01

Construindo o alicerce da aplicação

Transcrição

Quando desenvolvemos no server-side, organizamos nosso código em camadas para facilitar a manutenção, o reaproveitamento e a legibilidade de nosso código. É muito comum aplicarmos o modelo **MVC** (**M***odel,* ***V**iew,* **C**ontroller), que consiste na separação de tarefas, facilitando assim a reescrita de alguma parte e a manutenção do código.

Porém, não é raro o mesmo desenvolvedor deixar de lado essas práticas quando codifica no client-side. Mesmo aqueles que procuram organizar melhor seu código acabam criando soluções caseiras que nem sempre são bem documentadas.

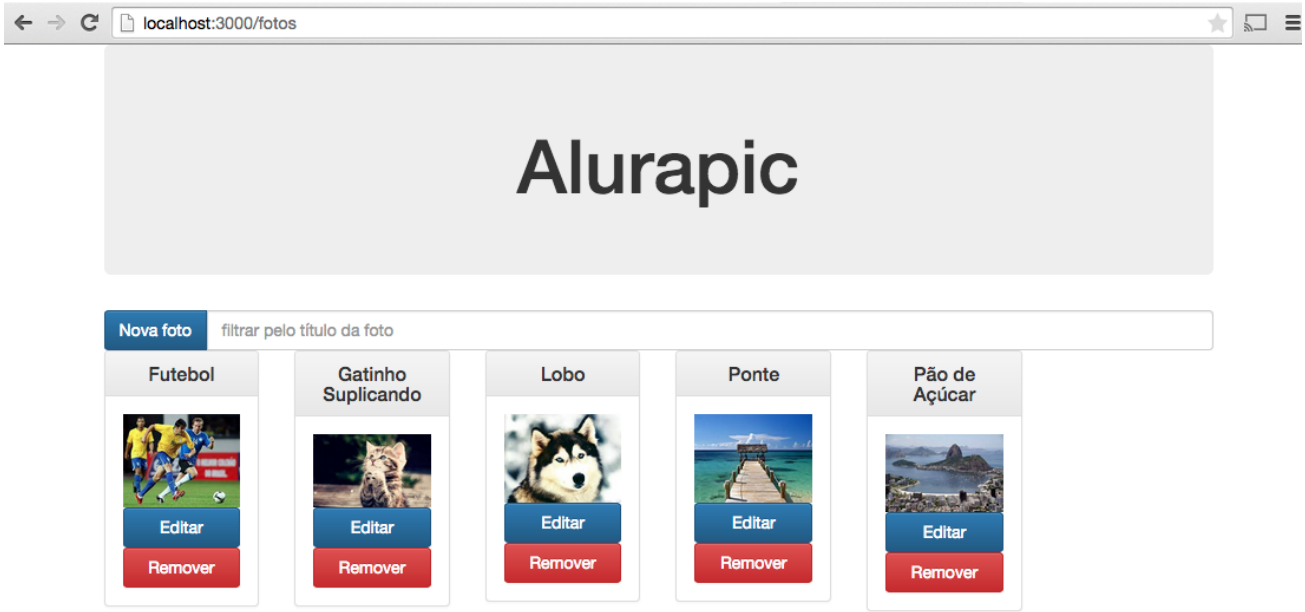
Tendo como base este cenário, frameworks MVC client-side foram criados. Entre eles temos o Backbone (http://backbonejs.org), Ember (http://emberjs.com), Knockout (http://knockoutjs.com/), CanJS (http://canjs.com), Batman (http://batmanjs.org/), entre outros.

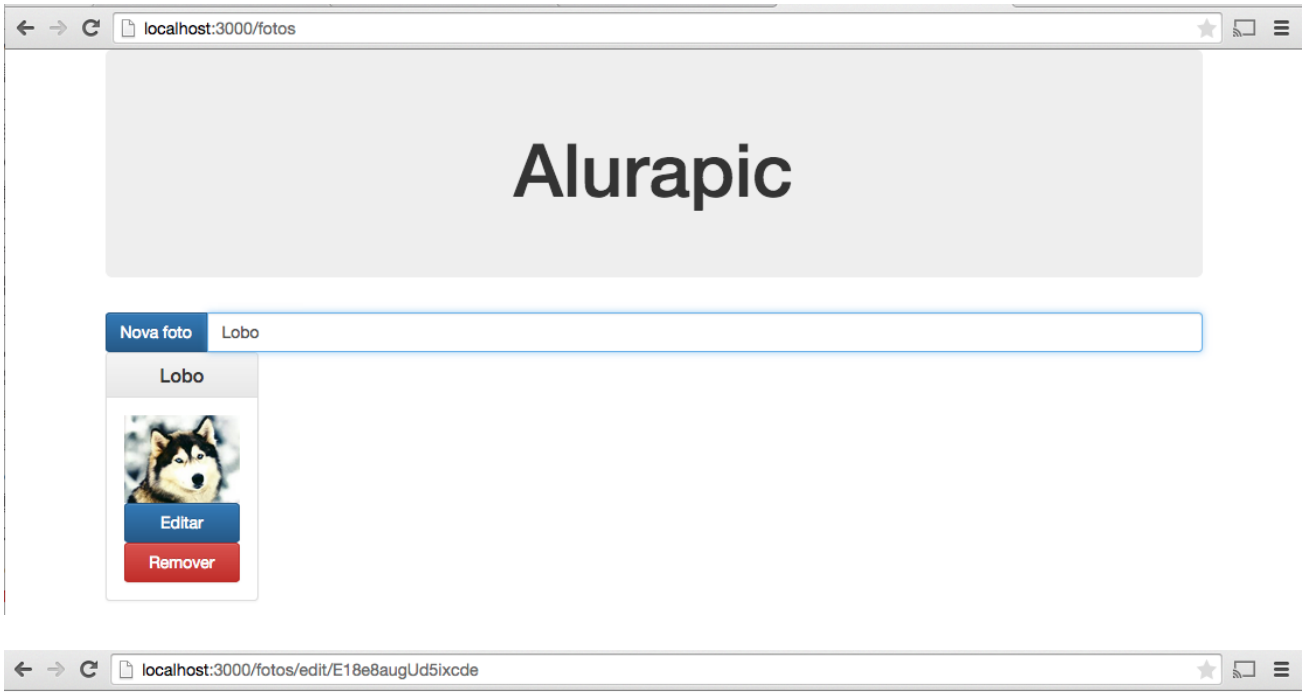
Angular, o framework MVC da Google

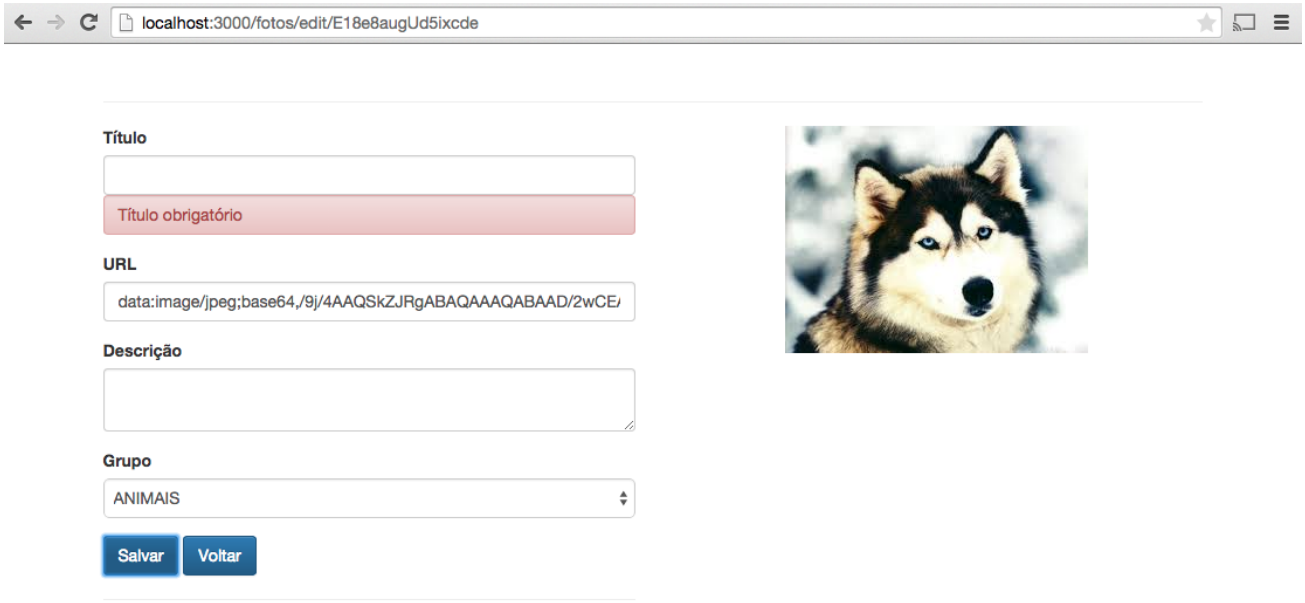
Um framework MVC no lado do cliente que tem ganhado muita atenção da comunidade é o Angular. Criado como um projeto interno da Google e liberado para o público em 2009, ele tem como foco a criação de **Single Page Applications** (SPA's). Este tipo de aplicação não recarrega a página durante seu uso, dando uma experiência mais fluída para o usuário. Não se preocupe se você é novo para este tipo de aplicação, você terá a oportunidade de entender melhor seus conceitos ao longo do treinamento.

Conhecendo um pouco da nossa aplicação, o Alurapic

Durante o treinamento construiremos a aplicação **Alurapic**, um sistema simples de gerenciamento de imagens, permitindo que o usuário busque por aquelas que seguem determinado critério. Mas não se engane: o domínio do problema, apesar de simples, será suficiente para empregarmos grande parte do "arsenal" que o Angular nos fornece, inclusive toda aplicação funcionará em cima de um servidor web já preparado.







Instalando e configurando toda infra necessária

Agora que você já conhece um pouco sobre a aplicação que construiremos, saiba que alguns recursos do Angular dependem de um servidor web para funcionarem, em nosso caso, um servidor local. A boa notícia é que já disponibilizamos um para você, livrando-o dos seus detalhes de configuração. Inclusive ele fará persistência de dados sem que você tenha que instalar um banco de dados específico para isso. Porém, para tudo funcionar, você precisa ter instalado o Node.js.

O arquivo do projeto, o tutorial de instalação do Node.js e as instruções de como levantar o servidor estão no primeiro exercício deste capítulo (https://cursos.alura.com.br/course/angularjs-mvc/task/9014). Esta é uma boa hora de fazê-lo antes de continuar. Vamos assumir a partir deste ponto que você realizou este primeiro exercício.

Por onde começar?

Angular é um framework que roda no lado do cliente, sendo assim, como qualquer outro script, deve ser importado na página que desejamos eleger como principal da aplicação, em nosso caso, a página já existente, `index.html`, que está salva dentro da pasta `alurapic/public` (não sabe que pasta é essa? Você fez o primeiro exercício

(https://cursos.alura.com.br/course/angularjs-mvc/section/2/task/3) do capítulo conforme solicitado?). Primeiramente vamos dar uma olhada em sua estrutura:

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="pt-br">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <title>Alurapic</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css">
    </head>
    <body>
        <div class="container">
            <h1 class="text-center">Alurapic</h1>
        </div> <!-- fim container -->
    </body>
</html>
```

Repare que a página já importa o CSS do Bootstrap (http://getbootstrap.com/). Se você ainda não o conhece, fique sabendo que ele permite aplicar um visual profissional em nosso projeto, com zero de esforço, apenas adicionando classes declaradas em seu arquivo CSS. Que classes são essas? Muita calma nessa hora! Elas serão introduzidas à medida que formos evoluindo nossa aplicação, mas já podemos adiantar que a classe `container` centraliza todo conteúdo da página e a `text-center` centraliza um elemento do tipo `block`, em nosso caso, a tag `h1`.

Agora que você já entendeu o papel do Bootstrap em nosso projeto, já podemos continuar. Todas as páginas, bibliotecas, scripts e qualquer outro arquivo dentro da pasta `alurapic/public` serão acessíveis através do seu navegador, inclusive já temos dentro da pasta `alurapic/public/js/lib` todos os arquivos do Angular que importaremos à medida que formos precisando. Vamos importar o script `angular.min.js`, o *core* do Angular dentro da tag `head`:

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="pt-br">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <title>Alurapic</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css">
        <script src="js/lib/angular.min.js"></script>
    </head>
    <body>
        <div class="container">
            <h1 class="text-center">Alurapic</h1>
        </div> <!-- fim container -->
    </body>
</html>
```

Pronto, mas ainda não escrevemos um código que utiliza o Angular! Para fazermos isso, primeiro precisamos criar um módulo.

Criando o alicerce da nossa aplicação, o módulo principal

A história é a seguinte: o framework nos ajuda a separar nosso código em pequenos grupos de código que podem ser combinados e reaproveitados quando necessário, esses grupos são chamados de **módulos**. Uma aplicação pode ter um, dez ou até mesmo mais de cinquenta módulos, tudo dependerá da complexidade da aplicação. Porém, há sempre um módulo que é o primeiro a ser inicializado assim que sua página é carregada pela primeira vez, inclusive ele também é o responsável pelo carregamento de outros módulos de que sua aplicação precisa para funcionar. É este que criaremos agora!

Vamos criar o arquivo `public/js/main.js` e mesmo sem escrever qualquer linha de código, vamos importá-lo sem demora na página `index.html` , abaixo da importação do core do Angular, para não corrermos o risco de esquecermos de importá-lo:

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="pt-br">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <title>Alurapic</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css">
        <script src="js/lib/angular.min.js"></script>
        <script src="js/main.js"></script>
    </head>
    <body>
        <div class="container">
            <h1 class="text-center">Alurapic</h1>
        </div> <!-- fim container -->
    </body>
</html>
```

Pronto, e agora? Como criamos nosso módulo? Bem, o Angular disponibiliza para nós no escopo global o objeto `angular` (nada criativo, não?) que possui uma série de funções que nos permite interagir com o framework, entre elas a função **module** responsável pela criação de módulos.

Você deve estar pensando "Ok, entendi que é através desse objeto global que eu crio módulos e um monte de coisas do framework, mas escopo global não é algo ruim?". Preocupação justa, mas não se preocupe! Em uma aplicação bem-feita em Angular, este é o único objeto disponível globalmente, todo restante fica confinado dentro dos módulos do Angular! Com o tempo isso ficará ainda mais claro para você.

Criando o módulo principal da aplicação:

```
// public/js/main.js

angular.module('alurapic', []);
```

Acabamos de criar nosso primeiro módulo. Perceba que a função `module` recebe dois parâmetros. O primeiro é o nome do módulo que desejamos criar e o segundo é uma array com todos os módulos de que nosso módulo depende. Como não avançamos ainda com o projeto, não temos nenhuma dependência ainda, porém, você **não deve omitir** este parâmetro. Mais tarde você entenderá o que acontece quando ele é omitido (suspense!).

Ensinando um truque novo para o navegador

Tudo certo, criamos nosso módulo, mas como o Angular saberá que deve carregá-lo? Será que importar o script `public/js/main.js` é suficiente? Com certeza não.

Precisamos indicar em nossa página qual será o escopo de atuação do Angular, isto é, se ele gerenciará a página inteira ou apenas parte dela. Isso é importante, porque outro framework MVC pode estar sendo utilizado (algo raro, porém pode acontecer) e não queremos que o Angular bagunce seu trabalho.

Como apenas utilizaremos o Angular, gerenciaremos a página inteira, isto, a tag `html` e todos seus elementos filhos! Tudo bem, mas como faremos essa associação? Através do atributo **ng-app**:

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="pt-br" ng-app="alurapic">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <title>Alurapic</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css">
        <script src="js/lib/angular.min.js"></script>
        <script src="js/main.js"></script>
    </head>
    <body>
        <div class="container">
            <h1 class="text-center">Alurapic</h1>
        </div> <!-- fim container -->
    </body>
</html>
```

Repare que o atributo `ng-app` tem como valor o nome do nosso módulo e não poderia ser diferente. Quando nossa página é carregada pela primeira vez o Angular encontrará esse atributo e carregará o módulo `alurapic`, tudo isso automaticamente, sem termos que nos preocupar em carregá-lo! É claro que se você esquecer de importar o arquivo `main.js` o Angular não será capaz de carregar o módulo, certo?

Ah, então isso é uma diretiva?

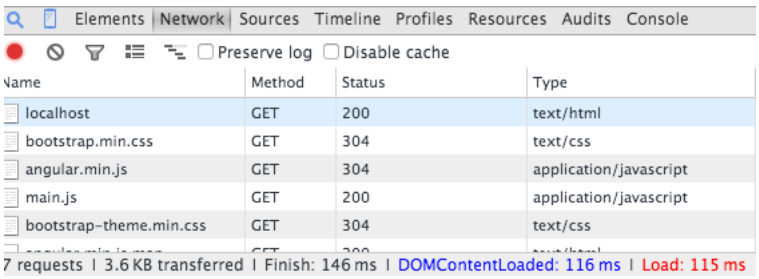
Agora, só uma coisa antes de continuarmos: o atributo `ng-app` existe no mundo HTML? Com certeza não, ele não faz parte da especificação da W3C. O "atributo" `ng-app` é na verdade uma **diretiva** do Angular.

Diretivas ampliam o vocabulário do navegador, ensinando-o novos truques, inclusive aprenderemos a criar nossas próprias diretivas ao longo do treinamento! Nesse caso a diretiva `ng-app` fornece a capacidade de nossa página carregar/iniciar o módulo principal da aplicação. Aliás, não vamos mais usar o termo página, usaremos **view**, termo mais correto quando estamos no universo do Angular!

Nossa primeira página dinâmica

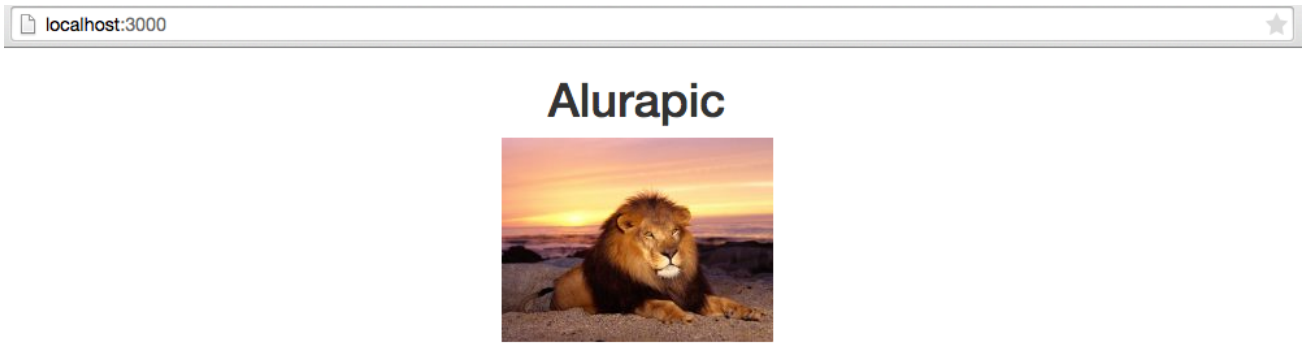
Legal, agora, com o servidor iniciado (não lembra como iniciá-lo? Veja o primeiro exercício (https://cursos.alura.com.br/course/angularjs-mvc/task/9014) do capítulo) vamos acessar o endereço `http://localhost:3000` e verificar o resultado. Como esperado, nada impressionante, pois apenas preparamos a

infraestrutura mínima de uma aplicação em Angular, que nada faz. No máximo, podemos ver através do console do navegador (eu uso Chrome, e você?) todos os arquivos carregados:



Bom, vamos começar a dar um colorido para nossa view `index.html`. Vamos adicionar uma foto, você pode escolher a URL de uma específica, não precisa ser igual a minha. Na tag `img`, utilizaremos as classes `img-responsive center-block` para que ela escale corretamente nos mais diversos dispositivos:

```
<!DOCTYPE html>
<html lang="pt-br" ng-app="alurapic">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <title>Alurapic</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css">
        <script src="js/lib/angular.min.js"></script>
        <script src="js/main.js"></script>
    </head>
    <body>
        <div class="container">
            <h1 class="text-center">Alurapic</h1>
            <img class="img-responsive center-block" src="http://www.fundosanimais.com/Minis/le
        </div><!-- fim container -->
    </body>
</html>
```



Bravo! Mas isso não impressiona, além do mais, se tivéssemos 100 imagens teríamos que repetir a tag `img` 100 vezes! Em nossa aplicação aprenderemos a cadastrar informações de imagens e a partir desses dados cadastrados montaremos dinamicamente uma lista de imagens! Só que antes de pensar em integração com o back-end, precisamos primeiro entender como o Angular fornece dados para a nossa view e como ela se constrói a partir desses dados.

Vamos realizar uma pequena mudança na tag `img`, alterando os atributos `src` e `alt`:

```
<!DOCTYPE html>
<html lang="pt-br" ng-app="alurapic">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <title>Alurapic</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css">
        <script src="js/lib/angular.min.js"></script>
        <script src="js/main.js"></script>
    </head>
    <body>
        <div class="container">
            <h1 class="text-center">Alurapic</h1>
            <img class="img-responsive center-block" src="{{foto.url}}" alt="{{foto.titulo}}">
        </div><!-- fim container -->
    </body>
</html>
```

A primeira coisa é entender que ** abrimos lacunas em nossa view index.html** através da sintaxe `{{ }}` . Se temos uma view que agora possui lacunas, podemos chamá-la de **template**. Quando alguém envia um memorando para outra, raramente ela começa do zero, ou seja, ela adota um template, toda uma estrutura já pronta e seu único trabalho é preencher essas lacunas que variam de acordo com a situação. Isso se aplica no mundo Angular! O que acontece se visualizarmos nossa página agora? **Nenhuma imagem será exibida e nenhum erro ocorrerá!** Aliás, o termo lacuna é muito genérico, no mundo Angular usamos **Angular Expression (AE)**. Todo `{{ }}` que encontrarmos chamaremos de AE. Combinado?

Quando temos um template que precisa de algum dado através de uma AE e não encontra, simplesmente aquela expressão fica em branco. Agora, a pergunta que não quer calar: quem, no modelo MVC é o responsável em disponibilizar dados para a views? Se você respondeu **controller** acertou!

Nosso primeiro controller

Precisamos criar um controller que disponibilize para a view o dado que ela precisa, no caso um objeto que contenha como chave o título e o endereço de uma foto, por exemplo `{ titulo: 'Leão', url : 'http://www.fundosanimais.com/Minis/leoes.jpg'}` .

Lembra do nosso módulo principal da aplicação? Podemos criar um ou mais controllers diretamente nele, porém é uma boa prática declarar cada controller em arquivos separados, mesmo que eles façam parte do módulo `alurapic` . Vamos criar dentro de `public/js/controllers/fotos-controller.js` .

Criando nosso controller:

```
// public/js/controllers/fotos-controller.js

angular.module('alurapic').controller('FotosController', function() {
  // definição do controller aqui
});
```

Veja que chamamos novamente `angular.module` , só que dessa vez sem passar o segundo parâmetro, o array vazio. Quando fazemos isso, indicamos que queremos acessar o módulo e não criar um novo. Faz todo sentido, se queremos que nosso controller faça parte do módulo `alurapic` . Em seguida encadeamos uma chamada à função **controller** que recebe dois parâmetros. O primeiro é o nome do controller que estamos criando na convenção *PascalCase*, o segundo uma função que define o controller.

Sabemos que o controller deve fornecer o objeto `foto` para a view e que esse objeto deve conter as chaves `titulo` e `url` . Faremos isso agora!

```
// public/js/controllers/fotos-controller.js

angular.module('alurapic').controller('FotosController', function() {

    var foto = {
        titulo : 'Leão',
        url : 'http://www.fundosanimais.com/Minis/leoes.jpg'
    };

});
```

Excelente, agora precisamos importar o novo arquivo js que acabamos de criar em nossa view principal `index.html` :

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="pt-br" ng-app="alurapic">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <title>Alurapic</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css">
        <script src="js/lib/angular.min.js"></script>
        <script src="js/main.js"></script>
        <script src="js/controllers/fotos-controller.js"></script>
    </head>
    <body>
        <div class="container">
            <h1 class="text-center">Alurapic</h1>

             <img class="img-responsive center-block" src="{{foto.url}}" alt="{{foto.titulo}}">

        </div><!-- fim container -->
    </body>
</html>
```

Será que isso é suficiente? Não, precisamos indicar dentro da nossa view `index.html` qual fragmento será gerenciado pelo nosso controller. Angular permite associarmos diferentes controllers para diferentes partes de nossa view, uma maneira de separar responsabilidades. Porém, neste exemplo, queremos que o controller gerencie a tag `body` e todos os seus elementos filhos e fazemos isso através da diretiva **ng-controller**, que deve ter como valor o nome exato do controller que criamos:

```
<!DOCTYPE html>
<html lang="pt-br" ng-app="alurapic">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width">
        <title>Alurapic</title>
        <link rel="stylesheet" href="css/bootstrap.min.css">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css">
        <script src="js/lib/angular.min.js"></script>
        <script src="js/main.js"></script>
        <script src="js/controllers/fotos-controller.js"></script>
    </head>
    <body ng-controller="FotosController">
        <div class="container">
            <h1 class="text-center">Alurapic</h1>

             <img class="img-responsive center-block" src="{{foto.url}}" alt="{{foto.titulo}}">

        </div><!-- fim container -->
    </body>
</html>
```

Será que funciona? Visualizando no navegador, a imagem não é exibida e também não temos uma mensagem de erro. Por quê? O objeto `foto` não foi criado dentro do nosso controller? Sim, mas como ele foi declarado com `var` dentro de uma função, ele possui escopo privado. Ah, então para resolvermos vamos remover `var` , fazendo com que ele caia no escopo global? Nem pensar! Porém, o Angular disponibiliza uma ponte de ligação entre o controller e a view chamada `$scope` e tudo que for "pendurado" neste objeto será enxergado pela view. Mas como teremos acesso a esse objeto tão especial dentro do mundo Angular? Pedindo! Como? Recebendo-o na função que declara o controller:

```
angular.module('alurapic').controller('FotosController', function($scope) {

});
```

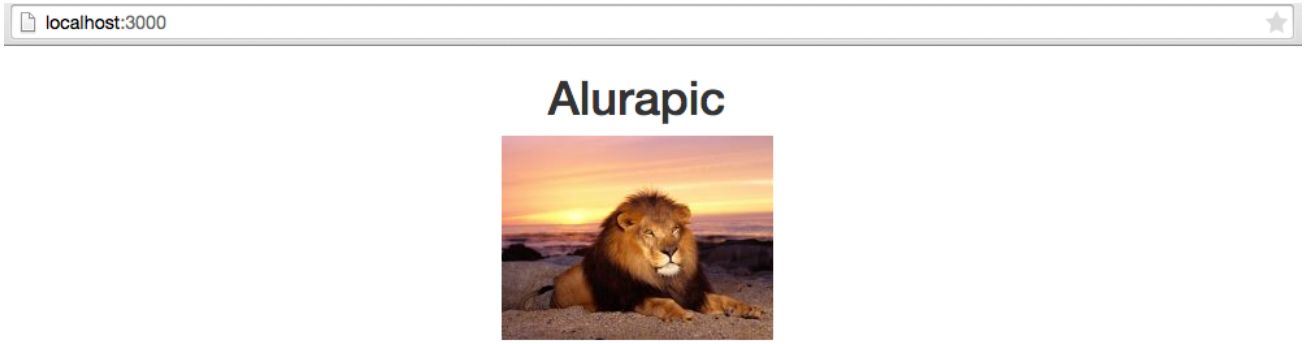
Angular encontra na declaração do controller `$scope` e sabe que tem que criar um para nós. Se tivéssemos escrito o nome do parâmetro de outra maneira, o framework não o criaria. Ou seja, Angular sabe o que deve buscar de sua infraestrutura de acordo com o nome do parâmetro que recebemos em nosso controller. Agora que já temos acesso à `$scope` , a ponte de ligação do controller com a view, podemos pendurar os dados da foto como sua propriedade.

```
angular.module('alurapic').controller('FotosController', function($scope) {

    $scope.foto = {
        titulo : 'Leão',
        url : 'http://www.fundosanimais.com/Minis/leoes.jpg'
    };

});
```

Lembre-se que tudo pendurado em `$scope` será acessível em nossa view, em nosso caso , através da angular expression. Duvida? Só testar e verificar o resultado.



Ah, isso é data binding?

Angular possui um termo apropriado para associação de um dado disponibilizado por um controller para a view: **data binding** (associação/ligação de dados). Qualquer alteração no dado do controller dispara uma atualização da view sem que o desenvolvedor tenha que se preocupar ou intervir no processo.

Excelente! Conseguimos um resultado semelhante ao que tínhamos antes, com a diferença de que agora a AE (Angular Expression) de nossa view foi processada com os dados fornecidos por `FotosController`. Pode parecer pouco, mas isso abre a porteira para que possamos avançar ainda mais no framework da Google.

O que aprendemos neste capítulo?

- o papel do modelo MVC
- o objeto global angular
- importação do Angular e criação do módulo principal da aplicação
- as diretivas ng-app e ng-controller
- Angular Expression (AE)
- o conceito de template
- criação de um controller
- o conceito de data binding