

# AngularJS

na prática



Daniel Schmitz  
Douglas Lira

# AngularJS na prática

Crie aplicações web com AngularJS

Daniel Schmitz and Douglas Lira

This book is for sale at <http://leanpub.com/livro-angularJS>

This version was published on 2014-01-13



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2013 - 2014 Daniel Schmitz and Douglas Lira

# Tweet This Book!

Please help Daniel Schmitz and Douglas Lira by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

Comprei o livro AngularJS na prática do autor Daniel Schmitz. Mais informações em:  
<https://leanpub.com/livro-angularJS>

The suggested hashtag for this book is [#livroangularjs](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#livroangularjs>

# Conteúdo

Errata . . . . .	i
Introdução . . . . .	ii
O que é AngularJS? . . . . .	ii
Código fonte . . . . .	ii
Uma nota sobre pirataria . . . . .	iii
 Parte 1 - AngularJS . . . . .	 1
Preparando o ambiente . . . . .	2
Editor de textos . . . . .	2
Servidor Web . . . . .	2
Instalando o AngularJS . . . . .	3
Principais características do AngularJS . . . . .	4
DataBind . . . . .	4
Controller . . . . .	5
Métodos do controller . . . . .	6
Loops . . . . .	7
Formulários . . . . .	8
Rotas e Deep linking . . . . .	9
Documentação do AngularJS . . . . .	14
Conectando AngularJS ao servidor . . . . .	15
Uso do \$http . . . . .	16
Exemplo com \$http . . . . .	17
Uso do \$resource . . . . .	19
Exemplo simples com \$resource . . . . .	20
 Parte 2 - Sistema de vendas . . . . .	 24
Preparando o sistema . . . . .	25
Técnicas comuns para o uso dos arquivos javascript . . . . .	25

## CONTEÚDO

Como carregar javascript? . . . . .	25
Preparando servidor . . . . .	26
Banco de dados . . . . .	26
Preparando o backend . . . . .	27
Arquivos iniciais do backend . . . . .	28
Arquivo DB.php . . . . .	28
Arquivo config.php . . . . .	32
Arquivo index.php . . . . .	33
Arquivo customer.php . . . . .	34
<b>Preparando o cliente . . . . .</b>	<b>37</b>
Estrutura de pastas . . . . .	37
Arquivo index.html . . . . .	38
Arquivo app.js . . . . .	40
<b>Cadastro de clientes . . . . .</b>	<b>45</b>
Templates . . . . .	45
Controller . . . . .	49
<b>Cadastro de funcionários . . . . .</b>	<b>54</b>
Servidor . . . . .	54
Cliente . . . . .	56
<b>Demais cadastros . . . . .</b>	<b>60</b>
<b>Validação . . . . .</b>	<b>61</b>
Validando com html . . . . .	61
Validando com ng-pattern . . . . .	63
 <b>Parte 3 - Node.JS . . . . .</b>	 <b>65</b>
<b>Introdução ao NODE.JS . . . . .</b>	<b>66</b>
Como funciona o NODE.JS? . . . . .	66
O que definitivamente não é NODE.JS . . . . .	66
<b>Instalação . . . . .</b>	<b>67</b>
Instalando o NODE.JS no Windows . . . . .	67
Instalando o NODE.JS no Linux . . . . .	67
<b>Exemplos . . . . .</b>	<b>69</b>
Criando o primeiro script com NODE.JS . . . . .	69
<b>Sockect.IO . . . . .</b>	<b>72</b>
O que é socket.IO . . . . .	72
Criando um script com socket.io . . . . .	72

# Errata

Esta é uma obra em constante evolução, e erros podem aparecer. Caso encontre algo errado, por favor envie um email para [vendas@danielschmitz.com.br](mailto:vendas@danielschmitz.com.br).

Todos os erros corrigidos serão atualizados automaticamente, e publicados em futuras versões da obra. Você poderá a qualquer momento obter uma nova versão, visitando o site [leanpub.com](http://leanpub.com)

# Introdução

A linguagem JavaScript vem tornando-se uma das mais usadas em todas as áreas de programação que estão ligadas a Web. É uma linguagem que praticamente nasceu com a Web, criada inicialmente para os navegadores *Netscape*, por volta de 1995.

Mais de 15 anos depois, javascript evolui de forma significativa com a criação de novos frameworks, tornando-se a cada dia mais poderosa e utilizada. Um deles é o jQuery, amplamente conhecido e praticamente obrigatório se você deseja controlar os elementos de uma página HTML, também chamado de DOM.

Nesta “onda”, diversos frameworks de qualidade surgem a cada dia, e dentre eles temos o AngularJS, que será o nosso principal objeto de estudo.

## O que é AngularJS?

Este framework é mantido pelo Google e possui algumas particularidades interessantes, que o fazem um framework muito poderoso.

Uma dessas particularidades é que ele funciona como uma extensão ao documento HTML, adicionando novos parâmetros e interagindo de forma dinâmica com vários elementos. Ou seja, com AngularJS podemos adicionar novos atributos no html para conseguir adicionar funcionalidades extras, sem a necessidade de programar em javascript.

AngularJS é quase uma linguagem declarativa, ou seja, você usa novos parâmetros na linguagem html para alterar o comportamento padrão do html. Estes parâmetros (ou propriedades) são chamados de diretivas, na qual iremos conhecer cada uma ao longo desta obra.

Além disso, é fornecido também um conjunto de funcionalidades que tornam o desenvolvimento web muito mais fácil e empolgante, tais como o DataBinding, templates e fácil uso do Ajax. Todas essas funcionalidades serão abordadas ao longo desta obra.

## Código fonte

O código fonte desta obra encontra-se em:

<https://github.com/danielps/livro-angular>

## Uma nota sobre pirataria

Todos nós sabemos que a pirataria existe, seja ela de filmes, músicas e livros. Eu, como autor, entendo o porquê disso acontecer, pois a maioria das distribuidoras cobra um preço além do valor, pois eles precisam obter uma margem de lucro satisfatória. Não é atoa que existem ebooks pelo preço de 50 reais ou até mais.

Como autor independente, eu consigo retirar a maioria dos custos envolvidos na produção do livro. Retiro custos de produção, design e até impostos (sim, eu posso retirar os impostos por estar produzindo conhecimento).

Tudo isso é retirado e o que você paga nos meus ebooks é o preço do meu desenvolvimento, que nunca vai passar de 10 dólares.

Se você comprou o livro, mais uma vez *obrigado!!*, mas se você obteve esta cópia por outros meios, eu faço um convite para que leia a obra e, caso goste, visite <https://leanpub.com/livro-angularJS> e compre o nosso livro.

Ajude o autor a produzir mais livros de qualidade por um preço justo, contribua para que você possa encorajar outros autores a produzirem conteúdo nesse formato.



# **Parte 1 - AngularJS**

# Preparando o ambiente

É preciso muito pouco para começar a aprender AngularJS. Em um nível mais básico, você precisa de um editor de textos e de um navegador web.

Como o nosso objeto é criar sistemas, iremos adicionar como requisito o uso de um servidor web capaz de processar páginas em PHP.

## Editor de textos

Aqui deixamos livre a sua escolha por um editor de textos ou uma IDE. Lembre-se que todo o nosso desenvolvimento é focado em HTML e JavaScript, ou seja, você não precisará de algo “poderoso” para aprender AngularJS, apenas algo que complemente o código HTML já está ótimo.

Nesta obra estamos usando extensivamente o *Sublime Text 2*, inclusive para escrever o próprio livro, então nos sentimos confortáveis em recomendá-lo.

## Servidor Web

O Servidor Web é necessário para processar as páginas PHP dos sistemas que iremos construir. O AngularJS é um framework para visualização de dados e informações, ele não possui a funcionalidade de prover dados dinâmicos, ou persistir informações em um banco de dados. Estas características são provenientes de um servidor Web.

Um dos servidores mais comuns nos dias de hoje é o Apache. O banco de dados que utilizaremos é o MySQL. Para cada sistema operacional, existe uma forma bastante fácil de possuir estes programas.

### Windows

Se você utiliza Windows, poderá instalar o **Wamp Server**, disponível neste link<sup>1</sup>. Faça o download da versão mais recente e instale o Wamp na configuração padrão.

Após instalado, você poderá incluir arquivos na seguinte pasta `C:\wamp\www`, e poderá utilizar o Apache, bem como o MySQL e outros utilitários acessando `http://localhost/`.

**Mac** Assim como existe o Wamp para o Windows, existe o Mamp<sup>2</sup> para o Mac, e você pode instalá-lo caso julgue necessário.

## Linux

Se usa Linux, acredito que instalar Apache e MySQL no sistema é uma tarefa extremamente simples para você :)

Os exemplos desta obra foram criados utilizando o Wamp Server como servidor Web.

## Instalando o AngularJS

É preciso apenas duas alterações na estrutura de um documento HTML para que possamos ter o AngularJS instalado. A primeira, e mais óbvia, é incluir a biblioteca javascript no cabeçalho do documento. A segunda, e aqui temos uma novidade, é incluir a propriedade **ng-app** no elemento html em que queremos “ativar” o angularJS. Neste caso, começamos inserindo na tag <html> do documento. O exemplo a seguir ilustra este processo.

### HTML default para o AngularJS

```
1 <html ng-app>
2   <head>
3     <title>Lista de compras</title>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular\
5 r.min.js"></script>
6   </head>
7   <body>
8   </body>
9 </html>
```

Na linha 1, temos o uso do **ng-app** que habilita o AngularJS a alterar o comportamento das tags html abaixo dele, permitindo assim que possamos utilizar mais propriedades do AngularJS.

Na linha 4 incluímos a biblioteca angularJS diretamente, através de um endereço CDN. Pode-se também realizar o download da biblioteca e inserir no projeto.

---

<sup>1</sup><http://www.wampserver.com/en/>

<sup>2</sup><http://www.mamp.info/en/index.html>

# Principais características do AngularJS

Agora que temos o básico em funcionamento, vamos aprender as principais regras do AngularJS. Através delas será possível realizar diferentes tarefas que irão tornar o desenvolvimento web muito mais simples e prazeroso.

## DataBind

Uma das principais vantagens do AngularJS é o seu DataBind. Este termo é compreendido como uma forma de ligar automaticamente uma variável qualquer a uma outra. Geralmente, o DataBind é usado para ligar uma variável do JavaScript (ou um objeto) a algum elemento do documento HTML.

No exemplo a seguir, estamos usando o AngularJS para ligar uma caixa de texto (o elemento input do html) à um cabeçalho.

### Exemplo de DataBind

---

```
1 <html ng-app>
2   <head>
3     <title>Hello World</title>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular.min.js"></script>
5   </head>
6   <body>
7     Hello <input type="text" ng-model="yourName"/>
8     <hr/>
9     <h1>Hello {{yourName}}</h1>
10  </body>
11 </html>
```

---

Além da propriedade **ng-app** (linha 1), utilizamos para DataBind a propriedade **ng-model**, para informar que este elemento estará ligado a uma variável do AngularJS, através da variável `yourName`, na linha 8. Isso significa que qualquer alteração na caixa de texto irá atualizar o valor da variável.

Na linha 10, temos a chamada à variável através do comando `{{yourName}}`, que imprime o valor da variável. Como o DataBind é dinâmico, ao mesmo tempo que algo é escrito na caixa de texto, o seu referido bind é realizado, atualizando instantaneamente o seu valor.

Bind também pode ser realizado em objetos, mas antes de começar a aumentar a complexidade do código, vamos criar um *controller* para melhorar a organização do código.

## Controller

Um controller é, na maioria das vezes, um arquivo JavaScript que contém funcionalidades pertinentes à alguma parte do documento HTML. Não existe uma regra para o controller, como por exemplo ter um controller por arquivo HTML, mas sim uma forma de sintetizar as regras de negócio (funções javascript) em um lugar separado ao documento HTML.

Inicialmente, vamos criar um exemplo simples apenas para ilustrar como usar o controller.

### Uso do controller

---

```
1 <html ng-app>
2   <head>
3     <title>Hello World</title>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular.min.js"></script>
5     <script src="simpleController.js"></script>
6   </head>
7   <body ng-controller="simpleController">
8     Hello <input type="text" ng-model="user.name"/>
9     <hr/>
10    <h1>Hello {{user.name}}</h1>
11  </body>
12 </html>
```

---

Neste exemplo, após incluir o arquivo `simpleController.js`, usamos a propriedade **ng-controller** (linha 8) para dizer que, todo elemento abaixo do `<body>` será gerenciado pelo controller.

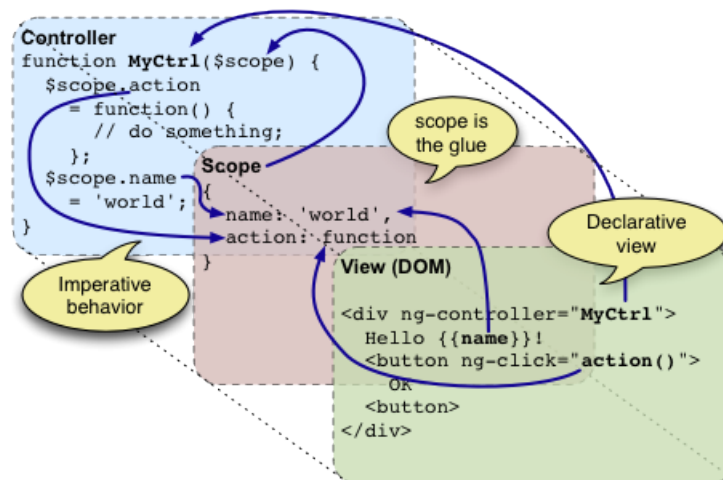
Na linha 9, adicionamos uma caixa de texto utilizando a propriedade **ng-model** e atribuímos o valor `user.name`. Neste caso, `user` é um objeto e `name` é uma propriedade a este objeto. Na linha 11 temos o uso do bind no objeto `user`.

**simpleController.js**

```
1 function simpleController($scope)
2 {
3     $scope.user = {name: "Daniel"}
4 }
```

O arquivo `simpleController.js` contém uma função chamada `simpleController`, que deve coincidir com o nome do controller. O nome do arquivo JavaScript não precisa ser o mesmo, e com isso pode-se ter vários controllers em somente um arquivo JavaScript.

Nesta função, temos o parâmetro `$scope` que é um “ponteiro” para a aplicação em si, ou seja, `$scope` significa a própria página html. Como o controller foi declarado no elemento `<body>`, `$scope` é usado para todo este elemento. Usa-se o `$scope` para criar uma conexão entre o model e a view, como foi feito no exemplo utilizando o objeto `user`. A imagem a seguir ilustra este processo.



Como o controller funciona

## Métodos do controller

O controller é usado também para manipular regras de negócio que podem ou não alterar os models. No exemplo a seguir, usamos o controller para definir uma variável e um método para incrementar em 1 esta variável.

### Simples exemplo de contador

---

```
1 <html ng-app>
2   <head>
3     <title>Hello Counter</title>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular.min.js"></script>
5     <script src="countController.js"></script>
6   </head>
7   <body ng-controller="countController">
8     <a href="#" ng-click="addOne()">Add 1</a>
9     <hr/>
10    <p>Counter value: {{counter}}</p>
11  </body>
12 </html>
```

---

Neste exemplo, usamos a propriedade **ng-click** que irá chamar um método dentro do seu respectivo controller.

### countController.js

---

```
1 function countController($scope)
2 {
3     $scope.counter = 0;
4
5     $scope.addOne = function(){
6         $scope.counter++;
7     }
8 }
```

---

No controller, criamos a variável **counter** e também o método **addOne**, que manipula a variável, de forma que o seu valor é refletido automaticamente na view (html) através do **dataBind**.

## Loops

Outra característica do AngularJS é utilizar templates para que se possa adicionar conteúdo dinâmico. Um loop é sempre realizado através da propriedade **ng-repeat** e obedece a uma variável que geralmente é um array de dados.

O exemplo a seguir ilustra este processo, utilizando a tag **li** para exibir uma lista qualquer.

### Usando loops

```
1 <html ng-app>
2   <head>
3     <title>Hello Counter</title>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular\
5 r.min.js"></script>
6     <script type="text/javascript">
7       function loopController($scope){
8         $scope.fruits = ['banana', 'apple', 'orange'];
9       }
10    </script>
11  </head>
12  <body ng-controller="loopController">
13    <ul>
14      <li ng-repeat="fruit in fruits">{{fruit}}</li>
15    </ul>
16  </body>
17 </html>
```

Como visto neste exemplo, pode-se adicionar JavaScript no arquivo HTML, mas não recomenda-se esta prática. É uma boa prática de programação utilizar o arquivo HTML apenas como a camada view, e o arquivo JavaScript como a camada de controller, de acordo com os padrões MVC.

## Formulários

Existem diversas características que um formulário contém, tais como validação, mensagens de erro, formato dos campos, entre outros. Neste caso, usamos o AngularJS de diferentes formas, e usamos vários parâmetros **ng** para controlar todo o processo.

O exemplo a seguir exibe apenas algumas dessas propriedades, para que você possa entender como o processo funciona, mas durante a obra iremos verificar todos os detalhes necessários para construir um formulário por completo.



### Formulário com validação

---

```
1 <html ng-app>
2 <head>
3   <title>Simple Form</title>
4   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular.mi\
5 n.js"></script>
6 </head>
7 <body>
8
9   <form name="myForm">
10     <span ng-show="myForm.$invalid">
11       Found erros in the form!
12     </span>
13     <input type="text" ng-model="name" name="Name" value="Your Name" required\
14 />
15     <button ng-disabled="myForm.$invalid">Save</button>
16   </form>
17
18 </body>
19 </html>
```

---

Neste formulário, usamos mais algumas propriedades, como por exemplo **ng-show** que irá exibir ou não a tag `<span>` contendo a mensagem de erro do formulário, e **ng-disabled** que desativa o botão de submissão do formulário.

O uso do `myForm.$invalid` é um recurso do AngularJS que define se um formulário está inválido ou não. Como usamos uma caixa de texto com a propriedade `required`, se o campo não estiver preenchido, o formulário ficará inválido.

## Rotas e Deep linking

O AngularJS possui um recurso chamado Deep Linking, que consiste em criar rotas na URI do documento HTML para manipular partes do código HTML de forma independente, podendo assim separar ainda mais as camadas da sua aplicação. No caso mais simples, suponha que exista uma lista de dados que são exibidos em uma tabela, e que ao clicar em um item desta lista, deseje-se exibir um formulário com os dados daquela linha.

No documento HTML criado, existem dois componentes bem definidos pela aplicação. O primeiro é a tabela com as informações, e o segundo, o formulário para edição dos dados.

O uso de DeepLinking usa Ajax para carregar templates de forma dinâmica, então é necessário que todo o exemplo seja testado em um servidor web.

Se organizarmos esta pequena aplicação em arquivos, teremos:

#### **index.html**

O arquivo principal da aplicação, que contém o código html, o **ng-app**, a inclusão do AngularJS, entre outras propriedades.

#### **app.js**

Contém todo o código javascript que define as regras de negócio da aplicação.

#### **list.html**

Contém a tabela que lista os dados.

#### **form.html**

Contém o formulário de edição e criação de um novo registro.

#### **index.html**

```
1 <html ng-app="app">
2 <head>
3     <title>DeepLinking Example</title>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular.min.js"></script>
5 </script>
6     <script src="app.js"></script>
7 </head>
8 <body>
9     <h1>DeepLink Example</h1>
10    <div ng-view></div>
11
12 </body>
13 </html>
```

Inicialmente criamos o arquivo `index.html`, que contém a chamada aos arquivos javascript da aplicação. Além dos arquivos javascript, também usamos a propriedade **ng-app**, que já aprendemos a usar em qualquer aplicação que use o framework, mas agora estamos criando um módulo, que é uma forma de organizar algumas configurações da página html.

Este módulo é criado através da definição de um nome para o **ng-app**, ficando desta forma: `ng-app="app"`. Assim, estamos criando um módulo chamado `App` que deve estar definido pela aplicação.

Como podemos ver, o arquivo `index.html` não tem nenhum conteúdo, apenas o cabeçalho e uma `div` que possui a propriedade **ng-view**. Esta propriedade configura o AngularJS para que toda a geração de código seja renderizada dentro desta tag.

Esta definição é realizada no arquivo `app.js`, cuja parte inicial está descrita a seguir.

#### app.js

```
1 $app = angular.module('app', [ ]);
2
3 $app.config(function($routeProvider){
4     $routeProvider.
5     when('/', {controller: listController, templateUrl: 'list.html'}).
6     when('/edit/:name', {controller: editController, templateUrl: 'form.html'}).
7     when('/new', {controller: newController, templateUrl: 'form.html'}).
8     otherwise({redirectTo: '/'});
9 });
10
11 $app.run(function($rootScope){
12     $rootScope.fruits = ["banana", "apple", "orange"];
13 });
```

Nesta primeira parte, usamos o método `angular.module` para criar um módulo, cujo o nome é `app`. O segundo parâmetro é, por enquanto, um array vazio, mas pode conter configurações extras que serão abordadas em um tópico mais específico.

Não omite o `array` vazio do segundo parâmetro da definição do módulo.

Após criar o módulo e atribuí-lo a variável `$app`, usamos o método `config` para configurar o módulo, neste caso estamos configurando uma funcionalidade chamada *Router*, que possui a função de carregar templates e controllers de acordo com uma URI, ou seja, um endereço repassado pelo navegador.

Na linha 5 temos a primeira configuração através do método `when`, que informa ao *Router* que, ao acessar a raiz do endereço web que o arquivo `index.html` está, deve ser carregado o controller `listController` e o template `list.html`.

Tanto o template quanto o controller serão carregados no elemento html que contém a propriedade **ng-view** do index.html.

Na linha 6 adicionamos mais uma rota, e agora configuramos que quando a URI for /edit/:name, o controller `editController` e o template `form.html` serão carregados. O atributo `:name` será uma variável que poderá ser obtida no controller.

Tanto na linha 6 quanto na linha 7 usamos o mesmo template `form.html` que contém um formulário para edição ou inserção de um registro.

Na linha 8, configuramos a rota padrão da URI, que é ativada quando nenhuma rota configurada é encontrada.

Na linha 11 usamos o método `$app.run` para configurar a variável `$scope` da aplicação, em um contexto global ao módulo. Neste método criamos a variável `fruits` que possui um contexto global à aplicação.

Continuando no arquivo `app.js`, temos:

`app.js`

```
14 function listController($scope){
15
16 }
17
18 function editController($scope,$location,$routeParams){
19     $scope.title = "Edit fruit";
20     $scope.fruit = $routeParams.name;
21
22     $scope.fruitIndex = $scope.fruits.indexOf($scope.fruit);
23
24     $scope.save = function(){
25         $scope.fruits[$scope.fruitIndex]=$scope.fruit;
26         $location.path('/');
27     }
28 }
29
30 function newController($scope,$location){
31     $scope.title = "New fruit";
32     $scope.fruit = "";
33
```

```
34     $scope.save = function(){
35         $scope.fruits.push($scope.fruit);
36         $location.path('/');
37     }
38 }
```

---

Criamos três controllers para a aplicação, sendo que o primeiro, `listController`, ainda não é utilizado, mas pode ser útil em um momento futuro.

Na linha 18 temos o controller `editController` que possui três parâmetros:

- **scope** É o escopo da aplicação que pode ser utilizada no template do controller criado.
- **location** Usada para realizar redirecionamentos entre as rotas
- **routeParams** São os parâmetros repassados pela URI

Na linha 19 preenchemos a variável `$scope.title`, para que o título do formulário mude, lembrando que o formulário é usado tanto para criar um novo registro quando editá-lo.

Na linha 20 pegamos como parâmetro o nome da fruta que foi repassada pela URI. Este valor é pego de acordo com o parâmetro `:name` criado pela rota, na linha 6.

Na linha 22 obtemos o índice do item que está para ser editado. Usamos isso para poder editar o item no método `save` criado logo a seguir.

Na linha 24 temos o método `save` que é usado para “salvar” o registro no array global. Em uma aplicação real estaríamos utilizando `ajax` para que o servidor persistisse o dado. Na linha 26 redirecionamos a aplicação e com isso outro template será carregado.

Na linha 30, criamos o controller `newController`, que é semelhante ao `editController` e possui o método `save` onde um novo registro é inserido no array `fruits`.

Vamos agora analisar o arquivo `list.html` que é um template e carregado diretamente pelo roteamento do módulo (`app.js`, linha 5).

#### list.html

---

```
1 <h2>Fruits ({{fruits.length}})</h2>
2 <ul>
3     <li ng-repeat="fruit in fruits"><a href="#/edit/{{fruit}}">{{fruit}}</a></li>
4 </ul>
5 <a href="#/new">New</a>
```

---

O template não necessita informar o seu controller, pois isso já foi feito pelo módulo do AngularJS (app.js, linha 5). Como a variável `fruits` possui um escopo global, ela pode ser usada pelo template e na linha 1, contamos quantos itens existem no array.

Na linha 3 iniciamos a repetição dos elementos que pertencem ao array `fruits` e incluímos na repetição um link para `#/edit/`. Esta é a forma com que o roteamento do AngularJS funciona, iniciando com `#` e repassando a URI logo a seguir. Na linha 5, criamos outro link, para incluir um novo registro. Novamente usamos a URI que será utilizada pelo roteamento do AngularJS.

O último arquivo deste pequeno exemplo é o formulário que irá editar ou inserir um novo registro.

form.html

---

```
1 <h2> {{title}} </h2>
2 <form name="myForm">
3     <input type="text" ng-model="fruit" name="fruit" required>
4     <button ng-click="save()" ng-disabled="myForm.$invalid">Save</button>
5 </form>
6 <a href="#/">Cancel</a>
```

---

Na linha 1 usamos o `{{title}}` para inserir um título que é criado pelo controller. O formulário possui apenas um campo cujo **ng-model** é `fruit` que será utilizado pelo controller (app.js, linhas 25 e 35). Neste formulário também utilizamos **ng-disabled** para que o botão seja ativado somente se houver algum texto digitado na caixa de texto. O botão `save` possui a propriedade **ng-click**, que irá chamar o método `save()` do controller.

## Documentação do AngularJS

Se você deseja saber mais sobre AngularJS, o site oficial provê uma ótima documentação e uma ótima API para que seja consultada sempre.

Os endereços estão listados a seguir, recomendo que salve-os e consulte-os sempre que for necessário.

<http://docs.angularjs.org/guide/overview>

<http://docs.angularjs.org/api>

# Conectando AngularJS ao servidor

Agora que conhecemos um pouco sobre o AngularJS, podemos entender como funciona a sua comunicação com o servidor. Assim como é feito com jQuery e até com javascript puro, a melhor forma de obter e enviar dados para o servidor é através de Ajax e o formato de dados para se usar nesta comunicação é JSON.

Existem diversas formas de conexão entre cliente (neste caso, AngularJS) e servidor, e nesta obra estaremos utilizando um conceito chamado RESTful, que é uma comunicação HTTP que segue um padrão bastante simples, utilizando cabeçalhos HTTP como POST, GET, PUT, DELETE.

Na forma mais simples de comunicação de dados, onde temos um objeto e as ações de criar, editar, listar e deletar objetos, resumimos o padrão RESTful às seguintes ações:

Método	http://site.com/produtos
GET	Listar todos os produtos
POST	Editar uma lista de produtos
PUT	Criar um novo produto na lista de produtos
DELETE	Excluir uma lista de produtos

Método	http://site.com/produto/1
GET	Obter o produto cujo id é 1
POST	Em teoria nao possui funcionalidade
PUT	Edita ou cria um novo produto
DELETE	Exclui um produto cujo o id é 1

As tabelas acima são uma sugestão que pode ser seguido ou não. Mesmo se simplificarmos os métodos acima, o mínimo que podemos estabelecer é que métodos GET não alterem dados, pois um método GET pode ser facilmente acessado através do navegador.

O AngularJS fornece duas formas distintas de trabalhar com estas conexões. A primeira delas, e mais simples, é através do serviço `$http`, que pode ser injetado em um controller. A segunda forma é através do serviço `$resource` que é uma abstração RESTful, funcionando como um *data source*.

## Uso do \$http

O uso do \$http não deve ser ignorado, mesmo que o \$resource seja mais poderoso. Em aplicações simples, ou quando deseja obter dados de uma forma rápida, deve-se utilizar \$http.

\$http é uma implementação ajax através do XMLHttpRequest utilizando JSONP. Iremos sempre usar JSON para troca de dados entre cliente e servidor.

A forma mais simples de uso do \$http está descrito no exemplo a seguir:

### \$http na sua forma mais simples

```
1 $http({method: 'GET', url: '/someUrl'}).success(function(data){  
2  
3 });
```

O método get pode ser generalizado para:

### \$http.get

```
1 $http.get('/someUrl').success(function(data){  
2  
3 });
```

Assim como existe o get, existem os outros também, conforme a lista a seguir:

- \$http.get
- \$http.head
- \$http.post
- \$http.put
- \$http.delete
- \$http.jsonp

Para todos estes métodos, o AngularJS configura automaticamente o cabeçalho da requisição HTTP. Por exemplo, em uma requisição POST os cabeçalhos preenchidos são:



- Accept: application/json, text/plain, \* / \*
- X-Requested-With: XMLHttpRequest
- Content-Type: application/json

Além dos cabeçalhos, o AngularJS também serializa o objeto JSON que é repassado entre as requisições. Se um objeto é enviado para o servidor, ele é convertido para JSON. Se uma *string* JSON retorna do servidor, ela é convertida em objeto utilizando um parser JSON.

## Exemplo com \$http

Vamos criar um exemplo utilizando o serviço \$http, para obter uma lista de dados e preencher uma tabela. Inicialmente, criamos um arquivo simples no servidor que, em teoria, retornaria com informações de uma lista de objetos em JSON. Como ainda não estamos trabalhando com banco de dados, esta lista será criada diretamente no arquivo, em formato json.

listFruits.html

---

```
1 { "fruits":  
2   [  
3     {  
4       "id": 1,  
5       "name": "Apple",  
6       "color": "Red"  
7     },  
8     {  
9       "id": 2,  
10      "name": "Banana",  
11      "color": "Yellow"  
12    },  
13    {  
14      "id": 3,  
15      "name": "watermelon",  
16      "color": "Green"  
17    },  
18    {  
19      "id": 4,  
20      "name": "Orange",  
21      "color": "Orange"  
22    }  
23   ]  
24 }
```

---

Neste exemplo, estamos inserindo o arquivo `listFruits.html` na pasta `c:\wamp\www\http-example\` e podemos acessá-lo através da url `http://localhost/http-example/listFruits.html`. O próximo passo é criar o arquivo `index.html`, que contém a camada view do exemplo.

#### index.html

---

```
1 <html ng-app>
2   <head>
3     <script src="http://code.angularjs.org/1.1.4/angular.min.js"></script>
4     <script src="app.js"></script>
5   </head>
6   <body>
7     <div ng-controller="appController">
8       <button ng-click="getData()">Get Data</button>
9       <h2 ng-show="fruits.length>0">Fruits</h2>
10      <ul>
11        <li ng-repeat="fruit in fruits" >
12          {{fruit.id}} - {{fruit.name}} ({{fruit.color}})
13        </li>
14      </ul>
15    </div>
16  </body>
17 </html>
```

---

No arquivo `index.html` não temos nenhuma novidade, pois a regra de negócio está no seu controller, onde realizamos o Ajax.

#### app.js

---

```
1 function appController($scope,$http)
2 {
3     $scope.fruits = Array();
4
5     $scope.getData = function(){
6         $http.get("listFruits.html").success(function(data){
7             $scope.fruits = data.fruits;
8             console.log($scope.fruits);
9         }).error(function(data){
10             alert("Error...");
11             console.log(data);
12         });
13     }
```

---

```
13     }  
14 }
```

---

No controller da aplicação, criamos o método `getData`, que é executado quando clicamos no botão “GetData” da view (`index.html`). Neste método, usamos a variável `$http` para as requisições Ajax. Repare que ela é repassada pelo parâmetro do controller, após o `$scope`. Neste caso, o AngularJS encarrega-se de injetar o serviço `http` nesta variável.

Na linha 6 temos o método `$http.get` onde estamos realizando uma requisição Ajax acessando o arquivo `listFruits.html`, que contém a resposta Json. Neste método, podemos concatenar outro método chamado `success`, que é executado se a requisição HTTP GET for realizada com sucesso. Neste caso, a resposta do servidor estará armazenada na variável `data`, e poderemos acessar a variável `data.fruits` que contém o array de objetos que serão usados no loop da view.

Na linha 8 temos o uso do `console.log` que pode ser usado em conjunto com o Firbug (Firefox) ou com o Google Chrome, para verificar resultados no console da janela “developer tools”. Pode-se usar o *developer tools* para analisar as chamadas ajax também, geralmente na aba *Network*.

Com este simples exemplo conseguimos mostrar como é fácil realizar uma requisição Ajax para obter dados do servidor. Pode-se usar o serviço `$http` para toda a sua aplicação, mas quando estamos utilizando RESTfull, existe outro serviço que torna o acesso ao servidor mais abstrato, chamado de `resource`, no qual veremos a seguir.

## Uso do \$resource

Aprendemos a realizar chamadas Ajax através do `$http` e caso haja necessidade, podemos abstrair ainda mais a forma como o AngularJS acessa o servidor.

Neste contexto entra o `$resource` que estabelece um padrão de comunicação RESTfull entre a aplicação e o servidor.

Para que possamos usar esta biblioteca é preciso adicionar o arquivo `angular-resource.js` no documento HTML. Ou seja, além da biblioteca padrão também incluímos a biblioteca `resource`, conforme o exemplo a seguir.

### Adicionando a biblioteca angular-resource

---

```
1 <html ng-app>
2   <head>
3     <title>Lista de compras</title>
4     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular\
5 r.min.js"></script>
6     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular\
7 r-resource.min.js"></script>
8   </head>
9   <body>
10  </body>
11 </html>
```

---

## Exemplo simples com \$resource

Com a biblioteca devidamente instalada, devemos carregá-la, através de um parâmetro na criação do módulo da aplicação. Para acompanhar este processo, vamos criar um exemplo simples, utilizando operações CRUD com o \$resource.

### index.html

---

```
1 <html ng-app="app">
2 <head>
3   <title>Lista de compras</title>
4   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular.min.j\
5 s"></script>
6   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.1.4/angular-resou\
7 rce.min.js"></script>
8   <script src="app.js"></script>
9 </head>
10 <body ng-controller="phoneController">
11   <input type="text" ng-model="idPhone" value="1"/>
12   <button ng-click="getPhoneById()">GetPhone By Id</button>
13   <hr/>
14   <button ng-click="getPhones()">GetPhones</button>
15   <hr/>
16   <button ng-click="savePhone()">Save Phone</button>
17   <hr/>
```

```
18         <button ng-click="deletePhone()">Delete Phone</button>
19     </body>
20 </html>
```

---

Este código não apresenta nenhuma novidade. Estamos criando o *module* do AngularJS chamado *app* e criamos um formulário com alguns botões no controller *PhoneController*.

#### app.js

---

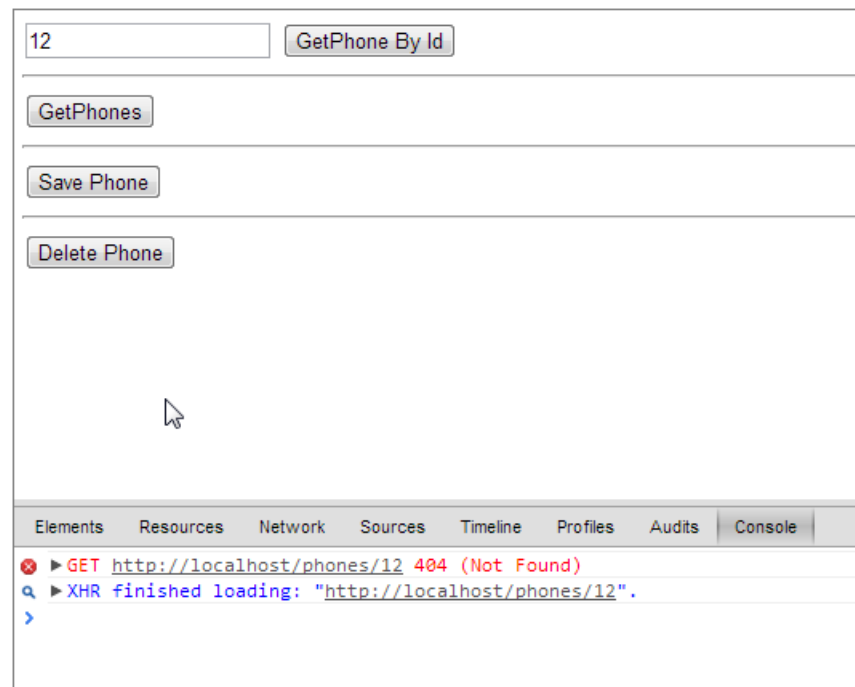
```
1 $app = angular.module('app', ['ngResource']);
2
3 function phoneController($scope,$resource){
4
5     Phone = $resource("/phones/:phoneId");
6
7     $scope.getPhoneById = function(){
8         Phone.get({phoneId:$scope.idPhone},function(data){
9             $scope.phone=data;
10         });
11     }
12
13     $scope.getPhones = function(){
14         Phone.query(function (data){
15             scope.phones = data;
16         });
17     }
18
19
20     $scope.savePhone = function(){
21         p = new Phone();
22         p.number="1111-2222"
23         p.$save();
24     }
25
26     $scope.deletePhone = function(){
27         Phone.delete({phoneId:10});
28     }
29
30 }
```

---

A primeira novidade deste código javascript está na linha 1, onde criamos o módulo *app* e já adicionamos o serviço *ngResource*.

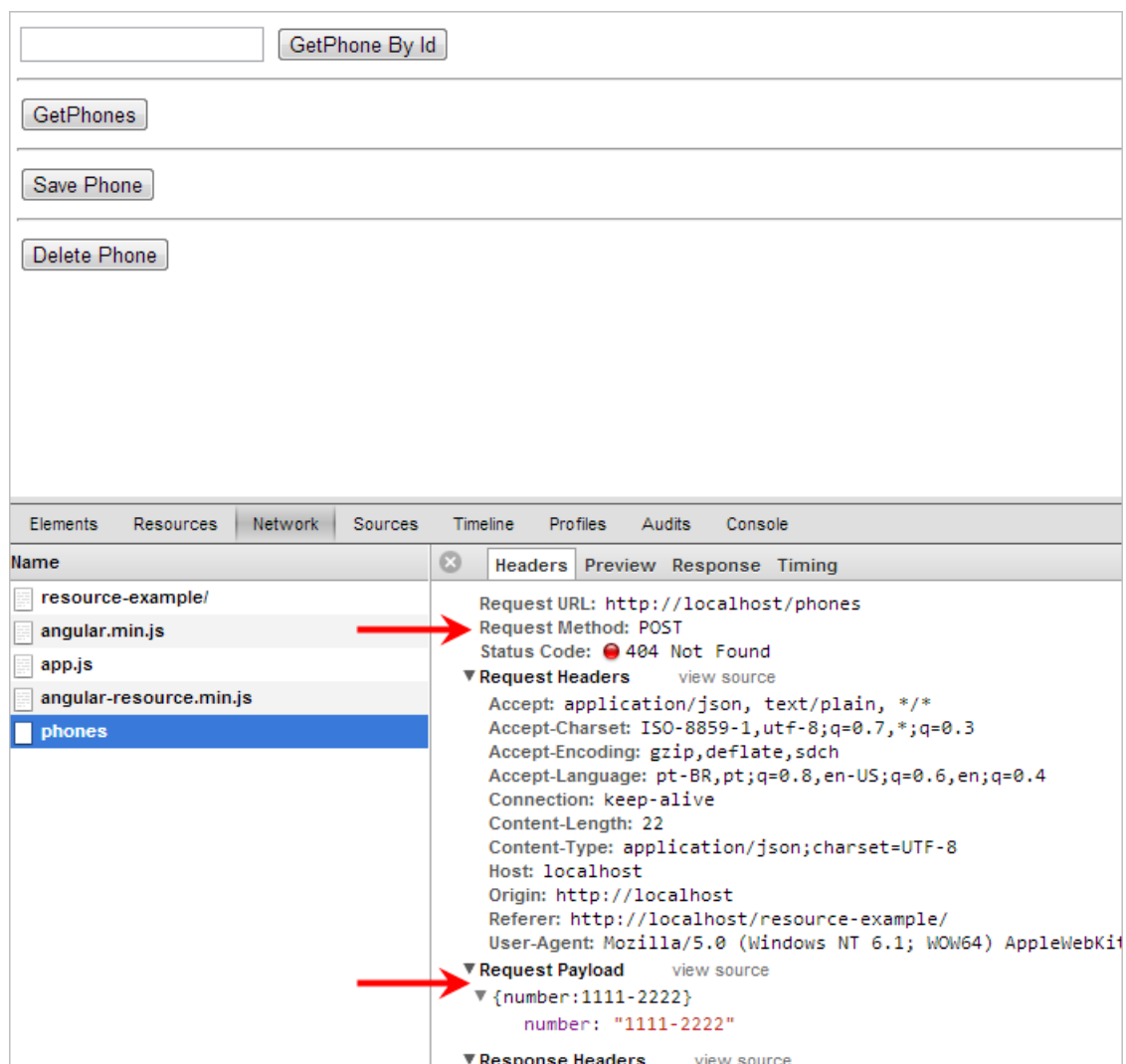
na definição do `phoneController` adicionamos a variável `$resource` como parâmetro da função, que será injetada pelo AngularJS. Usamos `$resource` para definir a criação da variável `Phone`. Esta variável é criada e configurada como um `resource`, sendo o primeiro parâmetro a url de acesso ao servidor.

Na linha 7 temos o método `getPhoneById` que é chamado pela view e usa `Phone.get` para realizar uma chamada Ajax ao servidor, conforme a figura a seguir.



Uso do Ajax pelo resources

O mesmo acontece com os outros métodos, e como podemos ver na figura a seguir, quando realizamos um save é realizado um POST no servidor, segundo as especificações do RESTfull.



Uso do Ajax pelo resources

Nesta figura, temos em detalhe o POST e a variável `number` sendo enviada como json (e não na URL do POST).

Lembre-se que, para testar o `resources`, você precisa utilizar um servidor web.

Este exemplo está em sua forma mais simples e possivelmente o `resource` não será criado em um `controller` da aplicação. Ainda nesta obra iremos exemplificar um processo um pouco mais complexo, porém utilizando a forma correta para criação de `resources` na aplicação.

## **Parte 2 - Sistema de vendas**



# Preparando o sistema

Neste capítulo iniciaremos a criação do sistema de vendas, que é baseado nas tabelas do banco de dados *northwind*. Este desenvolvimento tem como objetivo mostrar as várias práticas que podem ser empregadas no uso do javascript para a criação de sistemas web, então iremos separar cada técnica nos capítulos posteriores, de forma a facilitar o entendimento de cada método.

Não existe a técnica perfeita, mas sim a que mais se encaixa na sua realidade.

## Técnicas comuns para o uso dos arquivos javascript

O angular JS é uma biblioteca javascript e para que possamos trabalhar com ela nos mais diferenciados projetos precisamos inicialmente conhecer algumas técnicas que lidam principalmente com o carregamento de tais bibliotecas.

Como já foi dito, não existe o caminho perfeito e, dependendo do seu projeto, uma técnica será melhor que a outra. A abordagem que faremos a seguir é focada principalmente em como o javascript é carregado. Nosso principal “problema” é saber como carregar o javascript de forma eficiente, sem que prejudique a experiência com o usuário nem a manutenibilidade do sistema.

## Como carregar javascript?

Vimos nos exemplos até agora que um javascript é carregado da seguinte forma:

### Incluir todas os arquivos de uma vez

A forma mais fácil de incluir os arquivos javascript é fazer como estamos fazendo até agora, incluindo os arquivos na página `index.html`, de forma que todos os arquivos sejam carregados na inicialização da página. Isso será feito nas duas primeiras funcionalidades do sistema, de cadastro de clientes e de funcionários. Esta forma é recomendada se você possui um sistema muito pequeno, como no máximo até 5 tabelas. Separe cada tabela em arquivos distintos e adicione-os logo após incluir as bibliotecas jQuery e AngularJS.

### Incluir javascript dinamicamente

Existem prós e contras nesta técnica, mas ela geralmente não é recomendada pela maioria dos mentores em javascript. Por exemplo, o time que mantém o AngularJS não recomenda utilizar esta técnica, mas isso não significa que não iremos apresentá-la, muito menos que você possa utilizá-la. A principal vantagem desta técnica é que os arquivos javascript serão carregados na medida que forem utilizados, sem o risco de haver javascript em excesso no carregamento inicial da página. A desvantagem é garantir que esse arquivo javascript foi carregado corretamente, pois caso isso não aconteça você terá problemas no seu código.

### Incluir Javascript minificado

Através de algumas bibliotecas pode-se juntar todos os arquivos javascript em um só, e minificar ele, deixando praticamente tudo em uma linha, assim como é o arquivo `jquery.min.js` por exemplo. Esta técnica tem a vantagem de diminuir muito o carregamento da página, pois ao invés de carregar vários arquivos, carrega-se apenas um, e da forma mais otimizada possível.

### Usar o servidor para “montar” a página

Esta é uma técnica muito conhecida e usa o processamento do servidor para montar a página de forma dinâmica. Se utilizarmos a linguagem php, iremos criar arquivos separados, como por exemplo `cabecalho.php`, `menu.php`, `rodape.php` e assim montar a página html em blocos. Somente o javascript necessário será carregado. Esta técnica é considerada uma boa prática na programação de sistemas, com o único detalhe de necessitar que você domine a linguagem de servidor que irá utilizar.

## Preparando servidor

Independente da técnica empregada para carregamento do Javascript, você precisará de uma linguagem de servidor para carregar e persistir dados do banco de dados. Para isso, iremos utilizar a linguagem PHP e RESTful, como já foi comentado.

Inicialmente temos que preparar as tabelas no banco de dados. A priori, todo sistema começa na criação de tabelas, e a partir delas começos a codificar o sistema.

Existem diversas técnicas de desenvolvimento de software e, claro que, algumas abordam a criação do desenho das telas do sistema antes mesmo da criação das tabelas ou definição de tecnologias. Como não é o foco desta obra, estaremos partindo para um conjunto de tabelas prontas e usando a linguagem SQL para obter e persistir dados, de forma que possamos nos concentrar.

## Banco de dados

Como o Wamp Server está instalado, você pode usar o PhpMyAdmin para carregar as tabelas e dados que são fornecidos nesta obra. Existe um script SQL no código fonte da obra, chamado de `northwind.sql`<sup>3</sup> que você poderá importar para o banco de dados. Por conveniência chamaremos este banco de dados de `northwind`.

---

<sup>3</sup><https://github.com/danielps/livro-angular/blob/master/northSales/bancoDados/northwind.sql>

## Preparando o backend

O backend é formado pelo servidor apache, com a linguagem PHP. Utilizaremos RESTful e para isso, adicionaremos ao nosso projeto PHP o Slim Framework, uma ferramenta muito fácil para que possamos implementar as chamadas GET, POST, PUT e DELETE de forma rápida e simples.

Crie a pasta `sales-server` na pasta `c:\wamp\www` e crie a adicione a biblioteca Slim, que pode ser obtida no site oficial do Slim Framework<sup>4</sup>. Ao acessar o site do framework, clique no botão `Install Now` e depois clique no link `Download Stable Release`. Ao realizar o download, descompacte e copie a pasta Slim e o arquivo `.htaccess` para `c:\wamp\www\sales-server`.

A pasta Slim contém todo o framework e nesta obra não iremos abordar o que ela contém, mas é importante comentar sobre o arquivo `.htaccess`, já que ele é um “curinga” que usamos para criar url amigáveis no ambiente RESTful.

### Arquivo .htaccess

---

```
1 RewriteEngine On
2 RewriteCond %{REQUEST_FILENAME} !-f
3 RewriteRule ^ index.php [QSA,L]
```

---

O código deste arquivo é uma configuração para o servidor apache. Na linha 1 estamos ativando o comando `RewriteEngine`, que iniciar o processo de reescrita da URL. Na linha dois criamos uma condição, que caso seja satisfeita irá ativar a 3 linha. Esta condição abrange qualquer url exceto arquivos (`!=f`). Na linha 3, temos uma regra dizendo que deverá ocorrer um redirecionamento para o arquivo `index.php` e ainda algumas flags como `QSA`, dizendo que qualquer querystring deve ser mantida (por exemplo, `?chave=valor`) e `L`, finalizando a regra.

Você não precisa ser um expert em Rewrite para aprender AngularJS, mesmo pq este arquivo de regras vem pronto do Slim Framework, mas caso deseje se aprofundar no assunto pode consultar este link<sup>5</sup>



Para usar o `RewriteEngine`, é preciso referenciar uma biblioteca no Apache chamada `rewrite_module`. Com o Wamp Server, você consegue instalar a biblioteca de forma fácil, acessando o menu do Wamp Server na bandeja do Windows e indo no menu *Apache >> Apache Modules >> Rewrite Module*. Se não estiver utilizando o Wamp Server, você precisa descomentar a linha `LoadModule rewrite_module` do arquivo de configuração `httpd.conf` do apache.

---

<sup>4</sup><http://www.slimframework.com>

<sup>5</sup>[http://httpd.apache.org/docs/2.0/mod/mod\\_rewrite.html](http://httpd.apache.org/docs/2.0/mod/mod_rewrite.html)

## Arquivos iniciais do backend

O projeto no servidor precisa de alguns arquivos iniciais para que possamos trabalhar com o Slim Framework e persistir dados. Estes arquivos estão descritos a seguir:

### index.php

É o arquivo principal da aplicação, responsável em gerenciar todo o processo de requisição e resposta do servidor. Ele será responsável em instanciar o Slim Framework, o acesso ao banco de dados, chamar o método que irá persistir dados no banco e formalizar a resposta Json de volta ao cliente.

### DB.php

Este é um arquivo pronto que ajuda a persistir dados no banco MySQL. Foi criado com o intuito de servir a vários sistemas e você pode utilizá-lo em seus projetos. Veja que nosso interesse nesta obra é aprender AngularJS, então a forma como acessamos dados no servidor será a mais simples possível, sem usar camadas de persistência como o Doctrine ou frameworks mais complexos como o Laravel. Para acesso ao banco, usaremos PDO que é uma biblioteca nativa do PHP e de fácil uso.

### Config.php

Contém a configuração de acesso ao banco de dados. Pode conter mais configurações ao longo que o sistema é desenvolvido.

Além destes três arquivos iniciais, criaremos outros que serão adicionados ao arquivo index.php, de forma a possuir um fácil acesso pelo framework Slim.

## Arquivo DB.php

Vamos analisar passo a passo o arquivo DB.php, que tem a finalidade de persistir dados no MySQL através da biblioteca nativa PDO.

### Arquivo DB.php

---

```
1 <?php
2 /**
3  * Classe para conexao com o banco de dados MySQL,
4  * via acesso nativo do PHP/PDO.
5  * É necessário ter definido as seguintes constantes: DB_NAME, DB_HOST, DB_USER, \
6  DB_PASSWORD
7  */
8 class DB {
```

```
9
10  /**
11   * Instância singleton
12   * @var DB
13   */
14  private static $instance;
15
16  /**
17   * Conexão com o banco de dados
18   * @var PDO
19   */
20  private static $connection;
```

---

Inicialmente criamos a classe DB que, como o comentário já diz, espera que constantes globais como DB\_NAME já estejam criadas. Esta particularidade será vista logo adiante.

É criada a variável estática `$instance`, e se você lembrou do padrão de projeto *singleton*, é exatamente isso que estaremos fazendo. Caso não conheça o padrão, não tem problema, você não precisará saber disso para usar a classe. Também é criada a variável `$connection`, que representa a conexão com o banco de dados.

#### Arquivo DB.php

---

```
21  /**
22   * Construtor privado da classe singleton
23   */
24  private function __construct() {
25      self::$connection = new PDO("mysql:dbname=" . DB_NAME . ";host=" . DB_HOST, D\
26  B_USER, DB_PASSWORD, array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8"));
27      self::$connection->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
28      self::$connection->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_OBJ);
29  }
```

---

Criamos o construtor da classe, que pela definição do padrão *singleton* deve ser privada. No construtor, criamos a instância com o banco de dados, através da classe PDO. Aqui, na linha 25, usamos as constantes globais que devem estar definidas na aplicação. Na linha 28 definimos como deverá ser o tratamento de erros caso ocorra algum erro no SQL a ser executado, neste caso qualquer erro será gerado uma exceção de código PHP. Na linha 30 definimos o modo em como os dados serão agrupados pelos comandos SELECT. Neste caso, os dados serão agrupados em forma de objetos.

Arquivo DB.php

---

```
30 /**
31  * Obtém a instancia da classe DB
32  * @return type
33  */
34 public static function getInstance() {
35
36     if (empty(self::$instance)) {
37         self::$instance = new DB();
38     }
39     return self::$instance;
40 }
41
42 /**
43  * Retorna a conexão PDO com o banco de dados
44  * @return PDO
45  */
46 public static function getConn() {
47     self::getInstance();
48     return self::$connection;
49 }
```

---

O método `getInstance` é usado para se obter a instância da classe `DB`, isso porque no padrão *singleton* podemos ter apenas uma instância da classe. Este código é uma especie de receita de bolo do padrão *singleton* para classes PHP. O método `getConn` obtém a conexão com o banco de dados.

Arquivo DB.php 2

---

```
51 /**
52  * Prepara a SQL para ser executada posteriormente
53  * @param String $sql
54  * @return PDOStatement stmt
55  */
56 public static function prepare($sql) {
57     return self::getConn()->prepare($sql);
58 }
59
60 /**
61  * Retorna o id da última consulta INSERT
```

```
62  * @return int
63  */
64  public static function lastInsertId() {
65      return self::getConn()->lastInsertId();
66  }
```

---

O método `prepare` é usado para preparar a SQL pelo PDO, funcionando da mesma forma que a classe nativa. Se você conhece a biblioteca PDO certamente conhece este método. O método `lastInsertId` é auto explicativo, retornando o id da chave primária da última consulta `INSERT` da conexão com o banco de dados.

#### Arquivo DB.php 3

---

```
68  /**
69   * Inicia uma transação
70   * @return bool
71   */
72  public static function beginTransaction(){
73      return self::getConn()->beginTransaction();
74  }
75
76  /**
77   * Comita uma transação
78   * @return bool
79   */
80  public static function commit(){
81      return self::getConn()->commit();
82  }
83
84  /**
85   * Realiza um rollback na transação
86   * @return bool
87   */
88  public static function rollBack(){
89      return self::getConn()->rollBack();
90  }
```

---

Os métodos de controle de transação também estão presentes na classe, sendo que, como desenvolvedor, você já deva conhecer as operações de *commit* e *rollback* das consultas SQL.

#### Arquivo DB.php 4

---

```
93  /**
94      * Formata uma data para o MySql (05/12/2012 para 2012-12-05)
95      * @param type $date
96      * @return type
97      */
98  public static function dateToMySql($date)
99  {
100      return implode("-",array_reverse(explode("/", $date)));
101  }
102
103  /**
104      * Formata uma data do MySql (2012-12-05 para 05/12/2012)
105      * @param type $date
106      * @return type
107      */
108  public static function dateFromMySql($date)
109  {
110      return implode("/",array_reverse(explode("-", $date)));
111  }
```

---

Para finalizar a classe, existem dois métodos relativos a campos no formato data, para que possam ser convertidos de forma correta do banco para o sistema e vice-versa.

Você pode conferir esta implementação no seguinte endereço [do github](https://github.com/danielps/livro-angular/commit/edffadb02e17c2fc7cfe1225eb7d5cb0b4b66bb8)<sup>6</sup>

## Arquivo config.php

Este arquivo possui as configurações do sistema, inicialmente possuindo as informações de acesso ao banco de dados.

---

<sup>6</sup><https://github.com/danielps/livro-angular/commit/edffadb02e17c2fc7cfe1225eb7d5cb0b4b66bb8>



### Arquivo config.php

---

```
1 <?php
2
3 //Configuracao do banco de dados
4 define("DB_HOST","localhost");
5 define("DB_NAME","northwind");
6 define("DB_USER","root");
7 define("DB_PASSWORD","");
```

---

O acesso ao banco de dados do Wamp Server é realizado através do login root e a senha em branco.

## Arquivo index.php

Este é o arquivo principal do sistema, responsável em tratar todas as particularidades do mesmo.

### Arquivo index.php

---

```
1 <?php
2 session_start();
3
4 require 'config.php';
5 require 'DB.php';
6
7 require 'Slim/Slim.php';
8 \Slim\Slim::registerAutoloader();
9 $app = new \Slim\Slim(array(
10     'debug' => false
11 ));
12
13 $app->contentType("application/json");
14
15 $app->error(function (Exception $e = null) use ($app) {
16     echo '{"error":{"text":"' . $e->getMessage() . '"}}';
17 });
18
19 function formatJson($obj)
20 {
21     echo json_encode($obj);
```

```
22 }
23
24 //Includes
25 include("customer.php");
26
27
28 $app->run();
```

---

Na linha 2 temos o início da sessão do PHP, porque geralmente sempre usamos sessão para definir algum parâmetro, como a pessoa que está logada no sistema. Nas linhas 4 e 5 incluímos os arquivos `config` e `db` e, logo após na linha 7 iniciamos o uso do Slim Framework. Primeiro incluímos o arquivo `Slim.php`, e depois na linha 9 usamos o método `registerAutoloader()` para registrar as bibliotecas internas do Slim.

Na linha 13 definimos o `contentType` da resposta http à requisição, no caso será via JSON, que é um formato universal de dados.

Na linha 15 definimos um método do Slim Framework chamado `error` que será chamado caso ocorra algum erro no sistema. Desta forma, quando algum erro acontecer, ao invés de dar um erro na tela, será formatada uma mensagem em JSON com a descrição do erro.

O método `formatJson` na linha 19 é usado como uma forma de serializar o objeto em questão no formato JSON, que será retornado de volta ao cliente que fez a requisição HTTP do sistema.

Para finalizar, a partir da linha 25 incluímos as bibliotecas responsáveis em persistir os dados no banco, de forma um pouco mais organizada para que tudo não fique no mesmo arquivo. Esta é uma solução simples para que possamos usar o servidor de forma rápida e podermos nos concentrar no AngularJS.

## Arquivo customer.php

Este é o primeiro arquivo específico que tem a responsabilidade de trabalhar com os dados do cliente (tabela `customers`). Em um nível mais básico, temos as seguintes operações:

### Arquivo customers.php

---

```
1 <?php
2
3 $app->get("/customers",function (){
4
5     $sql = "SELECT CustomerID,ContactName,Phone FROM customers";
6     $stmt = DB::prepare($sql);
7     $stmt->execute();
```

```
8         formatJson($stmt->fetchAll());
9     });
10
11     $app->get("/customer/:id",function ($id){
12
13         $sql = "SELECT CustomerID,ContactName,Phone FROM customers WHERE CustomerID='$id\
14     '";
15         $stmt = DB::prepare($sql);
16         $stmt->execute();
17         formatJson($stmt->fetch());
18     });
19
20     $app->post("/customer/:id",function ($id){
21
22         $data =json_decode(\Slim\Slim::getInstance()->request()->getBody());
23
24         if ($data->isUpdate)
25         {
26             $sql = "UPDATE customers SET ContactName=?,Phone=? WHERE CustomerID=?";
27             $stmt = DB::prepare($sql);
28             $stmt->execute(array(
29                 $data->ContactName,
30                 $data->Phone,
31                 $data->CustomerID
32             )
33         );
34         }
35         else
36         {
37             $sql = "INSERT INTO customers (CustomerID,ContactName,Phone) VALUES (?, ?, ?)";
38             $stmt = DB::prepare($sql);
39             $stmt->execute(array(
40                 $data->CustomerID,
41                 $data->ContactName,
42                 $data->Phone
43             )
44         );
45
46         }
47
48         formatJson($data);
49     });
```

```
50 });  
51  
52 $app->delete("/customer/:id",function ($id){  
53     $sql = "DELETE FROM customers WHERE CustomerID=?";  
54     $stmt = DB::prepare($sql);  
55     $stmt->execute(array($id));  
56     formatJson(true);  
57 });
```

---

Este arquivo usa o Slim Framework para que possamos implementar o RESTful. Por exemplo, na linha 3 dizemos que, se houver uma requisição GET para “/customers”, o método na linha 3 deverá ser executado. Este método, que começa na linha 5, faz um SELECT no banco de dados e usa a classe DB para obter esta informação do banco de dados. Após obter a informação, usamos o método `formatJson` para formatar estes dados de forma correta.

Na linha 11 configuramos o endereço “/customer/:id”, que fica responsável em obter um cliente específico. Na linha 19 configuramos o método POST da URL “/customer/:id” responsável em persistir um cliente no banco, podendo ser tanto `insert` quanto `update`.

O último método é o delete, e está na linha 51, onde realizamos o delete do registro no banco de dados.

Veja que todas as operações são simples, e não estamos adicionando complexidade ao sistema no servidor para que possamos dar ênfase ao AngularJS. A medida que criarmos as outras funcionalidades no frontend iremos voltar a programação no servidor e escrever os outros arquivos.

Após criar o código que gerencia a aplicação, poderemos então criar as telas do sistema. Começaremos com o cadastro de clientes, no qual será visto no próximo capítulo.

# Preparando o cliente

Nesta primeira parte, iremos criar uma tela para cadastrar clientes e explicar os principais conceitos que envolvem o AngularJS. Não estaremos, a princípio, envolvendo questões como segurança, paginação, validação de dados, entre outros, para que não haja uma sobrecarga, pelo menos no início do sistema.

A medida em que avançarmos no sistema, iremos explicar diversos pontos que farão com que cada tela tenha uma implementação nova, de forma que possa ser assimilado naturalmente.

## Estrutura de pastas

Para que possamos criar a aplicação inicial no cliente, precisamos criar uma estrutura de arquivos e pastas (tanto javascript, quanto html, imagens e css) para que toda a aplicação funcione.

Uma estrutura básica é apresentada a seguir.

### **arquivo index.html**

Será o ponto de entrada do sistema, onde criamos a estrutura html inicial da aplicação e incluímos todos os arquivos necessários para que o AngularJS funcione.

### **pasta css**

Esta pasta contém todo o css necessário para estilizar a aplicação. Nossa aplicação usará a biblioteca Twitter Bootstrap e a interface Metro, que são facilmente configuráveis pelo css.

### **pasta img**

Contém as imagens do sistema

### **pasta js**

Contém todo o javascript do sistema, tanto relativo as bibliotecas prontas, como o jQuery e AngularJS, quanto o javascript das telas que iremos programar

### **pasta js/lib**

Contém as bibliotecas prontas do sistema, tais como o jQuery, twitter bootstrap, AngularJS e outros.

### **arquivo js/app.js**

Contém as configurações iniciais da aplicação, a criação da instância do AngularJS e diversas funcionalidades que serão adicionadas na medida em que o sistema cresça.

Para que você possa criar esta estrutura com mais facilidade, acesse o seguinte link <sup>7</sup> e faça o download da aplicação inicial pronta, clicando no botão zip.

Após fazer o download, copie a pasta `base-app-client` para `c:\wamp\www` e renomeie ela para `sales-client-1`, pois será a primeira versão de aplicação que iremos criar, conforme comentado no capítulo anterior.

Nesta primeira versão, iremos nos preocupar apenas com o AngularJS, então incluiremos todos os arquivos JS no arquivo `index.html`, sem a preocupação com performance ou segurança. Após termos uma boa base do AngularJS, podemos partir para outras soluções mais robustas.

## Arquivo `index.html`

Como já comentamos, o arquivo `index.html` é o ponto inicial da aplicação. Ele contém a toda a estrutura necessária para que possamos criar o sistema, por isso é importante que possamos analisar cada parte do arquivo em separado, de forma a compreender o que cada código representa. Inicialmente, temos:

`index.html`

---

```
1 <!DOCTYPE html>
2 <html ng-app="app">
3 <head>
4   <meta charset="utf-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Sales App</title>
7   <!-- Styles -->
8   <link href="css/bootstrap.min.css" rel="stylesheet">
9   <link href="css/bootstrap-responsive.min.css" rel="stylesheet">
10  <link href="css/font-awesome.min.css" rel="stylesheet">
11  <link href="css/bootswatch.css" rel="stylesheet">
12 </head>
```

---

Inicialmente criamos a tag `<html>` e já definimos a tag `ng-app` como `app` (linha 2), onde dizemos a aplicação AngularJS a se carregada. Isso significa que no código javascript, em alguma parte do documento html, teremos que configurar a aplicação com o nome `app`. Isso é realizado quando incluímos as bibliotecas JavaScript.

No cabeçalho do documento html, temos a definição do charset (sempre use UTF-8), a definição do viewport que irá reconfigurar a página para um melhor enquadramento caso esteja em um dispositivo mobile, e a inclusão de bibliotecas css para estilizar a aplicação. Estes arquivos css já

---

<sup>7</sup><https://github.com/danielps/livro-angular/tree/master/base-app-client>

vem prontos, basta incluí-los. Após incluir o cabeçalho, iremos incluir o corpo do documento html, veja:

index.html

```
13 <body>
14
15 <div class="navbar navbar-fixed-top navbar-inverse">
16   <div class="navbar-inner">
17     <div class="container">
18       <a class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">
19         <span class="icon-bar"></span>
20         <span class="icon-bar"></span>
21         <span class="icon-bar"></span>
22       </a>
23       <a class="brand" href="#">Sales - 1</a>
24       <div class="nav-collapse">
25         <ul class="nav">
26
27           <li><a href="#">Home</a></li>
28           <li><a href="#/clientes">Clientes</a></li>
29           <li><a href="#/funcionarios">Funcionários</a></li>
30           <li><a href="#/transportadoras">Transportadoras</a></li>
31
32         </ul>
33       </div>
34       <form class="navbar-form pull-right">
35         
36       </form>
37     </div>
38   </div>
39 </div>
40 <div class="container" ng-view style="margin-top:50px"></div>
```

O código do corpo do documento html é uma código de um exemplo retirado do Twitter Bootstrap. Você não precisa, a princípio, conhecer as classes do Bootstrap tais como navbar, container, bastando apenas copiar o código que está pronto.

Na linha 28 temos a criação dos itens de menu para este projeto, no qual iremos criar as telas de Cliente, Funcionários e Transportadoras. Veja que já estamos usando **deeplinking**, um conceito aprendido nos capítulos anteriores do livro, no qual possamos criar estruturas html que serão carregadas dinamicamente.

Na linha 36, criamos uma área onde será exibida uma imagem quando houver um método ajax no servidor, para indicar ao usuário quando o sistema estiver consultando dados. Faremos com que esta forma de exibição seja a mais dinâmica possível.

Na linha 42, incluímos uma `div` utilizando o atributo `ng-view`, configurando essa `div` para que nela seja renderizada todo o conteúdo que o *deeplinkg* fornecer.

Continuando com o arquivo `html`, temos a inclusão das bibliotecas JavaScript, veja:

`index.html`

---

```
43 <!-- JavaScript -->
44 <!-- Libs -->
45 <script src="js/lib/jquery.min.js"></script>
46 <script src="js/lib/jquery.smooth-scroll.min.js"></script>
47 <script src="js/lib/bootstrap.min.js"></script>
48 <script src="js/lib/bootswatch.js"></script>
49 <script src="js/lib/angular.min.js"></script>
50 <script src="js/lib/angular-resource.min.js"></script>
51 <script src="js/app.js"></script>
52 <!--Controllers-->
53 <script src="js/clientesController.js"></script>
54 </body>
55 </html>
```

---

Finalizando o arquivo `html`, incluímos todas as bibliotecas javascript, tais como jQuery e AngularJS. Também incluímos o arquivo `app.js`, que contém as configurações da aplicação e vamos, na medida em que criarmos mais funcionalidades, adicionar novos arquivos ao sistema.

## Arquivo `app.js`

O arquivo `app.js` é definido pela tag `ng-app='app'`, no começo do documento `html` e adicionado logo após adicionar todas as bibliotecas javascript estarem adicionadas no sistema. Este arquivo é responsável em criar o módulo `app` juntamente com algumas outras funcionalidades.



**app.js**

---

```
1 //URL de acesso ao servidor RESTful
2 SERVER_URL = "http://localhost/sales-server";
3
4 //Criação ao $app que é o modulo que representa toda a aplicação
5 $app = angular.module('app', []);
```

---

Inicialmente criamos a variável `SERVER_URL` que é uma variável global e será usada para acesso ao servidor. Com esta variável não precisaremos digitar o endereço do servidor nas requisições Ajax que fizermos, e caso haja alguma mudança neste endereço, basta alterar em um único local.

Na linha 5 criamos o módulo do AngularJS que irá gerenciar toda a aplicação. Este módulo se chama 'app', que é o mesmo nome definido em `<html ng-app='app'>`. O segundo parâmetro do método `modules` é um array vazio `[]` que deve estar presente para que o AngularJS funcione. Neste segundo parâmetro iremos inserir, quando for necessário, bibliotecas adicionais do próprio AngularJS.

**app.js**

---

```
7 $app.config(function($routeProvider,$httpProvider){
8
9     //Configura o route provider
10     $routeProvider.
11     when('/',{templateUrl:'view/main.html'}).
12     when('/clientes',{templateUrl:'view/clientes/main.html',controller:clientesContr\
13 oller}).
14     when('/clientes/new',{templateUrl:'view/clientes/update.html',controller:cliente\
15 sController}).
16     when('/cliente/:id',{templateUrl:'view/clientes/update.html',controller:clientes\
17 Controller}).
18     otherwise({redirectTo:'/'});
```

---

Na linha 7, utilizamos o método `config` do AngularJS para configurar algumas funcionalidades no sistema. Veja que, como parâmetros, utilizamos `$routeProvider` que irá fornecer funcionalidades ligadas ao roteamento (deeplinking) das urls e telas que iremos acessar, e `$httpProvider` que provê funcionalidades de acesso ajax ao servidor.

Essas duas variáveis serão automaticamente injetadas com suas respectivas instâncias, tudo realizado pelo próprio AngularJS.

Na linha 11, iniciamos o roteamento de urls, de uma forma bastante simples. Com o método `when`, configuramos a url que será acessada e posteriormente os arquivos html e o controller a serem carregados.

Por exemplo, na linha 14 configuramos que, ao acessar a url `/clientes/new` o template `html view/clientes/update.html` será carregado, juntamente com o controller `clientesController`.

Na linha 14 incluímos o parâmetro `id` na url, que poderá ser utilizado pelo controller. Na linha 18, usamos o método `otherwise` para configurar o roteamento de qualquer url que não estava mapeada nos mapeamentos anteriores.

A medida em que formos adicionando mais funcionalidades ao sistema, adicionaremos mais rotas ao `$routeProvider`.

#### app.js

```
19 //configura o RESPONSE interceptor, usado para exibir o ícone de acesso ao servid\
20 or
21 // e a exibir uma mensagem de erro caso o servidor retorne algum erro
22 $httpProvider.responseInterceptors.push(function($q,$rootScope) {
23     return function(promise) {
24         //Always disable loader
25         $rootScope.hideLoader();
26         return promise.then(function(response) {
27             // do something on success
28             return(response);
29         }, function(response) {
30             // do something on error
31             $data = response.data;
32             $error = $data.error;
33             console.error($data);
34             if ($error && $error.text)
35                 alert("ERROR: " + $error.text);
36             else{
```

```

37             if (response.status=404)
38                 alert("Erro ao acessar servidor. Página não encontrada. Veja
39 erros para maiores detalhes");
40             else
41                 alert("ERROR! See log console");
42         }
43         return $q.reject(response);
44     });
45 }
46 });
47 });

```

---

Neste código um pouco mais complexo, estamos criando uma funcionalidade para capturar os eventos ajax do sistema e exibir mensagens de erro personalizada, caso existam. Isso é feito para que não precisemos tratar erros em qualquer chamada ajax que houver.

#### app.js

---

```

46 $app.run(function($rootScope){
47
48     //Uma flag que define se o ícone de acesso ao servidor deve estar ativado
49     $rootScope.showLoaderFlag = false;
50
51     //Força que o ícone de acesso ao servidor seja ativado
52     $rootScope.showLoader=function(){
53         $rootScope.showLoaderFlag=true;
54     }
55     //Força que o ícone de acesso ao servidor seja desativado
56     $rootScope.hideLoader=function(){
57         $rootScope.showLoaderFlag=false;
58     }
59
60     //Método que retorna a URL completa de acesso ao servidor.
61     // Evita usar concatenação no conteroller
62     $rootScope.server=function(url){
63         return SERVER_URL + url;
64     }
65
66 });
67
68 //We already have a limitTo filter built-in to angular,

```

```
69 //let's make a startFrom filter
70 $app.filter('startFrom', function() {
71     return function(input, start) {
72         if (input==null)
73             return null;
74         start = +start; //parse to int
75         return input.slice(start);
76     }
77 });
```

---

Na linha 46, incluímos mais funcionalidades ao sistema através do método `run`, no qual configura algumas variáveis e métodos globais. Por exemplo, criamos o método `showLoader` que poderá ser chamado de qualquer controller para exibir a imagem de carregamento do sistema. O método `server` irá configurar de forma correta o caminho completo da URL de acesso ao servidor.

Repare nos parâmetros que são injetados pelo AngularJS. No caso do método `run` o parâmetro `$rootScope` é injetado e pode ser usado para manipular as variáveis globais do sistema.

Finalizando, na linha 70, é criado um método chamado `filter` que será usado para realizar a paginação de alguns grids do sistema.

# Cadastro de clientes

Após criar a base do sistema, contendo a lógica inicial no arquivo `index.html` e `app.html` vamos criar o cadastro de clientes. Este cadastro será realizado através de três arquivos, que já foram citados no mapeamento de roteamento da aplicação:

`app.js`

---

```
7 $app.config(function($routeProvider,$httpProvider){
8
9     //Configura o route provider
10     $routeProvider.
11     when('/',{templateUrl:'view/main.html'}).
12     when('/clientes',{templateUrl:'view/clientes/main.html',controller:clientesContr\
13 oller}).
14     when('/clientes/new',{templateUrl:'view/clientes/update.html',controller:cliente\
15 sController}).
16     when('/cliente/:id',{templateUrl:'view/clientes/update.html',controller:clientes\
17 Controller}).
18     otherwise({redirectTo:'/'});
```

---

## Templates

### Listagem de clientes

O acesso a página de listagem de clientes é realizada através da URL `/clientes`, que está definido no arquivo `index.html`, mais precisamente no link: ‘<li></li>’ onde usamos a ‘#’ para que o AngularJS possa interceptar a URL e aplicar o roteamento.

Neste caso, será carregado o template `view/clientes/main.html` e este template será renderizado na div que contém a propriedade `ng-view` da aplicação. O arquivo `main.html` é representado a seguir:

## view/clientes/main.html

---

```

1  <h2>Clientes</h2>
2  <div ng-init="loadAll()">
3
4  <div>
5      <a href="#/clientes/new" class="pull-right ">Novo</a>
6  </div>
7  <table id="tableData" class="table table-bordered table-hover table-striped">
8      <thead>
9          <tr>
10             <th width="10%">Id</th>
11             <th>Nome</th>
12             <th>Fone</th>
13             <!-- <th width="10%"></th>-->
14          </tr>
15      </thead>
16      <tbody>
17          <tr ng-repeat="row in rows | startFrom: currentPage*pageSize| limitTo: pa\
18 geSize">
19              <td>{{row.CustomerID}}</td>
20              <td><a href='#/cliente/{{row.CustomerID}}'>{{row.ContactName}}</a></t\
21 d>
22              <td>{{row.Phone}}</td>
23          </tr>
24      </tbody>
25  </table>
26  <button class="btn btn-primary" ng-click="currentPage=currentPage-1" ng-disabled=\
27 "currentPage == 0" >&larr; Voltar</button>
28  <button class="btn btn-primary pull-right" ng-click="currentPage=currentPage+1" n\
29 g-disabled="currentPage >= rows.length/pageSize - 1">Avançar &rarr;</button>
30 </div>

```

---

Como visto neste template, o código html corresponde apenas a listagem de clientes e será renderizado na div com o atributo ng-view. Neste html, criamos uma tabela simples com a tag <table> e utilizamos algumas classes que são nativas ao Twitter Bootstrap, tais como table-bordered e table-striped.

Todas as classes dão “vida” ao componente, e é responsabilidade do Twitter Bootstrap fazer isso, como por exemplo a classe pull-right que irá pegar a tag que a usou e colocá-la à direita da página.

A parte mais importante deste template está na linha 18, no qual usamos o comando ng-repeat

que irá repetir o comando `<tr>` de acordo com a lista `rows`. Neste caso, `rows` será definido pelo seu respectivo controller, que será visto logo a seguir.

O comando `row in rows` é um loop, no estilo do `foreach` do php, e significa: “interaja entre cada item de `rows` atribuindo o respectivo valor à variável `row`”. O comando `startFrom` e `limitTo` são usados para a paginação de dados, que neste caso é feito totalmente por javascript. Ou seja, todos os registros de clientes são obtidos de uma vez só, o que não pode ser muito bom quando tivermos centenas de registros.

Na criação das células da tabela, usamos a variável `row` como objeto e pode-se incluir os dados, como por exemplo `row.ContactName` para inserir o nome do cliente.

Na linha 27 e 29 criamos os botões para paginação, que são utilizados pelo controller para paginar os dados na tabela. O resultado deste código HTML pode ser visto na imagem a seguir:

Sales - 1   Home   Clientes   Funcionários   Transportadoras

Clientes

Novo

Id	Nome	Fone
	0Teste	teste
a	a222	a2222
ALFKI	Maria Anders 23333333	030-00743212
ANATR	Ana Trujillo 2	(5) 555-4729
ANTON	Antonio Moreno 2	(5) 555-3932
AROUT	Thomas Hardy	(171) 555-7788
BERGS	Christina Berglund	0921-12 34 65
BLAUS	Hanna Moos	0621-08460
BLONP	Frdrique Citeaux	88.60.15.31
BOLID	Martn Sommer	(91) 555 22 82
BONAP	Laurence Lebihan	91.24.45.40
BOTTM	Elizabeth Lincoln	(604) 555-4729
BSBEV	Victoria Ashworth	(171) 555-1212
CACTU	Patricio Simpson	(1) 135-5555
CENTC	Francisco Chang	(5) 555-3392

← Voltar

Avançar →

### Listagem de clientes

## Inserir novo cliente

Ao acessarmos `/cliente/new`, segundo o roteamento, o template `view/clientes/update.html` será carregado, que corresponde ao formulário para cadastro e alteração de clientes. Usamos o mesmo formulário para ambas as ações. O formulário é exibido a seguir:

view/clientes/update.html

```
1 <h2>Cliente</h2>
2 <div ng-controller="clientesController" ng-init="loadRow()">
3   <form ng-submit="save()" class="form-horizontal" ng-show="row!=null">
4     <div class="control-group">
5       <label class="control-label">ID: </label>
6       <div class="controls" >
7         <input type="text" ng-model="row.CustomerID" class="input-small" \
8 required ng-disabled="row.isUpdate" />
9       </div>
10    </div>
11    <div class="control-group">
12      <label class="control-label">Nome: </label>
13      <div class="controls">
14        <input type="text" ng-model="row.ContactName" class="input-xxlarge" \
15 e" />
16      </div>
17    </div>
18    <div class="control-group">
19      <label class="control-label">Telefone: </label>
20      <div class="controls">
21        <input type="text" ng-model="row.Phone"/>
22      </div>
23    </div>
24
25    <div class="form-actions">
26      <button type="submit" class="btn btn-primary">Salvar</button>
27      <a href="#/clientes" class="btn">Voltar</a>
28    </div>
29  </form>
30  <button class="btn btn-danger pull-right" ng-click="del()">Apagar</button>
31 </div>
```

O formulário de cadastro possui diversas novidades relativas ao AngularJS. Na linha 2, usamos o atributo `ng-init` que irá chamar o método `loadRow`, responsável em carregar o objeto `row` com as informações do cliente.

Na linha 3 criamos o formulário com a tag `form` e usamos o atributo `ng-submit` que indica qual o método do controller que será executado quando o formulário for submetido. Ainda nesta linha, usamos o atributo `ng-show` que irá exibir o formulário somente quando o objeto `row` existir. Isso é



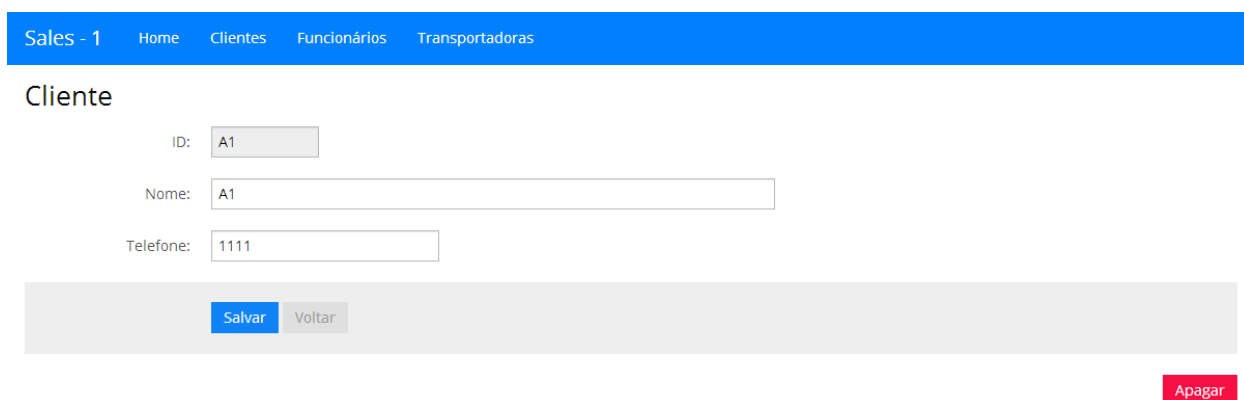
necessário por que, ao abrir o formulário, é preciso carregar os dados via Ajax, e nesse momento a visualização do formulário não é necessária.

Nos campos criados, da linha 4 à 23, usamos o Twitter Bootstrap para formatar os campos, e o atributo `ng-model` para dizer ligar estes campos ao modelo de cliente, isto é, `ng-model=row.CustomerID` informa que este campo possui o ID do cliente.

Na linha 8 usamos o atributo `ng-disable` para desabilitar o campo caso o formulário esteja em modo de edição, já que o ID não pode ser editado.

Na linha 26 temos o botão **Salvar**, que irá fazer o *submit* do formulário, e consequentemente irá chamar o método `save()` do controller.

Na linha 30, criamos um botão para excluir o cliente, que irá chamar o método `del()` do controller. O template criado é semelhante a figura a seguir:



O formulário é exibido dentro de uma barra de navegação azul com links para Sales - 1, Home, Clientes, Funcionários e Transportadoras. O título 'Cliente' está no topo do formulário. Há três campos de entrada: 'ID:' com o valor 'A1', 'Nome:' com o valor 'A1' e 'Telefone:' com o valor '1111'. Abaixo dos campos, há dois botões: 'Salvar' (em azul) e 'Voltar' (em cinza). Um botão 'Apagar' (em vermelho) está localizado à direita do formulário.

Formulário para atualizar cliente

## Controller

Agora que criamos os dois templates relativos a listar clientes e salvar clientes, podemos conferir como o controller será criado, de acordo com cada tela.

Na página `index.html`, adicionamos o arquivo `clientesController.js` que irá possuir o método `clientesController` que é referenciado no roteamento do arquivo `app.js`.

Ou seja, para criarmos o controller de clientes, temos que criar o seguinte método.

**clientesController.js**

---

```
1 //Clientes Controller
2 function clientesController($scope,$http,$routeParams,$location)
3 {
4     //lista de clientes
5     $scope.rows = null;
6
7     //um cliente
8     $scope.row = null;
9
10    //Pagination
11    $scope.currentPage = 0;
12    $scope.pageSize = 15;
13
14    $scope.numberOfPages =function(){
15        return Math.ceil($scope.rows.length/$scope.pageSize);
16    }
```

---

Inicialmente, criamos o método e já definimos alguns parâmetros para este método, como `$scope` e `$http`. Vamos a seguir explicar cada um destes parâmetros, lembrando que o AngularJS se encarrega de injetar as instâncias de cada um desses parâmetros automaticamente.

**`$scope`**

É o escopo do controller, onde poderemos criar variáveis e métodos que serão usados pela view. Então, ao criarmos a variável `$scope.rows` poderemos usá-la na view como `rows`.

**`$http`**

Usada para realizar requisições Ajax ao servidor.

**`$routeParams`**

Algumas rotas podem possuir parâmetros, como por exemplo `/cliente/:id/`, e obtemos o valor desses parâmetros através do `routeParams`. Neste caso, ao acessarmos por exemplo a rota `/cliente/5` a variável `$routeParams.id` será preenchida com o valor 5.

**`$location`**

Usada para redirecionar de uma rota para outra, através do comando `$location.path(<rota>)`.

Os parâmetros do controller não precisam estar nesta ordem, e podem ser adicionados na medida em que precisamos deles.

Na linha 5, criamos a variável `$scope.rows`, que é a variável usada na tabela de clientes, quando fazemos o loop com o `ng-repeat` (`ng-repeat="row in rows"`). Na linha 8, criamos a variável `$scope.row` que é a variável usada no formulário para salvar um cliente específico. A paginação de clientes é configurada nas linhas 11 e 12, juntamente com o método `numberOfPages` na linha 14.

Após a criação destas variáveis, criamos o primeiro método que irá acessar o servidor e buscar na tabela todos os clientes cadastrados. Este método é descrito a seguir:

#### clientesController.js

```
17 $scope.loadAll = function(){  
18     $scope.showLoader();  
19     $http.get($scope.server("/customers")).success(function(data){  
20         $scope.rows = data;  
21     });  
22 }
```

O método `loadAll` pode ser chamado pelo controller ou pela view, e possui a responsabilidade de obter dados do servidor. Inicialmente, na linha 18, é chamado o método global `showLoader()` que exibirá uma imagem de carregamento na tela do usuário, para que ele possa saber que o sistema está carregando dados.

Na linha 19, usamos o método `$http.get` que irá realizar uma requisição Ajax, cujo o primeiro parâmetro é a URL dessa requisição. Para definir essa URL, usamos o método global `server`, criado no arquivo `app.js`, que quando chamado retornará o seguinte valor:

```
http://localhost/sales-server/customers
```

Após realizar uma requisição GET a este endereço, é chamado o método `success`, que está encadeado logo após o método `get`. O método `success` é executado quando a requisição Ajax retorna com os dados do servidor, sem erros. Quando isso acontece, o código da linha 20 é executado e a variável `rows` é preenchida. No momento em que esta variável é preenchida, a tabela na view que usa esta variável no `ng-repeat` será preenchida.

A requisição Ajax sabe se houve erros ou não pelo status da resposta, que é um número que identifica o erro. Por exemplo, o status 404 significa que a página não foi encontrada enquanto que o status 500 significa um erro no servidor. Quando a requisição não dá erro, o status é 200.

Quando ocorre um erro no servidor, o método `success` não será executado, e já foi programado no arquivo `app.js` para que esse erro seja capturado e exibido ao cliente.

**clientesController.js**

---

```
23 $scope.loadRow = function(){
24     if ($routeParams.id!=null)
25     {
26         $scope.showLoader();
27         $http.get($scope.server("/customer/"+$routeParams.id))
28             .success(function(data){
29                 $scope.row = data;
30                 $scope.row.isUpdate = true;
31             });
32     }
33     else
34     {
35         $scope.row = {};
36         $scope.row.CustomerID = null;
37         $scope.row.isUpdate = false;
38     }
39 }
```

---

O próximo método se chama `loadRow` e é responsável em carregar os dados de um cliente específico. Neste caso, o método pode ser usado para carregar um registro específico, buscando-o no servidor (linha 27), ou criar um novo registro (linhas 13 a 15).

Para sabermos se estamos criando um novo ou carregando um registro específico, analisamos a variável `$routeParams.id` que é repassada pelo roteamento da aplicação.

**clientesController.js**

---

```
40 $scope.save = function(){
41     $scope.showLoader();
42     $http.post($scope.server("/customer/"+$routeParams.id),$scope.row)
43         .success(function(data){
44             alert("Salvo com sucesso");
45             $scope.row.isUpdate = true;
46         });
47 }
```

---

O método `save` é usado para realizar um POST no servidor, repassando como parâmetro o objeto `row`, que contém os dados do formulário. Se tudo correr bem, o método `success` será executado e uma mensagem ao usuário será exibida.

**clientesController.js**

---

```
23 $scope.del = function(){
24   if (confirm("Deseja excluir " + $scope.row.CustomerID + "?")){
25     $http.delete($scope.server("/customer/"+$routeParams.id))
26       .success(function(s){
27         alert("Excluído com sucesso");
28         $location.path("/clientes");
29       });
30   }
31 }
```

---

Finalizando o cadastro de clientes, temos a rotina para apagar um registro. Após fazer a confirmação com o usuário, realizamos uma requisição DELETE no servidor. Se tudo correr bem, o registro é excluído e a página é redirecionada para a listagem de clientes.

Terminamos o básico de um cadastro (CRUD) e é muito importante que você domine tudo que foi aprendido até agora. Caso tenha alguma dúvida, entre em contato com *ventas@danielschmitz.com.br* para que possamos lhe ajudar.

# Cadastro de funcionários

Na tela de cadastro de funcionários, usamos os mesmos conceitos apreendidos nos exemplos anteriores, só que agora vamos focar um pouco mais no dataBind, fazendo com que a tela de listagem de funcionários e a tela de editar funcionário seja a mesma, conforme a figura a seguir.

Sales - 1HomeClientesFuncionáriosTransportadoras

Funcionários

Davolio, Nancy 2

Fuller, Andrew 33

Leverling, Janet

Peacock, Margaret

Buchanan, Steven

Suyama, Michael

King, Robert

Callahan, Laura

Dodsworth, Anne

Schmitz, Daniel

de Tal, Fulando

, LOL

Fuller, Andrew 33

Nome:Andrew 33

Sobrenome:Fuller

Telefone:(206) 555-9482

Salvar

Novo

Tela de Funcionários

## Servidor

No servidor, criamos o arquivo `employee.php` que contém as funcionalidades para alterar os dados no banco. O arquivo é apresentado a seguir.

sales-server/employee.php

---

```
1  <?php
2
3  $app->get("/employees",function (){
4
5      $sql = "SELECT EmployeeID,FirstName,LastName,HomePhone FROM employees";
6      $stmt = DB::prepare($sql);
7      $stmt->execute();
8      formatJson($stmt->fetchAll());
9  });
10
11 $app->get("/employee/:id",function ($id){
12
13     $sql = "SELECT EmployeeID,FirstName,LastName,HomePhone FROM employees WHERE Emplo\
14 yeeID=?";
15     $stmt = DB::prepare($sql);
16     $stmt->execute(array($id));
17     formatJson($stmt->fetch());
18 });
19
20 $app->post("/employee/",function (){
21
22     $data =json_decode(\Slim\Slim::getInstance()->request()->getBody());
23
24     if ($data->EmployeeID!=0){
25         $sql = "UPDATE employees SET FirstName=?,LastName=?,HomePhone=? WHERE EmployeeID\
26 =?";
27         $stmt = DB::prepare($sql);
28         $stmt->execute(array(
29             $data->FirstName,
30             $data->LastName,
31             $data->HomePhone,
32             $data->EmployeeID
33         ));
34     };
35 }
36 else
37 {
38     $sql = "INSERT INTO employees (FirstName,LastName,HomePhone) VALUES (?, ?, ?)";
39     $stmt = DB::prepare($sql);
40     $stmt->execute(array(
```

```
41         $data->FirstName,
42         $data->LastName,
43         $data->HomePhone
44     )
45 );
46 $data->EmployeeID = DB::lastInsertId();
47 }
48
49 formatJson($data);
50 });
51
52 $app->delete("/employee/:id",function ($id){
53 $sql = "DELETE FROM customers WHERE CustomerID=?";
54     $stmt = DB::prepare($sql);
55     $stmt->execute(array($id));
56 formatJson(true);
57 });
```

---

As funcionalidades relativas ao funcionário são bastante semelhantes ao cliente, possuindo os métodos GET e POST para realizar as operações no banco de dados.

## Cliente

O primeiro passo para criar a tela de funcionários consiste na definição do mapeamento da rota, que deve ser feito no arquivo `app.js`.

`js/app.js`

---

```
9  //Configura o route provider
10 $routeProvider.
11 when('/',{templateUrl:'view/main.html'}).
12 when('/clientes',{templateUrl:'view/clientes/main.html',controller:clientesContro\
13 ller}).
14 when('/clientes/new',{templateUrl:'view/clientes/update.html',controller:clientes\
15 Controller}).
16 when('/cliente/:id',{templateUrl:'view/clientes/update.html',controller:clientesC\
17 ontroller}).
18 when('/funcionarios',{templateUrl:'view/funcionarios/main.html',controller:funcio\
19 nariosController}).
20 otherwise({redirectTo:'/'});
```

---



Na linha 15, criamos a entrada para `/funcionarios` que irá redirecionar para o template `main.html` e o controller `funcionariosController` também será carregado.

O controller `funcionariosController` será criado no arquivo `funcionariosController.js` e deve ser incluído no arquivo `index.html`, logo a seguir do arquivo `clientesController.js`.

#### `funcionariosController.js`

---

```
1  function funcionariosController($scope,$http,$routeParams,$location){
2
3  //lista de funcionarios
4  $scope.rows = null;
5
6  //um funcionario
7  $scope.row = null;
8
9  $scope.loadAll = function() {
10     $scope.showLoader();
11     $http.get($scope.server("/employees")).success(function(data){
12         $scope.rows = data;
13     });
14 }
15
16 $scope.loadRow = function(id){
17     $scope.showLoader();
18     $http.get($scope.server("/employee/"+id)).success(function(data){
19         $scope.row = data;
20         $scope.row.isUpdate = true;
21     });
22 }
23
24 $scope.save = function() {
25     $scope.showLoader();
26     $http.post($scope.server("/employee/"),$scope.row).success(function(data){
27         alert("Salvo com sucesso");
28         $scope.loadAll();
29     });
30 }
31
32 $scope.new = function() {
33     $scope.row = {
34         EmployeeID:0,
35         FirstName:"",
```

```

36         LastName: "",
37         HomePhone: ""
38     }
39 }
40
41
42
43 }

```

---

No controller, criamos as operações principais de uma tela de cadastro, manipulando o array de funcionários através da variável `rows` e um funcionário específico através da variável `row`. Veja que como o AngularJS torna todo o código muito mais simples, não é necessário conhecimento adicional para entender o código. Se compreendeu bem o cadastro de clientes, o cadastro de vendedores é até mais fácil de programar.

A novidade agora está no layout que mostra tanto a listagem quanto a edição de um funcionário.

#### funcionarios/main.html

---

```

1  <h2>Funcionários</h2>
2  <button class="btn btn-primary pull-right" ng-click="new()">Novo</button>
3  <div ng-init="loadAll()">
4  <div class="span3">
5      <ul class="nav nav-tabs nav-stacked">
6          <li ng-repeat="row in rows">
7              <a ng-click="loadRow(row.EmployeeID)">
8                  {{row.LastName}}, {{row.FirstName}}
9              </a>
10         </li>
11     </ul>
12 </div>
13 <div class="span7" ng-show="row!=null">
14     <h3>{{row.LastName}}, {{row.FirstName}}</h3>
15     <hr/>
16     <form ng-submit="save()" class="form-horizontal" ng-show="row!=null">
17         <div class="control-group">
18             <label class="control-label">Nome: </label>
19             <div class="controls">
20                 <input type="text" ng-model="row.FirstName"
21                                     class="input-xxlarge" />
22             </div>
23     </div>

```

```
24         <div class="control-group">
25             <label class="control-label">Sobrenome: </label>
26             <div class="controls">
27                 <input type="text" ng-model="row.LastName"
28                                     class="input-xxlarge" />
29             </div>
30         </div>
31         <div class="control-group">
32             <label class="control-label">Telefone: </label>
33             <div class="controls">
34                 <input type="text" ng-model="row.HomePhone"/>
35             </div>
36         </div>
37
38         <div class="form-actions">
39             <button type="submit" class="btn btn-primary">Salvar</button>
40         </div>
41     </form>
42 </div>
43 </div>
```

---

Esta tela é dividida na listagem de vendedores, realizada pelo Bootstrap que está na div da esquerda, separada pela classe span3 que é uma forma de dividir a tela em colunas (esse recurso se chama *grid system*).

Nesta listagem, usamos classes do Twitter Bootstrap para exibir a listagem em forma de stacks, mas pode-se perceber que a lista é formada apenas pelas tags `ul` e `li` do html. Cada `li` da lista possui um link (linha 7) que chama o método `loadRow` e ainda repassa o ID do funcionário para que os seus dados sejam recuperados. No Controller, esse método acessa o servidor e obtém os dados do funcionário em questão, atualizando a variável global `row`.

Quando a variável `row` é atualizada, o formulário com os dados do funcionário são exibidos na parte direita da tela. Os campos do formulário são ligados a variável `row` pelo `ng-model`, fazendo com que o `data-bind` seja realizado perfeitamente.

Perceba que, tanto o `input` que possui o nome do vendedor quanto o título do formulário (linha 14) ligam à mesma variável, fazendo com que ao alterar o `input`, o título é alterado dinamicamente.

# Demais cadastros

Como as telas de cadastro possuem a mesma metodologia, as demais telas deixamos como um exercício para que você possa tentar implementar sozinho, e conte com a nossa ajuda caso encontre alguma dúvida ou problema.

# Validação

Validar a entrada de dados é uma tarefa importante para manter a integridade dos dados que serão enviados ao servidor. Existem diversas técnicas de validação de dados, incluindo inclusive uma validação dupla, tanto no servidor quanto no cliente.

Como o nosso foco é o AngularJS, iremos validar dados somente no cliente, mas é vital que a mesma validação seja realizada no servidor. Isso é necessário porque o formulário que está sendo processado em um navegador pode ser facilmente manipulável através de Javascript, ou seja, um usuário mal intencionado pode manipular a sua validação de dados e enviar um post modificado.



Outra questão importante é, a validação no cliente possui apenas uma característica cosmética, destinada a informar ao usuário de forma clara e rápida sobre pequenos erros de entrada de dados. Não valide dados complexos, como por exemplo dados relativos a cartão de crédito, qualquer tipo de regra que envolva acesso ao banco de dados, regras que envolvem cálculos matemáticos, entre outros.

## Validando com html

A forma mais simples de validar dados com o AngularJS é usar o próprio HTML a nosso favor. Neste ponto existe uma receita de bolo que podemos usar sempre que precisarmos.



Existe uma desvantagem muito grave nessa técnica, pois a validação depende do navegador que o cliente usa, e a validação css não funciona no Internet Explorer, apenas no Google Chrome, Mozilla Firefox e outros baseados no webkit.

A validação que temos disponível pode ser usada para os seguintes tipos de campos: `text`, `number`, `url`, `email`, `radio`, `checkbox` e as diretivas para esta validação são: `required`, `pattern`, `minlength`, `maxlength`, `min`, `max`.

Vamos implementar a técnica no formulário para cadastrar clientes. O que precisamos saber é que devemos utilizar CSS para validar, e para isso basta inserir informações adicionais no arquivo `view\clientes\update.html`

A primeira regra que deve ser usada é utilizar o atributo `name` no formulário e nos seus respectivos campos. Isso dará ao AngularJS uma forma de acessar as propriedades de cada objeto, de forma a poder manipulá-los.

A segunda regra é usar os atributos padrão do html 5 para validação, então se você deseja que um campo seja requerido, deve-se utilizar o atributo `required`, e se deseja que um campo seja validado como email, deve-se usar `type='email'`.

Com estas duas regras, podemos criar a validação para o primeiro campo, `CustomerID`. Este campo possui o preenchimento obrigatório, e adicionamos a ele uma mensagem de erro, conforme o código a seguir:

view/clientes/update.html

```
<div class="control-group">
  <label class="control-label">ID:</label>
  <div class="controls" >
    <input name="CustomerID" type="text" ng-model="row.CustomerID"
          class="input-small" required ng-disabled="row.isUpdate" />
    <span class="help-inline text-error"
          ng-show="form.CustomerID.$dirty && form.CustomerID.$invalid">
      Campo obrigatório
    </span>
  </div>
</div>
```

Além de adicionar o atributo `name` ao campo, incluímos um bloco de texto através da tag `span` que contém a mensagem de erro para o campo requerido. Esta mensagem possui as classes `help-inline` e `text-error` que irão formatar a mensagem. Além disso, precisamos configurar se a mensagem irá aparecer ou não, e isso é feito através do `ng-show`, onde usamos algumas propriedades do formulário que são obtidas através do AngularJS.

A primeira propriedade é `form.CustomerID.$dirty` onde é verificado se o campo possui modificações. Em caso verdadeiro, é usado `form.CustomerID.$invalid` para determinar se o campo está inválido ou não.

O mesmo acontece com o campo email, que é programado da seguinte forma:

view/clientes/update.html

```
<div class="control-group">
  <label class="control-label">Email:</label>
  <div class="controls">
    <input name="Email" type="email" ng-model="row.Email" required/>
    <span class="help-inline text-error"
          ng-show="form.Email.$dirty && form.Email.$invalid">
      <span ng-show="form.Email.$error.required">Campo obrigatório</span>
    </span>
  </div>
</div>
```

```
<span ng-show="form.Email.$error.email">Email inválido</span>
</span>
</div>
</div>
```

Neste código, criamos a mesma estrutura do exemplo anterior, mas como precisamos validar o email, é necessário adicionar `type='email'` no campo. Além disso, usamos também o comando `form.Email.$error.email` para verificar se o email é válido. O resultado da validação é exibido a seguir.

Tela de clientes com validação

## Validando com ng-pattern

Se você deseja estender um pouco a validação dos campos do seu formulário, pode usar expressões regulares em javascript com o comando `ng-pattern`. A seguir listamos alguns patterns mais conhecidos, que serão adicionados no formulário de clientes apenas como exemplo, e não serão persistidos no banco de dados. Os exemplos foram criados na aba “Exemplos de validação”

view/clientes/update.html

```
1 <div class="control-group">
2   <label class="control-label">Url:</label>
3   <div class="controls">
4     <input name="Url" type="url" ng-model="row.Url"
5       ng-pattern="/https?:\/\/.+\/"/>
6     <span class="help-inline text-error"
7       ng-show="form.Url.$dirty && form.Url.$invalid">
8       <span>Url inválida</span>
9     </span>
10  </div>
11 </div>
12
13 <div class="control-group">
14   <label class="control-label">Data simples:</label>
15   <div class="controls">
16     <input name="DataSimples" ng-model="row.DataSimples"
17       type="text" ng-pattern="/\d{1,2}/\d{1,2}/\d{4}/"/>
18     <span class="help-inline text-error"
19       ng-show="form.DataSimples.$dirty && form.DataSimples.$invalid">
20       <span>Data inválida</span>
21     </span>
22   </div>
23 </div>
```



É válido lembrar que toda validação deve ser feita também no servidor, para que o sistema RESTful possa ser funcional.



## **Parte 3 - Node.JS**

# Introdução ao NODE.JS

*Por Douglas Lira*

Tradicionalmente o javascript é executado no lado do cliente, mas recentemente surgiu-se um considerável interesse em trazê-lo para o lado do servidor, tendo como um dos principais resultados a criação do NODE.JS.

Servidores web, como o Apache, são usados para interpretar scripts CGI, PHP entre outros, gerando threads para cada solicitação de entrada.

Eis que surge o NODE.JS que é uma plataforma construída sobre o motor Javascript do Google Chrome, que funciona no lado servidor facilitando o desenvolvimento de aplicações de rede rápidas e escaláveis. Node.JS usa um modelo de entrada e saída direcionada a evento não bloqueante tornando-o leve e eficiente, ideal para aplicações em tempo real com troca intensa de dados através de dispositivos distribuídos.

## Como funciona o NODE.JS?

Node.js usa o conceito de ciclo de eventos ao invés de threads, e é capaz de suportar milhões de conexões simultâneas. Ele tira proveito do fato de que os servidores passam a maior parte do seu tempo de espera processando operações de entrada e saída, como a leitura de um arquivo de um disco rígido, o acesso a um serviço web externo ou à espera de um arquivo para terminar a ser carregado, isso porque essas operações são muito mais lentas do que em operações de memória.

Cada operação de entrada e saída em Node.js é assíncrona, isso significa que o servidor pode continuar processando os pedidos recebidos durante a operação de entrada e saída. Este modelo baseado em eventos faz do Node.js uma ferramenta rápida, tornando aplicações mais eficientes e obtendo o resultado real-time mais eficaz.

## O que definitivamente não é NODE.JS

Já sabemos que o NODE.JS é um programa que roda no lado do servidor, que seu objeto é fornecer uma maneira fácil de criar programas de redes escaláveis (sem a intenção de substituir o Apache ou Tomcat).

# Instalação

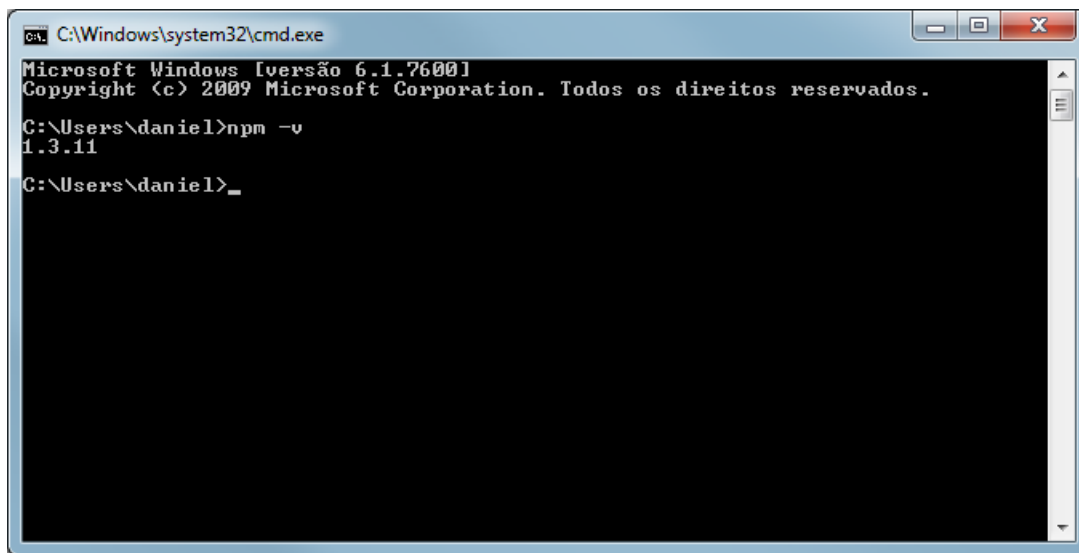
## Instalando o NODE.JS no Windows

Faça o download no site oficial<sup>8</sup> e execute o instalador. Após devidamente instalado, utilize o recurso do node conhecido como npm - *Node PackagedModules* - para instalar módulos que iremos ver mais adiante.

Para verificar a versão do node.js instalado, abra um prompt de comando e digite o seguinte:

```
1 C:\>npm -v
```

O resultado do comando é exibido na imagem a seguir:

A screenshot of a Windows command prompt window. The title bar reads 'C:\Windows\system32\cmd.exe'. The window content shows the following text: 'Microsoft Windows [versão 6.1.7600] Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.' followed by the command 'C:\Users\daniel>npm -v' and its output '1.3.11'. The prompt 'C:\Users\daniel>' is visible on the next line.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [versão 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.
C:\Users\daniel>npm -v
1.3.11
C:\Users\daniel>
```

Resultado do npm

## Instalando o NODE.JS no Linux

Antes de instalar o NODE.JS é preciso instalar algumas bibliotecas extras, segue abaixo o passo-a-passo.

- 1 – Instalando algumas bibliotecas necessárias. `sudo apt-get install g++ curl libssl-dev apache2-utils`
- 2 – Após instalar as bibliotecas extras, vá até o site do node.js faça o download do arquivo `node-v0.10.17.tar.gz` e em seguida descompacte.

---

<sup>8</sup><http://www.nodejs.org>

```
1 tar -xf node-v0.10.17.tar.gz
```

3 – Instalando.

```
1 ./configure
2 Make
3 sudo make install
```



Utilizaremos o Windows como base para criar e executar os scripts. A versão utilizada para instalar o node no Linux foi o Ubuntu 12.04

# Exemplos

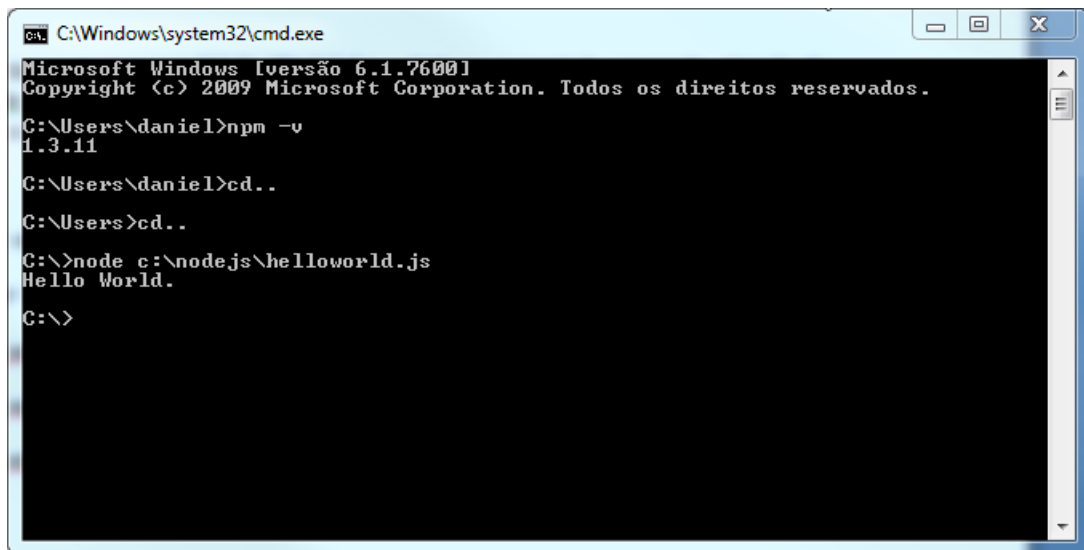
## Criando o primeiro script com NODE.JS

Crie uma pasta como nome node na unidade C: e em seguida crie um arquivo com o seguinte conteúdo.

```
1 console.log("Hello World");
```

Salve o arquivo com o nome HelloWorld.js. Vá até o prompt de comando, navegue até onde foi salvo o arquivo e execute o script da seguinte maneira:

```
1 node HelloWorld.js
```

A screenshot of a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The window shows the following commands and output: "Microsoft Windows [versão 6.1.7600] Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.", "C:\Users\daniel>npm -v" followed by "1.3.11", "C:\Users\daniel>cd.." followed by "C:\Users>cd..", and "C:\>node c:\nodejs\helloworld.js" followed by "Hello World.". The prompt "C:\>" is visible at the bottom.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [versão 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.
C:\Users\daniel>npm -v
1.3.11
C:\Users\daniel>cd..
C:\Users>cd..
C:\>node c:\nodejs\helloworld.js
Hello World.
C:\>
```

Resultado do npm

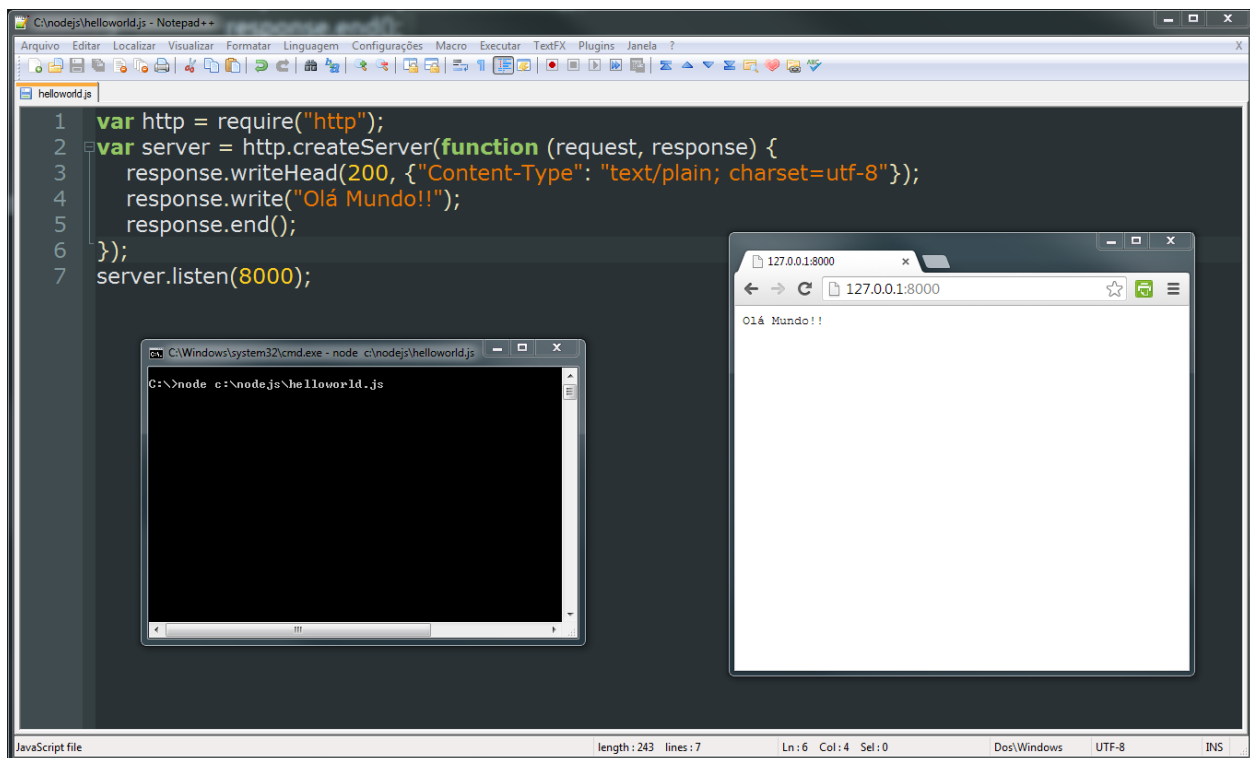
Observe que a mensagem Hello World foi exibida no prompt de comando, através do comando console.log. Vamos estender um pouco mais o helloworld criado e modificar o arquivo para o seguinte código:

## HelloWorld.js

```
1 var http = require("http");
2 var server = http.createServer(function (request, response) {
3     response.writeHead(200, {"Content-Type": "text/plain; charset=utf-8"});
4     response.write("Olá Mundo!!");
5     response.end();
6 });
7 server.listen(8000);
```

Salve o arquivo e execute. O código acima irá imprimir o resultado do script no browser. Na primeira linha utilizamos a biblioteca HTTP nativa do node que será responsável em criar uma comunicação entre o servidor e o browser. Na ultima linha é configurado o servidor para ser executado na porta 8000.

Para visualizar o resultado do script no browser, basta executar o script no prompt de comando e em seguida abrir o browser com a url `http://127.0.0.1:8000`. A função passada como primeiro parâmetro é executado sempre que a requisição for solicitada ao servidor na porta 8000 e a mensagem "Olá Mundo!!" será impressa no browser através do comando `response.write`.



Node.JS no browser



O script pode ser executado em qualquer porta, desde que a mesma não esteja sendo utilizada pelo sistema operacional ou software.

# Socket.IO

## O que é socket.IO

Socket.IO é uma biblioteca criada para o Node.js que faz websockets em tempo real, possível em todos os browsers, permitindo enviar vários sinais ou fluxos de informações sob uma mesma conexão, essa técnica é conhecida como *multiplexing* e, além disso, o Socket.IO favorece de forma clara e simples a escalabilidade horizontal.

Com Node.js e Socket.IO, implementar sistema de mensagens em tempo real faz com que seja menos trabalhoso por conter uma API de comandos que facilitam este desenvolvimento. Os comandos mais comuns e mais utilizados são `sockets.on` que serve para declarar um evento e o `sockets.emit` que serve como *trigger* para executar um evento.

## Criando um script com socket.io

Crie um arquivo com o nome “websocket.js” e insira o seguinte conteúdo:

websocket.js

---

```
1 var io = require('socket.io').listen(8000);
2 io.sockets.on('connection', function (socket) {
3     socket.on('emissor', function(texto) {
4         io.sockets.emit('receptor', texto);
5     });
6 });
```

---

O código acima é responsável em criar uma conexão websocket com Socket.IO utilizando a porta “8000”, em seguida é criado um evento chamado “connection” que sempre ao conectar-se ao Socket.IO irá criar um evento chamado “emissor”, que por sua vez sempre que for executado irá emitir uma solicitação para o evento “receptor” passando o parâmetro “texto”.

Para testarmos o websocket.js, crie o seguinte arquivo html:



index.html

---

```
1 <html>
2 <head>
3 <meta charset="UTF-8">
4
5 <title>AngularJS na pratica - SOCKET.IO por Douglas Lira</title>
6 <script src="http://localhost:8000/socket.io/socket.io.js"></script>
7
8 <script>
9
10 var socket = io.connect('http://localhost:8000');
11 socket.on('receptor',function(texto){
12   alert(texto);
13 });
14
15   function enviar(texto) {
16     socket.emit('emissor', texto);
17   }
18
19 </script>
20 </head>
21
22 <body>
23   <input type="button" onclick="enviar('Olá mundo!');" value="Enviar">
24 </body>
25
26 </html>
```

---

O código em destaque conecta-se ao servidor websocket através do evento “connection” criado no arquivo “websocket.js”. Também foi criado o evento “receptor” que ao ser executado irá exibir um alerta sempre que a função “enviar” for executada.

Para testar o código, execute no prompt de comando e navegue até a pasta onde encontra-se os arquivos criados e instale o socket.io utilizando o npm.

```
C:\node> npm install socket.io
```

Após devidamente instalado, execute o arquivo “websocket.js”. Execute o arquivo “index.html” em diferentes browsers e clique no botão “enviar”, observando que o evento receptor foi executado duas vezes por ter dois clientes conectados ao mesmo websocket.