

**Bootstrap + AngularJS + Tiles:
crie aplicações web escaláveis**

**Aprenda a integrar
frameworks front-end
com tecnologias do servidor**

WEBSOCKETS COM SOCKJS

Crie um chat web assíncrono



Promises

Conheça a API JavaScript para
programação assíncrona

Otimização web

Veja as principais técnicas e ferramentas
para otimizar suas páginas web

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 à 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.

Conteúdo sobre Boas Práticas, Artigo no estilo Solução Completa

04 – Como criar um chat web com WebSockets, Spring eSockJS

[Júlio Sampaio]

Conteúdo sobre Boas Práticas

15 – Programação assíncrona em JavaScript com Promises

[Júlio Sampaio]

Artigo no estilo Solução Completa

25 – Como criar uma aplicação responsiva com Bootstrap

[Jorge Rodrigues]

Conteúdo sobre Boas Práticas

36 – Como otimizar a performance no front-end

[Sueila Sousa]

Sumário

Como criar um chat web com WebSockets, Spring e SockJS

Veja como integrar os três frameworks e desenvolver ferramentas de mensageria em tempo real

A web tem sido largamente construída em torno do chamado paradigma de request/response (requisição/resposta) da HTTP. Um cliente carrega uma página web e, em seguida, nada acontece até que o usuário clique para a próxima página. Por volta de 2005, o Ajax começou a fazer a web ficar mais dinâmica, com seus processamentos assíncronos, sem a necessidade de recarregar as páginas sempre que algo deve ser buscado no servidor. Ainda assim, toda a comunicação HTTP sempre foi dirigida pelo cliente, o que exigia a interação do usuário ou de sondagem periódica (existem vários algoritmos JavaScript para isso) para carregar novos dados a partir do servidor, como por exemplo nos casos de páginas de jornais ou notícias que precisam mostrar ao cliente informações em tempo real sem que o mesmo precise estar recarregando a página o tempo todo.

A solução via Ajax, se beneficiava de um recarregamento simples que simulava o click no botão de load do browser via JavaScript. Todavia, as tecnologias que permitem ao servidor enviar dados para o cliente, no exato momento em que ele sabe que novos dados estarão disponíveis, é algo bem recente no universo de programação front-end. Os nomes mais comuns para defini-las são "Push" ou "Comet" (essa última representa um novo modelo de protocolo, baseado no HTTP, que permite ao servidor enviar dados para um browser, sem a necessidade de uma requisição).

Um dos hacks mais comuns para criar a ilusão de uma conexão de servidor iniciada é chamado de *long polling* (ou chamada seletiva, em tradução livre). Com ele, o

Fique por dentro

Este artigo se mostra útil para desenvolvedores que desejam implementar recursos de mensageria assíncrona full-duplex em suas aplicações web via WebSockets. A arquitetura full-duplex se caracteriza pelo envio e recebimento de mensagens ou outros tipos de dados do cliente para o servidor e vice-versa, quebrando o velho conceito HTTP onde as respostas só chegam ao cliente quando requisitadas antes. O exemplo mais comum disso está nos chats web, já que precisamos saber sempre que alguém enviou uma mensagem e não podemos recarregar o browser o tempo todo, pois o custo disso seria muito alto.

Ao final deste você saberá como integrar suas APIs tanto no front-end quanto no back-end, verá que linguagens de programação implementam tal recurso, bem como entenderá como implementar todos os serviços usando o SockJS, a API mais usada para WebSockets em JavaScript.

cliente abre uma conexão HTTP com o servidor que a mantém aberta até que o envio da resposta seja efetuado. Sempre que o servidor realmente tem novos dados ele envia a resposta (outras técnicas envolvem Flash, requests multipart do tipo XHR e os chamados HtmlFiles). O long polling e as outras técnicas funcionam muito bem. Um dos usos mais famosos desse tipo de tecnologia está no chat do Gmail, já que o Google adota a mesma em várias de suas ferramentas e soluções.

No entanto, todas estas soluções alternativas compartilham um problema: elas precisam lidar com o overhead (sobrecarga) que a HTTP traz consigo, o que as tornam inadequadas para aplicações de baixa latência. Por exemplo, pense em um jogo multiplayer de tiro em primeira pessoa no browser ou qualquer outro jogo

on-line com componentes em tempo real; o protocolo HTTP é muito pesado para lidar com o tráfego de tantas informações pesadas e oferecer um retorno eficiente ao mesmo tempo.

Para ir de encontro a esse problema, o W3C lançou em meados de 2011 a tecnologia dos **WebSockets**, que basicamente é um novo protocolo que fornece canais de comunicação full-duplex (nas duas direções – servidor e cliente) sobre uma conexão TCP (*Transmission Control Protocol*). O TCP é o protocolo núcleo do IPS (*Internet Protocol Suite*), famoso pela sua combinação com o IP (*Internet Protocol*) para formar o modelo mais usado nas redes de computadores: TCP/IP. Ele é extremamente mais rápido que o HTTP e, ao contrário deste, não precisa criar pacotes de cabeçalhos e configurações nas requisições para que o servidor reconheça os pedidos e devolva respostas apropriadas.

A tecnologia em si é dividida em duas partes: a WebSocket API (que fornece todo o código fonte necessário para implementações via JavaScript) e o WebSocket Protocol (que padroniza a forma como todos os browsers devem implementar a tecnologia). Na seção **Links** do artigo você encontra as URLs oficiais de cada parte. Ambas são um padrão no mundo front-end e, portanto, estão disponíveis nos principais browsers do mercado, como Chrome, Safari, IE e Firefox.

Enquanto os WebSockets foram projetados originalmente para ser implementados em ambos navegador e servidor web, os mesmos fornecem benefícios arquiteturais significativos para o ambiente cliente-servidor em si, mas também para arquiteturas de aplicações móveis que precisam se comunicar diretamente com os servidores, ou entre os próprios dispositivos.

Na **Figura 1** temos uma representação de como os WebSockets estão arranjados dentro da web. A melhor maneira de pensar no WebSocket é como uma camada de transporte no topo do qual qualquer outro protocolo pode ser executado. A API do WebSocket suporta a capacidade de definir sub-protocolos: bibliotecas do protocolo que possam interpretar protocolos mais específicos. Exemplos destes incluem XMPP, STOMP e AMQP. Desta forma, os desenvolvedores não têm que pensar em termos de paradigma solicitação-resposta da HTTP, mas em vez disso, eles podem escolher o protocolo mais apropriado para o tipo de aplicações que estão escrevendo. O único requisito do lado do navegador é

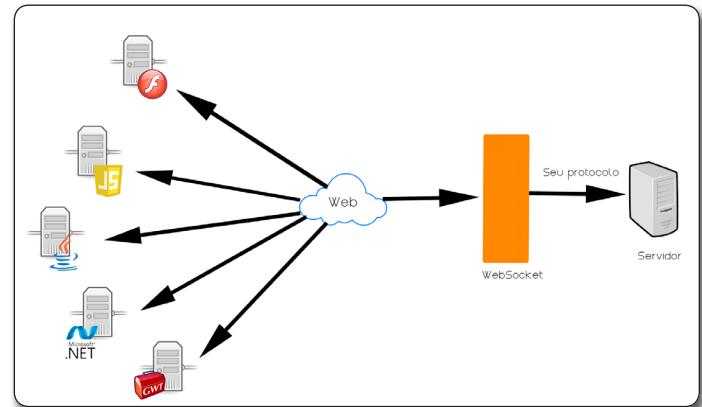


Figura 1. Arquitetura dos WebSockets

executar uma biblioteca JavaScript que pode interpretar os pacotes do WebSocket, estabelecer e manter uma conexão com o mesmo e interpretar protocolos específicos que venham a ser usados. No lado do servidor, o padrão da indústria é a utilização de bibliotecas do protocolo existentes que rodam em cima do TCP e que implementam algum gateway WebSocket de terceiros, como os protocolos da Kaazing WebSocket Gateways, por exemplo.

As implementações nos lados cliente e servidor que temos exigidas na figura usam bibliotecas específicas para tratar os WebSockets. Na **Tabela 1** você encontra uma relação das tecnologias que os suportam, suas APIs oficiais e a URL para cada uma.

Neste artigo trataremos de entender os principais conceitos referentes ao mundo WebSocket através do desenvolvimento de um web chat completo usando as tecnologias Spring com Java no lado servidor; SockJS, Stomp e AngularJS no lado cliente, para lidar com os serviços necessários para recebimento e envio das mensagens assincronamente.

Trazendo os Sockets para a web

A especificação WebSocket define uma API para estabelecer conexões do tipo “socket” entre um navegador e um servidor. Em outras palavras, há uma conexão persistente entre o cliente e o servidor e ambas as partes podem começar a enviar dados a qualquer momento.

Tecnologia	API Suportada	URLs
Node.js	Socket.IO WebSocket-Node Ws	http://socket.io/ https://github.com/Worlize/WebSocket-Node https://github.com/einaros/ws
Java	Jetty	http://www.eclipse.org/jetty/
Ruby	EventMachine	http://github.com/igrigorik/em-websocket
Python	Pywebsocket Tornado	http://code.google.com/p/pywebsocket/ https://github.com/facebook/tornado
Erlang	Shirasu	https://github.com/michilu/shirasu
C++	Libwebsockets	http://git.warmcat.com/cgi-bin/cgit/libwebsockets/
.NET	SuperWebSocket	http://superwebsocket.codeplex.com/

Tabela 1. Lista de tecnologias que suportam WebSockets

Como criar um chat web com WebSockets, Spring e SockJS

Para abrir uma conexão WebSocket você simplesmente precisa chamar o construtor do objeto WebSocket:

```
var conexao = new WebSocket('ws://seuwebsite.websocket.org/echo', ['soap', 'xmpp']);
```

Observe que o valor “ws:” representa o novo esquema de URL para conexões WebSocket. Há também o wss: para conexões com WebSocket seguro da mesma forma que o https é usado para conexões HTTP seguras.

Ao anexar alguns manipuladores de eventos imediatamente à conexão podemos saber quando a mesma foi aberta, recebeu mensagens de entrada, ou se houver um erro.

O segundo argumento aceita sub-protocolos opcionais. Pode ser uma string ou um array de strings. Cada string deve representar um nome de um sub-protocolo distinto e o servidor aceita apenas um dos passados no mesmo vetor, o primeiro que ele encontrar no browser. Para definir quais sub-protocolos você deseja aceitar na aplicação basta usar a propriedade *protocol* do objeto WebSocket. Os nomes dos sub-protocolos, por sua vez, precisam estar devidamente registrados na IANA, um órgão que padroniza os protocolos de WebSockets (vide seção [Links](#)). O parâmetro “soap” que definimos para comunicação com um Web Service é aceito pela mesma, tal como o “xmpp”.

Uma vez com o objeto de conexão instanciado, basta cadastrar os ouvintes para cada uma das ações padrão do mesmo, como abrir conexão, capturar erros ou mensagens vindas do servidor. Veja na [Listagem 1](#) um exemplo disso.

Listagem 1. Criando ouvintes para o objeto de conexão.

```
// Quando a conexão é aberta, envia alguns dados ao servidor
conexao.onopen = function () {
    connection.send('Olá'); // Envia a mensagem "Olá" ao servidor
};

// Loga erros
conexao.onerror = function (erro) {
    console.log('Erro no WebSocket' + erro);
};

// Loga mensagens no servidor
conexao.onmessage = function (e) {
    console.log('Servidor:' + e.data);
};
```

Veja como a implementação é muito próxima da que temos para métodos de frameworks JavaScript como o jQuery, que “ouve” sempre que determinado evento ocorre no objeto em questão.

Uma vez conectados com o servidor (quando o evento ‘open’ é disparado), podemos começar a enviar dados para o mesmo usando o método *send(“sua mensagem”)* no objeto de conexão. Ele é utilizado para receber apenas strings, mas na última especificação agora podemos enviar mensagens de binários também. Para enviar dados binários, você pode usar um Blob ou ArrayBuffer ([Listagem 2](#)).

Listagem 2. Exemplo de WebSocket enviando dados binários.

```
// Enviando uma string
conexao.send('sua mensagem');

// Enviando um canvas ImageData como um ArrayBuffer
var img = canvas_context.getImageData(0, 0, 400, 320);
var binario = new Uint8Array(img.data.length);
for (var i = 0; i < img.data.length; i++) {
    binary[i] = img.data[i];
}
conexao.send(binario.buffer);

// Enviando um arquivo como um Blob
var arquivo = document.querySelector("input[type='file']").files[0];
conexao.send(arquivo);
```

A listagem faz uso de objetos já conhecidos do JavaScript e HTML5. Isso prova que a tecnologia suporta muito mais que somente texto, como é o caso do HTTP. O suporte a formatos binários se faz útil principalmente para aplicações que desenham muitos gráficos no browser, como jogos e animações gráficas.

Igualmente o servidor pode enviar mensagens a qualquer momento. Sempre que isso acontece o método *ouvinte (callback) onMessage* é acionado. O método de callback recebe um objeto de evento e a mensagem real se torna acessível através da propriedade *data*.

O WebSocket também pode receber mensagens de binários da mesma forma que as envia. Frames binários podem ser recebidos nos mesmos formatos Blob e ArrayBuffer. Para especificar o formato do binário recebido, defina a propriedade *binaryType* do objeto WebSocket para ‘blob’ ou ‘ArrayBuffer’. O formato padrão é “blob”. (Não é necessário alinhar os bytes de *binaryType* antes do envio). Veja na [Listagem 3](#) um exemplo de como receber estes tipos de dados como resposta do servidor no mesmo exemplo anterior. Nele, a função “onmessage” só recebe o objeto genérico e, se for binário, será convertido automaticamente para ArrayBuffer, imprimindo a quantidade de bytes enviados.

Listagem 3. Código para receber o objeto binário enviado na Listagem 2.

```
// Configura binaryType para aceitar binários como 'blob' ou 'arraybuffer'
conexao.binaryType = 'arraybuffer';
conexao.onmessage = function(e) {
    console.log(e.data.byteLength); // Objeto ArrayBuffer se for binário
};
```

Outra característica recém-adicionada ao WebSocket são as extensões, por onde é possível enviar frames comprimidos, multiplexados, etc. Você pode encontrar extensões aceitas no servidor examinando a propriedade “extensions” do objeto WebSocket após o evento ser aberto:

```
console.log(connection.extensions);
```

Configuração do ambiente

Uma vez baixados o Eclipse IDE for Java EE e o Tomcat, você só precisa descompactar os arquivos em um diretório selecionado para o projeto e iniciar o Eclipse.exe. Depois vá na aba “Servers” e clique com o botão direito na opção “New > Server” e, na janela que abrir, selecione “Apache > Tomcat 7” e clique em *Next*. Na próxima tela, clique em *Browse*, busque o diretório onde extraiu o seu Tomcat 7 e clique em *Finish*. Após isso, clique no botão *Start* da mesma aba e verifique se o servidor inicia corretamente na aba *Console*.

Para criar a aplicação, vá até o menu “File > New > Maven Project”. Clique em *Next*, e na próxima tela digite “web” no campo *Filter* para filtrar somente pelos arquétipos de projetos Java Web. Selecione o arquétipo de *Group Id* “org.apache.maven.archetypes” e *Artifact Id* “Maven-archetype-webapp”, clique em *Next* novamente. Na tela que se suceder preencha os campos tal como demonstrado na **Figura 2**. Clique em *Finish*.

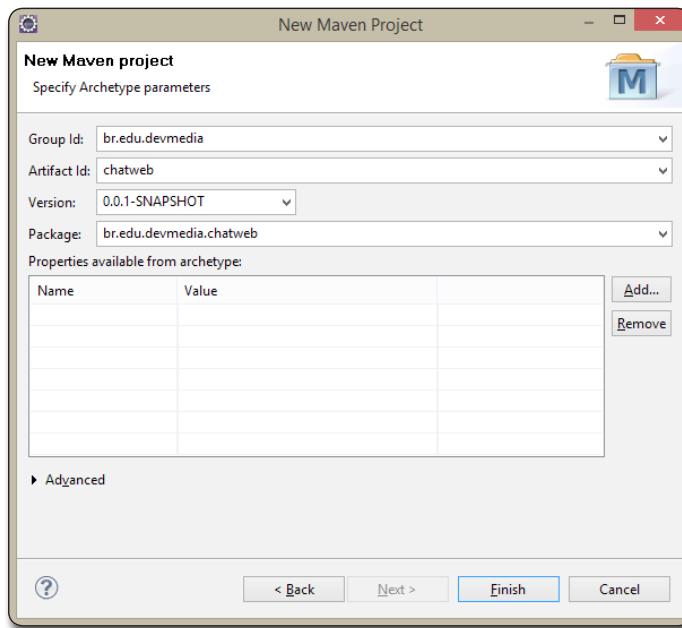


Figura 2. Tela de definição das propriedades do projeto Maven

Aguarde até que o Maven carregue todas as dependências básicas do projeto. O mesmo virá por padrão somente com as libs do JEE padrão, mais a lib do junit para testes unitários.

A primeira configuração importante consiste em adicionar as bibliotecas do framework Spring MVC, e do Spring Messaging que disponibilizará todas as demais para o recurso de WebSocket. Também precisaremos de um serializador JSON como o Jackson, por exemplo, uma vez que os dados devem ser empacotados nos padrões do WebSocket antes de serem enviados via rede. Para isso, abra o arquivo pom.xml presente na raiz do projeto e modifique o seu conteúdo para o ilustrado na **Listagem 4**.

As primeiras linhas trazem as configurações autogeradas pelo Maven como os ids do artefato e grupo, versões e URL do projeto. Nas configurações de dependências temos as três libs

Listagem 4. Código XML do pom para gerência de dependências.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.edu.devmedia</groupId>
  <artifactId>chatweb</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>chatweb Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>4.1.1.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-websocket</artifactId>
      <version>4.1.1.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-messaging</artifactId>
      <version>4.1.1.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>javax.websocket</groupId>
      <artifactId>javax.websocket-api</artifactId>
      <version>1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.2</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-core</artifactId>
      <version>2.3.3</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.3.3</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.jaxrs</groupId>
      <artifactId>jackson-jaxrs-json-provider</artifactId>
      <version>2.3.3</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>chatweb</finalName>
  </build>
</project>
```

Como criar um chat web com WebSockets, Spring e SockJS

do Spring (Web MCV, Messaging e WebSocket) todas na versão 4.1.1.RELEASE; também precisamos das libs do Servlet para fazer a integração entre os sockets e o HTTP, da API do JSTL para gerenciar os valores na tela via JSP, e finalmente as libs do Jackson para manutenção dos métodos de Web Services e comunicação cliente-servidor.

Para a parte front-end, precisaremos configurar algumas bibliotecas, como:

- **AngularJS**: para lidar com alguns serviços assíncronos do chat, como o recebimento das mensagens e solicitações de status ao servidor;
- **SockJS**: que fornece uma biblioteca com objetos que encapsulam a lógica do WebSocket e a traduzem para JavaScript.
- **STOMP-WebSocket**: é um protocolo de mensageria baseada em texto que possibilita a comunicação do cliente e servidor de forma independente.
- **Lodash**: uma biblioteca JavaScript de funções utilitárias, tais como empacotamento de classes, alta performance no código, integração com outros frameworks JS, etc.

Para isso, faremos uso do framework gerenciador de pacotes Bower. Crie um novo arquivo bower.json na raiz do projeto e adicione o conteúdo da **Listagem 5** ao mesmo. Certifique-se de usar sempre as versões compatíveis apresentadas na documentação oficial do framework, pois algumas delas podem não funcionar em versões antigas por falta de recursos e classes.

Listagem 5. Código JSON para dependências JS no Bower.

```
{  
  "name": "spring-web-chat",  
  "version": "0.0.1-SNAPSHOT",  
  "dependencies": {  
    "sockjs": "0.3.4",  
    "stomp-websocket": "2.3.4",  
    "angular": "1.3.8",  
    "lodash": "2.4.1"  
  }  
}
```

A instalação do Bower dever ser feita a nível de Sistema Operacional e, como o mesmo tem dependência direta no Node.js, baixe o arquivo de instalação do Node.js disponível via seção **Links** e instale-o seguindo os passos sem modificar nenhum deles. No final, o próprio instalador se encarregará de adicionar o npm (gerenciador de pacotes do Node) às variáveis de ambiente da máquina e disponibilizar os comandos do mesmo via prompt de comandos.

Para verificar se tudo ocorreu bem, execute o comando a seguir:

```
npm -v
```

A versão do Node.js irá aparecer. Após, execute o seguinte comando para instalar o Bower na máquina:

```
npm install -g bower
```

Quando o processo finalizar, verifique se tudo ocorreu bem via comando:

```
bower -v
```

A versão do Bower aparecerá (1.3.8, a mais recente). Agora só precisamos configurar o nosso projeto como um projeto do tipo Bower. Para isso, acesse via terminal de comandos (comando *cd*) o diretório onde você criou o seu projeto Maven e execute o seguinte comando na raiz do mesmo:

```
bower install
```

Lembre-se que o diretório deve ser o mesmo onde você salvou o arquivo bower.json, pois é dele que o framework buscará as dependências e respectivas versões a ser instaladas. O resultado será o exibido na **Listagem 6**.

Listagem 6. Resultado da execução do comando de instalação.

```
D:\workspace\chatweb>bower install  
[?] May bower anonymously report usage statistics to improve the tool over time?  
[?] May bower anonymously report usage statistics to improve the tool over time? No  
bower not-cached git://github.com/lodash/lodash.git#2.4.1  
bower resolve git://github.com/lodash/lodash.git#2.4.1  
bower not-cached git://github.com/myguidingstar/bower-sockjs.git#0.3.4  
bower resolve git://github.com/myguidingstar/bower-sockjs.git#0.3.4  
bower not-cached git://github.com/jmesnil/stomp-websocket.git#2.3.4  
bower resolve git://github.com/jmesnil/stomp-websocket.git#2.3.4  
bower not-cached git://github.com/angular/bower-angular.git#1.3.8  
bower resolve git://github.com/angular/bower-angular.git#1.3.8  
bower download https://github.com/myguidingstar/bower-sockjs/archive/0.3.4.tar.gz  
bower download https://github.com/jmesnil/stomp-websocket/archive/2.3.4.tar.gz  
bower download https://github.com/lodash/lodash/archive/2.4.1.tar.gz  
bower download https://github.com/angular/bower-angular/archive/v1.3.8.tar.gz  
bower extract sockjs#0.3.4 archive.tar.gz  
bower deprecated Package sockjs is using the deprecated component.json  
bower resolved git://github.com/myguidingstar/bower-sockjs.git#0.3.4  
bower extract stomp-websocket#2.3.4 archive.tar.gz  
bower invalid-meta stomp-websocket is missing "ignore" entry in bower.json  
bower resolved git://github.com/jmesnil/stomp-websocket.git#2.3.4  
bower extract angular#1.3.8 archive.tar.gz  
bower resolved stomp-websocket#2.3.4 archive.tar.gz  
bower resolved lodash#2.4.1 archive.tar.gz  
bower resolved git://github.com/angular/bower-angular.git#1.3.8  
bower install git://github.com/lodash/lodash.git#2.4.1  
bower install lodash#0.3.4  
bower install stomp-websocket#2.3.4  
bower install angular#1.3.8  
bower install lodash#2.4.1  
  
sockjs#0.3.4 bower_components\sockjs  
  
stomp-websocket#2.3.4 bower_components\stomp-websocket  
  
angular#1.3.8 bower_components\angular  
  
lodash#2.4.1 bower_components\lodash
```

Perceba que no início o Bower pergunta se deseja reportar à central quaisquer problemas relacionados ao uso indevido do mesmo. Resposta com *Yes* ou *No* e logo após o mesmo começa a baixar as dependências.

Se o leitor não desejar efetuar todos estes passos, poderá usar as mesmas URLs da listagem para baixar cada um dos arquivos diretamente no browser, assim terá de fazer seu próprio controle interno de dependências. Em alguns dos arquivos, temos alguns status sendo exibidos:

- **not-cached:** significa que o download foi feito diretamente dos servidores do Bower, em vez de em terceiros;
- **resolve:** o Bower não encontrou uma versão de alguma dependência informada, mas conseguiu resolver com versões mais recentes;
- **deprecated:** o arquivo em questão encontra-se depreciado. Em usos futuros é aconselhado usar versões mais recentes. Se não souber qual a versão em específico que quer usar e tiver certeza de que a última sempre funcionará para você, use o valor “*last version*” no arquivo JSON.

No fim, podemos ver que ele gerou uma pasta “bower_components” e salvou tudo dentro dela. No diretório físico ela já vai aparecer, porém não no Eclipse, portanto clique com o botão direito no projeto e selecione a opção *Refresh*, isso irá recarregar todos os arquivos e pastas, inclusive esta. Entretanto, a pasta foi gerada na raiz do projeto e, como se trata de um projeto web, precisamos fazer estes arquivos acessíveis de dentro da pasta “src > main > webapp”. Logo, acesse essa pasta e crie uma nova dentro chamada “libs”, e então mova todas as subpastas de “bower_components” para dentro dela, tal como temos na **Figura 3**.

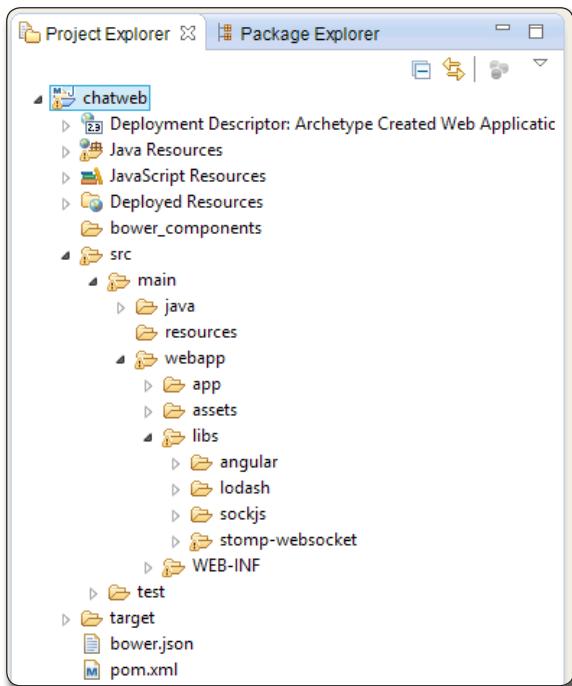


Figura 3. Estrutura de pastas dentro da WEB-INF

Dê uma olhada nos arquivos que foram gerados pelo Bower, os mais importantes são os .js core. O Bower também gera versões minificadas, onde o código está contido em uma linha só para melhorar o carregamento nos browsers e aumentar a performance das páginas.

Agora precisamos começar a configuração das classes Java que executarão no servidor. Vamos começar pelas classes de configuração do Spring. A primeira delas é a *WebAppInitializer*, responsável por recuperar os objetos do Java EE e disponibilizá-los para gerenciamento automático do Spring. Crie um novo pacote “br.edu.devmedia.chat.config” e após isso a classe. Adicione o conteúdo da **Listagem 7** à mesma.

Listagem 7. Conteúdo da classe WebAppInitializer.

```
import java.nio.charset.StandardCharsets;  
  
import javax.servlet.Filter;  
import javax.servlet.ServletRegistration;  
  
import org.springframework.web.filter.CharacterEncodingFilter;  
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;  
  
public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {  
  
    @Override  
    protected void customizeRegistration(ServletRegistration.Dynamic registration) {  
        registration.setInitParameter("dispatchOptionsRequest", "true");  
        registration.setAsyncSupported(true);  
    }  
  
    @Override  
    protected Class<?>[] getRootConfigClasses() {  
        return new Class<?>[] { AppConfig.class, WebSocketConfig.class };  
    }  
  
    @Override  
    protected Class<?>[] getServletConfigClasses() {  
        return new Class<?>[] { WebConfig.class };  
    }  
  
    @Override  
    protected String[] getServletMappings() {  
        return new String[] { "/" };  
    }  
  
    @Override  
    protected Filter[] getServletFilters() {  
        CharacterEncodingFilter characterEncodingFilter = new CharacterEncodingFilter();  
        characterEncodingFilter.setEncoding(StandardCharsets.UTF_8.name());  
        return new Filter[] { characterEncodingFilter };  
    }  
}
```

Essa classe criará o contexto do Spring e fornecerá os métodos para retornar as demais classes de configuração para o scanner do Spring (métodos *getRootConfigClasses()* e *getServletConfigClasses()*). Já o método *getServletMappings()* diz que todas as URLs que chegarem à aplicação deverão ser direcionadas para o Spring tratar.

Como criar um chat web com WebSockets, Spring e SockJS

O método getServletFilters() retorna o filtro de encoding que será usado em todas as requisições, no caso UTF-8. Como vamos tratar texto no chat, é importante que o Spring reconheça o padrão de texto em português.

Não se preocupe se o Eclipse mostrar alguns erros nas classes AppConfig (**Listagem 8**), WebSocketConfig (**Listagem 9**) e Web-Config (**Listagem 10**), pois as criaremos agora.

A classe AppConfig apenas se responsabiliza por definir qual o pacote raiz do projeto que será scaneado pelo Spring e que tipos de filtros serão considerados: as classes Controller e Configuration do próprio Spring.

Listagem 8. Conteúdo da classe AppConfig.

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.stereotype.Controller;

@Configuration
@ComponentScan(basePackages = "br.edu.devmedia", excludeFilters = {
    @ComponentScan.Filter(value = Controller.class, type =
        FilterType.ANNOTATION),
    @ComponentScan.Filter(value = Configuration.class, type =
        FilterType.ANNOTATION)
})
public class AppConfig {
```

Listagem 9. Conteúdo da classe WebSocketConfig.

```
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.AbstractWebSocketMessageBrokerConfigurer;
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker
@ComponentScan(basePackages = "br.edu.devmedia.chat.controller")
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topico");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/chat").withSockJS();
    }
}
```

Perceba que no início da classe WebSocketConfig estamos usando a anotação @EnableWebSocketMessageBroker da API do Spring, que configura que URLs serão direcionadas para o broker (gerente) das mensagens que chegarem via sockets, no

nosso caso “/topico”. Além disso, definimos no último método qual o endpoint que o SockJS vai usar para mandar as mensagens que chegarem do servidor, no nosso caso “/chat”. A anotação @ComponentScan define qual o pacote que conterá os controllers do Spring que gerenciarão a comunicação direta com as telas (métodos de fachada).

Listagem 10. Conteúdo da classe WebConfig.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.mvc.WebContentInterceptor;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "br.edu.devmedia.chat.controller")
public class WebConfig extends WebMvcConfigurerAdapter {

    @Bean
    public InternalResourceViewResolver getInternalResourceViewResolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }

    @Override
    public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }

    @Bean
    public WebContentInterceptor webContentInterceptor() {
        WebContentInterceptor interceptor = new WebContentInterceptor();
        interceptor.setCacheSeconds(0);
        interceptor.setUseExpiresHeader(true);
        interceptor.setUseCacheControlHeader(true);
        interceptor.setUseCacheControlNoStore(true);

        return interceptor;
    }

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/libs/**").addResourceLocations("/libs/");
        registry.addResourceHandler("/app/**").addResourceLocations("/app/");
        registry.addResourceHandler("/assets/**").addResourceLocations("/assets/");
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(webContentInterceptor());
    }
}
```

A classe WebConfig traz as principais configurações do projeto, a saber:

- Método `getInternalResourceViewResolver()`: Define em quais URLs os arquivos de conteúdo web (js, jsp, css, etc.) estarão contidos, bem como em quais diretórios deverão ser buscados;
- Método `configureDefaultServletHandling()`: Habilita o controle da aplicação que deverá ser feito diretamente pelo servidor (Tomcat);
- Método `webContentInterceptor()`: Define as configurações gerais do cabeçalho das requisições, como tempo máximo de cache, token de expiração do cabeçalho e controle de cache genérico;
- Método `addResourceHandlers()`: Mapeia as URLs de requisição aos diretórios onde estão os recursos;
- Método `addInterceptors()`: Associa o interceptor criado ao contexto do Spring.

Agora precisamos criar as classes que guardarão os dados finais para transporte de um lado para o outro. Vamos usar o padrão DTO (*Data Transfer Object*) com objetos POJOs simples, apenas com métodos get's e set's. Na **Listagem 11** temos a classe Mensagem que conterá a string de mensagem e o seu id, e na **Listagem 12** temos a classe MensagemSaida, que estenderá de Mensagem, mas terá também um objeto Date para guardar a hora e minuto da mesma.

Listagem 11. Conteúdo da classe Mensagem.

```
public class Mensagem {  
  
    private String message;  
    private int id;  
  
    public Mensagem() {}  
  
    public Mensagem(int id, String message) {  
        this.id = id;  
        this.message = message;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
  
    public void setMessage(String message) {  
        this.message = message;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
}
```

Portanto, crie as duas classes num novo pacote “br.edu.devmedia.chat.dto”.

Em seguida, crie também uma nova classe no pacote “controller” que lidamos antes para gerenciar o controle das views, as páginas JSP. Dê o nome da classe de “ChatController” e adicione o conteúdo da **Listagem 13** à mesma.

Listagem 12. Conteúdo da classe MensagemSaida.

```
public class MensagemSaida extends Mensagem {  
    private Date time;  
  
    public MensagemSaida(Mensagem original, Date time) {  
        super(original.getId(), original.getMensagem());  
        this.time = time;  
    }  
  
    public Date getTime() {  
        return time;  
    }  
  
    public void setTime(Date time) {  
        this.time = time;  
    }  
}
```

Listagem 13. Conteúdo da classe ChatController.

```
import org.springframework.messaging.handler.annotation.MessageMapping;  
import org.springframework.messaging.handler.annotation.SendTo;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
  
import br.edu.devmedia.chat.dto.Mensagem;  
import br.edu.devmedia.chat.dto.MensagemSaida;  
  
@Controller  
@RequestMapping("/")  
public class ChatController {  
  
    @RequestMapping(method = RequestMethod.GET)  
    public String viewApplication() {  
        return "index";  
    }  
  
    @MessageMapping("/chat")  
    @SendTo("/topico/mensagem")  
    public MensagemSaida sendMessage(Mensagem mensagem) {  
        System.out.println("Mensagem enviada");  
        return new MensagemSaida(mensagem, new Date());  
    }  
}
```

Na listagem temos basicamente dois mapeamentos: o primeiro (método `viewApplication()`) diz que todas as requisições da aplicação serão gerenciadas por uma só URL e página JSP, no caso “index”; o segundo (método `sendMessage()`) diz que as mensagens que chegarem da URL “/chat” deverão ser lançadas para a URI interna “/topico/msg” que criamos antes. No fim, um objeto do tipo MensagemSaida será criado com a hora atual e a mensagem recebida para enviar para a tela.

Como criar um chat web com WebSockets, Spring e SockJS

Telas e Views

Agora vamos focar na construção das páginas web e código JavaScript. Comecemos pela página index.jsp que deverá ser criada no diretório “webapp > WEB-INF > views”. Adicione o conteúdo da **Listagem 14** à mesma.

A primeira coisa a fazer é importar a fonte de texto e o CSS que queremos para a aplicação. Então criamos a tag body com o atributo ng-app com o nome da aplicação (appChat) para o AngularJS. Na aplicação teremos um controller do AngularJS (controladorChat); não se confunda com os controllers do Spring que criamos, pois aquele irá lidar com o controle no lado front-end.

Listagem 14. Conteúdo da JSP de index.

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<link href="http://fonts.googleapis.com/css?family=Open+Sans:400,300,600,700" rel="stylesheet" type="text/css"/>
<link href="assets/style.css" rel="stylesheet" type="text/css"/>
</head>
<body ng-app="appChat">
<div ng-controller="controladorChat" class="container">
<form ng-submit="addMensagem()" name="messageForm">
<input type="text" placeholder="Digite a sua mensagem..." ng-model="message"/>
<div class="info">
<span class="count" ng-bind="max - message.length"
ng-class="{danger: message.length > max}">>140</span>
<button ng-disabled="message.length > max || message.length === 0">Enviar</button>
</div>
</form>
<hr />
<p ng-repeat="message in messages | orderBy:'time':true" class="message">
<time>{{message.time | date:'HH:mm'}}</time>
<span ng-class="{self: message.self}">{{message.message}}</span>
</p>
</div>

<script src="libs/sockjs/sockjs.min.js" type="text/javascript"></script>
<script src="libs/stomp-websocket/lib/stomp.min.js" type="text/javascript">
</script>
<script src="libs/angular/angular.min.js"></script>
<script src="libs/lodash/dist/lodash.min.js"></script>
<script src="app/app.js" type="text/javascript"></script>
<script src="app/controllers.js" type="text/javascript"></script>
<script src="app/services.js" type="text/javascript"></script>
</body>
</html>
```

Após isso, precisamos criar um formulário que contenha um campo de texto input e associá-lo ao modelo “message” do próprio AngularJS, assim salvamos todas as informações nesse objeto e podemos usá-la mais adiante. Quando o formulário for submetido, o método addMensagem() será chamado no AngularJS que irá montar nossa mensagem e enviar usando WebSockets.

Também adicionamos um contador para verificar a quantidade máxima de caracteres na mensagem, tal como o que temos no Twitter. No momento que a quantidade de chars atingir o valor de “max”, o submit no form é bloqueado até que os mesmos chars sejam removidos.

Abaixo do form iteramos sobre cada uma das mensagens exibindo seu conteúdo e hora que foi enviada. Se a mensagem tiver sido enviada pelo usuário teremos uma classe CSS específica para diferenciar das demais mensagens. Por fim, importamos os arquivos de JS das bibliotecas do Bower.

Agora precisamos dos arquivos de JavaScript para lidar com os serviços do AngularJS, do SockJS e do Stomp. Para isso, acesse os mesmos dentro do diretório “src > main > webapp > app” no arquivo de download deste artigo e copie a pasta inteira para o seu projeto.

Por fim, também precisaremos importar um estilo padrão ao chat web. Crie uma nova pasta “assets” dentro do diretório “main > webapp” e, dentro dela, um novo arquivo chamado “style.css”. Adicione o conteúdo da **Listagem 15** ao mesmo. Tratam-se apenas de configurações de estilo, e o leitor pode customizar à vontade.

Antes de executar a aplicação precisamos nos certificar de que o context root do projeto no Eclipse está configurado corretamente para o valor “spring-ng-chat”. Para isso, clique com o botão direito no projeto, selecione “Properties > Deployment Assembly” e certifique-se de ter adicionado as dependências do Maven lá, caso já não estejam (**Figura 4**). Após, vá até a opção “Web Project Settings” e mude o valor de “Context root” para o informado.

Pronto, agora adicione o projeto no Tomcat, inicie-o e execute a aplicação no browser via URL: <http://localhost:8080/spring-ng-chat/>. Você deverá ver uma tela igual à da **Figura 5**.

Perceba que ao colocarmos o mouse sobre o botão de Enviar o mesmo é desabilitado, uma validação que já vem por padrão com o AngularJS. Ao digitar qualquer mensagem no campo você verá que o contador ao lado do botão começará a incrementar e o mesmo botão sairá do estado de desabilitado. Faça um teste,

digite uma mensagem e clique em Enviar, você verá ela aparecer logo abaixo da div tal como na **Figura 6**.

Tanto a mensagem quanto a hora em que ela foi criada serão exibidas na tabela abaixo do form. As mensagens não são cumulativas, portanto se você abrir a página em outro browser, verá que nenhuma aparecerá, o funcionamento normal de qualquer chat.

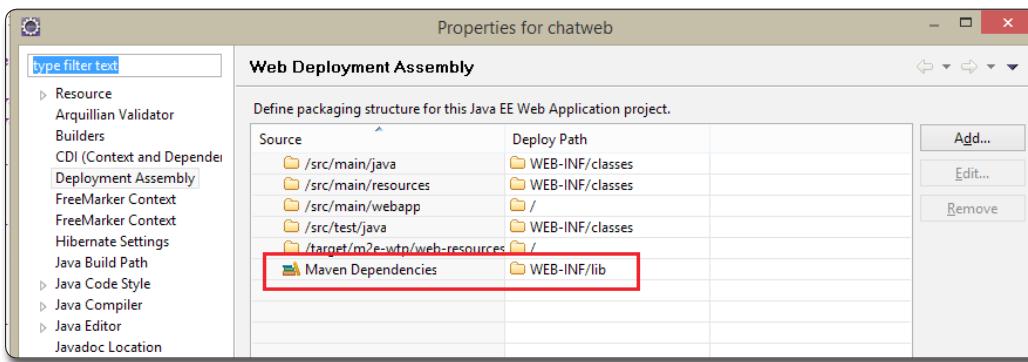
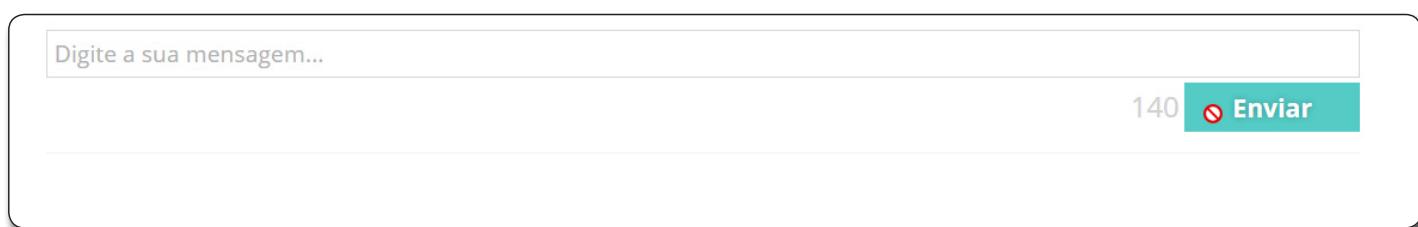


Figura 4. Dependências do Maven no projeto

Listagem 15. Conteúdo da página CSS do chat.

```
body, * {  
    font-family: 'Open Sans', sans-serif;  
    box-sizing: border-box;  
}  
  
.container {  
    max-width: 1000px;  
    margin: 0 auto;  
    width: 80%;  
}  
  
input[type=text] {  
    width: 100%;  
    border: solid 1px #D4D4D1;  
    transition: .7s;  
    font-size: 1.1em;  
    padding: 0.3em;  
    margin: 0.2em 0;  
}  
  
input[type=text]:focus {  
    -webkit-box-shadow: 0 0 5px 0 rgba(69, 155, 231, .75);  
    -moz-box-shadow: 0 0 5px 0 rgba(69, 155, 231, .75);  
    box-shadow: 0 0 5px 0 rgba(69, 155, 231, .75);  
    border-color: #8cc53e;  
    outline: none;  
}  
  
.info {  
    float: right;  
}  
  
form:after {  
    display: block;  
    content: " ";  
    clear: both;  
}  
  
button {  
    background: #8cc53e;  
    color: #FFF;  
    font-weight: 600;  
    padding: .3em 1.9em;  
    border: none;  
    font-size: 1.2em;  
    margin: 0;  
    text-shadow: 0 0 5px rgba(0, 0, 0, .3);  
    cursor: pointer;  
    transition: .7s;  
}  
  
button:focus {  
    outline: none;  
}  
  
button:hover {  
    background: #1c82dd;  
}  
  
button:disabled {  
    background-color: #49c5bf;  
    cursor: not-allowed;  
}  
  
.count {  
    font-weight: 300;  
    font-size: 1.35em;  
    color: #CCC;  
    transition: .7s;  
}  
  
.count.danger {  
    color: #a94442;  
    font-weight: 600;  
}  
  
.message time {  
    width: 80px;  
    color: #999;  
    display: block;  
    float: left;  
}  
  
.message {  
    margin: 0;  
}  
  
.message .self {  
    font-weight: 600;  
}  
  
.message span {  
    width: calc(100% - 80px);  
    display: block;  
    float: left;  
    padding-left: 20px;  
    border-left: solid 1px #F1F1F1;  
    padding-bottom: .5em;  
}  
  
hr {  
    display: block;  
    height: 1px;  
    border: 0;  
    border-top: solid 1px #F1F1F1;  
    margin: 1em 0;  
    padding: 0;  
}
```



The screenshot shows a simple web-based chat interface. At the top, there is a text input field with placeholder text 'Digite a sua mensagem...'. Below the input field is a teal-colored button containing the number '140' and a small red circular icon with a white minus sign, followed by the word 'Enviar' (Send). The overall design is clean and modern.

Figura 5. Tela inicial do chat web

Como criar um chat web com WebSockets, Spring e SockJS

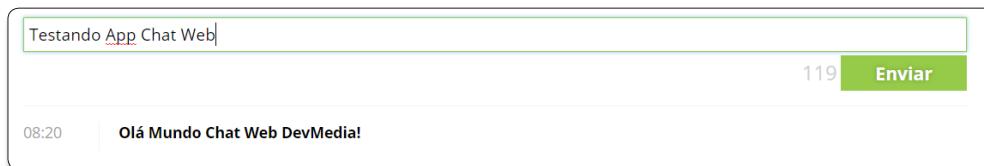


Figura 6. Enviando primeira mensagem no chat web

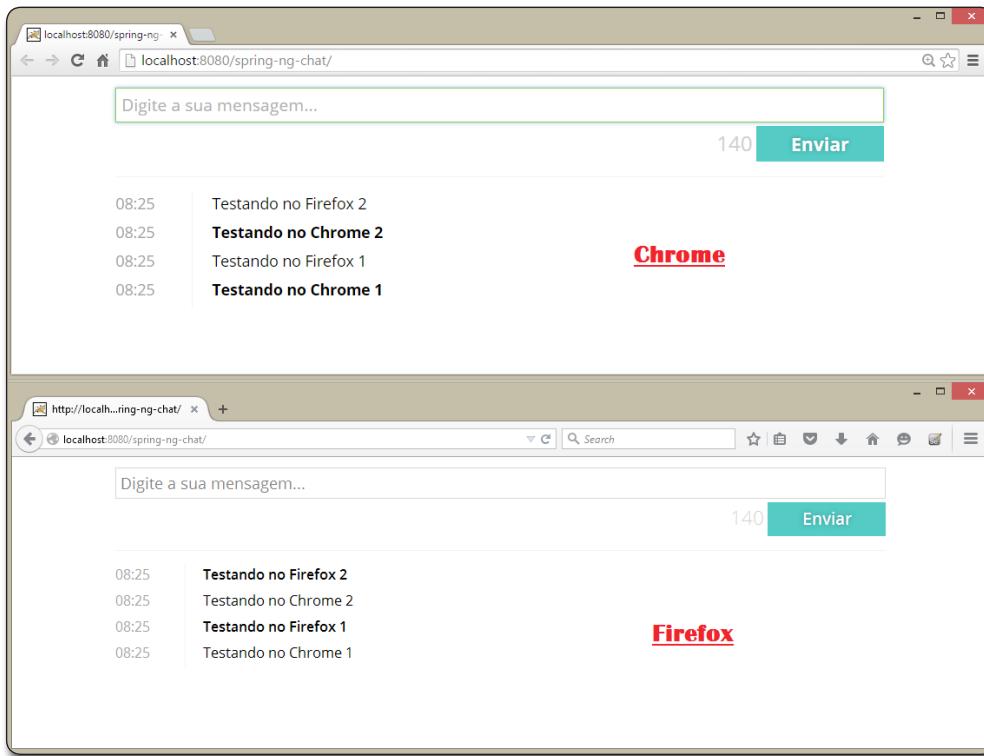


Figura 7. Telas da aplicação testadas em diferentes browsers ao mesmo tempo

Para verificar o funcionamento dos WebSockets, abra a mesma página em dois browsers distintos e mande mensagens entre os mesmos. Você verá que elas são sincronizadas à medida em que chegam no servidor (Figura 7).

Aplicação finalizada. O leitor pode ainda adicionar mais recursos comuns a chats como cadastro e login de usuários, customização de contas, mais cores e identificadores para cada um, bem como um sistema de captcha para validar se são mesmo humanos usando o sistema. Porém, o coração de um chat web é basicamente este: WebSockets funcionando no lado cliente e sua API respectiva no lado servidor. No nosso exemplo, usamos Java e Spring, mas você pode usar quaisquer outras listadas na Tabela 1 e se beneficiar dos recursos da mesma tecnologia. Bons estudos!

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 da API do WebSockets.

<http://dev.w3.org/html5/websockets/>

Página oficial do protocolo WebSockets.

<http://tools.ietf.org/html/rfc6455>

Página da IANA.

<http://www.iana.org/assignments/websocket/websocket.xml>

Página de download do Node.js

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

Programação assíncrona em JavaScript com Promises

Aprenda a usar todo o poder assíncrono da API de Promises do JavaScript, Bluebird e jQuery

O universo front-end está acostumado a lidar com JavaScript assíncrono em toda parte. Ajax, WebRTC e Node.js são alguns exemplos de onde APIs assíncronas são encontradas. Embora seja fácil de escrever uma função rápida para lidar com o resultado de uma solicitação HTTP, também é fácil se perder em um mar imprevisível de retornos de chamadas à medida que uma base de código cresce e mais pessoas passam a contribuir. É aí que uma boa abordagem para lidar com código assíncrono entra e muitos desenvolvedores estão escolhendo usar as **Promises** para tal abordagem.

Aplicações Web carregam dados e scripts no navegador de forma assíncrona. O Node.js e seus derivados fornecem uma série de APIs para I/O assíncrono. E novas especificações da web para *Streams*, *Service Workers*, e *Font Loading* (Carregamento de Fontes) incluem chamadas assíncronas em suas especificações. Estes avanços ampliam as capacidades de aplicações JavaScript, mas usá-los sem compreender como os trabalhos assíncronos funcionam pode resultar em código imprevisível e difícil de manter. Essa parte do ciclo de software é importante pois sem ela as coisas podem funcionar como esperado em ambientes de desenvolvimento ou teste, mas falhar quando implantadas em produção por causa de variáveis como velocidade da rede ou o desempenho do hardware.

O maior desafio de se construir código JavaScript assíncrono é coordenar a ordem de execução das funções e lidar com o gerenciamento de erros que podem (e vão) ser lançados. Se um erro é lançado durante uma execução síncrona, podemos capturá-lo, tratá-lo e prosseguir com a execução normalmente, porém se o mesmo acontece num processamento assíncrono, não teremos como saber

Fique por dentro

Este artigo é útil para desenvolvedores que desejam adicionar recursos assíncronos em suas aplicações sem necessariamente usar Ajax. O Ajax resolve muitos dos problemas quando precisamos nos comunicar com um servidor e trocar requisições e respostas, porém não se mostra muito bom para lidar com tais manipulações dentro do código JavaScript no cliente. Você verá o que são as Promises e como elas podem ser usadas para atacar esse problema, bem como suas principais características, prós e contras e como você pode fazer a integração das mesmas com frameworks como Bluebird e jQuery.

em que parte do código aconteceu e no máximo uma mensagem no Console aparecerá, além do fato de uma execução importante para o sistema não ter funcionado e não sabermos o que houve lá. As Promises encapsulam esse problema fornecendo formas para organizar callbacks (funções de retorno) que são fáceis de implementar e manter. Além disso, quando erros acontecem podemos gerenciá-los de fora da lógica primária da aplicação sem a necessidade de código sujo para checar se algo de errado aconteceu há cada novo passo da execução.

Uma promise é um objeto que serve como uma “lacuna” para um valor. Esse valor é usualmente o resultado de uma operação assíncrona como uma requisição HTTP ou a leitura de um arquivo no disco. Quando uma função assíncrona é chamada ela pode imediatamente retornar um objeto Promise. Usando esse objeto, você pode registrar callbacks que executarão quando a operação ocorrer com sucesso ou com erros.

Neste artigo trataremos de explorar os principais recursos, funções e gerenciamento de estado das promises. Você verá como usar todo o potencial da API de comunicação assíncrona, bem como entender, via exemplos básicos, quais os melhores fluxos de uso

de uma promise. Também entenderemos o que são web workers, objetos deferidos e como eles se integram com as promises e com outras bibliotecas como o jQuery.

JavaScript assíncrono

Vamos começar com um trecho de código muito comum nos projetos JavaScript. O código da **Listagem 1** faz uma solicitação HTTP usando o objeto XMLHttpRequest (XHR) e usa um loop while que se estende por três segundos. Embora seja geralmente uma má prática implementar “um atraso” com o loop while, é uma boa maneira de ilustrar como o JavaScript é executado. Veremos no código quando o callback do *listener* será disparado.

Listagem 1. Exemplo de código assíncrono com XHR.

```
01 // Faz um request assíncrono
02 var async = true;
03 var xhr = new XMLHttpRequest();
04 xhr.open('get','dados.json',async);
05 xhr.send();
06
07 // Cria um timing de três segundos
08 var timestamp = Date.now() + 3000;
09 while (Date.now() < timestamp);
10
11 // Quando os três segundos passarem,
12 // add um listener aos eventos de xhr.load e xhr.error
13 function listener() {
14   console.log('Boas vindas do listener');
15 }
16 xhr.addEventListener('load', listener);
17 xhr.addEventListener('error', listener);
```

O código cria um novo objeto XHR que representa a base de tudo que é Ajax no JavaScript, logo mesmo APIs que encapsulam requisições Ajax simplificadas, como o jQuery por exemplo, usam esses objetos por detrás. Na linha 4 abrimos uma nova conexão usando o método HTTP “GET” (já que se trata do modelo request-response tradicional do HTTP, porém assíncrono, sem loading de página) e informamos o endereço final para onde a requisição será enviada. Você pode substituir esse valor por qualquer outro endpoint que reconheça o request.

Na linha 5 enviamos a requisição. Precisamos aguardar até que ela retorne para processar o resultado. Como o JavaScript, diferente de outras linguagens server side, não espera um método retornar para prosseguir a execução no chamador, criamos um loop de três segundos (linhas 8 e 9) via objeto Date do JavaScript. Após isso, quando retornar adicionamos o listener e cadastramos seu método ouvinte para quando estiver carregado (“load”) e algum erro estourar (“error”) (linhas 16 e 17).

O leitor pode se perguntar se o listener será mesmo “sempre” chamado no exemplo anterior, já que o timing via while também não é uma opção 100% garantida. Mas sim, ele será sempre chamado porque o callback assume prioridade na execução para o JavaScript, uma vez que ele precisa dar um retorno sempre da chamada à função.

As funções de callback, por sua vez, podem ser invocadas tanto de forma síncrona quanto assíncrona (isto é, antes ou depois da função para qual foram passadas retornar). Vejamos na **Listagem 2** dois exemplos de cada uma dessas formas.

Listagem 2. Exemplos de callbacks síncrono e assíncrono.

```
01 // Callback síncrono
02 function callback(msg) {
03   console.log(msg);
04 }
05 msgs.forEach(callback);
06
07 // Callback assíncrono
08 function reallocarElemento() {
09   console.log('realocando...');
```

Na primeira função (linha 2) a função `callback()` recebe uma string `msg` para exibir no Console o valor. Logo após usamos um `forEach` para iterar sobre todas as mensagens e exibi-las. Por padrão, essa função executa de forma síncrona, logo a função também o será.

Um exemplo clássico de callbacks assíncronos é a função `window.requestAnimationFrame()` usada para redesenhar algo numa tela quando um objeto de pintura, como o Canvas, assim solicitar. Na linha 8 temos uma função simples de callback (`reallocarElemento`) que só imprime no Console uma mensagem também correspondente. Após a chamada direta no objeto `window` da linha 12, a função de callback só será de fato invocada entre os intervalos de repintura do browser. No exemplo, a mensagem “Última linha do script” é escrita no Console antes do “realocando...” porque a função `requestAnimationFrame` retorna imediatamente e invoca `reallocarElemento()` mais a frente.

Uso básico das Promises

Comecemos o entendimento sobre o uso de Promises com um exemplo que invoca funções de callback de “sucesso” ou “erro” dependendo do retorno da execução, como mostra a **Listagem 3**.

A função `carregarImg()` usa um objeto `Image` da HTML para carregar uma imagem qualquer setando diretamente o atributo `src`. O browser assincronamente carrega a imagem baseado no seu atributo `src` e enfileira ambas funções de callback (`onload` e `onerror`) após finalizada a ação.

Uma vez que o método `carregarImg()` é assíncrono, ele aceita callbacks em vez de retornar a imagem imediatamente via função. Entretanto, se a função `carregarImg()` foi modificada para retornar uma promise, você deve anexar as callbacks à promise em vez de passá-las como argumentos de função. Veja na **Listagem 4** como ficaria a função `carregarImg()` quando se retorna uma promise.

Listagem 3. Exemplo de execução com sucesso e erro.

```
01 carregarImg('devmedia.png',
02   function onsuccess(img) {
03     // Add a imagem à página web atual
04     document.body.appendChild(img);
05   },
06   function onerror(e) {
07     console.log('Aconteceu um erro ao carregar a imagem');
08     console.log(e);
09   }
10 );
11
12 function carregarImg(url, success, error) {
13   var img = new Image();
14   img.src = url;
15
16   img.onload = function () {
17     success(img);
18   };
19   img.onerror = function (e) {
20     error(e);
21   };
22 }
```

Listagem 4. Código XML do pom para gerência de dependências.

```
01 // Assume que a função carregarImg retorna uma promise
02 var promise = carregarImg('devmedia.png');
03
04 promise.then(function (img) {
05   document.body.appendChild(img);
06 });
07
08 promise.catch(function (e) {
09   console.log('Ocorreu um erro ao carregar a imagem');
10   console.log(e);
11 });
```

Podemos transcrever o código na seguinte afirmação: “Carregue uma imagem, então adicione-a ao documento HTML ou exiba um erro se ela não puder ser carregada.”. A promise que a função carregarImg() retorna tem um método *then()* que registra um callback para usar quando a operação suceder e um método *catch()* para lidar com os erros. Todavia, ambos retornam objetos promise, logo o registro de callback é comumente feito através do encadeamento da chamada desses dois métodos juntos, como podemos ver na **Listagem 5**.

Perceba como esse tipo de estrutura se assemelha às do jQuery. Nele podemos chamar funções encadeadas que sempre retornam o mesmo objeto. As Promises funcionam da mesma forma.

Na **Listagem 6** temos a implementação para carregarImg() que retorna uma promise.

A função construtor global chamada “Promise” expõe toda a funcionalidade das promises. Nesse exemplo, a função carregarImg() cria um novo objeto desse tipo e o retorna. Quando *Promise* é usado como um construtor ele requer um callback conhecido como “*resolver function*”. O resolver atende a dois propósitos:

- Receber os argumentos *resolve* e *reject*, que são funções usadas para atualizar a promise uma vez que o resultado seja conhecido;
- E lidar com qualquer erro lançado a partir desse resolver. A função *reject* será implicitamente usada para rejeitar a promise caso isso aconteça.

Listagem 5. Encadeamento de chamadas usando *then* e *catch*.

```
01 loadImage('devmedia.png').then(function (img) {
02   document.body.appendChild(img);
03 }).catch(function (e) {
04   console.log('Ocorreu um erro ao carregar a imagem');
05   console.log(e);
06 });
```

Listagem 6. Função carregarImg retornando uma promise.

```
01 function carregarImg(url) {
02   var promise = new Promise(
03     function resolver(resolve, reject) {
04       var img = new Image();
05       img.src = url;
06
07       img.onload = function () {
08         resolve(img);
09       };
10
11       img.onerror = function (e) {
12         reject(e);
13       };
14     }
15   );
16   return promise;
17 }
```

Agora, toda a lógica que antes era feita em carregarImg() agora é feita dentro do resolver. Dessa forma, encapsulamos qualquer regra de negócio para também ser efetuada de forma assíncrona dentro da estrutura da promise. Conclusão: a função *resolve()* é chamada quando a imagem for carregada e a função *reject()* é chamada se a imagem não puder ser carregada. Quando uma operação representada por uma promise completar, o resultado é armazenado e disponibilizado para quaisquer callbacks que a promise invocar. O resultado é passado para a promise como um parâmetro da função *resolve* ou *reject*. No caso da nossa função, a imagem é passada ao *resolve()*, então qualquer callback registrado com *promise.then()* receberá a imagem em questão.

Múltiplos consumidores

Quando múltiplos pedaços de código estão interessados na saída de uma mesma operação assíncrona, eles podem usar a mesma promise. Por exemplo, você pode recuperar o perfil de um usuário do servidor e usá-lo para exibir o seu nome em uma barra de navegação. Esse dado também pode ser usado numa página de conta de usuário para exibir seu nome completo de perfil. O código da **Listagem 7** demonstra isso através do uso de uma promise para rastrear sempre que o perfil do usuário tiver sido recebido. Duas funções independentes usam a mesma promise para exibir dados sempre que estiverem disponíveis.

Aqui criamos um objeto simples com uma propriedade “*promisePerfil*” e um método “*getPerfil*” para retornar a promise que será usada como um objeto contendo o perfil de usuário. Então, o script passa o usuário para os objetos *navbar* (linha 14) e *account* (linha 22) que, por sua vez, exibirão informações sobre o respectivo perfil.

Como a promise funciona como uma lacuna para o resultado de uma operação, neste caso o `user.promiseProfile` funciona como uma lacuna usada pelas funções `navbar.show()` e `account.show()`. Essas funções podem ser chamadas a qualquer momento antes ou depois dos dados de perfil estarem disponíveis na execução. Ao mesmo tempo, os callbacks que elas usam para imprimir os dados no Console serão somente invocados uma vez que o perfil esteja carregado. Isso acaba com a necessidade de uma cláusula `if` em cada função para checar quando os dados estão prontos.

O resultado da execução pode ser visto na **Figura 1**.

Gerenciamento de estados

O estado de uma operação representada por uma promise é armazenado dentro da mesma. Em qualquer momento de uma execução, uma operação varia entre os estados de “não iniciada”, “em progresso”, “finalizada”, ou “parada e não pode completar”. Essas condições são representadas por três estados mutuamente exclusivos:

- **Pending**: A operação não iniciou ou está em progresso;
- **Fulfilled**: A operação foi completada;
- **Rejected**: A operação não pôde ser completada.

A **Figura 2** traz uma representação de tais estados e seus relacionamentos uns com os outros.

Listagem 7. Uma promise com múltiplos consumidores.

```
01 var user = {
02   promisePerfil: null,
03   getPerfil: function () {
04     if (!this.promisePerfil) {
05       // Assume que a função ajax() retorna uma promise que é eventualmente
06       // preenchida com {nome:'DevMedia', subscribedToSpam: true}
07       this.promisePerfil = ajax('/seuURL/');
08     }
09     return this.promisePerfil;
10   }
11 };
12
13 var navbar = {
14   show: function (user) {
15     user.getPerfil().then(function (perfil) {
16       console.log('*** Navbar ***');
17       console.log('Nome: ' + perfil.nome);
18     });
19   }
20 };
21 var account = {
22   show: function (user) {
23     user.getPerfil().then(function (perfil) {
24       console.log('*** Informações da Conta ***');
25       console.log('Nome: ' + perfil.nome);
26       console.log('Envia muitos emails? ' + perfil.subscribedToSpam);
27     });
28   }
29 };
30
31 navbar.show(user);
32 account.show(user);
```

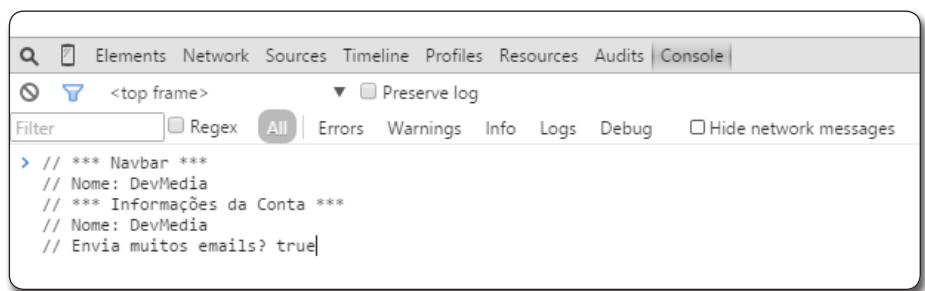


Figura 1. Resultado da execução do exemplo no Console

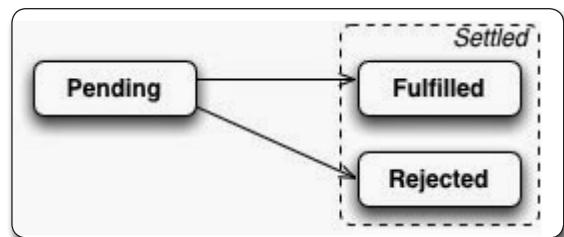


Figura 2. Variância de estados entre as promises

Nos exemplos anteriores nos referimos aos estados “fulfilled” e “rejected” como *success* e *error*, respectivamente. Existe uma diferença entre estes termos. Uma operação pode completar com um erro (apesar de não ser a forma mais apropriada) ou pode não completar porque foi cancelada mesmo que nenhum erro tenha ocorrido. Ainda assim, os primeiros termos originais são os mais usados e são padrão para se referir a ações com sucesso e erro.

Quando uma promise sai do estado de “pending” ela entra no estado de “settled” (liquidada). É só um termo genérico para dizer que a promise atingiu seu estado final; além disso, a liquidação de uma promise é permanente. Isso porque tal comportamento precisa se alinhar com o que temos na vida real: uma operação completada não pode voltar a ser “incompleta” e seu resultado, portanto, não muda. Por outro lado, uma operação que falhou deve ser “refeita” e isso significa que múltiplas tentativas retornam múltiplos valores diferentes. Conclusão: uma promise nova representa cada “tentativa”, logo um jeito melhor de descrever uma promise seria “um espaço (lacuna) para o resultado de uma tentativa de uma operação”.

Na **Listagem 8** temos um exemplo de como uma promise pode ser modificada apenas uma vez. O código chama as funções de estado `resolve()` e `reject()` que falamos antes no mesmo construtor da promise, para simplificar a implementação. A chamada à função `resolve()` modifica o estado da promise para “pendente” ou “completado”. Após isso, quaisquer novas chamadas para as mesmas funções serão ignoradas porque a promise já terá finalizado.

Perceba que após resolver o valor da promise com a constante PI da API Math do JavaScript (linha 2), as chamadas posteriores das linhas 3 e 4 não alterarão nem chamarão a respectiva função em detrimento desse controle de estados das promises.

A imutabilidade de uma promise que entra no estado “settled” torna o código mais fácil de se trabalhar. Se tivéssemos o contrário,

ou seja, promises podendo alterar seu estado ou valor após ser completada ou rejeitada, aí teríamos problemas de *race conditions* (condições de corrida), onde perdemos a noção de qual código executa primeiro e, portanto, o controle da execução como um todo. Felizmente, as regras de transição de estado para as promises previnem esse tipo de problema e essa é uma das principais vantagens de se utilizá-las.

Listagem 8. Exemplo de promise com estado que nunca muda depois de completada ou rejeitada.

```
01 var promise = new Promise(function (resolve, reject) {  
02   resolve(Math.PI);  
03   reject(0); // Não faz nada  
04   resolve(Math.sqrt(-1)); // Não faz nada  
05});  
06  
07 promise.then(function (numero) {  
08   console.log('O número é ' + numero);  
09});  
10  
11 // Saída do Console:  
12 // O número é 3.141592653589793
```

Encadeamento de promises

Já vimos como *then* e *catch* retornam promises para um fácil encadeamento de chamadas de funções, porém eles não retornam uma referência para a mesma promise. Sempre que qualquer um destes métodos é chamado uma nova promise é criada e retornada encapsulando a nova funcionalidade do método que a chamou. Vejamos a **Listagem 9**, onde temos um exemplo de uma função *then* retornando uma nova promise.

Listagem 9. Exemplo de função *then* sempre retornando uma nova promise.

```
01 var p1, p2;  
02  
03 p1 = Promise.resolve();  
04 p2 = p1.then(function () {  
05   // ...  
06});  
07  
08 console.log('p1 e p2 são objetos diferentes: ' + (p1 !== p2));  
09  
10 // Saída do Console:  
11 // p1 e p2 são objetos diferentes: true
```

Como vimos, após chamar a função *resolve()* a promise não pode mais ser modificada. Logo, a referência *p1* aponta para um espaço em memória diferente da referência *p2*, já que a função *then* também cria seu próprio objeto.

Também é possível, além de encadear chamadas sucessivas a *then* e *catch*, variar entre chamadas de mesma função: vários *then* e/ou *catch* chamando uns aos outros (**Listagem 10**).

Cada chamada à função *then* retorna uma nova promise que você pode usar para atrelar a um outro callback. Esse padrão permite a cada “passo” enviar seu próprio valor de retorno para o próximo passo. Se um passo retorna uma promise em vez de um valor, então os próximos passos recebem qualquer valor que

tenha sido usado para “completar” aquela promise. Vejamos na **Listagem 11** todas as formas possíveis de “completar” uma promise criada pela função *then*.

Listagem 10. Exemplo de encadeamento de funções *then*.

```
01 passo1().then(  
02   function passo2(resultadoDoPasso1) {  
03     // ...  
04   }  
05 ).then(  
06   function passo3(resultadoDoPasso2) {  
07     // ...  
08   }  
09 ).then(  
10   function passo4(resultadoDoPasso3) {  
11     // ...  
12   }  
13 );
```

Listagem 11. Passando valores em uma sequência de passos.

```
01 Promise.resolve('ola devmedia!).then(  
02   function passo2(resultado) {  
03     console.log('passo 2 recebido ' + resultado);  
04     return 'Saudações do passo 2'; // Valor de retorno explícito  
05   }  
06 ).then(  
07   function passo3(resultado) {  
08     console.log('passo 3 recebido ' + resultado); // Sem valor de retorno explícito  
09   }  
10 ).then(  
11   function passo4(resultado) {  
12     console.log('passo 4 recebido ' + resultado);  
13     return Promise.resolve('valor fulfilled'); // Retorna uma promise  
14   }  
15 ).then(  
16   function passo5(resultado) {  
17     console.log('passo 5 recebido ' + resultado);  
18   }  
19 );  
20  
21 // Saída do Console:  
22 // passo 2 recebido ola devmedia!  
23 // passo 3 recebido Saudações do passo 2  
24 // passo 4 recebido undefined  
25 // passo 5 recebido valor fulfilled
```

Um valor de retorno explícito é devolvido na primeira função, *passo2()*, do tipo string. Uma vez que a função *passo3()* não retorna um valor explicitamente, a função é “completada” com o valor *undefined*. Já em relação à função *passo4()*, que retorna uma nova promise já resolvida, seu valor de retorno será usado como parâmetro para a função seguinte, *passo5()*, imprimindo assim a mensagem.

Trata-se de um encadeamento de chamadas que se comunicamumas com as outras apenas com as estruturas das próprias promises, sem efetuar chamada direta a cada função.

Propagação e tratamento de erros

Independente da linguagem ou plataforma usada, o tratamento e programação de erros são recursos básicos que devem ser sempre entendidos e usados para organizar a sua implementação.

Programação assíncrona em JavaScript com Promises

Rejeições e erros são propagados através das cadeias de promises. Quando uma promise é rejeitada todas as promises subsequentes da cadeia são rejeitadas em um efeito dominó até um manipulador *onRejected* ser encontrado. Na prática, uma função *catch* é utilizada no fim de uma cadeia (ver **Listagem 12**) para tratar todas as rejeições. Esta abordagem trata a cadeia como uma única unidade que a última promise “cumprida” ou “rejeitada” representa.

Listagem 12. Lidando com handlers de rejeição no fim de uma cadeia.

```
01 Promise.reject(Error('más notícias')).then(  
02   function passo2() {  
03     console.log('Isso nunca é executado');  
04   }  
05 ).then(  
06   function passo3() {  
07     console.log('Isso também nunca é executado');  
08   }  
09 ).catch(  
10   function (error) {  
11     console.log('Algo de errado aconteceu. Inspecione o erro para mais  
detalhes.');//  
12     console.log(error); // Objeto de erro com mensagem:'más notícias'  
13   }  
14 );  
15  
16 // Saída do Console:  
17 // Algo de errado aconteceu. Inspecione o erro para mais detalhes.  
18 // [Error object] { message:'más notícias' ... }
```

O código inicia uma cadeia de promises através da criação de uma promise de estado rejected usando o método *Promise.reject()*. Duas novas promises são criadas em seguida através da adição de chamadas à função *then* e finalizadas com uma chamada a *catch* para lidar com as exceções.

Observe que o código das funções *passo2()* e *passo3()* nunca executam. Essas funções são chamadas apenas quando a promise a que elas estão atreladas é “completada”. Logo, considerando que a promise no topo da cadeia foi rejeitada, todas as funções de callback subsequentes na cadeia são ignoradas até que o *catch* seja chamado.

As promises também são rejeitadas quando um erro é lançado numa função de callback passada para o *then* ou em uma função *resolver* qualquer passada para o construtor de Promise. O código da **Listagem 13** é similar ao último, porém lançando uma exceção em vez da função *Promise.reject()*.

Ambos os exemplos fornecem um objeto JavaScript de erro quando uma promise é rejeitada. Apesar de qualquer valor estar habilitado ao rejeitar uma promise, inclusive undefined, é aconselhado sempre usar um objeto de erro. Ao criar um erro podemos capturar a pilha de chamadas dentro do *catch* que acabou o originando e rastrear a raiz do problema para corrigi-lo.

Efeito Cascata Assíncrona

O uso de funções e promises assíncronas é contagiante. Quando começa a usá-las, naturalmente elas se espalham pelo código. Quando você tem uma função assíncrona qualquer código que a chama passa a conter também partes assíncronas dentro dele. O

processo de outras funções se tornarem assíncronas através de extensões cria um efeito chamado de “efeito cascata” que continua até que a pilha de chamadas que originou o mesmo termine. Vejamos o exemplo exibido na **Listagem 14**, nele usamos três funções. Observe como a função assíncrona *ajax* força as demais a também serem assíncronas.

Listagem 13. Rejeitando uma promise através de um erro.

```
01 rejeitarCom('más notícias').then(  
02   function passo2() {  
03     console.log('Isso nunca é executado');  
04   }  
05 ).catch(  
06   function (error) {  
07     console.log('Erro!!');  
08     console.log(error); // Objeto erro com mensagem:'más notícias'  
09   }  
10 );  
11  
12 function rejeitarCom(val) {  
13   return new Promise(function (resolve, reject) {  
14     throw Error(val);  
15     resolve('Não usado'); // Nunca será executada  
16   });  
17 }  
18 // Saída do Console:  
19 // Erro!!  
20 // [Error object] { message:'más notícias' ... }
```

Listagem 14. Exemplo demonstrando o efeito cascata assíncrona.

```
01 showTexto().then(function () {  
02   console.log('Olá Efeito Cascata Assíncrona!');  
03 });  
04  
05 function showTexto() {  
06   return getTexto().then(function (texto) {  
07     console.log(texto);  
08   });  
09 }  
10  
11 function getTexto() {  
12   // Assume que ajax() retorna uma promise que é  
13   // completada pelo json para {conteúdo:'A loja está fechada!'}  
14   return ajax('/seuURL*').then(function (json) {  
15     var texto = JSON.parse(json);  
16     return texto.conteúdo;  
17   });  
18 }  
19  
20 // Saída do Console:  
21 // A loja está fechada!  
22 // Olá Efeito Cascata Assíncrona!
```

O esforço para recuperar e exibir os textos está dividido em três funções: *showTexto()*, *getTexto()* e *ajax()*. As funções formam uma cadeia de promises que inicia com *ajax()* e termina com o objeto retornado por *showTexto()*. A função *ajax()* retorna uma promise representando o resultado de uma requisição assíncrona XHR. Se a função *ajax()* retornasse o JSON de forma síncrona, *getTexto()* e *showTexto()* não consumiriam ou retornariam nenhuma promise.

Como regra geral, qualquer função que use uma promise deve também retorná-la. Quando uma promise não é propagada, o

código que a chamou não consegue saber quando a promise está completada e, portanto, não pode fazer nada em seguida. É fácil imaginar cenários onde o chamador da função não se importa quando o trabalho assíncrono está terminado, mas isso não é uma regra universal. É mais fácil retornar a promise em cada função em vez de ter de armazená-las em listas ou objetos de sessão para recuperar no futuro.

Considere fluxos normais de execução com estruturas condicionais envolvidas. Por exemplo, algumas ações podem exigir a autenticação de um usuário. No entanto, uma vez que um usuário é autenticado, ele não precisa repetir esse passo quando precisar entrar novamente. Podemos salvar suas informações e o logar automaticamente nas próximas vezes.

Como exemplo, vamos usar um leitor de livro eletrônico que requer autenticação antes que o usuário possa acessar quaisquer outros recursos. Existem várias maneiras de codificar esse cenário. A **Listagem 15** mostra os primeiros passos para isso.

Listagem 15. Fluxo assíncrono de condição.

```
01 var usuario = {  
02   autenticado: false,  
03   login: function () {  
04     // Retorna uma promise para o request de login  
05     // Configura "autenticado" para true e "completa" a promise quando o  
06     // /login acontece com sucesso  
07   };  
08 }  
09 // Evita o estilo de execução condicional assíncrona  
10 function showMenuPrincipal() {  
11   if (!usuario.autenticado) {  
12     usuario.login().then(showMenuPrincipal);  
13     return;  
14   }  
15 }  
16 // ... Código para exibir o menu principal  
17};
```

Nesta implementação de `showMainMenu`, o menu é exibido imediatamente se o usuário já está autenticado. Caso contrário, o processo de login assíncrono é executado e `showMenu` é chamado novamente uma vez que a autenticação tenha sido bem-sucedida.

Um problema aqui é que o menu não será exibido se o processo de login falhar. Isso porque `showMainMenu` depende de uma promise, mas não retorna uma como descrito na seção anterior.

Um segundo problema é que `showMainMenu` pode comportar-se de forma síncrona ou assincronamente, dependendo de se o usuário já está autenticado ou não. Consequentemente, este estilo de código cria vários caminhos de execução que podem ser difíceis de entender, além de criar comportamento inconsistente no código.

Como podemos ver na **Listagem 16**, os problemas em `showMainMenu` podem ser resolvidos substituindo uma *promise resolvida* se o usuário já estiver autenticado.

Agora o menu será sempre exibido assincronamente usando tanto a promise que o método `usuario.login()` retornou quanto a promise resolvida que foi substituída pelo processo de login.

Podemos eliminar a necessidade dessa promise substituta chamando a função `usuario.login()` em ambos os lugares (**Listagem 17**).

Veja como essa mudança simplificou o código. Isso não significa que todos os passos para efetuar o login precisem ser repetidos o tempo todo. A promise que `login()` retorna pode ser cacheada (salva temporariamente) e reusada sempre que quiser, como podemos ver na **Listagem 18**.

Listagem 16. Substituindo uma promise resolvida.

```
01 function showMenuPrincipal() {  
02   var p = (!usuario.autenticado) ? usuario.login() : Promise.resolve();  
03  
04   return p.then(function () {  
05     // ... Código para exibir o menu principal  
06   });  
07 }
```

Listagem 17. Encapsulando lógica condicional com uma promise.

```
01 function showMenuPrincipal() {  
02   return usuario.login().then(function () {  
03     // ... Código para exibir o menu principal  
04   });  
05 }
```

Listagem 18. Cacheando uma promise.

```
01 var usuario = {  
02   promiseLogin: null,  
03   login: function () {  
04     var eu = this;  
05  
06     if (this.promiseLogin === null) {  
07       this.promiseLogin = ajax('/suuaUrl/');  
08  
09     // Remove a promiseLogin cacheada quando uma falha ocorre  
10     // para evitar repetição  
11     this.promiseLogin.catch(function () {  
12       eu.promiseLogin = null;  
13     });  
14   }  
15   return this.promiseLogin;  
16 };
```

No exemplo, o `promiseLogin` é criado na primeira vez que o método `login()` é chamado. Todas as chamadas subsequentes a `login()` retornam a mesma promise a menos que o processo de login sofra alguma falha. Caso isso aconteça, a promise cacheada é removida e então o processo pode começar novamente.

Execução Paralela

Múltiplas tarefas assíncronas podem ser executadas em paralelo, como mostrado na **Listagem 19**. Considere um site financeiro que mostra um saldo atualizado para todas as suas contas bancárias e cartões de crédito cada vez que se efetuar o login. O saldo atualizado de cada instituição pode ser solicitado em paralelo e exibido assim que é recebido.

Perceba novamente o uso do `forEach` para iterar sobre todos os valores do vetor de contas e efetuar a chamada às respectivas

Programação assíncrona em JavaScript com Promises

URLs do serviço de cada uma que, por sua vez, irão retornar os dados do saldo daquela conta em específico. A função ajax() retorna uma promise diferente para cada conta, assim podemos usá-la para exibir as informações do saldo sem se preocupar com o sincronismo do processo. O foco da implementação em si é mostrar que as promises conseguem fazer todas as chamadas aos serviços de forma assíncrona.

Promises também são boas para consolidar tarefas paralelas em um único resultado. Suponha que uma mensagem deve ser exibida informando ao usuário quando todos os saldos de conta estão atualizados. Você pode criar uma promise consolidada usando a função *Promise.all()* que mapeia promises para seus resultados, conforme explicado na **Listagem 20**. Em suma, Promise.all() retorna uma nova promise que é cumprida quando todas as demais que ela recebe são também cumpridas. E se alguma das promises que ela recebe for rejeitada, a nova promise também o será.

Listagem 19. Executando tarefas assíncronas em paralelo.

```
01 // Define cada conta
02 var contas = ['Conferindo Conta', 'Programa de Milhas', 'Cartão Universitário'];
03
04 console.log('Atualizando informação do balanço...');

05 contas.forEach(function (account) {
06   // a função ajax() retorna uma promise completada para o balanço de conta
07   ajax(/*URL da Conta*/).then(function (balanco) {
08     console.log('Balanço:' + account + balanco);
09   });
10 });
11
12
13 // Saída do Console:
14 // Atualizando informação do balanço...
15 // Balanço Conferindo Conta: 5981
16 // Balanço Programa de Milhas: 1200
17 // Balanço Cartão Universitário: 0
```

Listagem 20. Tarefas assíncronas em paralelo com Promise.all().

```
01 var requests = accounts.map(function (account) {
02   return ajax(/*URL da Conta*/);
03 });
04
05 // Atualize a mensagem de status quando todos os requests completarem
06 Promise.all(requests).then(function (balances) {
07   console.log('Todos os ' + balances.length + ' balanços estão atualizados');
08 }).catch(function (error) {
09   console.log('Um erro ocorreu ao recuperar a informação de balanço');
10   console.log(error);
11 });
12
13 // Saída do Console:
14 // Todos os três balanços estão atualizados
```

Em vez de iterar sobre todas as contas usando um `forEach`, a função `map` é usada para criar uma série de promises que representam uma requisição de saldo da conta. Em seguida, a função `Promise.all()` consolida as promises em uma só. Uma matriz contendo todos os saldos de conta resolve a promise consolidada. Neste exemplo, a propriedade de comprimento dessa matriz é usada para exibir o número dos saldos recuperados.

Bibliotecas e Frameworks

Antes da API ES6 Promise existiam muitas bibliotecas JavaScript e frameworks implementando sua própria versão de Promises. Algumas bibliotecas foram escritas com o único propósito de fornecer promises enquanto outras, como o jQuery, as adicionaram para lidar com suas APIs assíncronas.

A biblioteca Promise também consegue atuar como um *polyfill* (estrutura que se adapta em versões anteriores) em navegadores mais antigos e outros ambientes onde as promises nativas não são fornecidas. Ela também consegue complementar a API padrão do navegador com um vasto conjunto de funções para gerenciamento das promises. Se o seu código utiliza apenas promises que você criou, então pode ser uma boa pedida escolher uma biblioteca e tirar o máximo proveito de sua API estendida. E se você está lidando com promises que outras bibliotecas produziram, pode encapsulá-las com as da sua biblioteca escolhida para acessar os recursos adicionais.

Todo o processo de interoperabilidade entre as diferentes bibliotecas Promises que existem se dá através de um contrato chamado de “*thenable*” (proveniente da função `then` que vimos). Qualquer objeto que tenha um método `then(onFulfilled, onRejected)` pode ser encapsulado por qualquer implementação padrão de promise. Vejamos o exemplo demonstrado pela **Listagem 21** para entender como isso funciona.

Listagem 21. Encapsulando um thenable para interoperabilidade.

```
01 function thenable(valor) {
02   return {
03     then: function (onfulfill, onreject) {
04       onfulfill(valor);
05     }
06   };
07 }
08
09 var promise = Promise.resolve(thenable('voila!'));
10
11 promise.then(function(resultado) {
12   console.log(resultado);
13 });
14
15 // Saída do Console:
16 // voila!
```

Veja como a função `thenable()` encapsula toda a regra de negócio e retorna tudo em função `then()`. Assim, independente da API de promises que você estiver usando (linha 9), quando tivermos de chamar a função `resolve()` basta passar `thenable()` como parâmetro.

Promises com Bluebird

A Bluebird é uma biblioteca open source com uma rica API e excelente performance. No seu repositório do GitHub (vide seção [Links](#)) é possível encontrar vários exemplos de implementações onde a API supera as demais em performance, incluindo versões nativas do motor V8 do JavaScript usado pelo Node.js e Google Chrome, por exemplo. Ela também oferece muitas outras features

incluindo um jeito elegante de gerenciar a execução de contextos, encapsulamento das APIs do Node.js, trabalhar com coleções de promises e manipular valores de preenchimento na linguagem, dentre outras.

Quando a Bluebird é incluída numa página web usando uma tag de script, ela automaticamente sobre escreve o objeto Promise global por padrão com sua própria versão desse objeto. Ela também pode ser carregada no browser de outras formas como um módulo AMD usando require.js, bem como está disponível como um pacote npm para uso em conjunto com o Node.js. Se você desejar usar a API da Bluebird porém com o objeto Promise nativo do browser, basta implementar o código `Promise.noConflict()` e o browser passará a ignorar a mesma. Vejamos então na **Listagem 22** um exemplo de uso da Bluebird com o código de interoperabilidade que criamos antes.

Listagem 22. Encapsulando uma promise nativa com uma promise Bluebird.

```
01 // Assume que a bluebird já tenha sido carregado via
<script src="bluebird.js"></script>
02 var Bluebird = Promise.noConflict();
// Restaura uma referência à Promise anterior
03 var nativePromise = Promise.resolve(); // Promise Nativa
04
05 var b = Bluebird.resolve(nativePromise);
// Encapsula uma promise nativa com uma promise Bluebird
06
07 console.log('Pending?' + b.isPending()); // Pending? false
08 console.log('Fulfilled?' + b.isFulfilled()); // Fulfilled? true
09 console.log('Rejected?' + b.isRejected()); // Rejected? false
```

Veja como é fácil encapsular uma promise nativa em uma da Bluebird (linha 5), basta ter o arquivo de script importado e usar o método `resolve()`. Toda as nomenclaturas de métodos dos fluxos de estado que vimos se mantêm de uma API para outra, inclusive na nativa. Sempre que você quiser saber os estados atuais da promise, a própria API ajuda com métodos de retorno de prefixo “is”, tal como vemos nas linhas 7 a 9.

Promises com jQuery

No jQuery, objetos deferidos representam operações assíncronas. Um objeto deferido é como uma promise cujas funções `resolve` e `reject` são expostas como métodos. A **Listagem 23** mostra uma função `carregarImg()` que faz uso de um objeto como esse.

A API padrão das Promises encapsula as funções `resolve` e `reject` dentro da promise em si. Por exemplo, se você tem um objeto `promise p`, você não pode chamar as funções `p.resolve()` ou `p.reject()` porque estas não estão atreladas ao objeto `p`. Qualquer código que recebe uma referência a `p` pode atrelar callbacks usando `p.then()` ou `p.catch()` mas o mesmo nunca poderá controlar quando `p` se tornará “completado” ou “rejeitado”.

Ao mesmo tempo, ao usar objetos deferidos não significa que você tenha que expor os métodos `resolve()` e `reject()` em toda parte.

Um objeto deferido no jQuery também expõe uma promise que pode ser enviada a qualquer código que não precise chamar às mesmas funções. Comparemos as duas funções da **Listagem 24**. A primeira é uma versão revisada da função `carregarImg()` da listagem anterior que retorna `deferido.promise()` enquanto a segunda é uma função equivalente implementada com uma promise padrão.

Listagem 23. Objeto deferido simples em jQuery.

```
01 function carregarImg(url) {
02   var deferido = jQuery.Deferred();
03   var img = new Image();
04   img.src = url;
05
06   img.onload = function() {
07     deferido.resolve(img);
08   };
09
10  img.onerror = function (e) {
11    deferido.reject(e);
12  };
13  return deferido;
14}
```

Listagem 24. Objeto deferido simples em jQuery.

```
01 function carregarImg(url) {
02   var deferido = jQuery.Deferred();
03   //
04   return deferido.promise();
05
06
07 function carregarImgSemDeferido(url) {
08   return new Promise(function resolver(resolve, reject) {
09     var image = new Image();
10     image.src = url;
11     image.onload = function () {
12       resolve(image);
13     };
14     image.onerror = reject;
15   });
16 }
```

A principal diferença entre as duas funções é que a primeira pode lançar um erro síncrono enquanto quaisquer erros lançados dentro da segunda (com a `Promise`) são capturados e usados para rejeitar a promise. As Promises criadas pela API de Deferred do jQuery não implementam os padrões de acordo com a API padrão ES6 de Promises do JavaScript. Além disso, alguns dos nomes de métodos nas promises do jQuery diferem das da especificação oficial como, por exemplo, `[jQueryPromise].fail()` que é equivalente a `[promisePadrao].catch()`.

Outra diferença importante está no tratamento de erros nos métodos de callback `onFulfilled` e `onRejected`. As promises padrão automaticamente capturam quaisquer erros lançados nestes métodos e os convertem em “rejeições”. Nas promises do jQuery, os mesmos erros são lançados acima para a pilha de métodos chamadores como exceções não capturadas.

Por tudo isso, mesmo muitos desenvolvedores considerando o uso de objetos deferidos mais fácil de entender, é importante

se ater que tais objetos muitas vezes criam situações onde não é possível resolver a promise no lugar em que foi criada, e isso é um problema. Por exemplo, suponha que você está usando um browser web para executar tarefas longas e pesadas. Você pode usar promises para representar a saída dessas tarefas; o código que recebe a resposta do browser é quem se responsabilizará por resolver a promise, portanto ele precisa de acesso às funções `resolve()` e `reject()`. Vejamos o código na **Listagem 25**.

A listagem exibe o conteúdo de dois arquivos: `task.js` para a definição das tarefas do web worker e `main.js` para o script que carrega o worker e recebe os resultados.

Listagem 25. Gerenciando os resultados web com objetos deferidos.

```
01 // Conteúdo do task.js
02 function onmessage(event) {
03   postMessage('completed', {
04     id: event.data.id,
05     result:'resultado computado'
06   });
07 }
08
09 // Conteúdo do main.js
10 var worker = new Worker('task.js');
11 var deferidos = {};
12 var cont = 0;
13
14 worker.addEventListener('completed', function onCompleted(event) {
15   var d = deferidos[event.data.id];
16   d.resolve(event.data.result);
17 });
18
19 function background(task) {
20   var id = cont++;
21   var deferred = jQuery.Deferred();
22   deferidos[id] = deferred; // Armazena deferido para uso futuro
23   console.log('Enviando tarefa para worker: ' + task);
24
25   worker.postMessage({
26     id: id,
27     task: task
28   });
29   return deferred.promise(); // Apenas expõe a promise do código chamado
30 }
31
32 background('Resolve para x').then(function (result) {
33   console.log('A saída é... ' + result);
34 }).fail(function(err) {
35   console.log('Não foi possível completar a tarefa');
36   console.log(err);
37 });
38
39 // Saída do Console:
40 // Enviando tarefa para worker: Resolve para x
41 // A saída é... resultado computado
```

O primeiro script é bem simples (linhas 1 a 7): sempre que ele receber uma mensagem ele responde com um objeto contendo o id da requisição original e um resultado *hard-coded* (fixo no código, no caso a string). Na linha 14 tratamos de adicionar o método no objeto ouvinte que se encarregará de “escutar” sempre que o evento de “completed” for executado. A função de `background` (linha 19) retorna uma promise completada sempre que o worker enviar uma mensagem “completed” para a mesma tarefa. Uma vez que o processamento da mensagem “completed” ocorre fora dessa função que cria a promise, um objeto deferido é usado para expor a função `resolve()` no método de callback `onCompleted()` (linha 14).

Enfim, as promises são o recurso do momento. Elas permitem principalmente que você perca menos tempo se preocupando em escrever código que execute de forma assíncrona sem saber onde que a linha de execução do seu algoritmo está. Alguns desenvolvedores optam por criar tal estrutura por si só e usar o recurso de `debugger` dos browsers para ajudar a rastrear o passo a passo de execução, mas isso definitivamente não é uma boa ideia.

O leitor pode ainda explorar outras APIs que usam as promises por padrão como os Service Workers ou Streams (muitos dos serviços de streaming online de música e vídeo fazem uso das promises para acelerar a comunicação assíncrona com os servidores), bem como outras bibliotecas que as implementam (como a Catilene, polyfill ou Yui, etc.) ou, mais que isso, pode criar a sua própria API de promises e disponibilizá-la para a comunidade web. Na seção **Links** deste artigo você encontra também a URL da API de Promises JavaScript oficial da W3C. Bons estudos!

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 da API Bluebird no GitHub.

<https://github.com/petkaantonov/bluebird/blob/master/API.md>

Página oficial das promises JS.

<https://www.promisejs.org/>

Como criar uma aplicação responsiva com Bootstrap

Crie uma aplicação responsiva de agenda de contatos para a web e o mundo mobile

Desde as primeiras aplicações web começarem a ser criadas, o processo de seleção das tecnologias que seriam usadas num projeto bem como as definições arquiteturais que envolviam tais cenários sempre foram alvo de muita discussão sobre preferências. Levando em consideração um paralelo entre duas das mais usadas tecnologias para web (cliente + servidor), temos:

- No PHP, todo o desenvolvimento se dá através da sua linguagem que mistura código HTML entrelaçado entre códigos PHP. Do ponto de vista de design, isso nunca foi uma boa prática, inclusive adotada por outras plataformas e abandonada futuramente, como o Java, por exemplo.
- No Java, inicialmente usavam-se as JSPs que se assemelhavam às páginas PHP citadas quando códigos Java eram criados em forma de scripts diretamente dentro das páginas web. Atualmente, soluções alternativas e mais elegantes já existem como as tags customizadas (JSTL, JSF, etc.), além de vários frameworks de integração como Spring, GWT e o Apache Tiles.

Essa complicaçāo toda aumenta quando, as mesmas tecnologias que antes eram focadas quase que completamente no servidor (salvos os casos em que precisávamos criar alguma validação de campos ou regras básicas de navegação via JavaScript no próprio browser), agora precisam dividir espaço com os diversos frameworks front-end que surgiram, muitos deles, inclusive, para substituir implementações inteiras que antes só eram possíveis no respectivo servidor. Podemos citar vários dos quais jQuery, Bootstrap, AngularJS ou Prototype se encaixam e hoje demandam cargos em empresas que focam exclusivamente nesse

Fique por dentro

Este artigo se mostra útil para todo desenvolvedor que necessita realizar integrações entre tecnologias de natureza server side, como Java, C# ou Ruby, e os frameworks front-end Bootstrap, AngularJS e jQuery. Você aprenderá, através da criação de uma agenda de contatos web, a integrar todas essas tecnologias tirando o máximo de proveito dos serviços do AngularJS para remover código de servidor no cliente; da responsividade e estrutura HTML do Bootstrap para evitar reinventar o design sempre que começar algo novo; bem como do Apache Tiles para fragmentar as páginas de conteúdo comum proporcionando produtividade e reaproveitamento de código na web.

universo: tecnologias front-end. Tudo isso, atrelado ao já sucesso do Node.js com a possibilidade de ter uma “mão” no lado do servidor que é totalmente JavaScript, fez com que empresas migrasses tudo para front-end, que é mais leve, rápido e performático.

Mas para os que ainda precisam fazer uso de todo o potencial acumulado de tecnologias como JEE, ASP.NET, Ruby, etc. é muito importante entender como fazer a integração das mesmas com os referidos frameworks *client side*. Este artigo trata de expor essa faceta da fase de desenvolvimento de um sistema web. Criaremos uma aplicação de agenda completa, que fará a integração de um módulo no servidor escrito em JEE + Spring (e seus módulos dependentes, como Security, Data e MVC) com os frameworks Bootstrap (para organização da responsividade e design nas páginas), AngularJS (para lidar com os serviços e comunicação com o servidor) e jQuery (para funções utilitárias e ajuda na seleção/manipulação dos campos da página). Também faremos uso do popular framework de templates para Java da Apache, o Tiles, que se integrará às nossas páginas JSP fornecendo simplicidade

e produtividade na exibição dos valores dinâmicos. O banco de dados usado para salvar as informações será o MySQL, mas você pode usar qualquer um (lembrando que não será foco do artigo expor detalhes de banco, por não caber no escopo).

Montagem do ambiente

Para o artigo será necessário ter o conjunto de tecnologias/ ferramentas instaladas no seu ambiente (vide seção **Links**):

- JDK 6 ou superior;
- IDE Eclipse Luna ou superior;
- Servidor Tomcat 7 ou superior (ou qualquer outro servidor de preferência);
- MySQL e MySQL Workbench (ou qualquer outros SGBD de preferência);
- Plugin do JBoss Tools para facilitar na criação dos tipos de projeto.
- Browser (de preferência uma versão recente do Chrome, que tem boas ferramentas de console e depuração *client side*).

Após baixar o zip do Eclipse, extraia todos os arquivos e execute o Eclipse.exe. Selecione um diretório para salvar o projeto e, uma vez dentro da IDE, vá até o menu *Help > Eclipse Marketplace...* e na caixa de buscas *Find* digite “JBoss Tools”. Nesse passo, atente para a versão da sua IDE e procure pelo plugin do JBoss Tools correspondente (pois ele tem uma versão para cada Eclipse diferente, o nome fica entre parênteses logo a frente) e clique em *Install*. Vá navegando nas janelas sempre clicando em *Next* (quando chegar na seleção dos pacotes do plugin, selecione as opções tal como na **Figura 1**), aceite os termos de condição, efetue a instalação e reinicie a IDE para ter tudo configurado.

Para importar o Tomcat no ambiente, extraia os arquivos do mesmo e vá até a aba *Servers* do Eclipse, clique com o botão direito em *New > Server* e depois em *Apache > Tomcat 7.0 Server* (ou o referente à sua versão), informe o diretório onde o extraiu e clique em *Finish*.

Se não tiver o JDK e MySQL (e SGBD) instalados execute os instaladores e siga os passos até o fim. Para o projeto faremos uso do Maven, gerenciador de dependências do Java que já vem por padrão no Eclipse (por isso a importância de usar uma versão recente).

Criação do Projeto

O tipo de projeto que criaremos é um projeto “Maven”. Isso porque é a partir dele que faremos o gerenciamento de dependências sem se preocupar em quais bibliotecas usar no servidor. Portanto, vá até o menu *File > New > Maven Project*, selecione a checkbox “Create a simple Project (skip archetype selection)” e depois clique em *Next*. Na próxima tela configure as propriedades tal como descrito na **Figura 2** (Group e Artifact Ids e mude o *Packaging* para war). Então clique em *Finish*.

A estrutura de pacotes e diretórios que ele gera é bem intuitiva e, basicamente, faremos uso somente de “Java Resources” e da pasta src que, por sua vez, contém os diretórios de arquivos web do projeto.

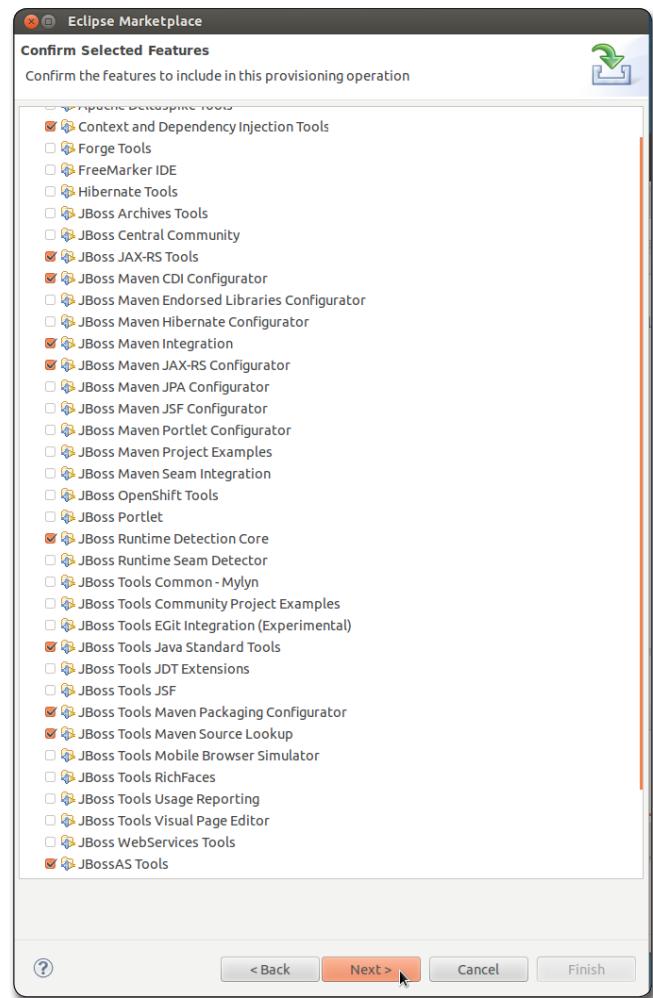


Figura 1. Tela de seleção dos pacotes do JBoss Tools a instalar

Problema de Build Path

Ao usar o Maven nesse processo, o mesmo configura os projetos sempre com a versão 1.5 do JDK que já está meio obsoleta. Portanto, para modificar isso e evitar problemas futuros, vamos alterar a versão para 1.7 ou superior clicando com o botão direito no projeto e indo em *Build Path > Configure Build Path...*. Clique na JRE que aparecer (1.5) e depois em *Edit*. Na combo *Execution Environment* selecione a opção “JavaSE-1.7 (...)” e clique em *Finish*. Em seguida vá até o menu *Project Facets* na mesma janela e mude o valor da versão da propriedade Java de 1.5 para 1.7.

O próximo passo consiste em editar o arquivo pom.xml referente às configurações do Maven com os dados apresentados na **Listagem 1**. Como se trata de um arquivo muito extenso, o dividiremos em algumas partes para melhor entendimento. A mesma listagem traz inicialmente detalhes de como o buil (construção) do nosso projeto deve ser feito pelo Maven, que incluem desde o servidor (Tomcat), porta HTTP (8080), diretórios de instalação, bem como as configurações de log e do conector de banco de dados que usaremos por padrão para o projeto. Salve todo o conteúdo após a tag *<packaging>* e ainda dentro da tag *<project>*.

Em seguida, precisamos adicionar as configurações do Hibernate e do Spring para que suas bibliotecas sejam carregadas em total compatibilidade. Adicione o conteúdo da **Listagem 2** ao fim do arquivo, porém antes da tag <project>.

O leitor pode consultar também o site do mvnrepository disponível na seção **Links** para verificar se já existem versões mais recentes das referidas bibliotecas e atualizar o conteúdo do pom.xml com as mesmas. Como opção, pode-se usar sempre o valor "LATEST" na tag <version> e assegurar, assim, que o Maven sempre carregue a última versão do repositório.

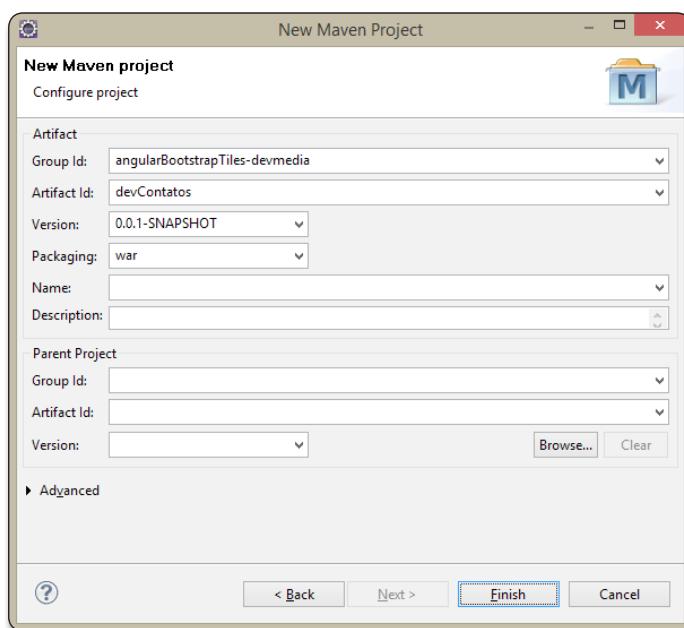


Figura 2. Tela de configuração das propriedades Maven do projeto

Dando continuidade, adicione o código da **Listagem 3** ao final da anterior, certificando de carregar todas as bibliotecas de utilitários, JSON, do Apache Tiles e da API de Servlets, que o nosso projeto fará uso.

Com isso, teremos todas as dependências devidamente baixadas e associadas ao projeto via Build Path. Em seguida, precisamos também criar o datasource do projeto, que nos ajudará a conectar com o banco sem a necessidade de todas aquelas configurações presas no código que temos quando usado só o JDBC. Para o projeto é pressuposto que o leitor tenha um banco chamado "de vContatos" criado no MySQL.

Então, crie a estrutura de diretórios definida na **Figura 3**, dentro do caminho *src > main*.

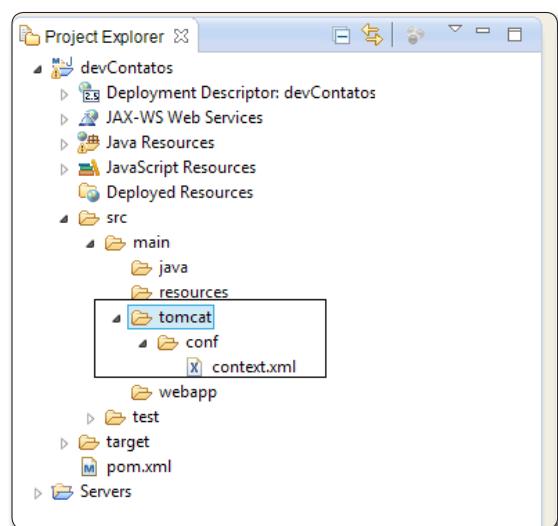


Figura 3. Estrutura de diretórios do projeto para Tomcat

Listagem 1. Conteúdo de build do projeto Maven – pom.xml.

```

<build>
  <finalName>devContatos</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <version>2.1</version>
      <configuration>
        <port>8080</port>
        <path>/${project.build.finalName}</path>
        <additionalConfigFilesDir>${basedir}/src/main/tomcat/conf</additionalConfigFilesDir>
        <systemProperties>
          <log4j.configuration>file:./src/main/tomcat/conf/log4j.xml</log4j.configuration>
        </systemProperties>
      </configuration>
    </plugin>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.16</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.1</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.1</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.14</version>
    </dependency>
  </dependencies>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Como criar uma aplicação responsiva com Bootstrap

Listagem 2. Configuração do pom.xml para dependências do Hibernate e Spring.

```
<dependencies>
    <!-- Hibernate -->
    <dependency>
        <groupId>org.hibernate.javax.persistence</groupId>
        <artifactId>hibernate-jpa-2.0-api</artifactId>
        <version>1.0.1.Final</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>4.2.4.Final</version>
    </dependency>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
        </exclusion>
    </exclusions>
    </dependency>

    <!-- Spring -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-core</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
    </dependency>
    <dependency>
        <version>3.1.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-taglibs</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-tx</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>3.2.3.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-oxm</artifactId>
        <version>3.1.1.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
        <version>1.2.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib</artifactId>
        <version>2.2</version>
    </dependency>
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjrt</artifactId>
        <version>1.7.0</version>
    </dependency>

```

Listagem 3. Configuração do pom.xml para as demais dependências.

```
<!-- JSON -->
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-jaxrs</artifactId>
    <version>1.8.2</version>
</dependency>

<!-- Tiles -->
<dependency>
    <groupId>org.apache.tiles</groupId>
    <artifactId>tiles-api</artifactId>
    <version>2.2.2</version>
</dependency>
<dependency>
    <groupId>org.apache.tiles</groupId>
    <artifactId>tiles-core</artifactId>
    <version>2.2.2</version>
</dependency>
<exclusions>
    <exclusion>
        <groupId>org.slf4j</groupId>
        <artifactId>jcl-over-slf4j</artifactId>
    </exclusion>
</exclusions>
</dependency>
<dependency>
    <groupId>org.apache.tiles</groupId>
    <artifactId>tiles-jsp</artifactId>
    <version>2.2.2</version>
</dependency>
<dependency>
    <groupId>org.apache.tiles</groupId>
    <artifactId>tiles-servlet</artifactId>
    <version>2.2.2</version>
</dependency>
<dependency>
    <groupId>org.apache.tiles</groupId>
    <artifactId>tiles-template</artifactId>
    <version>2.2.2</version>
</dependency>
<!-- Servlet -->
```

Continuação: Listagem 3. Configuração do pom.xml para as demais dependências.

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.2</version>
    <scope>provided</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>

<!-- Utilities -->
<dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.4</version>
</dependency>
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>10.0.1</version>
</dependency>
<dependency>
    <artifactId>commons-logging</artifactId>
    <groupId>commons-logging</groupId>
    <version>1.1.1</version>
</dependency>
</dependencies>
```

Em seguida, abra o arquivo context.xml (que você também deve criar) e o preencha com o código da **Listagem 4**.

Esse datasource será usado diretamente pelo Spring. É através dele que as conexões com o banco de dados iniciam e terminam, mas não vamos entrar em muitos detalhes sobre isso. Se desejar dar outro nome ao banco, edite a propriedade *url* com o respectivo nome no final. Edite também os dados de usuário (*username*) e senha (*password*) para os do seu MySQL. As demais configurações são padrão e você pode encontrá-las na documentação oficial do Tomcat.

Listagem 4. Código de configuração do datasource para o Tomcat.

```
<?xml version='1.0' encoding='utf-8?'>
<Context>
    <WatchedResource>WEB-INF/web.xml</WatchedResource>
    <Resource
        factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
        name="jdbc/tomcatDataSource"
        auth="Container"
        type="javax.sql.DataSource"
        initialSize="1"
        maxActive="20"
        maxIdle="3"
        minIdle="1"
        maxWait="5000"
        username="root"
        password="root"
        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/devContatos"
        validationQuery="SELECT 'OK'"
        testWhileidle="true"
        testOnBorrow="true"
        numTestsPerEvictionRun="5"
        timeBetweenEvictionRunsMillis="30000"
        minEvictableIdleTimeMillis="60000"/>
</Context>
```

Configurações do Spring

As configurações de projetos Spring podem ser encontradas aos montes no GitHub e, principalmente, no site oficial do projeto. Para não perdermos o foco, demonstraremos aqui apenas os passos

para importar as configurações já prontas do projeto final (vide seção **Links** para mais detalhes). Portanto, baixe o arquivo de fontes final no topo da página deste artigo e extraia o projeto do mesmo. Navegue até o diretório *src > webapp > WEB-INF > spring* e copie os quatro arquivos XML presentes para o mesmo diretório no seu projeto (se a estrutura não existir, crie-a). São eles:

- *spring.xml*: guarda as configurações mais básicas e genéricas do Spring, como os pacotes que serão scaneados para varredura de classes de configuração, e as referências aos demais arquivos.
- *spring-mvc.xml*: guarda as configurações do Spring MVC como um todo, tais como gerenciamento de interceptors, filtros, referência ao arquivo de configuração do Tiles, internacionalização, View Resolvers, dentre outras.
- *spring-jpa.xml*: configurações gerais do JPA e conexão com o banco, tais como definição de JNDI do datasource, gerenciador de transações e propriedades de persistência.
- *spring-security.xml*: configurações gerais de segurança do módulo Security, tais como definição de página de erro, controle de navegação, login de usuário e seleção de regras de autenticação.

Após isso, copie o diretório *META-INF > persistence.xml* para a sua pasta *src/main/resources*, bem como o arquivo *web.xml* do diretório *WEB-INF*. Ele conterá as demais informações de configuração do JPA, bem como o dialeto do Hibernate a ser usado.

Certifique-se de não modificar nenhum valor desses arquivos, a não ser que seja um usuário experiente do Spring/JPA, pois isso acarretará em erros graves ao build do projeto. Se estiver usando outro banco que não o MySQL, verifique o arquivo *persistence.xml* e *spring-data.xml* para alterar as informações do mesmo.

Apache Tiles

O Tiles funciona como uma espécie de “montador de pedaços de páginas”. Em outras palavras, ele consegue criar templates baseados em fragmentos de páginas que podem ser incluídas em outras usando *includes* simples, diminuindo a neces-

Como criar uma aplicação responsiva com Bootstrap

sidade de duplicar conteúdo. Também é possível associá-lo a outros elementos Tiles já criados e produzir, assim, uma série de componentes reusáveis. Uma das grandes vantagens é que ele não é intrusivo, ou seja, não influencia nos componentes front-end que estamos usando.

Na aplicação teremos basicamente dois locais onde o código HTML será igual para todas as páginas: o cabeçalho (que conterá o menu de navegação, botão de logout, etc.) e o rodapé (que conterá uma descrição básica de Copyrights e alguns scripts JavaScript que devem ser carregados sempre no fim da página).

Para usá-lo em nossos projetos a primeira configuração já foi importada (arquivo `spring-mvc.xml`), que se refere a um bean do Spring (o Tiles tem integração facilitada com o Spring) com informações básicas do que usar do mesmo. Na tag `<list>` dessa configuração informamos que o arquivo `tiles.xml` presente na pasta WEB-INF se encarregará de subir tudo do framework que precisamos. Vamos criar então este arquivo no mesmo diretório e incluir o conteúdo da **Listagem 5** nele.

Listagem 5. Código de configuração do Apache Tiles.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
"http://tiles.apache.org/dtds/tiles-config_2_1.dtd">
<tiles-definitions>
<!-- Principal -->
<definition name="principal.page" template="/public/template/principal.jsp">
<put-attribute name="cabecalho" value="/public/template/cabecalho.jsp"/>
<put-attribute name="rodape" value="/public/template/rodape.jsp"/>
</definition>

<!-- Páginas -->
<definition name="welcomePage" extends="principal.page">
<put-attribute name="corpo" value="/protected/welcomePage.jsp"/>
</definition>
<definition name="contactsList" extends="principal.page">
<put-attribute name="corpo" value="/protected/contatos/contatos.jsp"/>
</definition>
<definition name="login" extends="principal.page">
<put-attribute name="cabecalho" value="" />
<put-attribute name="rodape" value="" />
<put-attribute name="corpo" value="/public/login.jsp"/>
</definition>
</tiles-definitions>
```

A primeira consideração importante é lembrar que os templates do Tiles têm sempre uma página *master*, que funciona como base do template. Aqui demos o nome de `principal.page`. As tags `<definition>` definem quais serão os atributos globais do template e para quais arquivos físicos eles apontam. Esses atributos, por sua vez, definem as chamadas “sessões”, que são setores da página.

Cada definition recebe um atributo `name` que deve ser lembrado quando precisarmos chamar tais propriedades diretamente dos controladores do Spring. Veja como nas chamadas seguintes não precisamos mais informar tais atributos, isso porque estamos importando as mesmas configurações via herança através do atributo `extends`. Na última delas informamos os atributos com o

valor vazio, dessa forma também é possível “sobrescrever” tais definições como numa herança JavaScript comum.

Perceba que ainda não temos criados os três arquivos de template: `principal.jsp` (**Listagem 7**), `cabecalho.jsp` e `rodape.jsp`. Antes de criá-los, verifique a estrutura de diretórios da **Listagem 6** e faça o mesmo no seu projeto.

Listagem 6. Estrutura de diretórios das páginas de template.

```
src
|--- main
    |--- webapp
        |--- protected
            |--- contatos
        |--- public
            |--- template
```

Listagem 7. Código do template da página `principal.jsp`.

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<!doctype html>
<html lang="pt-BR" id="ng-app" ng-app="">
<head>
    <title><spring:message code="project.title" /></title>
    <link href=<c:url value='/resources/css/bootstrap.min.css' />
        rel="stylesheet"/>
    <link href=<c:url value='/resources/css/bootstrap-responsive.min.css' />
        rel="stylesheet"/>
    <link href=<c:url value='/resources/css/project_style.css' />
        rel="stylesheet"/>
    <script src=<c:url value='/resources/js/jquery-1.9.1.min.js' />></script>
    <script src=<c:url value='/resources/js/angular.min.js' />></script>
</head>
<body>
    <div class="container">
        <tiles:insertAttribute name="cabecalho"/>
        <tiles:insertAttribute name="corpo"/>
    </div>
    <!--[if IE]>
        <script src=<c:url value='/resources/js/bootstrap.min.ie.js' />></script>
    <![endif]-->
    <!--[if IE]><!-->
        <script src=<c:url value='/resources/js/bootstrap.min.js' />></script>
    <!--<![endif]-->
        <tiles:insertAttribute name="rodape" />
    </body>
</html>
```

Nessa página podemos ver a importação das tags do JSTL, uma vez que o Tiles tem dependência direta nelas. Veja também que já importamos os arquivos de JavaScript e CSS referentes ao Bootstrap, jQuery e AngularJS, mas não se preocupe, faremos a importação física dos mesmos mais adiante.

O foco da listagem, entretanto, é mostrar como importar as respectivas páginas de template dentro da página principal. Perceba o uso da tag `<tiles:insertAttribute>` dentro da div container. Para importar, basta usar o mesmo nome que configurou no arquivo XML e pronto; o mesmo vale para o rodapé. Essa página representa apenas o esqueleto da aplicação, portanto nenhum componente do tipo texto ou botão será criado nela.

Antes de criarmos as próximas páginas do template precisamos configurar os objetos de modelo, uma vez que elas precisam destes para funcionar. Portanto, crie primeiro a estrutura de pacotes representada na **Figura 4** no seu projeto. Eles servirão para organizar melhor as responsabilidades de cada classe. Em seguida, crie um novo enum de nome *Role* (**Listagem 8**) que conterá as constantes dos tipos de usuário da aplicação e as classes *Usuario* (**Listagem 9**) e *Contato* (**Listagem 10**) que conterão os dados mapeados de cada entidade de usuário e contato, respectivamente. Detalhes sobre a implementação/mapeamentos do Hibernate serão omitidos.

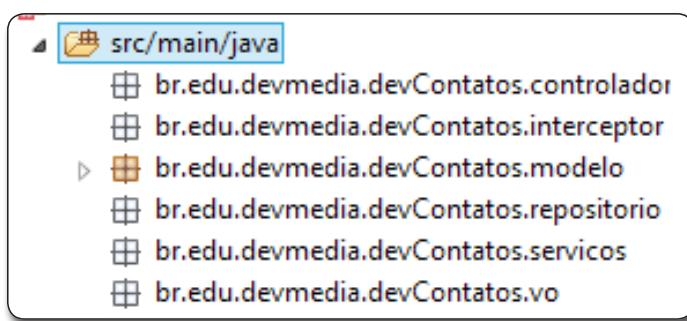


Figura 4. Lista de pacotes usados no projeto

Listagem 8. Código do enum de regras de usuário.

```
package br.edu.devmedia.devContatos.modelo;

public enum Role {
    ROLE_ADMIN, ROLE_USER
}
```

Listagem 9. Código da classe que representa um usuário.

```
package br.edu.devmedia.devContatos.modelo;

// Imports omitidos

@Entity
@Table(name = "tb_usuario")
public class Usuario {

    @Id
    @GeneratedValue
    private int id;

    private String email;
    private String nome;
    private String habilitado;
    private String senha;

    @Enumerated(EnumType.STRING)
    @Column(name = "user_role")
    private Role role;

    // Get's e Set's
}
```

Listagem 10. Código da classe que representa um contato.

```
package br.edu.devmedia.devContatos.modelo;

// Imports omitidos

@Entity
public class Contato {
    @Id
    @GeneratedValue
    private int id;
    private String nome;
    private String numeroFone;
    private String email;

    public Contato(){}

    public Contato(String nome, String numeroFone, String email, int id) {
        super();
        this.nome = nome;
        this.numeroFone = numeroFone;
        this.email = email;
    }

    // Get's e Set's

    @Override
    public boolean equals(Object object) {
        if (object instanceof Contato){
            Contato Contato = (Contato) object;
            return Contato.id == id;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return id;
    }
}
```

Em seguida, copie também as classes dos pacotes “*.servicos”, “*.repositorio” e “*.vo” para o seu projeto. Elas conterão os códigos de persistência da base que precisaremos para comunicar os controladores com o banco.

Controladores

Precisamos criar agora os controladores que receberão do front-end todas as requisições em conjunto com os métodos de action do Tiles. Vamos começar pelo mais complexo: o controlador de contatos. Crie, portanto, uma nova classe no pacote “*.controlador” e inclua o código da **Listagem 11** nela. Os métodos utilitários do final da mesma (omitidos para simplificação) podem ser encontrados no arquivo original do projeto.

Os métodos que constam da anotação `@RequestMapping` são os métodos que recebem as requisições diretamente do front-end. São seis métodos para listar os contatos, criar/atualizar/remover um contato, buscar por um em específico, etc. Os mesmos lidam com a exibição de mensagens de validação ao final identificando qual

Como criar uma aplicação responsiva com Bootstrap

Listagem 11. Código da classe controladora de contatos.

```
// Imports omitidos

@Controller
@RequestMapping(value = "/protected/contatos")
public class ControladorContatos {
    private static final String DEFAULT_PAGE_DISPLAYED_TO_USER = "0";

    @Autowired
    private ContatosService contatosService;

    @Autowired
    private MessageSource messageSource;

    @Value("5")
    private int resultadosMax;

    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView welcome() { return new ModelAndView("listaContatos"); }

    @RequestMapping(method = RequestMethod.GET, produces = "application/json")
    public ResponseEntity<?> listAll(@RequestParam int page, Locale locale) {
        return createListAllResponse(page, locale);
    }

    @RequestMapping(method = RequestMethod.POST, produces = "application/json")
    public ResponseEntity<?> create(@ModelAttribute("contact") Contato contato,
        @RequestParam(required = false) String searchFor, @RequestParam(
            required = false, defaultValue = DEFAULT_PAGE_DISPLAYED_TO_USER)
        int page, Locale locale) {
        contatosService.save(contato);
        if (isSearchActivated(searchFor))
            return search(searchFor, page, locale, "message.create.success");
        return createListAllResponse(page, locale, "message.create.success");
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.PUT,
        produces = "application/json")
    public ResponseEntity<?> update(@PathVariable("id") int contactId,
        @RequestBody Contato contato, @RequestParam(required = false) String searchFor,
        @RequestParam(required = false, defaultValue = DEFAULT_PAGE_DISPLAYED_TO_USER)
        int page, Locale locale) {
        if (contactId != contato.getId())
            return new ResponseEntity<String>("Bad Request", HttpStatus.BAD_REQUEST);
        contatosService.save(contato);
        if (isSearchActivated(searchFor))
            return search(searchFor, page, locale, "message.update.success");
        return createListAllResponse(page, locale, "message.update.success");
    }

    @RequestMapping(value = "/{contactId}", method = RequestMethod.DELETE,
        produces = "application/json")
    public ResponseEntity<?> delete(@PathVariable("contactId") int contactId,
        @RequestParam(required = false) String searchFor, @RequestParam(required = false,
            defaultValue = DEFAULT_PAGE_DISPLAYED_TO_USER) int page, Locale locale) {
        try {
            contatosService.delete(contactId);
        } catch (AccessDeniedException e) {
            return new ResponseEntity<Object>(HttpStatus.FORBIDDEN);
        }
        if (isSearchActivated(searchFor))
            return search(searchFor, page, locale, "message.delete.success");
        return createListAllResponse(page, locale, "message.delete.success");
    }

    @RequestMapping(value = "/{name}", method = RequestMethod.GET,
        produces = "application/json")
    public ResponseEntity<?> search(@PathVariable("name") String name,
        @RequestParam(required = false, defaultValue = DEFAULT_PAGE_DISPLAYED_TO_USER)
        int page, Locale locale) { return search(name, page, locale, null); }

    private ResponseEntity<?> search(String name, int page, Locale locale,
        String actionMessageKey) {
        ListaContatosVO listaContatosVO = contatosService.findByNameLike(
            page, resultadosMax, name);
        if (!StringUtil.isEmpty(actionMessageKey))
            addActionMessageToVO(listaContatosVO, locale, actionMessageKey, null);
        Object[] args = {name};
        addSearchMessageToVO(listaContatosVO, locale, "message.search.for.active", args);
        return new ResponseEntity<ListaContatosVO>(listaContatosVO, HttpStatus.OK);
    }

    // Demais métodos no arquivo original
}
```

o *Locale* do usuário via objeto de requisição recebido do browser e buscando o arquivo de propriedades respectivo.

É importante ressaltar que o código inteiro lida com a conversão de valores sempre em JSON. Por estarmos lidando com uma implementação diretamente vinculada ao JavaScript, este formato é sempre a melhor escolha por ser mais rápido e leve em comparação ao XML.

As três demais classes de controladores para gerenciamento das páginas de login, home e index se encontram no pacote “*.controlador” do projeto original. Copie-as para o seu projeto. Para finalizar as configurações, copie também a classe LoginInterceptor do pacote “*.interceptor”, que se encarregará de adicionar o usuário à sessão uma vez logado com sucesso.

Associando o AngularJS ao SpringMVC

Uma das principais características do AngularJS é fornecer um *binding* de objetos simplificado, ou seja, conseguimos refletir nas URLs da aplicação exatamente o que está acontecendo nos métodos de serviço do framework.

Para entender melhor, vamos criar a página de login.jsp no diretório *webapp > public* e adicionar o código da **Listagem 12** na mesma.

Na página podemos ver o uso de vários atributos *code* nas tags. Esse atributo acessa o arquivo de propriedades (*i18n_pt.properties*, copie-o para o seu projeto também) com todas as mensagens da aplicação para exibir seus conteúdos. Veja também o uso do atributo do AngularJS *ng-controller* na div após *row-fluid*. Ele é responsável por mapear o mesmo nome do controlador que criamos no Spring, por isso certifique-se de informá-lo com inicial minúscula.

É importante diferenciar entre um controller do Spring e do AngularJS: os do Spring são métodos Java que executam ações do lado servidor, como uma requisição a um Web Services; já os do AngularJS são simples métodos JavaScript que executam assincronamente. Veja que no fim da página importamos o arquivo *login.js* que está no diretório “resources...”, ele se encarrega de conter a função de login que também deverá ter o mesmo nome. Certifique-se de copiar todos os arquivos de CSS, JS e imagens no diretório *main > webapp > resources* para o seu projeto.

Listagem 12. Código da página de login.jsp.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<div class="row-fluid">
    <div class="jumbotron">
        <h1><spring:message code='project.name' /></h1>
    </div>
</div>
<div class="row-fluid">
    <div class="span4 offset4 well" ng-controller="controladorLogin">
        <legend><spring:message code="login.header" /></legend>
        <div class="alert alert-error" ng-class="{': displayLoginError == true,
            'none': displayLoginError == false}">
            <spring:message code="login.error" />
        </div>
        <form method="post" action="j_spring_security_check">
            <div>
                <input name="j_username" id="j_username" type="text" class="span12"
                    placeholder=<spring:message code='sample.email' />><br/>
                <input name="j_password" id="j_password" type="password"
                    class="span12" placeholder="Senha"><br/>
                <button type="submit" name="submit" class="btn btn-inverse
                    btn-block"><spring:message code="login.signIn" /></button>
            </div>
        </form>
    </div>
</div>
<script src=<c:url value='/resources/js/pages/login.js' />></script>
```

Os dois campos de input, por sua vez, referentes ao login e senha devem ter os nomes e ids iguais em detrimento da autenticação segura que o Spring implementa. Perceba que também estamos usando o atributo *ng-class* com frequência, em vista da definição de classes CSS do próprio AngularJS às páginas.

Ainda precisamos de mais três páginas, a saber:

1. *welcomePage.jsp*: terá apenas uma mensagem de boas-vindas acessando as mensagens do arquivo de propriedades;
2. *contatos.jsp*: guarda toda a lógica do CRUD de contatos;
3. *dialogosContatos.jsp*: cria um formulário de contato, onde o usuário pode enviar mensagens e visualizá-las depois.

Portanto, copie os três arquivos do projeto original e vejamos alguns detalhes sobre o código HTML5 e AngularJS dos mesmos. Na página *contato.jsp*, temos como principal trecho de código o representado na **Listagem 13**.

Suprimimos o restante do código HTML para simplificar o entendimento. No início da página declaramos mais uma vez o nome do controller: *controladorContatos*, que também terá sua função JavaScript de execução. Em seguida criamos a tabela que exibirá a lista de contatos da base, em especial pelo atributo *ng-repeat* do AngularJS que nos possibilita iterar por sobre a lista inteira utilizando o modelo de mapeamento que definimos antes. Cada atributo entre as chaves duplas `{}` estará associado diretamente ao objeto no lado do AngularJS. O valor *page.source*, por sua vez, define a fonte de dados que alimenta essa lista dire-

tamente na sessão do usuário. Depois é só sair preenchendo cada um dos valores em cada uma das colunas da tabela e, dentro de cada uma, definir as funções de serviço que efetuarão as demais ações do CRUD.

Listagem 13. Código principal da página de contato.jsp.

```
<div ng-controller="controladorContatos">
    ... <!-- mais HTML -->
    <div id="gridContainer" ng-class="{': state == 'list', 'none': state != 'list'}">
        <table class="table table-bordered table-striped">
            <thead>
                <tr>
                    <th scope="col"><spring:message code="contacts.name" /></th>
                    <th scope="col"><spring:message code="contacts.email" /></th>
                    <th scope="col"><spring:message code="contacts.phone" /></th>
                    <th scope="col"></th>
                </tr>
            </thead>
            <tbody>
                <tr ng-repeat="contato in page.source">
                    <td class="tdContactsCentered">{{contato.nome}}</td>
                    <td class="tdContactsCentered">{{contato.email}}</td>
                    <td class="tdContactsCentered">{{contato.numeroFone}}</td>
                    <td class="width15">
                        <div class="text-center">
                            <input type="hidden" value="{{contatos.id}}"/>
                            <a href="#" ng-click="selectedContact(contatos);"
                                role="button"
                                title=<spring:message code="update" />&nbsp;
                                <spring:message code="contact" />
                                class="btn btn-inverse" data-toggle="modal">
                                <i class="icon-pencil"></i>
                            </a>
                            <a href="#" ng-click="selectedContact(contatos);"
                                role="button"
                                title=<spring:message code="delete" />&nbsp;
                                <spring:message code="contact" />
                                class="btn btn-inverse" data-toggle="modal">
                                <i class="icon-minus"></i>
                            </a>
                        </div>
                    </td>
                </tr>
            </tbody>
        ... <!-- mais HTML -->
    </div>
```

Agora precisamos criar os dois últimos templates do Tiles para lidar com o cabeçalho e rodapé das páginas. Os códigos para ambos estão disponíveis nas **Listagens 14** e **15**, respectivamente.

Na página de cabeçalho criamos apenas o código HTML com a estrutura do Bootstrap para implementar a barra de navegação, com os menus de contexto e o menu de logout no fim. Perceba que usamos JSTL, tag *c:out*, para fornecer o mecanismo padrão de navegação para a URL home do projeto, representada pelo `/`.

Como criar uma aplicação responsiva com Bootstrap

Listagem 14. Código da página de cabecalho.jsp.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>

<div class="masthead">
  <h3 class="muted">
    <spring:message code='header.message'/>
  </h3>

  <div class="navbar">
    <div class="navbar-inner">
      <div class="container">
        <ul class="nav ng-controller="LocationController">
          <li ng-class="{active: activeURL == 'home', ': activeURL != 'home'}">
            <a href=<c:url value="/" /><br/>
              title=<spring:message code="header.home"/>
            </a>
          </li>
          <li ng-class="{gray: activeURL == 'contacts', ': activeURL != 'contacts'}">
            <a href=<c:url value="/protected/contatos/" /><br/>
              <p><spring:message code="header.contacts"/></p></a>
            </li>
          <ul class="nav pull-right">
            <li><a href=<c:url value="/logout" /><br/>title=<spring:message code="header.logout"/><br/><p class="displayInLine">
              <spring:message code="header.logout"/></p></a></li>
          </ul>
        </ul>
      </div>
    </div>
  </div>
</div>
```

Listagem 15. Código da página de rodape.jsp.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<script src=<c:url value="/resources/js/pages/header.js" />></script>
<p align="center" style="margin-top: 25px">
  Copyright @2015 - Todos os direitos reservados
</p>
```

Executando o projeto

Para efetuar o build do projeto temos duas opções: via interface de linha de comando e via IDE. Como a primeira opção é mais trabalhosa em vista de necessitar a instalação do Maven a nível de Sistema Operacional, além de obrigatoricamente ter de conhecer os comandos da ferramenta; vamos optar por fazer isso usando o Eclipse. Portanto, vá até o botão de seta ao lado do *Run* no topo da barra de ferramentas e clique em *Run Configurations....*. Na barra lateral esquerda clique com o botão direito em *Maven Build* e depois em *New*. No campo *Name* preencha com "devContatos", clique em *Browse Workspace...* e selecione o projeto terminando com *OK*. No campo *Goals* coloque o valor *clean install tomcat7:run* para que o Maven possa limpar e construir o projeto e então clique em *Run*.

Aguarde até que ele baixe todas as dependências, empacote o .war e inicie a instância do Tomcat. Quando finalizar acesse a

aplicação via URL: <http://localhost:8080/devContatos/>. Você verá a tela de login conforme demonstrada na **Figura 5**.

A tela já traz a exibição da mensagem de validação caso os dados não sejam preenchidos. Sempre que você tentar logar no estado atual essa mensagem aparecerá em detrimento de não termos salvo nenhum usuário na base ainda. Portanto, efetue um INSERT básico no banco, preenchendo todos os campos (exceto o de id, que é auto incremento) e tente logar novamente. O resultado do login com sucesso está expresso na **Figura 6**.

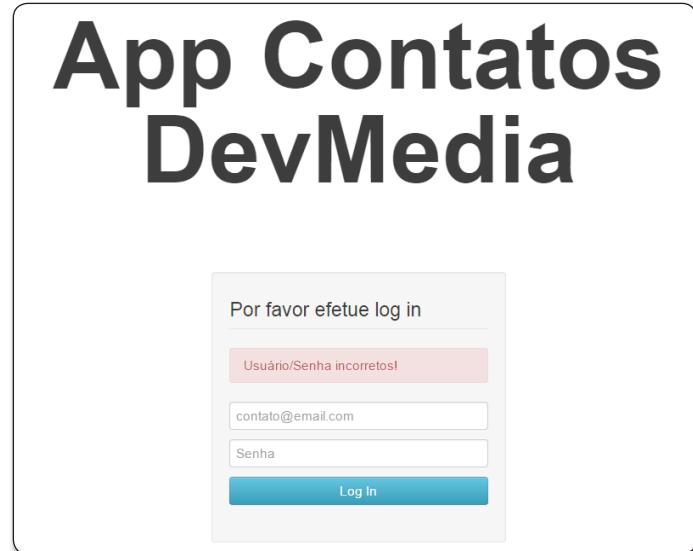


Figura 5. Página de login do aplicativo de contatos



Figura 6. Página inicial do aplicativo de contatos

Quando clicar no menu *Contatos* uma mensagem informando que nenhum contato foi cadastrado ainda aparecerá. Portanto, clique no botão *Criar Contato*, preencha os campos e clique em *Criar*. Faça testes criando vários contatos até que o número máximo por páginas (cinco, no nosso caso) seja ultrapassado e o componente do AngularJS de paginação em conjunto com o Bootstrap gere uma navegação paginada de dados igual à da **Figura 7**, onde os dados serão visualizados à medida que novas páginas são criadas. Ao mesmo tempo, certifique-se de verificar sempre na base de dados se as informações foram de fato salvadas.

Na última coluna da tabela temos dois botões para edição e exclusão dos registros. Faça alguns testes com eles também, e veja como eles se comportam alterando e removendo contatos tanto na tela front-end quanto na base de dados. Por fim, o usuário pode ainda fazer uma filtragem da lista de contatos através do menu de lupa acima da tabela. Nele, você informa o nome e o AngularJS passa a query de filtro conforme o mesmo, trazendo como listagem apenas as informações que atendem à condição (**Figura 8**).

Além disso, a ação de logout também está disponível no topo direito da barra de navegação. O que vimos até então é apenas uma pequena demonstração do quão poderosas podem ser essas tecnologias se usadas em conjunto.

Bem-vindo ao App Contatos

Home Contatos Logout

Contatos (1)

Total de dados encontrados na base:

Nome	Email	Fone	Ações
Dev	dev@dev.com	213423423	
Dev2	dev2@dev.com	23423423	
Dev3'	dev3@dev.com	23423423	
Jorge	jorge@root.com	8540122980	
Test 1	jorgerodrissantos@hotmail.com	+558587900136	

Próximo < 1 de 2 > Último

+ Criar Contato

Copyright ©2015 - Todos os direitos reservados

Figura 7. Página inicial do aplicativo de contatos

Buscar

Buscar por

Buscar Cancelar

Figura 8. Modal com campo de filtro para pesquisa por nome

O poder de processamento do lado do servidor pode muito bem ser alinhado com os frameworks front-end, principalmente se eles proveem funções de serviços, componentização e customização.

O AngularJS fornece toda a abstração de que precisamos para encapsular modelos de dados inteiros que estão no servidor sem injetar código de servidor nas páginas cliente. Isso é uma vantagem enorme, pois conseguimos reduzir toda a configuração de métodos de action e/ou web services a funções JavaScript e atributos de tags HTML. Já o Bootstrap nos ajuda com toda uma configuração responsiva e de design trabalhada para facilitar esse tipo de implementação nos aplicativos. Consequentemente, a prisão ao CSS e emaranhamento de tags HTML é uma recíproca, por isso o desenvolvedor deve conhecê-lo mais profundamente para dominar suas técnicas e compreender suas regras chave. E para completar a união, o Tiles é a pedida perfeita quando se trata de comunicar páginas JSP com esses frameworks, além de fornecer mecanismos para reaproveitamento de código da camada de visão como um todo.

Autor



Jorge Rodrigues

É especialista em SEO e marketing na web, e freelancer no universo mobile e front-end, tendo domínio sobre tecnologias como CoffeeScript, JavaScript, HTML5 e Meteor.js. Já trabalhou em diversas empresas de TI tendo acumulado mais de dois anos de experiência.



Links:

Página de download do Eclipse.

<https://eclipse.org/downloads/>

Página de repositório de dependências do Maven.

<http://mvnrepository.com/>

Site oficial dos projetos Spring.

<http://spring.io/>

Página de repositório de dependências do Maven.

<http://mvnrepository.com/>

Como otimizar a performance no front-end

Aprenda a usar as principais técnicas e ferramentas para deixar suas aplicações mais rápidas

O desenvolvimento de aplicações web que, até pouco tempo, focava em medições de performance apenas na arquitetura servidor e excluía toda a parte gráfica (principalmente por se tratar de GUIs desktop e estas serem deveras performáticas), se vê hoje forçado a entender como funciona a web, seus diversos protocolos, navegadores e, mais importante, as linguagens de programação que fazem tudo funcionar. Por muito tempo os testes de stress, famosos por definir até quanto a capacidade de um sistema aguenta uma quantidade expressiva de usuários gerando requisições e peso para o mesmo, eram uma das únicas formas de medir performance em aplicações web. O seu maior problema é que a medição focava tão-somente na carga de requisições HTTP geradas e no overhead de memória no servidor. Consequentemente, isso mascarou por muito tempo a ideia de que precisamos também focar na otimização da forma como desenvolvemos nosso front-end, pois é a partir dele que saem as requisições e é ele o responsável por lidar diretamente com o cliente navegador.

Desenvolver aplicações rápidas, elegantes, que funcionem bem e sejam fáceis de criar é uma premissa, e tais exigências atacam desde um cliente mais exigente (todos) à necessidade de um SEO satisfatório. Ninguém gosta de um site lento e, por mais que você saiba que o seu sistema tem muitos usuários em horários de pico, ou o servidor que a empresa disponibilizou não é tão robusto, ou ainda que os programadores não sabem usar boas práticas para criar código leve; ninguém ligará para isso. No mundo da TI, **performance**, medida pelo usuário, é tecnicamente definida como o medidor de resposta da sua aplicação e, a partir dela, podemos saber se é aceitável ou não.

Talvez o maior problema dessa questão esteja relacionado à inclusão de componentes e bibliotecas nas páginas web. Por exemplo, o site antigo da sua empresa era bem simples, tudo era estático, poucas imagens, nenhum vídeo, feio; porém performático. E aí o seu chefe vem até você e solicita vários novos recursos, como menus dinâmicos no topo da página, controle de

Fique por dentro

Este artigo se mostra útil para os desenvolvedores que desejam aumentar a performance de suas aplicações e não sabem muito bem quais as melhores estratégias e ferramentas para fazer isso. Neste artigo, você aprenderá a otimizar desde o cache básico dos browsers para que suas páginas não precisem recarregar tudo a cada novo request, até imagens, técnicas de compressão de arquivos JavaScript, HTML e CSS, dentre outros. Ao final, o leitor estará apto, inclusive, a medir a performance de cada estratégia usada, bem como definir qual delas se encaixa melhor em cada situação do ciclo de desenvolvimento front-end.

navegação com *breadcrumbs*, um slideshow no início com várias fotos em “alta resolução”, muitos ícones espalhados pela página, etc. Para cada necessidade, você vai precisar de uma (ou várias) bibliotecas JavaScript diferentes, mas você não entende a diferença entre a versão minificada e normal, sai criando cada ícone como um arquivo de imagem separado (e gera novas requisições para cada imagem importada = mais overhead), carrega as imagens do slideshow com tamanho enorme e sai ajustando largura e altura diretamente nos atributos da tag HTML, etc. Resultado disso? Uma sobrecarga exponencial para a sua aplicação que, agora tem vários recursos interessantes, mas falha em tempo de carregamento, em SEO, e, principalmente, em experiência com o usuário.

E aí você se pergunta: mas não podemos ter sites com vários recursos sob o risco de perder em performance? Negativo. A grande sacada é entender como tudo funciona e, sobretudo, como tudo funciona de forma integrada. Os frameworks se comunicam muitas vezes, existem arquivos que não precisam estar ali, código minificado é mais rápido, dentre várias outras coisas que podem ser considerados em uma otimização desse tipo.

O assunto é levado tão a sério pela comunidade web que a editora O'Reilly, responsável pela publicação de inúmeras bibliografias sobre o assunto, organiza anualmente o evento Velocity, que se dedica a reunir profissionais da área de otimização e escalabilidade web para debater os últimos lançamentos e técnicas do mercado. Em sua sétima edição, 2015, que acontece em quatro cidades do

globo, o evento traz trends variadas sobre o monitoramento de performance, cloud computing, otimização de UX, design responsivo, etc. focadas em redes, internet, mobile, gerência e métricas.

Veremos neste artigo uma série de conceitos e exemplos práticos de como aplicar as principais técnicas de otimização de aplicações web do mercado.

Cache no Client-Side

O uso de caches é uma parte muito importante da web moderna, mas é também uma questão cercada por confusão e desentendimento. *Caching* é um termo deveras genérico, mas geralmente se refere ao armazenamento de recursos web (documentos HTML, imagens, etc.) temporariamente em um local para aumentar a performance. Eles podem ser implementados pela maioria dos navegadores web, em proxies web dinâmicos, e até mesmo em gateways de intranets. Para entender melhor, vejamos alguns dos principais tipos de caches que existem.

Caching em Browsers

O cache de um browser se resume basicamente a salvar arquivos recentes de forma temporária deixando o acesso mais rápido, já que uma requisição ao servidor é bem mais custosa que buscar no disco da própria máquina. Cada servidor tem uma política própria de controle de cache e dita, muitas vezes, o que o browser pode ou não cachear. E os browsers obedecem esses termos muito bem.

O grande problema dessa estratégia é que o espaço reservado pelos navegadores para a mesma é consideravelmente pequeno e, mesmo com os contínuos avanços em armazenamento pessoal de máquina, os fabricantes de browsers parecem sempre modestos em relação a isso. Somando-se que as páginas web continuam crescendo em recursos e se tornando cada vez mais pesadas, é questão de tempo até que essa realidade mude. Veja na **Tabela 1** uma relação da quantidade máxima de cache disponibilizada pelos principais navegadores do mercado.

Navegador	Tamanho máximo de cache
Firefox 17+	1024MB
IE 6, 7 e 8	1/32 do espaço do disco, limitado a 50MB
IE 9	1/256 do espaço do disco, limitado a 250MB
Safari	Sem limites
Opera 10+	400MB
Chrome	300MB

Tabela 1. Lista de cache máximo por navegador

Ao mesmo tempo, quando o cache enche, o algoritmo que decide o que será removido é deveras bruto, isto é, é falho para decidir o que realmente é importante na página. Por exemplo, o carregamento de recursos JavaScript tipicamente bloqueia o carregamento do resto da página (razão pela qual devemos importá-los sempre no fim da mesma), logo tais arquivos deveriam ter prioridade no cache em vez de imagens. Mas isso é algo que as futuras versões dos navegadores já estão trabalhando para corrigir.

Além disso tudo, ainda temos o problema do usuário final. A maioria dos navegadores fornecem recursos para que seus usuários possam limpar o cache facilmente e, apesar de acharem que estão limpando espaço em disco para deixar seu navegador mais rápido, o que acontece de fato é que o browser terá de recarregar tudo de novo para as páginas mais usadas.

Caches intermediários

Os caches intermediários, ou proxy (chamados assim porque preenchem muito bem os requisitos de ambos), são comumente usados por ISPs (*Internet Service Provider*, fornecedores de Internet) e grandes organizações. Quando usados pelos ISPs eles tipicamente assumem a forma de proxies de cache transparente que silenciosamente interceptam qualquer tráfego HTTP. Quando o cliente faz uma requisição, o proxy a intercepta e checa seu cache local para fazer uma cópia do recurso. Se nada for encontrado, ele faz a requisição no lugar do cliente e então armazena uma cópia dele mesmo no processo. Assim, quando outro cliente pertencente ao mesmo ISP faz a mesma requisição, uma cópia cacheada é enviada rapidamente, sem ter de refazer o processo todo novamente.

Apesar desse tipo de cache fornecer benefícios significativos de performance (uma vez que a conexão entre o cliente e o proxy geralmente é mais rápida), perdemos em latência para recursos que não estão no cache intermediário, tendo de fazer checagens que não resultarão em nada.

Controlando o cache

Voltemos para o cache de browser. Como um navegador sabe se um recurso cacheado localmente ainda é válido quando o acessar novamente? O modelo padrão de verificação é dado através do envio de requisições condicionais GET. Se o servidor determina que o recurso tenha sido modificado desde a data dada no cabeçalho da requisição (que será setada pelo browser com a data de última modificação originária da resposta recebida antes), o recurso é retornado normalmente. Caso contrário, o status 304 *Not Modified* é retornado.

Vejamos um exemplo para entender melhor. Vamos efetuar um request de uma imagem no navegador e depois reenviar o mesmo request, analisando o que acontece via *Ferramentas de Desenvolvedor* do mesmo. Para isso, usaremos o browser Google Chrome pela simplicidade de uso da sua ferramenta de depuração. Portanto, abra uma nova aba do mesmo e digite F12, vá até a aba *Network* e acesse a URL a seguir via barra de endereços (se você acessa o site da DevMedia com frequência essa imagem já deve estar cacheada, logo clique com o direito na mesma aba e selecione a opção “*Clear browser cache*”):

<http://www.devmedia.com.br/Imagens/2013/logo.png>

Ao fazer isso, com o cache limpo, você verá o resultado expresso na **Figura 1**. Veja que na coluna *Status* temos o valor 200, informando que o arquivo foi recebido via método GET, bem como outras informações de tempo de resposta, tamanho do arquivo, etc.

Como otimizar a performance no front-end

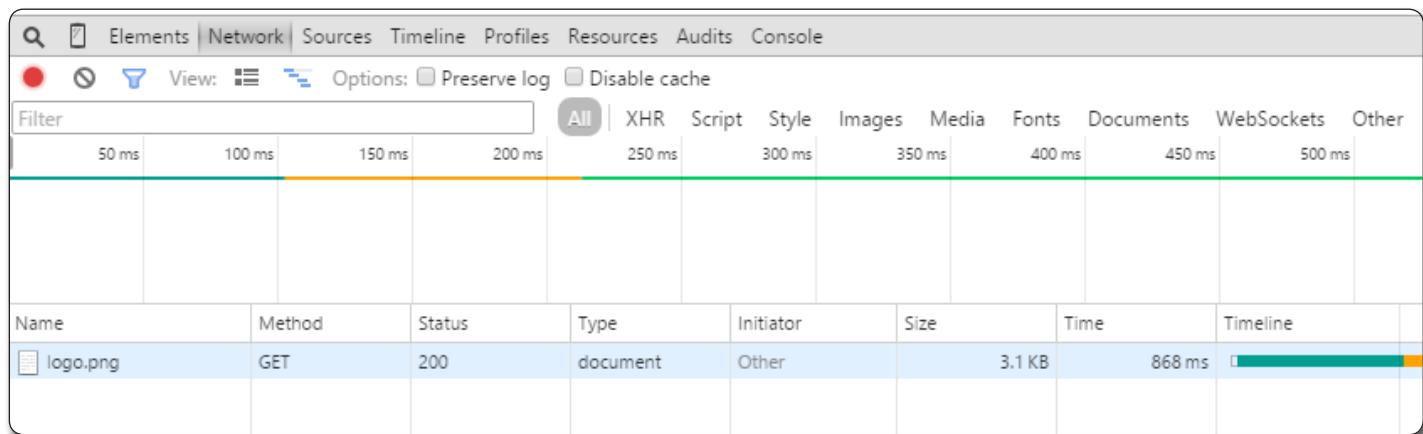


Figura 1. Resultado de request por imagem

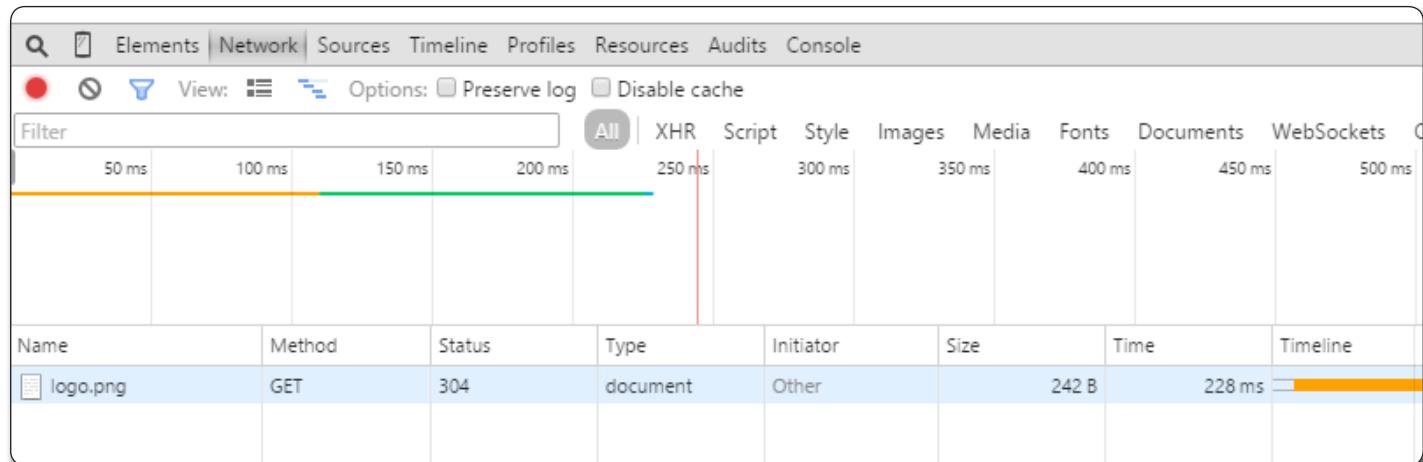


Figura 2. Resultado de request por imagem cacheada

Para visualizar o conteúdo do cabeçalho da requisição e resposta, basta clicar sobre o elemento da imagem com o botão direito e selecionar as opções “*Copy request headers*” e “*Copy response headers*”, respectivamente. Os resultados dessas ações podem ser visualizados na **Listagem 1**.

Na linha 3 temos a propriedade *connection* que define o estado de conexão aberta com o servidor, na linha 4 o componente de controle de cache informando o *max-age* igual a zero, já que não temos o dado salvo ainda. Já no header de resposta (a partir da linha 11) vemos as informações de *max-age* configuradas com a data do recurso, representada pela propriedade *Last-Modified* que informa a última data do arquivo no servidor.

Para verificar se o browser salvou ou não no cache, basta dar um *refresh* na página e analisar o resultado na mesma aba (**Figura 2**).

Veja que o status mudou para 304, tal como a mensagem dos cabeçalhos exibida na **Listagem 2**. Nela, temos dois novos atributos (linhas 10 e 11) que enviam um código de identificação do recurso que já está no cache, bem como a data que o cliente tem da última modificação do mesmo. Como resposta (linha 13) o servidor envia que nenhuma modificação aconteceu até então, através da mensagem “304 Not Modified”.

Listagem 1. Conteúdo dos cabeçalhos de request/response.

```
01 GET /Imagens/2013/logo.png HTTP/1.1
02 Host: www.devmedia.com.br
03 Connection: keep-alive
04 Cache-Control: max-age=0
05 Accept: text/html,application/xhtml+xml,application/xml;
q=0.9,image/webp,*/*;q=0.8
06 User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/43.0.2357.134 Safari/537.36
07 Accept-Encoding: gzip, deflate, sdch
08 Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.6,en;q=0.4,es;q=0.2,fi;
q=0.2,gl;q=0.2
09 Cookie: ...
10
11 HTTP/1.1 200 OK
12 Cache-Control: max-age=604800
13 Content-Type: image/png
14 Last-Modified: Fri, 14 Feb 2014 19:14:27 GMT
15 Accept-Ranges: bytes
16 ETag: "802b79fab829cf1:0"
17 Server: Microsoft-IIS/7.5
18 X-Powered-By: ASP.NET
19 Date: Mon, 20 Jun 2015 13:14:43 GMT
20 Content-Length: 2883
```

Listagem 2. Conteúdo dos cabeçalhos de request/response cacheados.

```
01 GET /Imagens/2013/logo.png HTTP/1.1
02 Host: www.devmedia.com.br
03 Connection: keep-alive
04 Cache-Control: max-age=0
05 Accept: text/html,application/xhtml+xml,application/xml;
q=0.9,image/webp,*/*;q=0.8
06 User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/43.0.2357.134 Safari/537.36
07 Accept-Encoding: gzip, deflate, sdch
08 Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.6,en;q=0.4,es;q=0.2,
fi;q=0.2,gl;q=0.2
09 Cookie: ...
10 If-None-Match: "802b79fab829cf1:0"
11 If-Modified-Since: Fri, 14 Feb 2014 19:14:27 GMT
12
13 HTTP/1.1 304 Not Modified
14 Cache-Control: max-age=604800
15 Last-Modified: Fri, 14 Feb 2014 19:14:27 GMT
16 Accept-Ranges: bytes
17 ETag: "802b79fab829cf1:0"
18 Server: Microsoft-IIS/7.5
19 X-Powered-By: ASP.NET
20 Date: Mon, 20 Jun 2015 13:23:56 GMT
```

Além disso, perceba que na **Figura 1** temos a linha colorida no topo da imagem representando o tempo que o browser levou para carregar aquela página e todos os seus recursos. No primeiro exemplo (sem cache) levamos 790 milissegundos, no segundo esse tempo caiu para quase 250 milissegundos. Por se tratar de um exemplo básico, não vemos tanta melhora de cara, mas multiplique isso pelo tamanho dos recursos (arquivos, imagens, downloads, etc.) e pela quantidade deles em cada página e teremos uma melhora significativa nos carregamentos.

Expires e Cache-Control

Muita gente ainda se confunde em entender qual a função destes atributos do cabeçalho no HTTP. Eles fazem a mesma coisa: controlam o cache e definem quando determinados tipos de recursos expiram ou não. A única diferença é que o primeiro foi definido no HTTP 1.0 e o segundo no HTTP 1.1.

Nos seus servidores, como o Apache, é possível configurar o tempo máximo de armazenamento pelos tipos MIME de arquivos. Por exemplo, suponha que você deseja configurar que todas as suas imagens no formato GIF, PNG e JPEG, uma vez salvas no cache do navegador, só expirem daqui a dois meses. Para esse exemplo vamos considerar uma página HTML comum que carrega duas imagens relativamente pesadas, conforme a **Listagem 3**. As imagens precisam ser de um recurso externo, caso contrário o browser não considerará a inclusão no cache.

Para configurar o tempo de expiração, bastaria configurar no seu arquivo .htaccess os seguintes valores:

```
ExpiresByType image/gif "access plus 2 months"
ExpiresByType image/png "access plus 2 months"
ExpiresByType image/jpeg "access plus 2 months"
```

Listagem 3. Página HTML de exemplo com duas imagens grandes.

```
<!doctype html>
<html lang="en-US">
<head>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0,
minimum-scale=1.0, maximum-scale=1.0, user-scalable=no" />
<title>Teste Performance DevMedia</title>
</head>
<body>

</body>
</html>
```

Ao executar a página no navegador, você veria os três recursos (a página em si e as duas imagens) sendo carregados e o tempo que levou para carregar tudo, como na **Figura 3**. Veja que o Chrome exibe não só o tempo para carregar no total (parte de baixo da figura) como também o tempo individual de cada recurso. Levamos 25.34 segundos para carregar três recursos que somam 6.5MB.

Se quisermos eliminar toda essa sobrecarga, inclusive de ficar enviando o request GET de verificação o tempo todo, ao usar o script que vimos no servidor, o processamento seria reduzido ao que vemos na **Figura 4**.

Veja a enorme diferença no processamento, nove milissegundos para carregar a mesma página, porém sem requisições adicionais.

Compressão de Conteúdo

Pelas seções anteriores fomos capazes de analisar que quanto menor o dado trafegado mais otimizada a aplicação. Mas e quando não podemos armazenar no cache as informações? E se quisermos que nossa aplicação seja rápida sempre no primeiro acesso? As compressões são uma opção excelente que fazem uso de técnicas para minificar arquivos e dados antes de enviá-los.

Para ter uma ideia de como os grandes sites funcionam com compressão, se um usuário acessar o site do youtube.com via internet dial-up ele levará cerca de 14 segundos para ter a tela toda carregada sem compressão, ao passo que se comprimir seus arquivos esse tempo cairia para dois segundos. Levando em consideração sites muito pesados, como o site de mídias digitais chinês sina.com.cn, o tempo de carregamento sem compressão sobe para mais de um minuto, enquanto se devidamente comprimido, cairia para 16 segundos.

Como muitos outros recursos do HTTP, a compressão de conteúdo somente ocorre quando o cliente adverte que quer usar e o servidor retorna dizendo que pode usar. O cliente faz isso através do envio do cabeçalho *Accept-Encoding* ao fazer requisições. Se o servidor aceitar compressão, ele então devolve uma resposta com o recurso comprimido. Já o *Content-Length* do cabeçalho de resposta que vimos antes retorna o tamanho do conteúdo comprimido, não o original.

Como otimizar a performance no front-end

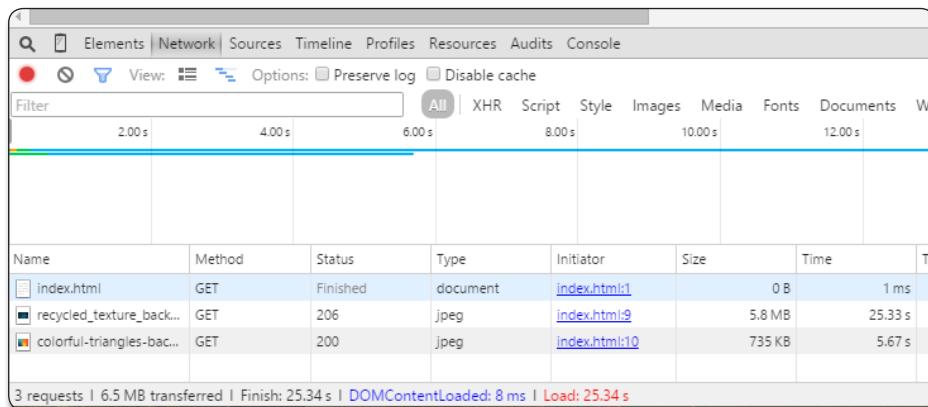


Figura 3. Resultado de carregamento dos três recursos

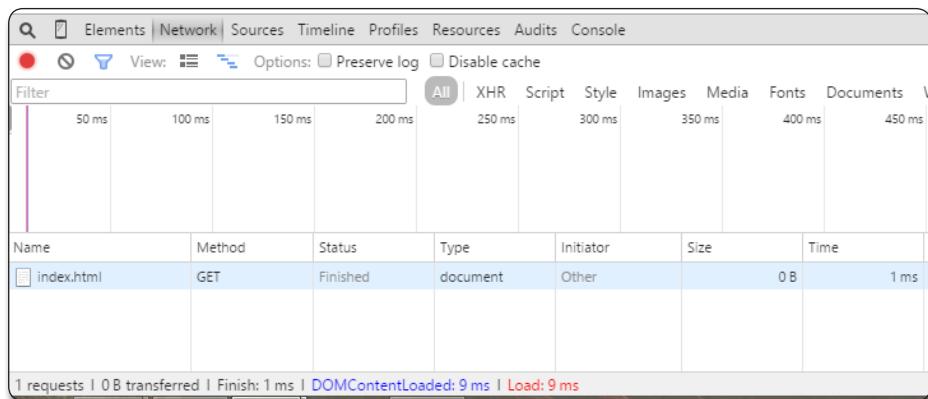


Figura 4. Resultado de carregamento dos três recursos, porém sem checagem GET

Método de compressão	Descrição
identity	Indica não-compressão. Se o cliente informa esse valor no cabeçalho Accept-Encoding, o servidor não deve comprimir o conteúdo da resposta.
compress	Esse método usa o método de compressão do UNIX, que é baseado no algoritmo de Lempel-Ziv Welch (LZW). O uso desse método está depreciado em detrimento dos próximos dois.
gzip	Por muito tempo foi o método mais popular de compressão, baseado no algoritmo LZ77, que tem base no LZW. Como o nome indica também é o mesmo algoritmo usado na ferramenta gzip do UNIX.
deflate	Essencialmente, é o mesmo que o gzip, porém sem a checagem de somas do cabeçalho (checksum). Ele é mais rápido que o gzip, mas menos eficiente.

Tabela 2. Lista de métodos de compressão do HTTP

Ao mesmo tempo, se o cliente suportar múltiplos métodos de compressão, ele deverá expressar uma preferência usando os valores q-. Por exemplo, considere o seguinte cabeçalho de resposta HTTP, onde o cliente expressa sua preferência pelo método *deflate* através do valor *q-value 1.0* em relação ao *gzip* que recebeu um 0.5:

Accept-Encoding: gzip;q=0.5, deflate; q=1.0

Veja na Tabela 2 os valores possíveis para compressões no HTTP.

O que comprimir?

Potencialmente, qualquer recurso pode ser comprimido, mas a compressão é geralmente apenas aplicada a texto com base em conteúdo, como HTML, XML, CSS e JavaScript. Existem duas razões para isso:

1. Texto tende a ser mais comprimido do que dados binários.

2. Muitos dos formatos binários em uso na web já usam compressão. GIF, PNG e JPEG são os principais exemplos, e não há praticamente nenhum ganho a ser adquirido através da aplicação de compressão de conteúdo para estes tipos. Isso simplesmente desperdiça ciclos de CPU e pode até fazer o tamanho ficar um pouco maior (porque o método de compressão pode adicionar seus próprios cabeçalhos aos dados).

Tendo em vista este último ponto, também não há nada a se ganhar ao comprimir recursos de textos menores que um determinado tamanho. As economias são mínimas, e cabeçalhos adicionados pelo método de compressão podem também tornar o recurso comprimido maior do que o original.

Desvantagens

Apesar de a compressão de conteúdo ser quase sempre uma coisa boa, existem situações onde ela deve ser bem analisada. Primeiro, você deve levar em consideração sempre o custo de uso da CPU tanto do lado cliente quanto do servidor, afinal o servidor deve comprimir o documento e o cliente descomprimi-lo. Existem formas de fazer com que os servidores enviem versões pré-comprimidas dos recursos, em vez de comprimir cada um deles sempre que uma requisição for feita.

Segundo, existe sempre uma pequena parcela dos usuários que usam navegadores antigos e que não suportam tal recurso. Então cabe a você analisar se o seu público-alvo atende as exigências.

Outros métodos de compressão

Apesar do gzip e deflate serem os métodos mais usados pela comunidade, alguns outros também trazem seus benefícios para situações bem específicas, a saber:

- **bzip2:** oferece compressão superior em relação ao primo gzip (tendo como consequência o consumo maior de CPU), muito comum aos usuários UNIX. Apesar de não ser suportado pelos browsers, pode ser usado em APIs de terceiros para prover melhores soluções.
- **SDCH:** comumente chamado de “sandwich”, a sigla vem de *Shared Dictionary*

Compression for HTTP (dicionário de compressão compartilhada do HTTP) e é um método de compressão desenvolvido pelo Google. Com ele é possível a implementação de redundâncias de alto nível através de domínios específicos em uma rede via dicionário de dados.

- **EXI:** focado em conteúdo XML, o EXI codifica XML em formato binário, drasticamente reduzindo o tamanho do arquivo. Apesar dos browsers não o adotarem, o W3C está em favor dessa adoção haja visto o grande poder de compressão que ele oferece e que pode ser explorado por outros algoritmos.

- **peerdist:** desenvolvido pela Microsoft, este método aplica a lógica *peer-to-peer* das redes de computadores na web através da habilitação de um cliente para recuperar conteúdo tanto dos peers quanto do servidor.

Compressão em PHP

Muitos desenvolvedores web usam como opção a compressão de arquivos HTML em linguagens de script antes de enviá-los ao browser. Essa estratégia tem uma grande vantagem que é não precisarmos ter de modificar nada no servidor para enviar algo comprimido e ter um código funcionando bem ao mesmo tempo.

Para comprimir páginas geradas pelo PHP, existem basicamente duas opções: buffer de saída e zlib. Essa última é a preferida pelos developers atuais, mas a primeira é a que mais se difundiu por ser mais antiga. Vejamos um exemplo de cada uma. Na **Listagem 4** temos o código de uma página PHP simples com uma chamada à função `ob_start()`.

Listagem 4. Exemplo básico de página PHP.

```
<?php  
ob_start("ob_gzhandler");  
?  
<html>  
<head>  
...  
...
```

Veja que o buffer de saída é usado junto à função de callback “`ob_gzhandler`” para comprimir a saída antes de enviá-la ao browser.

Você pode ir além, criando a sua própria função de callback para lidar com a compressão manualmente. Apesar disso, não se pode usar alguma alternativa de compressão como o bzip (ainda estamos limitados ao que o cliente suporta), dando a oportunidade de deployar uma minificação. Nossa função ficaria como a da **Listagem 5**.

O primeiro passo importante é manualmente adicionar o buffer de saída no topo do script. Ele será recebido como um parâmetro do método e deverá ser preenchido, para então podermos retorná-lo. Uma desvantagem dessa estratégia é que, como consequência do buffering, nenhuma compressão acontecerá até que o script finalize sua execução. Só assim os dados estarão completamente comprimidos e disponíveis para ir ao cliente.

Listagem 5. Mesmo exemplo com função de compressão.

```
<?php  
ob_start("minhaFuncao");  
function minhaFuncao($buffer) {  
    $buffer = str_replace(array("\r", "\n", "\n", "\t"), "");  
    return $buffer;  
}  
?  
<html>  
<head>  
...
```

A segunda forma, com o zlib, envolve uma extensão da linguagem PHP através da opção `ini` que pode ser setada no código em si, via função `ini_set()`, via arquivo `.htaccess`, ou via `php.ini`, conforme vemos a seguir:

```
zlib.output_compression = 1
```

O valor confere a característica de habilitado (`true`). Opcionalmente, você pode também configurar o nível do componente, como por exemplo:

```
zlib.output_compression_level = 9
```

O valor default é `-1`, que faz com que o PHP escolha um nível. O único problema aqui é que esse tipo de implementação é global, isto é, você não pode escolher quais arquivos deseja aplicar tais regras, elas valem para todos.

Minification

Minification (ou minificação) é o ato de remover todos os caracteres desnecessários (espaços em branco e comentários) do código afim de reduzir o seu tamanho final, e um *minifier* (minificador) é a ferramenta que faz isso. A técnica é comumente associada ao JavaScript, mas também é possível aplicá-la em arquivos CSS e HTML.

Esse tipo de estratégia sempre foi alvo de muitas críticas em comparação às técnicas de compressão, já que um arquivo minificado não apresenta tanta diferença de tamanho em relação a um comprimido. Mas levando em consideração as análises de alguns estudos feitos em diversas ferramentas, o ganho aproximado gira em torno de 10% que, dependendo da sua aplicação, pode ser bem significativo. Esse valor aumenta consideravelmente quando se trata de clientes navegadores que não permitem compressão.

Todavia, como consequência, temos código de arquivos inteiros resumido em uma linha só, o que o torna difícil de ler e modificar. Uma solução viável é guardar sempre duas versões dos arquivos, tal como fazem a maioria das empresas que tem uma tecnologia do tipo na comunidade, como o jQuery, Prototype, CoffeScript, dentre inúmeros outros. Assim, você modifica sempre a versão não-minificada e usa alguma ferramenta automatizada para gerar a versão minificada correspondente.

Como otimizar a performance no front-end

JavaScript Minification

Cada ferramenta minificadora usa suas próprias regras para encurtar os arquivos. Algumas delas, inclusive, além de remover espaços em branco e comentários também encurtam os nomes de funções, classes, variáveis e objetos para simplificar o código. Vejamos na **Listagem 6** um exemplo de código JavaScript que lida com uma função *toggle* para mudar a visibilidade de um elemento.

Salvando o código em um arquivo simples, seu tamanho total será de aproximadamente 295 bytes. Mas antes de rodar o arquivo em um minifier, é interessante que você o deixe mais organizado, como criando uma variável para o elemento recuperado pelo id, em vez de duplicar o código várias vezes, como na **Listagem 7**.

O melhor processo de minificação de código é conhecer como fazer isso, saber usar as boas práticas de programação. Ao salvar o mesmo arquivo diminuímos seu tamanho total para 241 bytes. Para finalizar a simplificação manual, podemos nos livrar dos blocos if/else trocando por uma operação ternária, como na **Listagem 8**.

Salve novamente e veja o tamanho que o arquivo ficou: 189 bytes. Você conseguiu reduzir quase pela metade e o código continua legível.

Listagem 6. Função para modificar a visibilidade de um elemento.

```
function toggle(idElemento) {
  if (document.getElementById(idElemento).style.display != 'none') {
    document.getElementById(idElemento).style.display = 'none';
  } else {
    document.getElementById(idElemento).style.display = '';
  }
}
```

Listagem 7. Mesma função refatorada.

```
function toggle(idElemento) {
  var el = document.getElementById(idElemento);
  if(el.style.display != 'none') {
    el.style.display = 'none';
  } else {
    el.style.display = '';
  }
}
```

Listagem 8. Mesma função refatorada novamente.

```
function toggle(idElemento) {
  var el = document.getElementById(idElemento);
  (el.style.display != 'none') ? el.style.display = 'none' : el.style.display = '';
}
```

YUI Compressor

Para minificar o arquivo usaremos o compressor da Yahoo!, o YUI Compressor. Na seção **Links** você encontra a referência ao site da ferramenta. Para usá-la é necessário ter o Java (4 ou superior) instalado na máquina e configurado para uso via linha de comando (na própria página você encontra tutoriais de como fazer isso).

Abra o prompt cmd do seu Windows (ou o respectivo nos outros SOs), navegue até o diretório onde você salvou o jar (é importante salvar o arquivo js no mesmo diretório do jar) e digite o seguinte comando:

```
java -jar yuicompressor-2.4.8.jar saida.js
```

Onde saida.js é o nome do arquivo que você salvou o código da **Listagem 8**. Você verá a saída representada pela **Figura 5**. Veja que o comando gera o código no próprio console para que você possa usar.

Em cenários onde tenhamos códigos bem mais extensos, essa não seria a melhor estratégia. Portanto, execute o seguinte comando:

```
java -jar yuicompressor-2.4.8.jar saida.js > saida_min.js
```

Ele não gerará nenhuma saída no cmd, mas criará um novo arquivo no mesmo diretório chamado saida_min.js com o mesmo código minificado:

```
function toggle(b){var a=document.getElementById(b);(a.style.display!="none")?a.style.display="none":a.style.display="";}
```

Veja como a ferramenta substitui as referências originais dos nomes de variáveis dentro da função. Por isso é aconselhado manter uma versão original, pois tal código, mesmo simples, é difícil de entender.

Google Closure

O Google também tem a sua própria solução de minifier para as mesmas finalidades, o Google Closure Compiler (seção **Links**). Ele fornece os mesmos recursos do YUI, porém com alguns plus como expansões inline e remoção de código não usado.

O *inlining* é um conceito adotado pelos compiladores da linguagem C e significa a ação de inserir conteúdo de uma função em um lugar onde a mesma seria chamada. Isso pode aumentar a velocidade de execução (ao cortar a sobrecarga envolvida em ter de chamar a função), e reduzir o tamanho do arquivo final. Vejamos o código da **Listagem 9** que traz uma função que exibe uma mensagem de alerta e é chamada em seguida.

No código exibimos a mensagem em detrimento do valor do campo de formulário (campo-



Figura 5. Resultado da compressão do arquivo via YUI Compressor

form). Caso a mensagem seja “no” chamamos a função de alerta. Para rodar o Closure também é necessário baixar o seu jar (vide seção [Links](#)) e, via prompt cmd, executar o seguinte comando:

```
java -jar compiler.jar --compilation_level ADVANCED_OPTIMIZATIONS --js ex-closure.js --formatting PRETTY_PRINT
```

O uso da opção PRETTY_PRINT no fim do comando serve para exibir o código formatado com os espaços em branco, você pode remover a opção caso deseje o contrário. Já ex-closure.js é o nome do arquivo em que salvamos o referido código. O resultado pode ser visto na [Figura 6](#).

Como você pode ver a função showalert() foi removida e seu conteúdo reposicionado *inline* no local onde a função antes era chamada. Para um exemplo simplista como esse, que faz uso somente de uma chamada à função, tudo bem, mas o que aconteceria se tivéssemos várias chamadas ao longo do arquivo, tal como temos na [Listagem 10](#)?

Basta reexecutar o comando e o resultado seria semelhante ao da [Listagem 11](#).

Dessa vez, o Closure se preocupou em não colocar a função *inline*, pois teria de duplicá-la caso isso acontecesse. Em vez disso, ele apenas encurtou os nomes das funções e variáveis tal como o YUI faria.

Estes foram apenas dois dos principais players do mercado para lidar com minificação de arquivos JavaScript. Veja na [Tabela 3](#) uma lista com todos os principais, bem como o tamanho total de um arquivo de 120KB após minificado.

Minifier	Descomprimido (KB)	Gzipped (KB)
Nenhum	122	28
YUI Compressor	71	21
JSMin	91	23
Closure (simples)	70	21
Closure (avançado)	55	18
Packer	90	23

Tabela 3. Lista de principais minifiers do mercado e suas performances

As duas opções que analisamos são as líderes. As ferramentas também lidam com arquivos de CSS e HTML, então faça alguns testes com os arquivos que criamos aqui. Veja como elas se comportam minificando-os.

Otimização de Gráficos e Imagens

As imagens de uma aplicação web geralmente são criadas por um web designer ou designer gráfico, para quem a qualidade é

```
C:\WINDOWS\system32\cmd.exe
D:\temp\yui>java -jar compiler.jar --compilation_level ADVANCED_OPTIMIZATIONS --
js ex-closure.js --formatting PRETTY_PRINT
x = document.getElementById("campo-form");
"no" == x.value && (el = document.getElementById("caixa-alerta"), el.innerHTML =
"Você realmente disse Não?", el.style.visibility = "visible");

D:\temp\yui>
```

Figura 6. Resultado da compressão do arquivo via Google Closure

Listagem 9. Exemplo de função que exibe mensagem de alerta.

```
function showalert(msg) {
  el = document.getElementById('caixa-alerta');
  el.innerHTML = msg;
  el.style.visibility = 'visible';
}

x = document.getElementById('campo-form');
if (x.value == "no") {
  showalert("Você realmente disse Não?");
}
```

Listagem 10. Exemplo de função que exibe mensagem de alerta com várias chamadas.

```
function showalert(msg) {
  el = document.getElementById('caixa-alerta');
  el.innerHTML = msg;
  el.style.visibility = 'visible';
}

x = document.getElementById('campo-form');
if (x.value == "no") {
  showalert("Você realmente disse Não?");
}

y = document.getElementById('campo-form1');
if (y.value == "no") {
  showalert("Você realmente disse Não?");
}

z = document.getElementById('campo-form2');
if (z.value == "no") {
  showalert("Você realmente disse Não?");
}
```

Listagem 11. Resultado da minificação do último arquivo js.

```
D:\temp\yui>java -jar compiler.jar --compilation_level ADVANCED_
OPTIMIZATIONS --
js ex-closure.js --formatting PRETTY_PRINT
function a0() {
  el = document.getElementById("caixa-alerta");
  el.innerHTML = "Você realmente disse Não?";
  el.style.visibility = "visible";
}
x = document.getElementById("campo-form");
"no" == x.value && a0();
y = document.getElementById("campo-form1");
"no" == y.value && a0();
z = document.getElementById("campo-form2");
"no" == z.value && a0();
```

um conceito primário. Então, não é surpresa que a otimização desse tipo de arquivo seja frequentemente ignorada. Você pode quase que sempre diminuir 10KB ou até 20KB de uma imagem sem nenhuma perda visível de qualidade.

Durante muito tempo os dois tipos de formatos de imagens que existiam se resumiam a GIF e JPEG. A primeira era usada para definir imagens feitas em computadores, como ícones, logos, etc. e a segunda para agrupar fotografias. Em 1996, uma nova especificação para imagens foi lançada: a PNG que logo se tornou um padrão adotado pelo W3C e, em seguida, pelos browsers. Hoje, ela é usada para as duas finalidades.

A primeira regra para lidar com performance de imagens na web é saber que ferramentas usar para fazer isso. Uma das opções mais usadas e recomendadas do mercado é o GIMP (*GNU Image Manipulation Program*); ele é gratuito, funciona muito bem no Windows, UNIX e Mac, e faz um ótimo trabalho comprimindo imagens de todos os formatos. Além dele, ferramentas como o Adobe Photoshop e o próprio Paint do Windows são usadas. Sempre atente para o tamanho final do arquivo, faça testes com as mesmas imagens e analise qual ferramenta se adequa melhor no paralelo tamanho-quality.

Geralmente, podemos dividir a otimização de imagens em quatro passos:

1. Remova cabeçalhos redundantes;
2. Reduza a profundidade das cores (por exemplo, mude o formato de RGB para palleted, ou reduza o número de cores na pallete);
3. Reimplemente ou use filtros alternativos;
4. Aumente a performance dos algoritmos LZ77 e deflate. Algumas vezes é necessário usar implementações customizadas.

Removendo cabeçalhos redundantes

Esse passo envolve a remoção de metadados que não pertençam de fato à imagem. Apesar de não termos perdas na qualidade da imagem, dados estão sendo retirados do arquivo, logo muito cuidado com ele.

O PNG suporta mais de uma dúzia de cabeçalhos adicionais, que não são requeridos e podem ser removidos sem efeitos colaterais. A seguir vemos uma lista dos possíveis candidatos a remoção:

- **bKGD**: especifica uma cor padrão de plano de fundo. Essa opção não é usada pelos navegadores web;
- **pHYs**: define as proporções de tela ou tamanho em pixel da imagem;
- **sBIT**: armazena uma número significativo de bits possibilitando ao browser reconstituir imagens convertidas de uma com baixo nível de profundidade;
- **sPLT**: armazena uma paleta sugerida quando o viewer bloqueia a capacidade de mostrar o intervalo completo de cores;
- **HIST**: contém um histograma da paleta, com gráficos do quanto frequente cada cor aparece na imagem;
- **tIME**: a última hora que a imagem foi modificada.
- **cHRM**: armazena as coordenadas x/y das cores primárias usadas na imagem.

Adicionalmente, existem três campos de texto (zTXT, tEXT e iTXT) que servem para guardar informações relacionadas à imagem. Tipicamente, isso inclui título, autor, descrição, informações de copyright, comentários, etc.

Reduzindo a profundidade das cores

O formato PNG se divide em outros três de acordo com a qualidade e tamanho da imagem: PNG8 (usa uma tabela de cores 8-bit, ou seja 256), PNG24 (substituta equivalente ao JPEG) e PNG32 (similar ao modo RGB, lida com imagens que têm muitas cores e geralmente são arquivos pesados). Quando for gerar suas imagens, ou recebê-las do web designer, solicite que as mesmas sejam criadas sob a base PNG8 que resolve 95% das necessidades. Salvo os casos em que uma qualidade estupenda seja necessária, certifique-se de criar o tipo corretamente.

Essa estrutura se assemelha muito à escolha de um tipo de dado primitivo quando se está programando. Você pode muito bem salvar o valor “2” em uma variável de tipo *double*, mas se sabe que o valor será sempre esse, é bem melhor salvá-lo numa de tipo *byte*, pois assim poupará um enorme espaço em memória.

Data URIs

Uma das estratégias de otimização para imagens mais antigas, mas que só começou a ser adotada recentemente, é o esquema de URIs. Este esquema permite que dados (como imagens) sejam embutidos *inline* nas páginas web, em vez de carregados de fontes externas. A sintaxe para isso pode ser vista a seguir:

```
data:<mime type>;base64,<data>
```

Por exemplo, considere a inserção de uma imagem pequena, com todas as definições de cores, tamanho, posição x/y no plano cartesiano, etc., como na **Listagem 12**.

Listagem 12. Exemplo de página com imagem em Data URI.

```
<!doctype html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
      minimum-scale=1.0, maximum-scale=1.0, user-scalable=no" />
    <title>Data URI Exemplo</title>
  </head>
  <body>
    
  </body>
</html>
```

Veja a forma como a tag é usada e seu atributo src setado. Dessa forma, não precisamos mais enviar um request para buscar o recurso no servidor, o que deixa nossa aplicação mais rápida, consequentemente. O resultado pode ser visto na **Figura 7**.

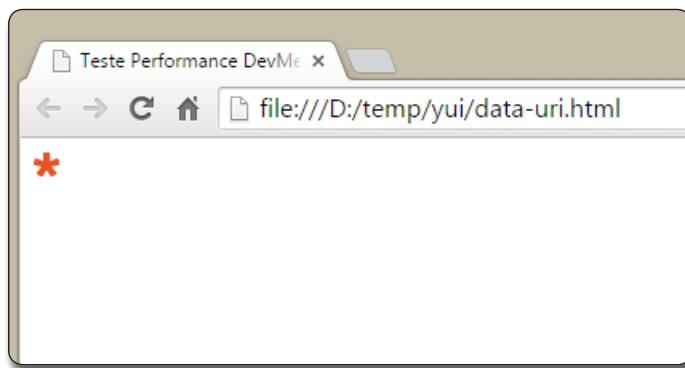


Figura 7. Resultado do exemplo com Data URI

Essa abordagem tem algumas desvantagens, como o limite que alguns browsers impõem a URIs internas (32KB para o IE 8, por exemplo) o que acaba limitando o escopo de atuação. Além do mais, quanto maior a imagem maior será a URI para exibi-la o que pode levar, em alguns casos, a imagem a ficar maior que quando importada do servidor.

CSS Sprites

A propriedade CSS *background-image* possibilita configurar uma imagem para ser usada como plano de fundo de um dado elemento. Uma das vantagens de configurar uma imagem com esse atributo é que podemos selecionar regiões da mesma através da propriedade *background-position*. Partindo para uma situação mais real, imagine que na sua página HTML você exiba dez ícones, cada um representado por um arquivo de imagem diferente. Quando a página for carregada, o browser terá de gerar dez requisições ao servidor para recuperar cada uma das dez imagens, mesmo que elas tenham sido referenciadas na mesma propriedade CSS. Isso traria uma sobrecarga desnecessária para a aplicação ao carregar arquivos tão pequenos.

Com a técnica do CSS Sprite você pode gerar apenas um arquivo de imagem com todas as demais uma ao lado da outra e importá-lo apenas uma vez no CSS, porém referenciando as regiões (posição x/y) onde cada uma está na imagem geral.

Por exemplo, a página de buscas do Google.com faz uso de sprites para exibir seus ícones. Na **Figura 8** temos a imagem que ele usa e na **Listagem 13** o código CSS para exibir os botões de notificações e dos Apps Google.

Veja como ele faz uso das propriedades que comentamos para posicionar a exibição nos referidos elementos HTML de div. Se você modificar os valores do *background-position* pode facilmente mudar os ícones. Por exemplo, modifique os valores do primeiro e segundo elementos para o código da **Listagem 14**.

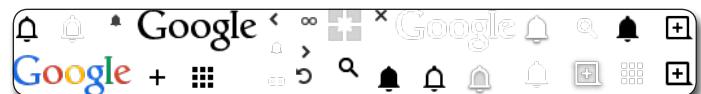


Figura 8. Imagem de sprites da página do Google.com

Listagem 13. Código CSS que o Google usa para exibir os dois botões.

```
/* Botão Google Apps */
.gb_6a.gb_ca {
    background-position: -132px -38px;
    opacity: .55;
}
.gb_7 {
    background-image: url('//ssl.gstatic.com/gb/images/i1_0430b5ba.png');
    -webkit-background-size: 528px 68px;
    background-size: 528px 68px;
}

/* Botão de Notificações */
.gb_da {
    background-color: rgba(0,0,0,.55);
    color: #fff;
    font-size: 12px;
    font-weight: bold;
    line-height: 20px;
    margin: 5px;
    padding: 0 2px;
    text-align: center;
    -webkit-box-sizing: border-box;
    box-sizing: border-box;
    -webkit-border-radius: 50%;
    border-radius: 50%;
    height: 20px;
    width: 20px;
}
.gb_7 {
    background-image: url('//ssl.gstatic.com/gb/images/i1_0430b5ba.png');
    -webkit-background-size: 528px 68px;
    background-size: 528px 68px;
}
```

Listagem 14. Código CSS que do Google modificado.

```
/* Botão Google Apps */
.gb_6a.gb_ca {
    background-position: -241px 0px;
    ...
}
/* Botão de Notificações */
.gb_da.gb_ea {
    background-position: -351px 22px;
}
.gb_7 {
    ...
    background-size: 528px 60px;
}
```

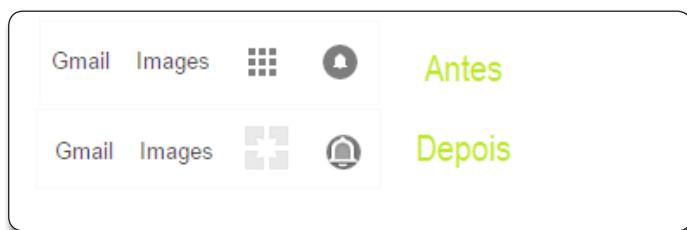


Figura 9. Imagem de sprites da página do Google.com modificados

O resultado do original e do alterado pode ser visualizado na **Figura 9**.

Há muitas outras práticas que o leitor pode usar para continuar otimizando e melhorando a performance de suas aplicações web, tais como:

- Agrupe seus arquivos JavaScript e CSS (se você usa muitas bibliotecas de terceiros e as que você mesmo criou, procure juntar e comprimir tudo no menor número de arquivos, assim menos requests serão gerados);
- Use o Yahoo Smush-it para remover todos aqueles cabeçalhos e paletas desnecessários das suas imagens;
- Nunca redimensione imagens diretamente na HTML a custo de gerar mais sobrecarga para o browser;
- Ponha os arquivos CSS no topo e JavaScript embaixo, para que eles não atrapalhem no processamento da página como um todo;
- Use ferramentas para medir sua performance, como o YSlow ou o PageSpeed, elas ajudam a fazer um diagnóstico completo da aplicação e te dá relatórios do que pode ser melhorado;
- Use CDNs quando estiver trabalhando com bibliotecas de terceiros, em vez de baixar os arquivos fisicamente no seu projeto.

Se estiver trabalhando com tecnologias server side como JSF para Java ou ASP.NET para .NET você terá mais trabalho para aplicar tais conceitos, visto que são ferramentas que geram códigos sujos e difíceis de modificar. Portanto, analise sempre muito bem cada situação antes de começar o projeto.

Autora



Sueila Sousa

É tester e entusiasta de tecnologias front-end. Atualmente trabalha como analista de testes na empresa Indra, com foco em projetos de desenvolvimento de sistemas web, totalmente baseados em JavaScript e afins. Possui conhecimentos e experiências em áreas como Gerenciamento de processos, banco de dados, além do interesse por tecnologias relacionadas ao desenvolvimento e teste client side.



Links:

Página do YUI Compressor.

<http://yui.github.io/yuicompressor/>

Página de download do YUI Compressor.

<https://github.com/yui/yuicompressor/releases/>

Página do Google Closure Compiler.

<https://developers.google.com/closure/compiler/?csw=1>

Página de download da última versão do Closure.

<http://dl.google.com/closure-compiler/compiler-latest.zip>

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