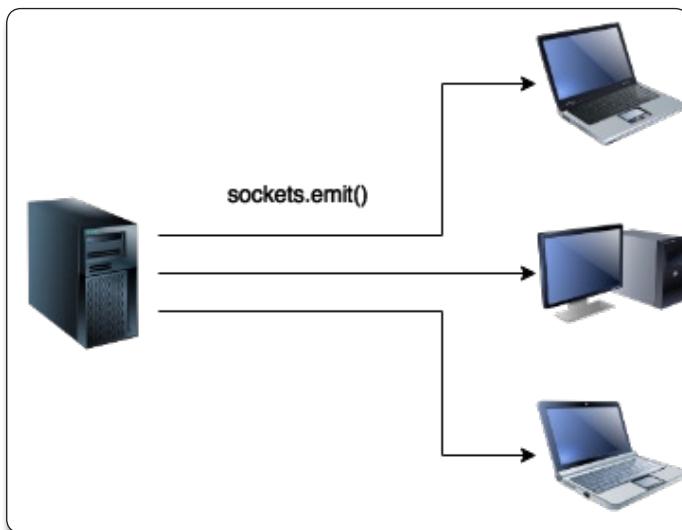


Figura 2. Broadcast de mensagem enviada pelo servidor

Quando o usuário acessar a página, um socket é criado com o servidor e é através deste socket que é realizada a troca de mensagens entre um cliente e um servidor. Este, por sua vez, pode tanto emitir um evento para um único Socket como para todos os sockets conectados a ele, o que chamamos de broadcast de mensagem.

Preparando o ambiente

Caso você não tenha ainda o ambiente do Node.js preparado, vamos então ver juntos nesta seção como deixar tudo pronto para começarmos a desenvolver aplicações do tipo. É um processo simples, já que vamos apenas instalar o Node.js, neste artigo não vamos precisar de banco de dados nem qualquer outra aplicação. Tudo se resumirá a um editor de texto simples e um terminal (ou prompt) do seu sistema operacional.

Windows/MAC

Acesse o site oficial do Node.js (vide seção **Links**) e baixe o arquivo de instalação de extensão msi para o seu tipo de Windows

(32 ou 64 bits), ou o arquivo .pkg para Mac. Abra o arquivo e execute a instalação normalmente. Ao final acesse o terminal (ou o prompt de comando) e execute o comando *node -v* (igual ao mostrado na **Figura 3**) e se a resposta for a versão instalada do Node.js isso quer dizer que o ambiente já está pronto.

Linux

Para o Linux, nas distribuições Debian e Fedora, o processo é bastante simples: a instalação é realizada pelo repositório de aplicações. Vejamos primeiro os comandos a serem executados no terminal do Ubuntu (válidos também para distribuições Debian), como mostra a **Listagem 1**. Na **Listagem 2** vemos os comandos para as distribuições Fedora.

Listagem 1. Comandos para instalação no Linux Ubuntu (Distribuições Debian)

```
sudo apt-get update  
sudo apt-get install Node.js  
sudo apt-get install npm
```

Listagem 2. Comandos de instalação para distribuições Fedora

```
sudo curl -silent -location https://rpm.nodesource.com/setup | bash -  
sudo yum -y install Node.js  
sudo yum -y install npm
```

Ao final do processo, basta acessar o terminal e digitar o comando *Node.js -v*: se a resposta for a versão instalada do Node.js então nosso ambiente já está pronto para começarmos. O segundo comando de instalação após o Node.js é referente ao NPM (*Node.js Package Manager*), um sistema para o gerenciamento de pacotes que é muito importante no desenvolvimento de aplicações Node.js.

Caso você tenha outra distribuição Linux, como OpenSuse, Gentoo, Arch, acesse a página de ajuda para instalação do Node.js no Linux (confira a seção **Links**). É importante também mencionar que no artigo colocamos o comando “node” em vez de “Node.js” por este ser o comando padrão do Windows. Tenha em mente que sempre que precisarmos inserir o comando node utilize o Node.js.

```
milleo@DevMedia:~$ nodejs --version  
v0.10.25  
milleo@DevMedia:~$ █
```

Figura 3. Conferindo a versão do Node.js

Começando o projeto

Para começar vamos criar um diretório chamado ChatJS onde lhe for mais conveniente, mas que seja de fácil acesso pelo prompt. É recomendável colocar o diretório do projeto no diretório */var/www/* (ou no *C:\www* para usuários Windows). Agora que já temos um diretório vamos criar um arquivo chamado *app.js*, que será o arquivo principal do nosso servidor.

Como criar um chat com Node.js

Como primeira parte vamos criar um servidor bastante simples que só vai apresentar na tela do navegador uma mensagem de sucesso, como mostra a **Listagem 3**.

O script cria um servidor HTTP (que estará escutando a porta 3000) que tem como método principal a ser requisitado a função resposta, que tem dois parâmetros: *req* (de requisição) e *res* (de resposta). Na resposta definimos um código 200 de sucesso e finalizamos a mesma com uma string avisando que o servidor está ok.

Listagem 3. Criando uma aplicação de servidor

```
var app = require('http').createServer(resposta);
app.listen(3000);
console.log("Aplicação está em execução...");
function resposta (req, res) {
  res.writeHead(200);
  res.end("Ola, o servidor esta funcionando corretamente.");
}
```

Logo após, vamos rodar o comando a seguir, que irá executar nossa aplicação no prompt cmd:

```
node app.js
```

Repare que quando executar este código no prompt ele não irá imprimir nenhuma outra linha, isto indica que nossa aplicação está em execução no momento.

Neste momento temos apenas o nosso servidor Node.js rodando, inclusive o terminal apresentou o conteúdo da função *console.log* avisando que a aplicação está em execução, conforme apresentado na **Figura 4**. Se você acessar no browser o endereço *http://localhost:3000/* ele só irá mostrar a mensagem que passamos no método *end*, como podemos observar na **Figura 5**.

```
milleo@DevMedia:/var/www/ChatJS$ nodejs app.js
Aplicação está em execução...
```

Figura 4. Mensagem que passamos via *console.log*

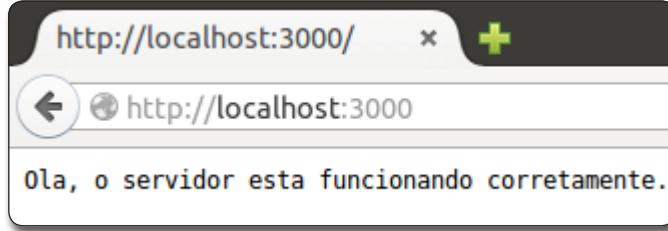


Figura 5. Resposta da aplicação no browser

Em seguida, vamos fazer nosso servidor apresentar uma resposta em HTML e que será a página principal do nosso chat. Para isso teremos de carregar o módulo do FileSystem, já que vamos navegar no diretório do projeto e abrir um arquivo. Portanto, vamos alterar o nosso *app.js* para que fique conforme a

Listagem 4. Antes de realizar as alterações, vá até o prompt e pressione Ctrl + C (ou command + C) para terminar a execução da nossa aplicação no servidor.

Listagem 4. Apresentando uma página HTML

```
var app = require('http').createServer(resposta);
var fs = require('fs');

app.listen(3000);
console.log("Aplicação está em execução...");
function resposta (req, res) {
  fs.readFile(__dirname + '/index.html',
    function (err, data) {
      if (err) {
        res.writeHead(500);
        return res.end('Erro ao carregar o arquivo index.html');
      }

      res.writeHead(200);
      res.end(data);
    });
}
```

Após estas alterações vamos novamente executar o comando *node app.js* no prompt. Ao acessarmos novamente o endereço *http://localhost:3000/* vamos nos deparar com a mensagem “*Erro ao carregar o arquivo index.html*” (**Figura 6**) isso por que ainda não temos um arquivo *index.html* dentro do nosso projeto.



Figura 6. Mensagem de erro para arquivo HTML não encontrado

É importante lembrar também que o servidor que criamos, até então, não diferencia o caminho, ou seja, você pode passar depois de *http://localhost:3000/* qualquer coisa que ele sempre irá responder da mesma forma porque não implementamos um modo de tratar estes caminhos. Logo, você pode muito bem chamar endereços como *http://localhost:3000/chat*, *http://localhost:3000/erro*, *http://localhost:3000/batata*, etc., que qualquer requisição que o servidor receber irá responder com o mesmo método (a função que chamamos de resposta, neste caso).

Vamos então criar uma interface bastante simples para o nosso chat. Crie um arquivo *index.html* dentro do diretório do projeto (diretório ChatJS). Neste arquivo insira um código igual ao demonstrado na **Listagem 5**.

Nosso *index*, por enquanto, só vai contar com uma div chamada *historico_mensagens* que é onde estarão dispostas todas as mensagens trocadas no chat e logo depois um formulário com uma caixa de texto e o botão de envio de mensagem. Uma estrutura bastante simples de chat até o momento.

Listagem 5. Código HTML da aplicação de chat

```
<!DOCTYPE html>
<html>
<head>
<title>ChatJS - FrontEnd Magazine - DevMedia</title>
<link rel="stylesheet" type="text/css" href="/css/style.css" />
</head>
<body>
<div id="historico_mensagens"></div>
<form id='chat'>
<input type='text' id='texto_mensagem' name='texto_mensagem' />
<input type='submit' value='Enviar mensagem!' />
</form>
</body>
</html>
```

Entretanto, se você agora tentar acessar o endereço `http://localhost:3000/` irá receber a mesma mensagem de erro. Isso acontece porque não reiniciamos nossa aplicação de servidor, então mais uma vez vamos até o prompt, pressionamos Ctrl + C e depois reexecutamos o comando `node app.js`.

Acostume-se com este procedimento **sempre** que realizar alterações de código no arquivo `app.js`. Já nos arquivos HTML e CSS não é preciso fazer isto porque eles atualizam com o *refresh* de página automaticamente. Agora que reiniciamos a aplicação de servidor a nossa página HTML está funcionando conforme a **Figura 7**.

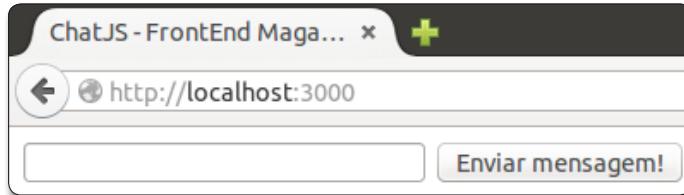


Figura 7. Página HTML sem estilo CSS

Como você deve ter percebido, já deixamos uma tag `link` na tag `<head>` da nossa aplicação para carregarmos o nosso CSS. Vamos então criá-lo para que fique com um design mais próximo de um chat. Dentro do diretório do nosso projeto crie um outro diretório chamado `css` e dentro dele crie o arquivo `style.css` com o conteúdo igual ao demonstrado na **Listagem 6**.

Se reiniciarmos a aplicação Node.js, o estilo ainda não estará aplicado à página index. A razão disso é que o nosso `app.js` só trata de um *path* de requisição até o momento. Para resolver isso vamos alterar o nosso arquivo `app.js` para que ele carregue os arquivos que são passados na URL da solicitação, ao invés de colocarmos cada uma das URLs manualmente. Vamos conferir melhor as alterações apontadas na **Listagem 7**.

Se reiniciarmos a aplicação node, desta vez o nosso sistema reconhecerá o estilo CSS que criamos anteriormente, conforme mostra a **Figura 8**.

Enviando mensagens

Agora temos o servidor funcionando, o estilo CSS na nossa página e toda a estrutura HTML pronta. Vamos a partir de agora

Listagem 6. Conteúdo do arquivo style.css

```
html, body{
  font-family: Arial, Tahoma, sans-serif;
  margin: 0;
  padding: 0;
}
body{
  background:#302F31;
  padding:10px;
}
form{
  margin:15px 0;
}
form input[type='text']){
  border:2px solid #45C5BF;
  border-radius: 5px;
  padding:5px;
  width:75%;
}
form input[type='submit']){
  background: #45C5BF;
  border:none;
  border-radius: 5px;
  color:#FFF;
  cursor:pointer;
  font-weight: bold;
  padding:7px 5px;
  width:19%;
}
#historico_mensagens{
  background: #FFF;
  border:2px solid #45C5BF;
  height: 550px;
}
```

Listagem 7. Alterações de caminhos no app.js

```
var app = require('http').createServer(resposta);
var fs = require('fs');

app.listen(3000);
console.log("Aplicação está em execução...");

function resposta (req, res) {
  var arquivo = "";
  if(req.url == "/"){
    arquivo = __dirname + '/index.html';
  }else{
    arquivo = __dirname + req.url;
  }
  fs.readFile(arquivo,
    function (err, data) {
      if (err) {
        res.writeHead(404);
        return res.end('Página ou arquivo não encontrados');
      }

      res.writeHead(200);
      res.end(data);
    });
}
```

trabalhar na função de envio de mensagens. Nossa aplicação vai funcionar se comunicando com o servidor node através da biblioteca client-side do Socket.IO com o jQuery fazendo a integração com a página.

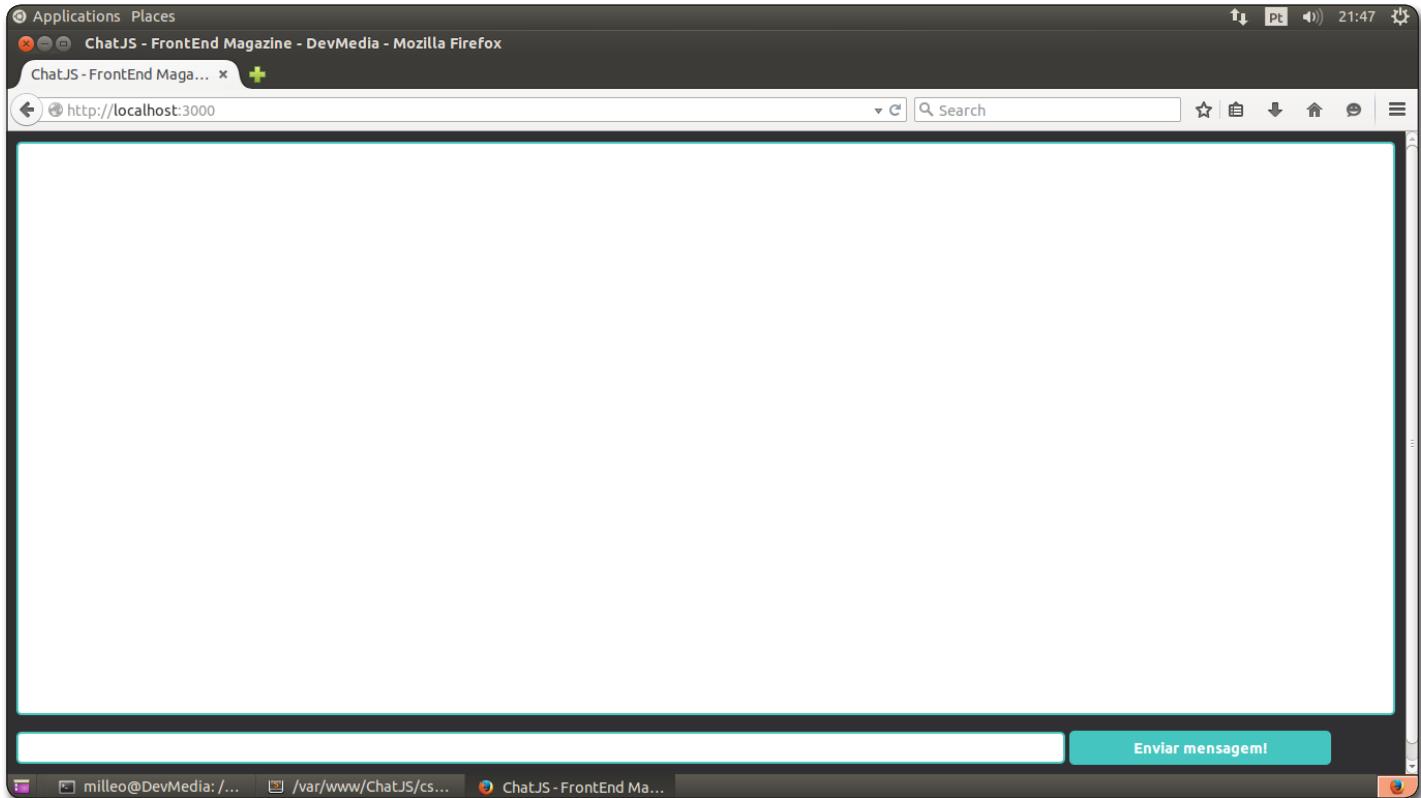


Figura 8. Layout da aplicação

Para isso vamos alterar o arquivo app.js, como demonstrado na **Listagem 8**, e incluir uma linha de um comando *require* logo no começo do arquivo informando que estamos incluindo na aplicação o Socket.IO e que o módulo será armazenado na variável *socket*.

Podemos ver que o *require* claramente chama o módulo socket.io e estamos passando a variável *app* (referente ao nosso servidor) no *require* do módulo a fim de facilitarmos uma parte do nosso desenvolvimento que veremos mais à frente.

Porém, para darmos um *require* em um módulo precisamos instalar o módulo na nossa aplicação. Para isso acessamos o terminal finalizando nossa aplicação Node.js com o Ctrl + C (ou command + C) e inserindo o seguinte código:

```
npm install socket.io
```

Logo depois de instalar rode o comando para iniciar nossa aplicação novamente e acesse o endereço <http://localhost:3000> para verificar se está tudo ok com a aplicação. Por enquanto nada estará funcionando porque não criamos nenhuma funcionalidade. Adicione também, antes do fechamento da tag de body do arquivo index.js, duas tags <script>, conforme a **Listagem 9**, com as nossas bibliotecas que realizarão as principais funções do chat. O jQuery facilitará o nosso processo de desenvolvimento e o Socket.IO para o client-side.

Você deve estar se perguntando o porquê de colocarmos uma tag script com o caminho para o Socket.IO que não existe, mas não se

preocupe, pois a biblioteca Socket.IO vai entender este caminho automaticamente e irá trazer para nossa aplicação a biblioteca client-side por si só.

Listagem 8. Incluindo o módulo Socket.IO

```
var app = require('http').createServer(resposta);
var fs = require('fs');
var io = require('socket.io')(app);
...
```

Listagem 9. Importando bibliotecas para nossa aplicação client-side

```
...
<script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/
jquery/2.1.4/jquery.min.js"></script>
<script type="text/javascript" src="/socket.io/socket.io.js"></script>
</body>
</html>
```

Lembre-se que já passamos antes aquela variável *app* no nosso *require*, exatamente para que a nossa aplicação se integre melhor com o módulo. Agora que está tudo pronto vamos ao envio e recebimento de mensagens. Primeiramente, vamos abrir uma tag script (desta vez sem o atributo *src*) onde estará todo o nosso código do lado cliente da nossa aplicação. Para começar temos que programar um evento de envio de mensagem, criando assim uma função que será atrelada ao *submit* do formulário de mensagem, como mostram as alterações na **Listagem 10**.

Listagem 10. Evento de envio de mensagens

```
...
<script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/
jquery/2.1.4/jquery.min.js"></script>
<script type="text/javascript" src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
var socket = io.connect();

$("form#chat").submit(function(e){
e.preventDefault();
socket.emit("enviar mensagem", $(this).find("#texto_mensagem").val(),
function(){
$("form#chat #texto_mensagem").val("");
});
});

</script>
</body>
</html>
```

No código JavaScript da página declaramos uma variável *socket* que é referente à biblioteca Socket.IO, que será responsável por todas as funcionalidades do socket. A seguir declaramos um evento *submit* do nosso formulário em jQuery e passamos um *preventDefault* para que o formulário não prossiga ao *action* do formulário, já que nós é quem vamos cuidar da resposta do formulário.

Em seguida, podemos ver que é invocado o método *emit* da biblioteca, no qual passamos como parâmetros três coisas: o nome do evento (isso será útil no servidor), os dados que estamos emitindo (no caso só estamos enviando o conteúdo do campo *mensagem*) e por último o *call-back*, uma função que vai ser executada assim que o evento for emitido. Este último, em específico, servirá apenas para limpar o campo de mensagem, assim o usuário não tem que ficar excluindo a mensagem depois que mandá-la.

Se testarmos agora nossa aplicação (reiniciando-a e acessando o <http://localhost:3000>) o envio de mensagens não vai funcionar, nem mesmo o *call-back* para limpar o campo de mensagem porque ainda não colocamos a funcionalidade do que o servidor tem que fazer assim que receber este evento. Para isso, edite o arquivo app.js colocando o código mostrado na **Listagem 11** no fim do mesmo.

Criamos um método que atuará em resposta à conexão do cliente ao servidor. Quando o cliente acessa a página ela dispara este método no servidor e quando este socket receber um método *Enviar Mensagem* acionamos um método que tem como parâmetros os dados enviados (o campo *mensagem*) e o *call-back* que criamos no lado cliente.

Dentro deste método colocamos a segunda parte da funcionalidade: o módulo vai emitir para todos os sockets conectados com o servidor (todos os usuários, por assim dizer) o evento *Atualizar Mensagens* e passará também qual mensagem nova foi enviada, com uma formatação de data e hora entre colchetes. Para fornecer a data e hora criamos uma função a parte porque ainda utilizaremos este método mais algumas vezes ao longo do desenvolvimento. Logo em seguida chamamos o *call-back* que criamos no lado cliente, que é o método para limpar os campos.

Finalmente, edite também o arquivo index.html e crie o método que vai atualizar as mensagens para os usuários. A ideia é bem simples: vamos dar um *append* na div *historico_mensagens* (as alterações se encontram na **Listagem 12**). As linhas a seguir devem ser inseridas logo depois do processamento do *submit* do formulário.

Listagem 11. Recebendo mensagens do cliente

```
...
io.on("connection", function(socket){
  socket.on("enviar mensagem",function(mensagem_enviada, callback){
    mensagem_enviada = "[" + pegarDataAtual() + "] " + mensagem_enviada;

    io.sockets.emit("atualizar mensagens", mensagem_enviada);
    callback();
  });
});

function pegarDataAtual(){
  var dataAtual = new Date();
  var dia = (dataAtual.getDate()<10 ? '0' : '') + dataAtual.getDate();
  var mes = ((dataAtual.getMonth() + 1)<10 ? '0' : '') + (dataAtual.getMonth() + 1);
  var ano = dataAtual.getFullYear();
  var hora = (dataAtual.getHours()<10 ? '0' : '') + dataAtual.getHours();
  var minuto = (dataAtual.getMinutes()<10 ? '0' : '') + dataAtual.getMinutes();
  var segundo = (dataAtual.getSeconds()<10 ? '0' : '') + dataAtual.getSeconds();

  var dataFormatada = dia + "/" + mes + "/" + ano + " " + hora + ":" + minuto + ":" + segundo;
  return dataFormatada;
}
```

Listagem 12. Atualizando histórico de mensagens

```
...
$("form#chat").submit(function(e){
  // Conteúdo da função
});
socket.on("atualizar mensagens", function(mensagem){
  var mensagem_formatada = $("<p />").text(mensagem);
  $("#historico_mensagens").append(mensagem_formatada);
});
```

O que percebemos aqui é que basicamente a conversa entre o servidor e o cliente é igual dos dois lados, isto é, os dois possuem eventos *emit* para emissão de eventos, e *on* para recepção de eventos. Acessando a aplicação no <http://localhost:3000> em duas abas (não se esqueça de reiniciar o aplicativo no prompt ou terminal antes de acessar no browser) é só enviar uma mensagem e ver o poder do Socket.IO em ação. A aplicação deve apresentar a mensagem como mostra a **Figura 9**.

A nossa aplicação ainda não atende a todas as necessidades, pois ainda não temos nomes de usuário, então a coisa fica um tanto quanto anônima, mas vamos cuidar disso agora.

[25/07/2015 22:04:56]: Olá? Tem alguém aí?

[25/07/2015 22:05:06]: Sim, estou aqui!

Figura 9. Envio de mensagens

Como criar um chat com Node.js

Dando nomes aos usuários

Toda aplicação de chat recebe o usuário com um formulário para inserir o apelido e nesta seção vamos criar um formulário simples onde o visitante vai colocar seu nome de usuário. Caso já exista alguém com o mesmo nome vamos então apresentar uma tela de erro avisando ao nosso visitante, do contrário, apresentaremos a tela de chat normalmente.

Para não ficarmos repetindo código e criar mais um arquivo HTML vamos programar nossa página index para que apresente primeiro o formulário de apelido. Se tudo der certo, a página esconde o formulário e apresenta a sala de chat, assim ganhamos tempo e linhas de código para aproveitar mais do nosso projeto.

Vamos começar então pela edição do arquivo index.html, embrulhando nossa sala de chat em uma div com o id `sala_chat`. Crie também outra div com o id `acesso_usuario` com um formulário bastante simples contendo apenas o campo de nome de usuário, conforme mostrado na **Listagem 13**.

Com o formulário inserido na página ainda precisamos editar o `style`. Portanto, faça as alterações vistas na **Listagem 14**.

Listagem 13. Adaptando a página principal para login de usuário

```
...
<div id='acesso_usuario'>
  <form id='login'>
    <input type='text' placeholder='Insira seu apelido' name='apelido' id='apelido' />
    <input type='submit' value='Entrar' />
  </form>
</div>
<div id='sala_chat'>
  <div id="historico_mensagens"></div>
  <form id='chat'>
    <input type='text' id='texto_mensagem' name='texto_mensagem' />
    <input type='submit' value='Enviar mensagem!' />
  </form>
</div>
...
...
```

Listagem 14. Alterando o CSS da aplicação

```
...
#sala_chat{
  display: none;
}
#acesso_usuario{
  height:30px;
  left:50%;
  margin-left:-160px;
  margin-top:-15px;
  position: fixed;
  top:50%;
  width:320px;
}
#acesso_usuario form{
  margin:0;
}
```

Se atualizarmos agora o navegador (como é alteração de HTML e CSS não é necessário reiniciar a aplicação rodando no nosso prompt) vemos que agora nossa aplicação só mostra um formulário como pretendímos, mas obviamente ele ainda não funciona, como mostra a **Figura 10**.

Figura 10. Formulário de entrada

Lembrando-se da implantação do nosso sistema de envio de mensagem podemos imaginar como vai funcionar o lado cliente desta funcionalidade, não? Vamos dar `emit` em um evento chamado `entrar`, que por sua vez, vai mandar para o servidor o nome do usuário que foi digitado. Já no retorno do servidor como `true` vamos simplesmente dar um `hide` no `form` e apresentar a nossa sala de chat.

Agora vejamos quais são as alterações que devemos fazer no nosso arquivo index.html, conforme a **Listagem 15**.

Nosso sistema agora emite o evento para o servidor, logo precisamos validar se o nome está disponível e depois armazená-lo junto aos nomes dos demais usuários. Portanto, vamos fazer as alterações conforme demonstrado na **Listagem 16**.

Listagem 15. Implantando funções client-side de acesso de usuário

```
...
$(“form#login”).submit(function(e){
  e.preventDefault();
  socket.emit(“entrar”, $(this).find(“#apelido”).val(), function(valido){
    if(valido){
      $(“#acesso_usuario”).hide();
      $(“#sala_chat”).show();
    }else{
      $(“#acesso_usuario”).val(“”);
      alert(“Nome já utilizado nesta sala”);
    }
  });
});
...
...
```

Listagem 16. Implantação da verificação de acesso dos usuários

```
var io = require(‘socket.io’)(app);
var usuarios = [];

io.on(“connection”, function(socket){
  socket.on(“entrar”, function(apelido, callback){
    if(!(apelido in usuarios)){
      socket.apelido = apelido;
      usuarios[apelido] = socket;
      callback(true);
    }else{
      callback(false);
    }
  });
  socket.on(“enviar mensagem”, function(mensagem_ enviada, callback){
    mensagem_ enviada = “[ “ + pegarDataAtual() + ” ]” + socket.apelido +
    “ diz: ” + mensagem_ enviada;
    io.sockets.emit(“atualizar mensagens”, mensagem_ enviada);
    callback();
  });
});
```

No começo podemos ver a declaração de um vetor chamado `usuarios`, que vai ser nossa base de nomes, uma estrutura bastante simples para armazenar apenas o nome de cada um que acessar nossa aplicação. Logo depois colocamos um método de resposta ao evento que criamos no client-side. Nos parâmetros passamos apenas o nome do usuário e o método *call-back* que criamos anteriormente.

Então fazemos uma verificação simples se o usuário consta no nosso vetor de nomes de usuários e se não constar primeiro criamos um atributo *apelido* dentro do socket, para assim acessarmos depois este valor em outros métodos. Também adicionamos um índice com o nome do usuário e armazenamos o *socket*, assim, podemos depois resgatar o mesmo através do nome de usuário; isso será útil mais adiante.

Além disso, também mudamos o formato da mensagem, colocamos o apelido do usuário que mandou a mensagem antes do texto propriamente dito. Confira como ficaram as alterações reiniciando a aplicação no prompt e posteriormente acessando o servidor local (<http://localhost:3000/>).

Agora vamos criar uma lista de usuários para sabermos quem está na nossa sala de chat e também colocar o nome do usuário na mensagem emitida, assim sabemos quem disse o quê.

Veja na **Figura 11** como irá ficar nossa lista de usuários, disposta ao lado direito do nosso painel de mensagens.



Figura 11. Lista de usuários

Para isto, devemos primeiro alterar o nosso `index.html` conforme a **Listagem 17**, colocando um `select` de tipo `multiple` (onde estará o nome dos usuários) que ficará ao lado direito do painel principal da conversa. Além do `select` precisamos também colocar a função que vai ser acionada quando o servidor emitir um evento para atualizar os usuários da lista.

Se atualizarmos o browser agora veremos que o design da página não ficou bom, por isso vamos alterar mais um pouco o nosso CSS conforme apresentado na **Listagem 18**. Para isso altere o estilo da div `historico_mensagens`, colocando junto o estilo da lista de usuários com alguns atributos novos e inserindo mais alguns trechos de código ao final do arquivo `style.css`.

Por fim, na **Listagem 19** vamos emitir o evento pelo servidor para atualizar a lista de usuários e mandar uma mensagem para todos avisando que um novo usuário entrou na sala.

Incluímos o código onde é emitido o evento para que todos os sockets avisem que a lista deve ser atualizada, além de termos passado todas chaves do nosso vetor de usuários. Logo em seguida também emitimos outro evento, que já utilizamos anteriormente, o *atualizar mensagens*. Desta vez especificamos a mensagem em si, passando o apelido do usuário.

Listagem 17. Adicionando lista de usuários e nome de usuário na mensagem enviada

```
...  
<div id="historico_mensagens"></div>  
<select multiple="multiple" id='lista_usuarios'><option value="">Todos</option></select>  
<form id='chat'>  
...
```

Listagem 18. Alterando o estilo da aplicação

```
...  
#historico_mensagens, #lista_usuarios{  
background: #FFF;  
border:2px solid #45C5BF;  
height: 550px;  
float:left;  
margin-bottom: 10px;  
width:75%;  
}  
...  
#historico_mensagens{  
border-right: 0;  
}  
#lista_usuarios{  
border-left: 1px solid #45C5BF;  
height: 554px;  
width: 20%;  
}
```

Listagem 19. Emitindo atualização de lista de usuários

```
...  
socket.on("entrar", function(apelido, callback){  
if(!(apelido in usuarios)){  
socket.apelido = apelido;  
usuarios[apelido] = socket;  
  
io.sockets.emit("atualizar usuarios", Object.keys(usuarios));  
io.sockets.emit("atualizar mensagens", "[" + pegarDataAtual() + "] " + apelido +  
" acabou de entrar na sala");  
  
callback(true);  
}else{  
callback(false);  
}  
});  
...
```

Vamos colocar o método de atualização dos usuários na nossa página `index.html`, conforme a **Listagem 20**. O código jQuery deve ser inserido no final do conteúdo da tag `<script>`.

Agora nossa aplicação de chat está ficando com cara de uma verdadeira sala de chat dos portais de internet com poucas linhas de código e o melhor, não utilizamos banco de dados, apenas simples comunicação entre cliente e servidor.

Vamos implantar mais uma funcionalidade que vai ser disparada quando o usuário sair da sala. Para isto vamos mais uma vez alterar o nosso arquivo `app.js` com a alteração da **Listagem 21** que será também colocada dentro do *call-back* do *io connection*.

Desta vez não precisamos criar um `emit` para este evento porque ele é nativo do Socket.IO. Além dele, temos o `connect` (quando o socket realiza uma conexão com o servidor) e o `message` (para o envio de mensagens - não utilizamos este método aqui no

Como criar um chat com Node.js

artigo para entendermos exatamente como funciona a criação de eventos).

Na função de *call-back* do *disconnect* removemos o socket armazenado e o apelido da lista de usuários. Logo após atualizamos a lista de usuários dos clientes e enviamos uma mensagem avisando que o usuário saiu da sala.

Listagem 20. Método de atualização da lista de usuários

```
...
socket.on("atualizar usuarios", function(usuarios){
  $("#lista_usuarios").empty();
  $("#lista_usuarios").append("<option value=>Todos</option>");
  $.each(usuarios, function(indice){
    var opcao_usuario = $("<option />").text(usuarios[indice]);
    $("#lista_usuarios").append(opcao_usuario);
  });
});
```

Listagem 21. Tratamento para quando o usuário sair da sala

```
...
io.on("connection", function(socket){
  ...
  socket.on("disconnect", function(){
    delete usuarios[socket.apelido];
    io.sockets.emit("atualizar usuarios", Object.keys(usuarios));
    io.sockets.emit("atualizar mensagens", "[" + pegarDataAtual() + "]"
      + socket.apelido + " saiu da sala");
  });
});
```

Finalizando o projeto

Precisamos atribuir uma funcionalidade à nossa lista de usuários que está ao lado direito do nosso painel de mensagens. Quando tivermos algum nome selecionado enviaremos uma mensagem privada ao usuário. Trata-se de uma alteração simples porque irá seguir o mesmo estilo da função de envio de mensagem, mas ao invés de usarmos um *emit* para todos os sockets, vamos emitir apenas para um socket em específico.

Temos de fazer uma alteração rápida também no método de emissão do evento “enviar mensagem”, porque estamos enviando apenas a mensagem que o usuário escreveu, mas a partir de agora teremos de mandar a mensagem e o usuário que ele selecionou. Conforme a **Listagem 22** vamos primeiro alterar o método *submit* do formulário de chat na página index.html.

E agora no servidor vamos adicionar o processamento para o usuário que deve receber a mensagem especificamente, como mostra a **Listagem 23**.

No programa, se o usuário for vazio, a mensagem é para todos e então colocamos o código para o *emit* ser enviado para todos os sockets. Mas se for para um usuário em específico vamos dar um *emit* no socket que está armazenado na lista de usuários com o usuário que foi enviado. Ele também será emitido para o próprio usuário que mandou a mensagem, para que a conversa fique visível para os dois.

Listagem 22. Alterando parâmetros a serem enviados no evento enviar mensagem

```
$(form#chat").submit(function(e){
  e.preventDefault();

  var mensagem = $(this).find("#texto_mensagem").val();
  var usuario = $("#lista_usuarios").val();

  socket.emit("enviar mensagem", {msg: mensagem, usu: usuario}, function(){
    $(form#chat #texto_mensagem").val("");
  });
});
```

Listagem 23. Enviando mensagem privada

```
socket.on("enviar mensagem", function(dados, callback){

  var mensagem_enviada = dados.msg;
  var usuario = dados.usu;
  if(usuario == null)
    usuario = "";

  mensagem_enviada = "[" + pegarDataAtual() + "] " + socket.apelido +
  " diz:" + mensagem_enviada;

  if(usuario == ""){
    io.sockets.emit("atualizar mensagens", mensagem_enviada);
  }else{
    socket.emit("atualizar mensagens", mensagem_enviada);
    usuarios[usuario].emit("atualizar mensagens", mensagem_enviada);
  }

  callback();
});
```

Agora vamos reiniciar nossa aplicação no terminal, abrir três abas diferentes no nosso navegador, entrar na sala nas três abas e verificar se está tudo ok com a sala de chat. Envie uma mensagem privada para um dos três usuários e uma mensagem para todos. O usuário que mandou as duas mensagens irá vê-las. O usuário que recebeu a mensagem privada também irá visualizar duas mensagens e o terceiro irá ver apenas uma, que foi dita a todos.

Contudo, ainda faltam duas coisas para o nosso chat: um aviso dizendo que a mensagem é privada e uma diferenciação das mensagens privadas, mensagens públicas e mensagens do sistema. Uma vez que já estamos concentrados em uma única função no client-side, o método de *atualizar mensagens*, vamos fazer uma alteração bastante simples nele.

Na **Listagem 24** podemos ver que a alteração consiste em apenas modificar os parâmetros de *atualizar mensagens* passando agora qual será a classe daquela mensagem: se é uma mensagem privada, uma mensagem de sistema, etc.

O lado ruim dessa implementação é que teremos de alterar todas as chamadas ao método *atualizar mensagens*. Devemos especificar, portanto, o tipo de mensagem enviada e na **Listagem 25** temos como ficará alguns trechos do app.js após as alterações de chamada do evento.

Se reiniciarmos a aplicação e acessarmos no browser, ao inspecionar as mensagens com a ferramenta de desenvolvedores, veremos que agora as mensagens estão com as devidas classes. Então vamos alterar o nosso arquivo style.css para termos uma diferenciação visual, como mostra a **Listagem 26**.

Listagem 24. Adicionando classes para cada tipo de mensagem

```
...  
socket.on("atualizar mensagens", function(dados){  
    var mensagem_formatada = $("<p>").text(dados.msg).addClass(dados.tipo);  
    $("#historico_mensagens").append(mensagem_formatada);  
});  
...  
...
```

Listagem 25. Alterando chamada do evento atualizar mensagens

```
...  
io.sockets.emit("atualizar usuarios", Object.keys(usuarios));  
io.sockets.emit("atualizar mensagens", {msg: "[" + pegarDataAtual() + "  
" + apelido + " acabou de entrar na sala", tipo:'sistema'});  
  
callback(true);  
...  
mensagem_enviada = "[" + pegarDataAtual() + "] " + socket.apelido + " diz: " +  
mensagem_enviada;  
  
if(usuario == ""){  
    io.sockets.emit("atualizar mensagens", {msg: mensagem_enviada, tipo:''});  
}else{  
    socket.emit("atualizar mensagens", {msg: mensagem_enviada, tipo:'privada'});  
    usuarios[usuario].emit("atualizar mensagens", {msg: mensagem_enviada,  
    tipo:'privada'});  
}  
callback();  
});  
  
socket.on("disconnect", function(){  
    delete usuarios[socket.apelido];  
    io.sockets.emit("atualizar usuarios", Object.keys(usuarios));  
    io.sockets.emit("atualizar mensagens", {msg: "[" + pegarDataAtual() + "  
" + socket.apelido + " saiu da sala", tipo:'sistema'});  
});
```

Listagem 26. Alterações no estilo das mensagens

```
...  
#historico_mensagens .sistema{  
background-color: #45C5BF;  
color: #FFF;  
font-weight: bold;  
}  
#historico_mensagens .privada{  
background-color: #CCC;  
color: #000;  
font-weight: bold;  
}
```

O resultado da nossa alteração de CSS vai ficar conforme a **Figura 12**. Lembrando que você pode modificar o CSS à vontade.

Outra coisa que deixa a desejar no sistema de chat é um histórico breve de mensagens assim que o usuário entra na sala. Seria bacana ele já ter pelo menos as últimas cinco mensagens enviadas na sala antes do mesmo entrar. Podemos fazer isto criando outra variável, abaixo da variável *usuarios* chamada *ultimas_mensagens*, que será um *array*. E no evento de *enviar mensagens* vamos inserir a mensagem no final do vetor e, caso o vetor tenha mais de cinco mensagens armazenadas, removemos a mais antiga.

```
[ 27/07/2015 23:13:34 ] Fulano acabou de entrar na sala  
[ 27/07/2015 23:13:42 ] Fulano diz: Olá pessoal!!  
[ 27/07/2015 23:14:04 ] Siclano diz: Fulano, depois preciso falar com você
```

Figura 12. Padrão das mensagens de chat

Todas as alterações vão se concentrar no arquivo *app.js*, mas vamos fazer por partes a alteração. Primeiramente, criaremos a variável das mensagens e uma função que vai armazenar as mesmas nesta mesma variável, conforme demonstrado na **Listagem 27**. A primeira parte vai logo no começo do script e a função será colocada no final do arquivo, logo depois da função *pegarDataAtual*.

É uma função bastante simples, se o tamanho do vetor for maior que 5, é retirado o primeiro valor da estrutura, e depois é colocado no final dela a mensagem que foi passada por parâmetro.

Agora precisamos fazer com que todos os eventos que emitem o evento de *atualizar mensagens* chamem esta função passando como parâmetro a mensagem em si. Vejamos os trechos a serem alterados no *app.js* primeiramente no evento de envio de mensagem, previsto na **Listagem 28**.

Tais alterações ajudarão a deixar o código mais enxuto e sem repetições. Vamos analisar o passo a passo: primeiro criamos um objeto chamado *obj_mensagem*, onde ficará armazenada a mensagem e o tipo dela; é importante armazenar o tipo para depois diferenciarmos quando for preciso apresentar o histórico para o usuário que acabou de entrar na sala.

Listagem 27. Guardando mensagens em um histórico

```
var app = require('http').createServer(resposta);  
var fs = require('fs');  
var io = require('socket.io')(app);  
var usuarios = [];  
var ultimas_mensagens = [];  
  
function armazenaMensagem(mensagem){  
    if(ultimas_mensagens.length > 5){  
        ultimas_mensagens.shift();  
    }  
  
    ultimas_mensagens.push(mensagem);  
}
```

Listagem 28. Chamando a função para guardar mensagens quando elas são enviadas

```
socket.on("enviar mensagem", function(dados, callback){  
    var mensagem_enviada = dados.msg;  
    var usuario = dados.usu;  
    if(usuario == null)  
        usuario = "";  
  
    mensagem_enviada = "[" + pegarDataAtual() + "] " + socket.apelido + " diz: "  
    + mensagem_enviada;  
    var obj_mensagem = {msg: mensagem_enviada, tipo:''};  
  
    if(usuario == ""){  
        io.sockets.emit("atualizar mensagens", obj_mensagem);  
        armazenaMensagem(obj_mensagem);  
    }else{  
        obj_mensagem.tipo = 'privada';  
        socket.emit("atualizar mensagens", obj_mensagem);  
        usuarios[usuario].emit("atualizar mensagens", obj_mensagem);  
    }  
    callback();  
});
```

Como criar um chat com Node.js

No *else* da nossa condicional, que diferencia se a mensagem é privada ou não, alteramos o valor do tipo de vazio para privada. E por fim armazenamos no nosso vetor a mensagem enviada pelo usuário. Agora na **Listagem 29** alteramos o método que é chamado na saída do usuário da sala de bate-papo, afinal este método também emite uma mensagem e seria interessante guardarmos no histórico.

Aqui a alteração seguiu a mesma linha que na nossa **Listagem 28**, onde separamos os dados em um objeto, emitimos a mensagem como antes e por fim chamamos a função. Esta é a alteração mais simples até o instante.

Na **Listagem 30** devemos alterar o método de entrada do usuário. Esta parte envolve mostrar as mensagens armazenadas no histórico e depois emitir a mensagem de que o usuário ingressou na sala.

Dessa vez, primeiro imprimimos as mensagens anteriores e depois armazenamos a mensagem de entrada. Mas por que temos de fazer isto nesta ordem? A razão é para não mostrarmos duas vezes a mesma mensagem de que o usuário entrou na sala. Senão é dado um *emit* na mensagem de entrada.

Feitas as alterações, um novo usuário que entrar na sala terá um histórico breve das mensagens trocadas antes dele entrar na sala.

Listagem 29. Armazenando mensagem de saída da sala

```
socket.on("disconnect", function(){
  delete usuarios[socket.apelido];
  var mensagem = "[" + pegarDataAtual() + "] " + socket.apelido + " saiu da sala";
  var obj_mensagem = {msg: mensagem, tipo:'sistema'};

  io.sockets.emit("atualizar usuarios", Object.keys(usuarios));
  io.sockets.emit("atualizar mensagens", obj_mensagem);

  armazenaMensagem(obj_mensagem);
});
```

Listagem 30. Alterando o método de entrada do usuário no chat

```
socket.on("entrar", function(apelido, callback){
  if(!(apelido in usuarios)){
    socket.apelido = apelido;
    usuarios[apelido] = socket;

    for(indice in ultimas_mensagens){
      socket.emit("atualizar mensagens", ultimas_mensagens[indice]);
    }

    var mensagem = "[" + pegarDataAtual() + "] " + apelido +
      " acabou de entrar na sala";
    var obj_mensagem = {msg: mensagem, tipo:'sistema'};

    io.sockets.emit("atualizar usuarios", Object.keys(usuarios));
    io.sockets.emit("atualizar mensagens", obj_mensagem);

    armazenaMensagem(obj_mensagem);

    callback(true);
  }else{
    callback(false);
  }
});
```

Você pode alterar o valor das mensagens que devem ser exibidas, basta alterar o método que salva as mesmas. Abra duas abas no navegador e faça uma troca de mensagens, depois abra uma terceira para verificar se o terceiro usuário recebeu as cinco últimas mensagens da sala, conforme demonstrado na **Figura 13**.

Todos	Fulano	Beltrano	Outra pessoa
[27/07/2015 21:27:29] Fulano acabou de entrar na sala			
[27/07/2015 21:27:43] Fulano diz: Olá pessoal! Tem alguém ai?			
[27/07/2015 21:27:53] Beltrano acabou de entrar na sala			
[27/07/2015 21:27:58] Beltrano diz: Agora tem! hehe			
[27/07/2015 21:28:03] Outra pessoa acabou de entrar na sala			
[27/07/2015 21:28:10] Outra pessoa diz: Oi pessoal!			

Figura 13. Painel de mensagens com histórico prévio

Com este projeto bastante simples e com poucas dependências e tecnologias conseguimos criar em poucas linhas uma aplicação bastante funcional e que serve bem ao propósito que estabelecemos. Se você gostou mesmo deste projeto, você pode estendê-lo, talvez colocando um painel de envio de *emojis* (o link para baixar a biblioteca está na seção **Links**), criando um painel de *admin* que consiga expulsar usuários, compartilhamento de imagens e vídeos dentre várias outras possibilidades.

Mas o poder do Socket.IO se estende a mais soluções que podem fazer com que sistemas que antes dependiam de páginas back-end e de um tempo de resposta de entrada e saída do servidor agora possam contar com o protocolo WebSocket para a troca mais ágil de informações. Possibilitando sistemas que acompanhem resultados de monitoramento ou até mesmo especificação em tempo real para os usuários.

Autor



Rafael Milleo

Analista de sistemas com experiência há sete anos no mercado de desenvolvimento web, formado pela Universidade Mackenzie com ênfase em Sistemas Hipermídia. Tem experiência em agências e grandes empresas do segmento de TI.



Links:

Site oficial do Socket.IO

<http://socket.io/>

Site oficial do Node.js para download da plataforma

<https://Node.js.org/>

Ajuda para instalação do Node.js no Linux

<https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager#enterprise-linux-and-fedora>

Repositório GitHub do projeto deste artigo

<https://github.com/Milleo/ChatJS>

Biblioteca EmojiArea

<https://github.com/diy/jquery-emojiarea>

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.



Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.



Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.



1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486