
Tabela de conteúdos

Introdução	1.1
Fundamentos	1.2
Funcionamento de uma Aplicação Web	1.3
HTML	1.4
CSS	1.5
JavaScript	1.6
Sintaxe	1.6.1
Tipos de dados	1.6.2
Condicionais	1.6.3
Laços de repetição	1.6.4
Funções	1.6.5
Objetos	1.6.6
HTML DOM	1.7
Eventos	1.7.1
Formulários	1.7.2
Front-end	1.8
Ferramentas	1.9
npm	1.9.1
TypeScript	1.10
Quickstart	1.10.1
Tipos de dados	1.10.2
Declaração de variáveis	1.10.3
Funções	1.10.4
Classes	1.10.5
Bootstrap	1.11
Fundamentos	1.11.1
Formulários	1.11.2
AngularJS	1.12
Iniciando	1.12.1
Diretiva ng-click	1.12.2

Diretiva ng-show	1.12.3
Diretiva ng-repeat	1.12.4
Model complexo	1.12.5
Validação de formulários	1.12.6
Serviço http	1.12.7
Módulo ngRoute	1.12.8
AngularJS Tutorial	1.13
Passo 0: Iniciando	1.13.1
Passo 1: Template estático	1.13.2
Passo 2: Templates do Angular	1.13.3
Passo 3: Filtrando Repeaters	1.13.4
Passo 4: Vinculação de Dados de Via dupla	1.13.5
Passo 5: Telas de lista e detalhes	1.13.6
Passo 6: XHR e Injeção de Dependência	1.13.7
Passo 7: Mais detalhes do telefone	1.13.8
Passo 8: Roteamento e múltiplas Views	1.13.9
Passo 9: REST e Serviços	1.13.10
Angular	1.14
Arquitetura	1.14.1
Estrutura do projeto	1.14.2
Observables	1.14.3
Servicos	1.14.4
Múltiplos componentes	1.14.5
Back-end	1.15
PHP	1.16
Sintáxe Básica	1.16.1
Tipos de Dados	1.16.2
string	1.16.2.1
array	1.16.2.2
Operadores	1.16.3
Aritméticos	1.16.3.1
Estruturas de Controle	1.16.4
if...else	1.16.4.1
while	1.16.4.2

Funções	1.16.5
Funções para manipulações de variáveis	1.16.5.1
Funções para	1.16.5.2
Funções para	1.16.5.3
Orientação a Objetos em PHP	1.16.6
Classes e Objetos	1.16.6.1
Construtores	1.16.6.2
Propriedades	1.16.6.3
Herança	1.16.6.4
Formulários HTML e PHP	1.16.7
Método GET	1.16.7.1
Método POST	1.16.7.2
Validação de Formularios	1.16.7.3
Sessão	1.16.8
Acesso a Banco de dados via PDO	1.16.9
Conexão via PDO	1.16.9.1
Inserção de dados	1.16.9.2
Seleção de dados	1.16.9.3
Alteração de dados	1.16.9.4
Exclusão de dados	1.16.9.5
Exemplos de projetos com PDO	1.16.9.6
PROJETO 1: Conexão com o banco de dados	1.16.9.6.1
PROJETO 2: operações DAO (1.16.9.6.2
API REST	2.1
Roteamento	2.1.1
Request	2.1.2
Acesso a Bancos de Dados	2.1.3
DBAL	2.1.4
Arquitetura de uma API	2.1.5

Desenvolvimento de software para a web

Este livro é um pouco mais do que uma fonte de referência em programação. Explico. As habilidades requeridas para desenvolvimento de software para a web vão além da programação. Bem mais do que entender código, o profissional que deseja lidar com este contexto de desenvolvimento de software precisa ter em mente que vai precisar entender uma pilha de tecnologias que irão, ao fim de um processo, gerar o resultado tão esperado.

É com este tipo de pensamento que iniciamos este livro.

O conteúdo do livro é dividido em partes:

- **Parte 1: Fundamentos.** Aqui você verá conteúdo sobre comunicação na Web, protocolos, HTML, CSS e JavaScript
- **Parte 2: Tecnologias Front-end.** Aqui o espaço é reservado para Bootstrap e AngularJS
- **Parte 3: Tecnologias Back-end.** Entram em ação as ferramentas para programação *server-side*, ou seja, PHP e Silex (microframework para criação de API REST).

Como um livro de programação, também estão disponíveis os códigos-fontes dos aplicativos de exemplo utilizados nos conteúdos. Para ter acesso a eles, acesse o repositório <https://github.com/jacksongomesbr/livro-web-codigo-fonte>.

De uma forma didática, os conceitos envolvidos nas áreas contempladas neste livro podem ser acompanhados por meio de um [mapa mental](#).

Fundamentos

A primeira seção deste livro, chamada **Fundamentos**, tem o objetivo de apresentar conceitos e ferramentas fundamentais para o desenvolvimento de habilidades para a construção de aplicativos web.

Em específico, este conjunto de habilidades envolve o conhecimento de:

- Comunicação cliente-servidor na web
- HTML
- CSS
- JavaScript
- HTML DOM

As seções seguintes apresentarão estes conceitos em detalhes.

Comunicação e serviços

A comunicação no ambiente web é feita sobre a internet e, portanto, utiliza protocolos de comunicação e intercâmbio de dados em rede. O principal protocolo de comunicação na internet é o HTTP (HyperText Transfer Protocol). Como protocolo, o HTTP estabelece uma série de critérios de comunicação entre duas partes, chamadas cliente e servidor. Para mais detalhes do HTTP, leia a [sua especificação](#).

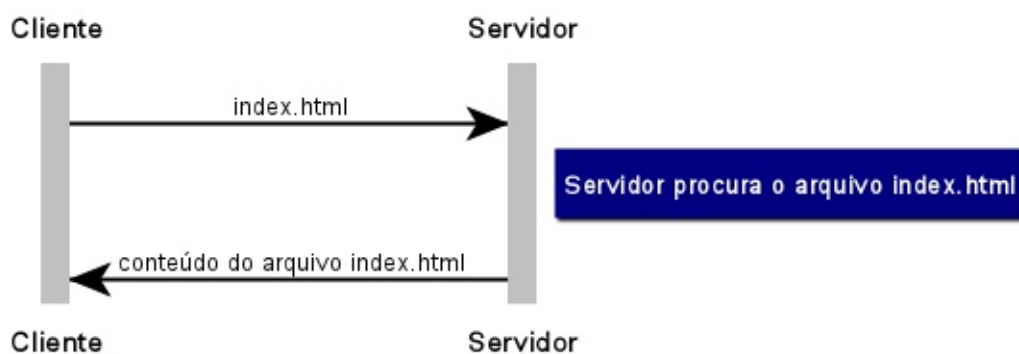
Comunicação na internet

O cliente gera uma **requisição** para o servidor, que a responde. Uma requisição é um pedido, uma solicitação para o servidor. O cliente solicita para o servidor um **recurso**, que pode ser, por exemplo, uma imagem ou um arquivo .html. Geralmente, o cliente é representado por um **browser**. Entre os browsers mais conhecidos e utilizados estão [Google Chrome](#), [Firefox](#), [Internet Explorer](#) e [Safari](#).

O servidor **responde** solicitações do cliente. É por isso que este processo de comunicação sempre se dá início no cliente. Entretanto, há tecnologias que permitem que a comunicação seja iniciada no servidor, como o serviço PUSH, utilizado em softwares que, por exemplo, enviam notificações para os clientes. Uma vez que o cliente é representado pelo browser, o servidor é representado pelo **servidor web (web server)**. Alguns dos servidores web mais conhecidos e utilizados são [Apache HTTP](#), [IIS](#) e [Nginx](#).

Este processo de comunicação cliente-servidor é conhecido como **solicitação e resposta**. A figura a seguir ilustra o processo.

Comunicação cliente-servidor na internet



www.websequencediagrams.com

Como tanto cliente quanto servidor são software (programas) é possível ter mais de um deles executando na mesma máquina (computador) e, até mesmo, é possível que tanto cliente quanto servidor estejam em execução na mesma máquina. Portanto, é importante perceber que a abstração é uma generalização e não requer que, obrigatoriamente, cliente e servidor estejam em máquinas diferentes.

Um elemento importante a ser considerado é que, embora se tenha popularizado o entendimento de utilizar a internet por meio do browser, são inúmeras as aplicações que utilizam a internet, seja para consultar ou enviar dados para uma parte remota. Por exemplo, ao utilizar um aplicativo mobile que consulta notícias a internet é utilizada. Ao consultar dados de uma rede social, o mesmo acontece. O interessante é, então, que a popularização destes recursos tornou a internet tão presente e tão inserida no dia-a-dia das pessoas que não se pode dizer que o cliente é sempre o browser. Neste sentido, recentemente (25/jun/2015) um dos presidentes do Google, Eric Schmidt, pronunciou: [A Internet vai desaparecer](#).

Desta forma, embora este curso esteja mais voltado para software cliente e servidor baseado em tecnologias da internet como HTML, parte do conteúdo aqui presente também serve para software mobile ou desktop.

Funcionamento de uma Aplicação Web

Diferentemente de uma aplicação Desktop (em uma máquina local), uma aplicação Web funciona sob uma arquitetura diferente: Cliente-Servidor. Na arquitetura Cliente-Servidor, o processamento das requisições é realizado por dois “lados” o Servidor (Server side) e Cliente (Client side). Assim, quando o usuário (cliente), através de um browser (navegador) faz acesso a uma página específica, o servidor faz o processamento da requisição, processa-a e envia de para o cliente arquivos em diferentes formatos, por exemplo, HTML ou CSS. Desta forma, uma aplicação web possui basicamente dois programas que podem ser executados:

- O código que está no servidor e responde às requisições HTTP;
- E o código que está no cliente (interpretado pelo browser) e responde às entradas dos usuários.

Códigos do “lado” do Servidor (Server side)

O código disponibilizado do “lado” do servidor possui algumas peculiaridades, dentre elas:

- São quaisquer códigos que podem ser executados no Servidor e que podem responder às requisições HTTP. Por exemplo, códigos Java, C# e PHP.
- Permite a persistência de dados;

- Não pode ser visualizado pelo usuário;
- Respondem às requisições HTTP através de uma particular URL;
- Caso a aplicação seja web, cria páginas que o usuário pode visualizar através de um navegador (browser).

Um Servidor Web tem como função receber requisições e, a partir delas, oferecer respostas para o cliente. Estas requisições podem ser realizadas através de um navegador de internet (browser) pelo usuário e, geralmente, recebem como resposta uma página HTML que é interpretada pelo browser e apresentada ao usuário.

Códigos do “lado” do Cliente (Client side)

O código disponibilizado do “lado” do cliente também possui suas particularidades, dentre elas

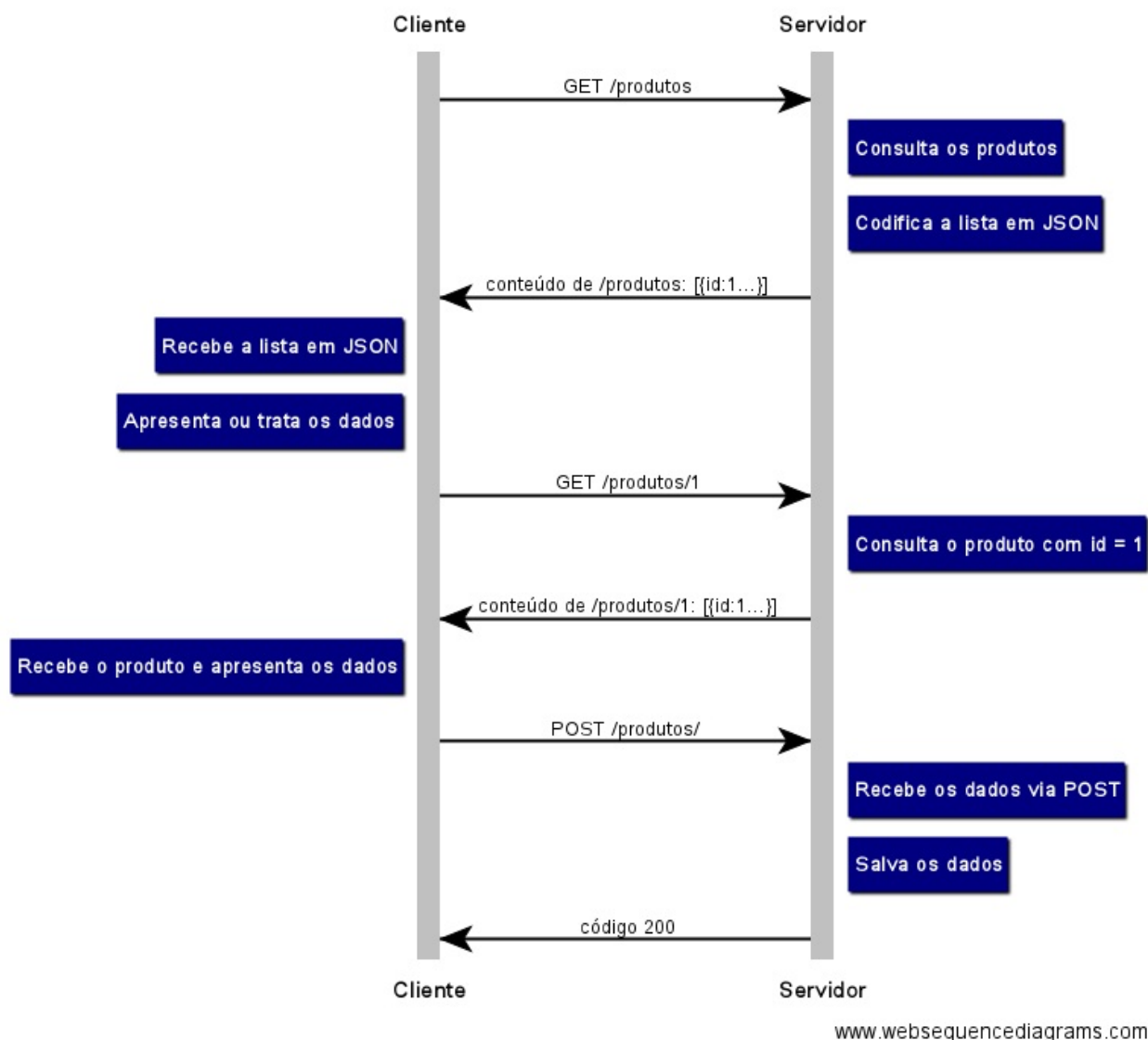
- Permite o uso das linguagens HTML, CSS e JavaScript;
- Código interpretado pelo browser do usuário;
- Reage às entradas dos usuários;
- Permite a leitura de arquivos de um servidor através de requisições HTTP;
- Torna possível a exibição de conteúdos (por exemplo, uma página web) para os usuários.

Os códigos do “lado” do cliente são analisados pelo próprio navegador do cliente e são, geralmente, gerados a partir de solicitações destes clientes ao servidor. Se por acaso o servidor hospedar um determinado documento HTML (processado do lado do cliente), o servidor então tira uma cópia deste arquivo HTML e o envia para o cliente.

REST

REST (Representational State Transfer) é estilo ou padrão de arquitetura de aplicações na internet. Surgiu como alternativa aos **Web Services**, um padrão de comunicação voltado para interoperabilidade entre sistemas baseado em HTTP e XML. Com o alto nível de complexidade dos sistemas baseados em Web Services, a utilização de REST procurou flexibilizar a comunicação entre sistemas. Principalmente, um fator importante foi a evolução dos padrões de arquitetura de software na internet e a adoção de novos padrões. Por exemplo, ao invés de utilizar exclusivamente XML, os softwares passaram a utilizar o formato **JSON**. Ao invés de complicados esquemas de comunicação e codificação de mensagens, o uso de REST popularizou o surgimento e a utilização das APIs REST. A figura a seguir ilustra a utilização de uma API fictícia.

API REST para dados de produtos



A figura ilustra três situações:

1. O cliente solicita a lista de produtos: em uma API REST o cliente está interessado nos dados, não no conteúdo de uma página, por exemplo. Assim, ao invés de pedir ao servidor o conteúdo de uma página que lista os produtos, o cliente solicita a lista de produtos. Os dados são codificados em JSON. Ao receber a lista, o cliente interpreta os dados e os apresenta.
2. O cliente solicita um produto: como no item anterior, o cliente solicita os dados do produto, não a página de produto.
3. O cliente quer cadastrar um produto: em uma API REST, o servidor trata situações de consulta e de cadastro de forma diferenciada. Neste caso, os dados são enviados para o servidor, que realiza uma atualização no banco de dados e retorna uma resposta com código HTTP 200, indicando que a requisição foi atendida com sucesso.

Alguns elementos importantes neste processo são os verbos e os códigos de status.

Verbos

Com a utilização de REST, os verbos HTTP ganham mais importância. Os verbos definem, especificamente, uma semântica de comunicação com o servidor. Os mais conhecidos ou utilizados são:

- **GET** é usado para solicitar dados.
- **POST** é usado para enviar dados.
- **PUT** é usado para enviar dados. A diferença entre POST e PUT fica a cargo de cada API. Por exemplo, a API pode definir que POST é usado para cadastro de produto, enquanto PUT é usado para atualizar dados de um produto existente.
- **DELETE** é usado para excluir dados. Por exemplo, uma requisição DELETE para a URL `/produtos/1` significa a solicitação da exclusão de um produto com identificador 1.

Códigos de status

Da mesma forma que os verbos, os códigos HTTP se tornaram mais populares. Os códigos mais utilizados são:

- **200**: indica que a solicitação foi atendida com sucesso.
- **404**: indica que a solicitação não pode ser atendida porque o recurso solicitado não foi encontrado. Por exemplo, quando o cliente solicitar `GET /produtos/1` o servidor pode responder com conteúdo vazio e código HTTP 400 para indicar que o produto solicitado não foi encontrado. O importante disso é que não é necessário retornar a mensagem textual indicando isso.
- **403**: indica que a solicitação não pode ser atendida por causa de falha de permissão. Por exemplo, quando o cliente solicitar `GET /produtos/1` o servidor pode retornar o código 403 para indicar que o cliente não pode acessar o recurso porque não possui permissão para fazê-lo.
- **500**: indica um erro no servidor. O servidor pode retornar o código 500 para, por exemplo, indicar que um acesso ao banco de dados não pode ser realizado.

API

A utilização de REST proporcionou um crescimento no número de softwares que consomem ou disponibilizam serviços por meio de APIs. Um repositório bastante popular de APIs é o www.programmableweb.com.

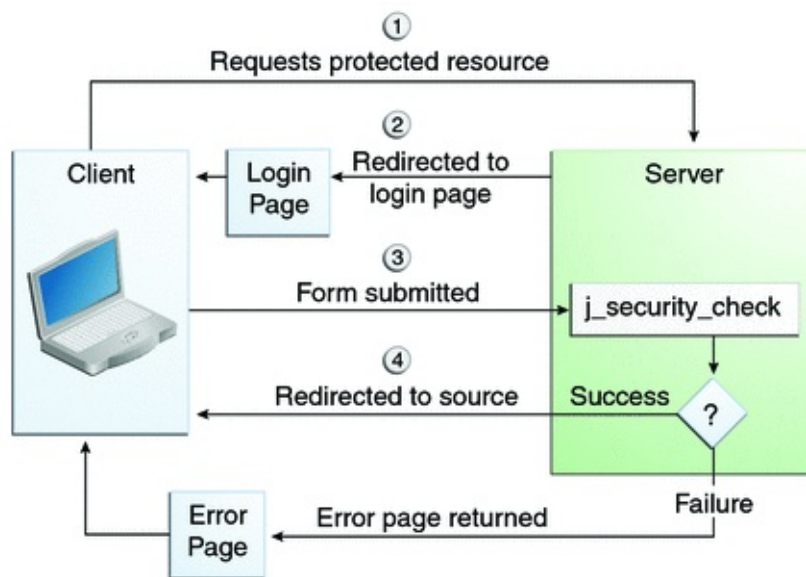
O Google é bastante conhecido por sua política de acesso aos seus produtos. Grande parte deles está disponível via API. Por exemplo, você pode criar um software que consulta os dados da agenda de um usuário, ou pode consultar as coordenadas de um ponto no mapa

a partir do endereço. Para conhecer mais sobre produtos do Google disponíveis via API acesse o [Google Developers](#).

Segurança

Quando se pensa em dados ou conteúdos disponíveis na internet, uma questão é justamente a proteção ou a garantia do método de acesso a estes dados ou conteúdos.

Basicamente, aplicações e sites web popularizaram a utilização de formulários como mecanismos de autenticação. A figura a seguir ilustra este processo.

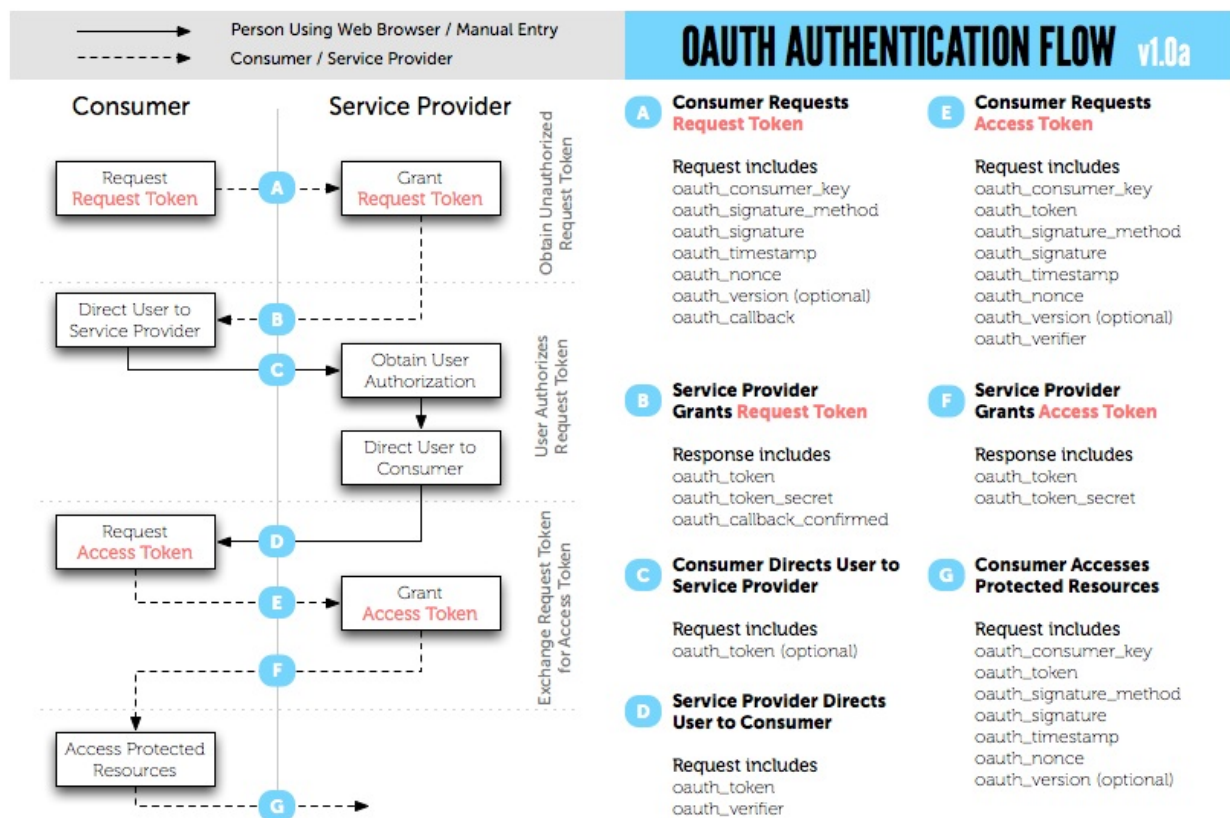


Ao acessar um recurso protegido, o servidor apresenta ao usuário uma interface de autenticação (por exemplo, um formulário por meio do qual o usuário informa seu login e senha). Se o login e a senha estiverem corretos, então o usuário continua acessando o recurso. Caso contrário, o formulário de autenticação é apresentado novamente, com uma mensagem de erro indicando a falha da autenticação.

Este processo funciona muito bem quando se trata da utilização de software por pessoas. Entretanto, no cenário de APIs a comunicação é feita via máquinas, ou seja, é software interagindo com software. Desta forma, novos mecanismos de autenticação precisaram ser estabelecidos, dentre os quais os mais populares são:

- Utilização de Cookies (de sessão ou persistentes)
- Autenticação baseada em POST (dados de autenticação ou identidade são enviados via POST em toda requisição)
- Autenticação baseada em OAuth.

Dentre estes mecanismos, o mais utilizado atualmente é, sem dúvidas, o protocolo [OAuth](#). OAuth é um padrão aberto de comunicação que estabelece formas rígidas e bem definidas para autenticação entre sistemas. A imagem a seguir ilustra a autenticação via OAuth.



Neste cenário, são comuns termos como **consumer**, **service provider** e **token**.

Por exemplo, ao desenvolver software que consome dados do Twitter, do Google Plus ou do Facebook, você precisará seguir um processo (considerado burocrático, até) para acessar os dados. Geralmente isso envolve o cadastro do seu software (também chamado **app**), a solicitação de uma chave e a utilização dela para a comunicação com a API.

É importante informar que a utilização deste mecanismo se dá mais por causa de uma característica do próprio REST: geralmente, não utiliza cookies de sessão. Desta forma, o cliente precisa solicitar um **token** para o servidor e o utilizará durante o processo de comunicação (que pode ser maior do que uma sessão tradicional, de 30min, por exemplo).

HTTPS

Para finalizar o conjunto de ferramentas e conceitos utilizados na comunicação na internet, um recurso adicional de segurança é o HTTP seguro, ou HTTPS. Na verdade, é comum uma API ser acessada apenas via HTTPS, ao invés de HTTP. O interessante em utilizar este mecanismo é que os dados trocados entre cliente e servidor são transmitidos via um canal de comunicação seguro, baseado na utilização de algoritmos criptográficos e dos **certificados digitais**. A figura a seguir ilustra este processo.



Para saber mais, leia este [artigo sobre HTTPS na Wikipedia](#).

HTML

[HTML \(HyperText Markup Language\)](#) é, basicamente, a linguagem base e universal para conteúdo na internet. A versão atual é chamada HTML5 e utilizar seus recursos não é, na verdade, uma obrigação. Entretanto, aprender os elementos principais permite que o conteúdo da página seja melhor organizado semanticamente.

Estrutura padrão

O código a seguir demonstra a estrutura de um documento HTML5.

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
<meta charset="UTF-8">
<title>Título do documento</title>
</head>

<body>
Conteúdo do documento.....
</body>

</html>
```

Na linha 1, a instrução de processamento `DOCTYPE` indica que o conteúdo do arquivo é HTML5.

Na linha 2, o elemento `html` possui o atributo `lang="pt-BR"`, indicando que o idioma usado no documento é Português do Brasil.

Na linha 4, o elemento `meta` possui o atributo `charset="UTF-8"`, indicando que o texto está codificado no padrão UTF-8.

Elementos

Os elementos mais importantes do HTML5 são:

- **Semânticos:** `header`, `footer`, `article` e `section`
- **Controles de Formulário:** tipos numéricos, data e hora, calendário e intervalo
- **Gráficos:** `svg` e `canvas`
- **Multimídia:** `audio` e `video`

Recursos da API de programação

Além de elementos, o HTML5 disponibiliza novos recursos para o desenvolvedor:

- Geolocalização
- Arrastar e soltar
- Armazenamento local
- Cache da aplicação
- Web workers
- Notificações Enviadas pelo Servidor

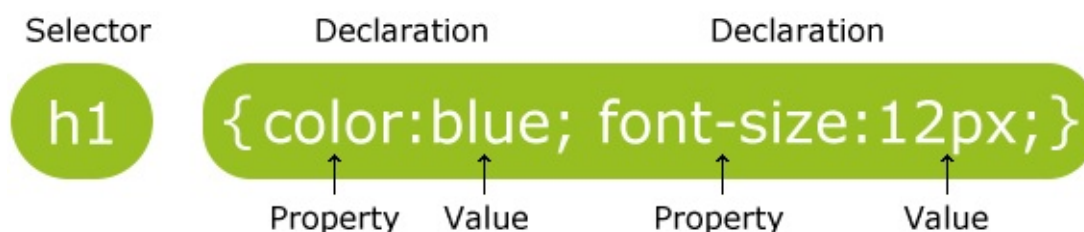
Para saber mais sobre o HTML5, veja o [curso de HTML do W3Schools](#).

CSS

CSS (Cascading Style Sheets) é a linguagem utilizada para formatação de conteúdo na internet. Enquanto o HTML define o conteúdo, em si, CSS é utilizado para estilizar ou apresentar o conteúdo. Desta forma, o objetivo de um documento CSS é definir regras de formatação para um conteúdo.

Sintaxe

Uma regra CSS consiste de um **seletor** e de um **bloco de declaração**:



Seletores

O seletor, como o nome indica, permite selecionar o elemento ou o conjunto de elementos do HTML que serão afetados pelas regras de formatação presentes no bloco de declaração. O seletor pode ser:

- **seletor de elemento:** exemplo: `h1` aplica-se ao elemento `h1`
- **seletor de id:** exemplo: `#pagina` aplica-se ao elemento com `id="pagina"`
- **seletor de classe:** exemplo: `.paragrafo` aplica-se a todos os elementos que tiverem `paragrafo` como uma de suas classes (atributo `class`).

Independentemente do seletor, ele também pode ser combinado, aplicando a regra a vários elementos. Para isso, basta separar os seletores por vírgula. Exemplo: `h1, .paragrafo, #pagina { ... }` aplica-se ao elemento `h1` , a todos os elementos que tiverem a classe `paragrafo` e ao elemento com id `pagina` .

O `id` é geralmente único no documento. A classe pode ser aplicada a mais de um elemento e um elemento pode ter mais de uma classe. Exemplo:

```
<p class="paragrafo destaque">...</p>
```

O elemento `p` possui as classes `paragrafo` e `destaque` .

Bloco de declaração

O bloco de declaração contém as declarações, separadas por ponto-e-vírgula. Cada declaração é composta por **propriedade** e **valor**. Parte da chave, bem como da dificuldade, em aprender CSS está em saber as propriedades disponíveis (que podem variar conforme o browser), seus valores possíveis e como eles afetam a apresentação do conteúdo.

Para saber mais sobre o CSS, consulte o [curso de CSS da W3Schools](#).

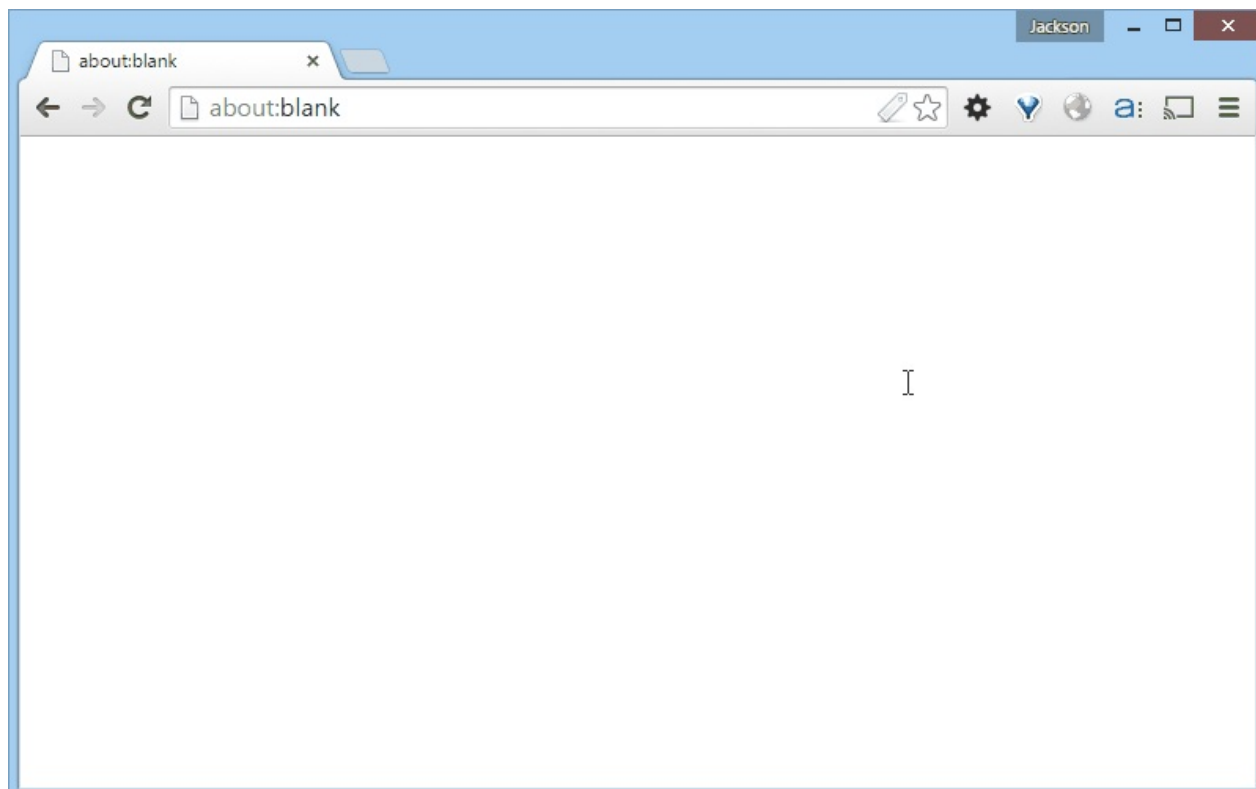
JavaScript

JavaScript (ou ECMAScript) é a principal linguagem de programação da internet. As semelhanças com a linguagem Java são bastante notáveis, bem como as diferenças.

Execução no browser

JavaScript é uma linguagem executada no browser de forma interpretada. Isso quer dizer que o código JavaScript não é compilado, mas interpretado em tempo de execução.

A ordem de execução do código é feita de cima para baixo, conforme o código estiver presente no documento. Entender esta ordem de execução é importante para facilitar não apenas o entendimento da execução do programa, mas também a correção de erros. Por ser uma linguagem interpretada, a execução das linhas subsequentes a uma linha com erro não é efetuada. Isso pode gerar problemas na execução do código. Para aprender melhor como lidar com o fluxo de execução do código, você pode utilizar a função **inspetor** do seu browser (está disponível na maioria dos navegadores modernos). A animação a seguir demonstra como utilizar o inspetor (ativado ao pressionar a tecla F12).



Inclusão no documento

Código JavaScript é inserido no documento por meio do elemento `script` . Exemplo:

```
<html>
<head>
  <script>
    alert('Javascript em execução!');
  </script>
</head>
<body>
</body>
</html>
```

Código externo

Além de incluir o código JavaScript diretamente no documento, junto do HTML, você pode manter o código em um arquivo separado, com a extensão `.js` e incluí-lo no seu documento. No exemplo a seguir, o código está no arquivo `script.js` e é incluído no documento por meio do elemento `script` e o atributo `src` :

```
<script src="script.js"></script>
```

Para aprender mais sobre o JavaScript, acesse o [curso de JavaScript do W3Schools](#).

Aqui no curso, o conteúdo de JavaScript será dividido entre entendimento da sintaxe, manipulação do HTML e utilização de bibliotecas e ferramentas, como jQuery e AngularJS.

Entender a sintaxe do JavaScript é uma tarefa fundamental para o andamento do curso. A seção seguinte (Javascript) apresenta os elementos principais da linguagem:

- [sintaxe](#)
- [tipos de dados](#)
- [condicionais](#)
- [laços de repetição](#)
- [funções](#)
- [objetos](#)

Sintaxe

As instruções de programa em JavaScript são dispostas em linhas, terminadas, opcionalmente, em ";" (ponto-e-vírgula). Exemplo:

```
var x = 5;
var nome = "José da Silva";
if (x == 5) {
    alert(x);
}
```

Variáveis

JavaScript, por ser uma linguagem interpretada, consegue fazer alocação dinâmica e automática de variáveis. Isso quer dizer que não é necessário, obrigatoriamente, declarar uma variável para utilizá-la. Entretanto, é uma boa prática de programação fazê-lo. Para declarar uma variável é utilizada a palavra `var`. Exemplo:

```
var x;
```

A atribuição de valor a uma variável ocorre por meio do **operador de atribuição**, o sinal `=`. Exemplo:

```
var x = 5;
x = 6;
```

Operadores

Os operadores aritméticos estão disponíveis:

Operador	Descrição
/	Divisão
*	Multiplicação
+	Adição
-	Subtração
%	Módulo
++	Incremento
--	Decremento

O operador `+` pode ter comportamento diferente conforme o tipo de dados. Se o dado for numérico, será realizada uma soma, se for literal (string) será realizada uma concatenação.

Precedência dos operadores aritméticos:

Operador	Precedência
()	Agrupamento de expressões
++ e --	Incremento e Decremento
*, / e %	Multiplicação, divisão e módulo
+ e -	Adição e subtração

Operadores de comparação e lógicos:

Operador	Descrição
==	Igual
===	Igual (valor e tipo iguais)
!=	Diferente
!==	Diferente (valor e tipo diferentes)
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que

Comentários

Comentários podem ser feitos:

- por linha
- por múltiplas linhas

Comentários por linha são feitos por meio de `//` . Exemplo:

```
var x = 5; // declara variavel x
// comentario de inicio de linha
```

Comentários de múltiplas linhas podem ser feitos iniciando-se com `/*` e concluindo com `*/` . Exemplo:

```
/* este é
um comentário
de múltiplas
linhas */
```

Identificadores

Identificadores (nomes de variáveis ou funções) seguem as seguintes regras:

- o primeiro caracter precisa ser: uma letra, um `_` ou um `$`
- caracteres subsequentes podem ser letras, dígitos, `_` ou `$`

Case Sensitive

JavaScript é uma linguagem "case sensitive", isto é, letras maiúsculas e minúsculas em identificadores são tratadas diferentemente. Por exemplo, as duas variáveis a seguir são diferentes:

```
var primeiroNome = 'Jose';
var primeironome = 'Jose';
```

Tipos de dados

Por ser uma linguagem interpretada, JavaScript não é fortemente tipada. Isso significa que as variáveis não possuem tipos ao serem declaradas e seu tipo é definido conforme seu valor. Os tipos considerados pelo JavaScript são: números, strings, booleanos e objetos. Exemplos:

```
var idade = 18; // numero
var temperatura = 37.5; // numero
var x = 10e2; // numero
var y = 10e-5; // numero
var nome = 'Jose'; // string
var v = true; // booleano
var f = false; // booleano
var carros = ['fusca', 'ferrari', 'gol']; // objeto (array)
var pessoa = { idade : 18, nome : 'Jose' }; // objeto
```

Além disso, o código a seguir também é aceito:

```
var x = 15;
x = 'Nome';
x = 1.8;
x = [1, 2, 3];
```

Em relação a strings, elas podem ser representadas utilizando aspas simples ou duplas. Exemplos:

```
var nome = 'Jose da Silva';
var local = "Arraial d'Ajuda";
local = "Arrail d\'Ajuda";
var mensagem = "\"José\" é o meu nome";
```

Os exemplos acima ilustram representações de strings, utilizando aspas simples e duplas e demonstram a utilização da `\` como forma de representar caracteres especiais dentro de strings.

Exercício

Qual o valor da expressão `10 + 8 + 'Anos'` ?

Arrays

Arrays (listas de valores) são representados em JavaScript por meio dos caracteres `[` e `]`, com valores separados por `,`. Exemplo:

```
var n = [1, 2, 3, 4, 5];
var balaio = ["gato", 1, {id:1}, false];
```

No último exemplo do código acima, vemos que um array pode conter valores com tipos diferentes.

A indexação de arrays inicia-se em 0, e a forma de acessar um índice é informando o índice entre `[]`. Exemplo:

```
var n = [1, 2, 3];
n[0] = 0;
```

Objetos

Objetos são representados com pares nome:valor, separados por vírgula, entre `{}`. Exemplo:

```
var pessoa = {id: 1, nome: 'José'};
pessoa.id = 2; // alterando valor do atributo do objeto
pessoa.idade = 18; // atribuir valor para atributo não declarado não gera erro e criar
á o atributo no objeto
```

Operador `typeof`

Em se tratando de valores, um operador bastante útil é o `typeof`, pois permite saber o tipo de um valor. Exemplo:

```
typeof "José" // retorna string
typeof 3.14 // retorna number
typeof true // retorna boolean
typeof [1, 2, 3] // retorna object
typeof {nome : "José" } // retorna object
```

Valores especiais

Em Javascript uma variável declarada sem valor possui, por padrão, o valor `undefined` (do tipo `undefined`). Exemplo:

```
var nome;
```

Ao atribuir um valor `undefined` a uma variável, o procedimento equivale a "limpar" a variável.

Outro valor especial é o `null`, que significa algo que não existe. Na verdade, por mais estranho que possa parecer, em JavaScript, o valor `null` é do tipo `object`. Desta forma, o valor `null` é utilizado quando se tratam de objetos. Exemplo:

```
var pessoa = { id : 1 };  
pessoa = null;
```

Na prática, `null` e `undefined` são iguais em valor, mas diferentes em tipo.

Exercício

Como identificar se uma variável é array?

Funções para conversões de tipos

As funções `parseInt()` e `parseFloat()`, respectivamente, convertem strings em números inteiros e reais. Para converter um valor para string, utiliza-se a função `toString()`.

Exemplos:

```
var n = '1';  
n = parseInt(n); // converte para numero  
var m = '1.5';  
m = parseFloat(m); // converte para numero  
  
var s = n.toString(); // converte para string
```

Exercício

Crie um objeto que represente a seguinte situação: um aluno é identificado pelo número acadêmico e possui nome, e-mail, data de nascimento, endereço e uma lista de turmas; cada turma é identificada por um número e possui nome da disciplina, ano e semestre; para cada turma, o aluno possui quatro notas.

Condicionais

As instruções condicionais em JavaScript utilizam as palavras `if`, `else`, e `switch`.

A instrução `if`

A instrução `if` permite definir um condicional que executa um código apenas se o condicional for verdadeiro. A sintaxe é a seguinte:

```
if (condicao) {  
    código a ser executado se condicao for true  
}
```

Exemplo:

```
var x = 5;  
if (x == 5) {  
    mensagem = "x é igual a cinco";  
}
```

Neste exemplo, a variável `x` possui valor `5`. Como o condicional da instrução `if` é baseado no valor de `x`, executando o código apenas se `x` for igual a `5`, a execução executa o código interior, pois o condicional é verdadeiro.

A instrução `else`

A instrução `else` não deve ser utilizada desacompanhada da instrução `if`. Na verdade, a instrução `else` complementa a instrução `if`, executando um bloco de código quando o condicional do `if` não for verdadeiro. A sintaxe é a seguinte:

```
if (condicao) {  
    código a ser executado se condicao for true  
} else {  
    código a ser executado se o condicao for false  
}
```

Exemplo:

```
var x = 1;
if (x == 5) {
    mensagem = "x é igual a cinco";
} else {
    mensagem = "x é diferente de cinco";
}
```

Neste exemplo, a variável `x` possui valor `1`. A execução do código fará com que o bloco do `else` seja executado, pois o condicional `x == 5` é falso.

A instrução else if

De forma similar às instruções anteriores, a instrução `else if` executa um código se uma condição for verdadeira. A sintaxe é a seguinte:

```
if (condicao1) {
    código a ser executado se condicao1 for true
} else if (condicao2) {
    código a ser executado se condicao2 for true e condicao1 for false
} else {
    código a ser executado se condicao1 e condicao2 forem false
}
```

Exemplo:

```
var x = 5;
if (x > 10) {
    mensagem = "x é maior que 10";
} else if (x > 5 && x < 10) {
    mensagem = "x é maior que cinco e menor que 10";
} else {
    mensagem = "x é menor que cinco";
}
```

A instrução switch

A instrução `switch` permite executar código com base em diferentes condições. De certo modo, é similar ao uso da estrutura `if...else if...else`. A sintaxe é:

```
switch (expressao) {  
  case n1:  
    código a ser executado se expressão for igual a n1  
    break;  
  case n2:  
    código a ser executado se expressão for igual a n2  
    break;  
  default:  
    código a ser executado se expressão for diferente dos casos anteriores  
}
```

Exemplo:

```
var dia = 0;  
switch (dia) {  
  case 0:  
    mensagem = "hoje é domingo";  
    break;  
  case 1:  
    mensagem = "hoje é segunda-feira";  
    break;  
  case 2:  
    mensagem = "hoje é terça-feira";  
    break;  
  case 3:  
    mensagem = "hoje é quarta-feira";  
    break;  
  case 4:  
    mensagem = "hoje é quinta-feira";  
    break;  
  case 5:  
    mensagem = "hoje é sexta-feira";  
    break;  
  case 6:  
    mensagem = "hoje é sábado";  
    break;  
}
```

No exemplo, a variável `dia` possui valor `0`. Assim, apenas o caso que corresponde ao valor `0` será executado (o primeiro).

Outro exemplo:

```
var dia = 0;
switch (dia) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
    mensagem = "hora de trabalhar!";
    break;
  default:
    mensagem = "hora de descansar!";
}
```

No exemplo, a variável `dia` possui o valor `0`. Como nenhum dos casos equivale ao valor da variável, o bloco `default` será executado.

A utilização de `switch` é adequada apenas nos casos em que é possível enumerar os valores da expressão. Em outros casos, melhor utilizar a instrução `if`.

Laços de Repetição

Laços de repetição podem ser: `for` , `for in` , `while` e `do while` .

Instrução `for`

A instrução `for` representa um laço de repetição. A sintaxe é a seguinte:

```
for (instrucao 1; instrucao 2; instrucao 3) {  
    código a ser executado  
}
```

A `instrucao 1` executa antes do laço de repetição. A `instrucao 2` define uma condição para a execução do laço de repetição. A `instrucao 3` é executada a cada passo do laço de repetição.

Exemplo:

```
var mensagem = "";  
for (i = 0; i < 10; i++) {  
    mensagem += "Numero: " + i + "\n";  
}
```

O uso do laço de repetição no código de exemplo faz com que o código da linha 3 seja executado enquanto a condição `i < 10` for verdadeira. Neste caso, como `i = 0` , o código executa 10 vezes.

Instrução `for...in`

A instrução `for...in` permite iterar pelos atributos de um objeto. A sintaxe é similar à da instrução `for` :

```
for (identificador in expressao) {  
    código a ser executado  
}
```

Exemplo:

```
var pessoa = {id : 1, nome : "José", idade : 18 };

var mensagem = "";
for (var a in pessoa) {
    mensagem += a + " - " + pessoa[a];
}
```

Neste caso, a sintaxe permite acessar um atributo do objeto semelhante ao acesso de um índice em um array.

A instrução `for...in` também pode ser utilizada para iterar pelos elementos de um array. Exemplo:

```
var n = [1, 2, 3, 4, 5];
var s = 0;
for (var i in n) {
    s += n[i];
}
```

Neste exemplo, o laço de repetição itera pelos elementos do array e calcula a soma dos seus números.

Instrução while

A instrução `while` executa um código enquanto uma condição for satisfeita. A sintaxe é a seguinte:

```
while (condicao) {
    código a ser executado
}
```

Exemplo:

```
var i = 0;
while (i < 10) {
    mensagem += "Número é " + i;
    i++;
}
```

Instrução do...while

A instrução `do...while` é uma variação da `while`. A diferença é que enquanto o `while` executa um código 0 ou mais vezes, `do...while` executa um código pelo menos 1 vez (1 ou mais vezes). A sintaxe é:

```
do {  
    código a ser executado  
} while (condicao);
```

Exemplo:

```
var i = 0;  
do {  
    mensagem += "Número é " + i;  
    i++;  
} while (i < 10);
```

Exercícios

Exercício 1: Calcule a média de um array de números. Utilize as instruções `for`, `for...in`, `while` e `do...while` (fazendo um código separado para cada instrução).

Exercício 2: Considere que um conjunto de dados armazena os seguintes dados: nome, idade e sexo de um grupo de pessoas. Crie um programa JavaScript que encontre e apresente:

- o nome e a idade da pessoa mais idosa;
- a média das idades dos homens; e
- a média das idades das mulheres.

Exercício 2A: Na implementação do **exercício 2** (alterando-a diretamente ou em uma versão alternativa), utilize objetos para representar os dados das pessoas.

Funções

Funções em JavaScript permitem a organização eficiente de código para realizar uma determinada tarefa. Uma função é **definida** e, posteriormente, pode ser **chamada** (ou **invocada**).

Sintaxe:

```
function nome(parametros) {  
    código da função  
}
```

Exemplo:

```
function soma(a, b) {  
    return a + b;  
}  
  
// utilização da função  
var x = 1;  
var y = 2;  
var z = soma(x, y);
```

Neste exemplo, a palavra `return` é utilizada para retornar um valor, ou seja, a função pode ser utilizada em uma expressão. A variável `z` recebe o valor da chamada da função `soma()` passando as variáveis `x` e `y` como parâmetros.

Exercício

Exercício 1: Crie uma função que aceita um array de números como parâmetro e retorna a média dos números.

Exercício 2: Crie uma função que aceita um array de números como parâmetro e retorna um array contendo: o menor valor, o maior valor, a média dos valores.

Exercício 3: Crie uma função que aceita uma quantidade não definida de parâmetros e retorna a soma deles. A função deve ser chamada mais ou menos assim:

```
var s = soma(1, 2, 3, 4, 5);
```

Objetos

Por ser uma linguagem multi-paradigma, o JavaScript também suporta o Paradigma Orientado a Objetos (POO), da mesma forma que outras linguagens como Python, C++, Java e C#. As diferenças principais para outras linguagens e as características principais de POO com JavaScript são apresentadas nesta seção.

Definição do objeto

Na seção sobre tipos de dados vimos que o JavaScript possui uma sintaxe que permite definir um objeto diretamente no código-fonte. Cada um dos atributos do objeto podem ser acessados diretamente, como por exemplo:

```
var pessoa = {nome: 'Jose', idade: 30};
pessoa.idade = 40; // modifica a idade
pessoa.idade++; // modifica a idade
alert(pessoa.nome); // apresenta o nome em um alert
```

Conforme esta maneira de acessar os atributos, algumas pessoas os chamam de **propriedades**. Para efeito de clareza e objetividade, continuaremos usando o termo **atributo** para nos referirmos aos membros de um objeto que representam características dele.

Também podemos definir métodos diretamente na declaração de um objeto. Por exemplo:

```
var pessoa = {
  nome: 'Jose',
  nascer: function() { // define o método
    alert('Pessoa nascendo!');
  }
};
pessoa.nascer(); // chama o método
```

O trecho de código ilustra a definição do método `nascer()` para o objeto `pessoa`. Perceba que a sintaxe é similar à de um atributo, com a diferença que o valor atribuído não é um número ou um literal, mas uma função. Na prática, o tipo deste valor (ao utilizar `typeof`) é `function`. Ainda em outro sentido, JavaScript permite que funções sejam atribuídas a variáveis, como um recurso de "ponteiro para função" em C++. Exemplo:

```
function somar(a, b) {  
    return a + b;  
}  
  
var s = somar; // s recebe a função somar()  
  
var c = s(1, 2); // chama a função somar() como s()
```

No trecho de código, a função `somar()` é atribuída à variável `s`. Neste sentido, a variável torna-se um "atalho" para a função. Assim, pode-se chamar a função como `somar()` ou `s()`.

A utilização de métodos em objetos também permite acessar seus atributos. Isso é feito por meio da palavra `this`. Por exemplo:

```
var pessoa = {  
    nome: 'Jose',  
    nascer: function() { // define o método  
        this.idade = 0;  
    }  
};  
pessoa.nascer(); // chama o método
```

No trecho de código, o método `nascer()` do objeto `pessoa` modifica (e cria) o atributo `idade`. Perceba que a utilização de `this` representa uma referência ao objeto em questão, ou seja, `pessoa`. Se não for utilizada a palavra `this` o código executará, mas considerando que `idade` seria apenas uma variável interna ao método.

Definição de tipo (classe)

A definição de objeto diretamente no código permite um recurso de programação interessante. Entretanto, seguindo o formato anterior, um objeto é uma variável, apenas. Se for necessário criar outro objeto, será necessário reproduzir a mesma estrutura de declaração. Para padronizar e estruturar os dados de forma adequada, como um tipo, ou, se preferir, uma **classe**, JavaScript permite uma instrução similar à de construtores em outras linguagens POO. Por exemplo:

```
function Pessoa(nome, idade) {  
  this.nome = nome;  
  this.idade = idade;  
  this.nascer = function() {  
    this.idade = 0;  
  }  
}
```

O trecho de código ilustra a declaração da estrutura do tipo `Pessoa` na forma de uma função. Os parâmetros da função `Pessoa()`, `nome` e `idade` são utilizados para **inicializar** um objeto (daí a utilização deste recurso como construtor). Perceba que o código, que é uma função, contém atribuições dos parâmetros para atributos (utilizando a palavra `this`) e a declaração do método `nascer()`. O processo de instanciação ocorre como mostram os seguintes exemplos:

```
var jose = new Pessoa('Jose', 30);  
var maria = new Pessoa('Maria');  
maria.nascer();
```

O trecho de código ilustra a instanciação (criação de objetos) por meio da palavra `new` -- de forma similar ao que é feito em outras linguagens POO. No momento da instanciação, são informados os parâmetros do construtor do tipo (neste caso, o tipo `Pessoa`). É importante destacar que JavaScript não requer que todos os parâmetros esperados sejam informados no momento de chamar uma função (o que vale também para o construtor), ficando a cargo do programador lidar com esta situação (identificar quando um parâmetro não é informado). Veja outro exemplo:

```
var pessoas = [new Pessoa('Jose'), new Pessoa('Maria')];  
for (var p in pessoas) {  
  pessoas[p].nascer();  
}
```

No trecho de código, a variável `pessoas` recebe um array de objetos. Perceba que a construção `new Pessoa()` é válida. Neste caso, cada índice do array possui (referencia) um objeto.

Exercício

Considere que, no contexto de uma escola:

- A escola é identificada por um código e possui nome e alunos; e
- Os alunos possuem nome, disciplinas e notas (uma nota para cada disciplina).

Represente este contexto na forma orientada a objeto, criando tipos de dados (classes).

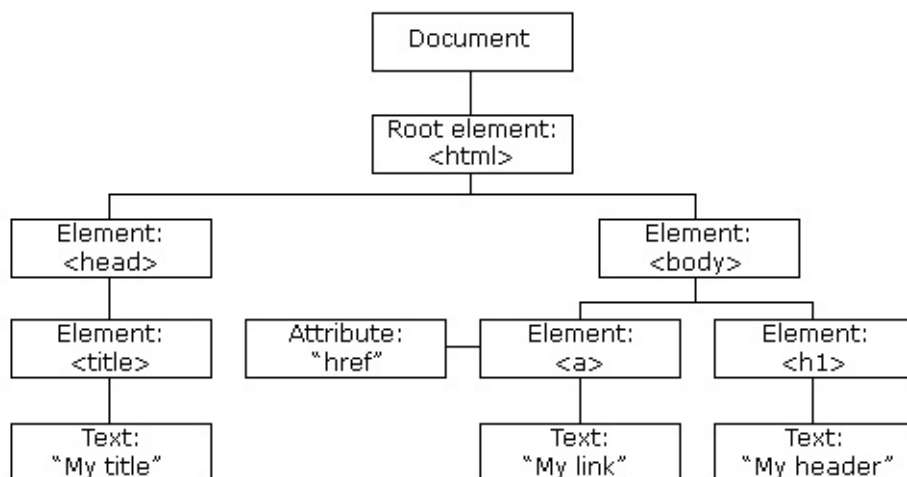
A classe que representa o aluno deve possuir um método que calcula e retorna a média geral do aluno, ou seja, a média das suas notas nas disciplinas.

Além disso, a classe que representa a escola deve possuir um método que retorna o nome do aluno com maior média geral.

HTML DOM

O DOM (Document Object Model) é um padrão da W3C especificado na forma de uma interface para manipulação de documentos cujo conteúdo é representado por objetos. O HTML DOM é uma interface voltada para manipulação de documentos HTML.

A figura a seguir apresenta uma árvore DOM.



A árvore DOM é uma representação de um documento na forma de árvore, representando relações entre elementos (ou objetos). Na árvore de exemplo acima, o nó raiz "Document" tem como filho o nó "html" (que é raiz do documento HTML) que, por sua vez, possui dois filhos: os nós "head" e "body".

Os nós podem ser de quatro tipos:

- Document
- Element
- Attribute
- Text

Destes tipos, apenas o tipo Document pode ter apenas uma instância na árvore DOM.

Ao representar o documento HTML na forma de objetos, também são representadas outras características além da relação entre elementos, como propriedades, métodos e eventos.

Manipulação da árvore DOM

Por meio da interface de programação do DOM é possível manipular o documento, o que quer dizer que você pode, usando JavaScript, por exemplo, criar nós, mover nós, adicionar ou modificar propriedades etc.

Este exemplo demonstra o funcionamento básico do DOM.

No exemplo, o trecho de código a seguir merece destaque:

```
<div id="conteudo"></div>

<script>
var conteudo = document.getElementById("conteudo");
conteudo.innerHTML = "Hello World!";
</script>
```

O elemento `div` possui `id` "conteudo". O atributo `id` representa uma das maneiras de se acessar o elemento por meio de uma consulta à árvore DOM.

No código JavaScript, o objeto `document`, que representa o nó raiz da árvore DOM, fornece o método `getElementById()`, que consulta a árvore DOM, em busca de um nó que tenha como identificador o fornecido por parâmetro.

O nó do tipo `Element` possui a propriedade `innerHTML`, que dá acesso ao conteúdo do nó. Este conteúdo é, na verdade, HTML.

Exercício

Modifique o código para que o conteúdo HTML do elemento `div` apresente um texto como uma página de uma notícia, com título, resumo, imagem e texto.

Métodos para encontrar elementos

Além do método `getElementById()` o objeto `document` fornece os seguintes métodos:

- `getElementsByName(tag)` : retorna um array de elementos, cujas tags na árvore DOM sejam conforme o nome da tag informada como parâmetro;
- `getElementsByClassName(classe)` : retorna um array de elementos cujas classes (atributo `class`) contenham a classe informada como parâmetro;
- `querySelectorAll(seletor)` : retorna um array de elementos com base no seletor. Um seletor utiliza a sintaxe de seletores do CSS.

Modificando elementos

Além da propriedade `innerHTML` o objeto `Element` possui:

- `setAttribute(atributo, valor)` : modifica o valor de um atributo;
- `removeChild(elemento)` : remove um elemento filho
- `appendChild(elemento)` : adiciona um elemento filho

Para criar elementos, o objeto `document` fornece o método `createElement(tag)` , que cria um novo elemento, para ser adicionado à árvore DOM.

[Este exemplo](#) apresenta uma situação em que vários métodos do DOM são chamados para manipular o documento.

Eventos

A programação orientada a eventos é um conceito importante da Ciência da Computação. Em termos simples, um evento representa uma interação com o usuário, ou um comportamento. Por exemplo, um botão (um elemento de interface) possui o evento que trata o comportamento de "clique" e permite executar um código quando tal situação ocorre.

No documento HTML, os eventos são representados por meio de atributos.

O exemplo a seguir utiliza o conceito de eventos para que, no clique de um botão, uma mensagem seja apresentada para o usuário.

```
<!DOCTYPE html>
<html>

<head>
<script>
function mostrarMensagem() {
    var mensagem = document.getElementById('mensagem');
    mensagem.style.display = 'block';
}
</script>
</head>

<body>
<p>Clique no botão abaixo:</p>

<button onclick="mostrarMensagem()">Clique-me!</button>

<div id="mensagem" style="display:none;background-color:yellow;margin-top:10px;padding:10px">
    <h2>Parabéns!</h2>
    <div>Você clicou no botão e apresentou a mensagem</div>
</div>
</body>

</html>
```

Veja o exemplo em funcionamento [aqui](#).

O código JavaScript declara a função `mostrarMensagem()` :

```
function mostrarMensagem() {  
    var mensagem = document.getElementById('mensagem');  
    mensagem.style.display = 'block';  
}
```

A função `mostrarMensagem()` encontra o elemento com id "mensagem" e modifica a propriedade `display` do CSS para `block`, tornando o elemento visível.

No código HTML, é importante ressaltar a maneira de disparar o evento de clique:

```
<button onclick="mostrarMensagem()">Clique-me!</button>
```

O atributo `onclick` representa a maneira de tratar o evento de clique. Ele não é exclusivo de botões. O conteúdo do atributo `onclick` é código JavaScript. Neste caso, o código representa a chamada da função `mostrarMensagem()`.

Outros eventos interessantes são:

- `onload` : aplicável apenas ao elemento `body`. Disparado no momento em que a página termina de carregar;
- `onchange` : aplicável a elementos de entrada de dados (`input` e `select`, por exemplo). Disparado no momento em que o valor do elemento é modificado;
- `onmouseover` e `onmouseout` : disparados quando o mouse está sobre o elemento e quando sai, respectivamente;
- `onmousedown` e `onmouseup` : disparados quando o mouse é pressionado e quando é liberado;
- `onfocus` : aplicável a elementos de entrada de dados (`input`, por exemplo). Disparado no momento em que o elemento ganha foco.

Formulários

Embora seja possível, por meio da árvore DOM, manipular qualquer tipo de documento HTML, algumas facilidades são fornecidas. Uma delas é a maneira de lidar com formulários. O objeto `document` possui a propriedade `forms`, um array que representa os formulários presentes na página. Este array pode ser acessado por índice numérico ou string (correspondendo ao nome do formulário).

Entrada básica

O exemplo a seguir ilustra a utilização do HTML DOM para acessar dados do formulário.

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      function enviarDados() {
        var form = document.forms[0];
        var primeiro_nome = form["primeiro_nome"].value;
        var segundo_nome = form["segundo_nome"].value;
        var mensagem = document.getElementById('mensagem');
        mensagem.innerHTML = primeiro_nome + ' ' + segundo_nome;
      }
    </script>

  </head>
  <body>

    <form onsubmit="return false;">
      <div>
        <label for="primeiro_nome">Primeiro nome:</label>
        <input type="text" name="primeiro_nome" id="primeiro_nome"
          placeholder="Informe seu primeiro nome">
      </div>
      <div>
        <label for="segundo_nome">Segundo nome:</label>
        <input type="text" name="segundo_nome" id="segundo_nome"
          placeholder="Informe seu segundo nome">
      </div>
      <button type="submit" onclick="enviarDados()">Enviar dados</button>
    </form>

    <p>Seu nome completo é:</p>
    <div id="mensagem"></div>

  </body>
</html>
```

Veja o exemplo em funcionamento [aqui](#).

No HTML, é importante destacar o trecho referente ao formulário:

```
<form onsubmit="return false;">
  <div>
    <label for="primeiro_nome">Primeiro nome:</label>
    <input type="text" name="primeiro_nome" id="primeiro_nome"
      placeholder="Informe seu primeiro nome">
    </div>
    <div>
      <label for="segundo_nome">Segundo nome:</label>
      <input type="text" name="segundo_nome" id="segundo_nome"
        placeholder="Informe seu segundo nome">
      </div>
    <button type="submit" onclick="enviarDados()">Enviar dados</button>
  </form>
```

No elemento `form` está sendo utilizado o evento `onsubmit`, que é disparado quando o formulário está enviando seus dados. O envio dos dados do formulário é causado pelo clique em um botão ou pelo pressionar da tecla `ENTER` em um campo de texto. O valor do atributo `onsubmit` é `return false;` e representa o código necessário para cancelar o envio de dados do formulário. Isto é necessário para que a página não faça um "refresh".

Os elementos `input` representam os campos de entrada de texto e possuem o atributo `name`, que indica o nome do campo e que será usado, posteriormente, para ter acesso ao valor informado pelo usuário.

O elemento `button`, que permite o envio dos dados, possui o atributo `onclick` com valor `enviarDados()`, ou seja, ao detectar o clique no botão, a página HTML chamará a função `enviarDados()`.

O código JavaScript da função `enviarDados()`:

```
function enviarDados() {
  var form = document.forms[0];
  var primeiro_nome = form["primeiro_nome"].value;
  var segundo_nome = form["segundo_nome"].value;
  var mensagem = document.getElementById('mensagem');
  mensagem.innerHTML = primeiro_nome + ' ' + segundo_nome;
}
```

Na função `enviarDados()` o destaque vai para o seguinte:

- o objeto `document` fornece o objeto `forms`, que é um vetor representando os formulários da página. Como só há um formulário, então acessa-se o índice `0`;
- a variável `form`, que representa o formulário, permite acessar cada campo pelo seu nome, como se a variável fosse uma espécie de array. A variável `primeiro_nome` recebe o valor do campo "primeiro_nome" por meio de `form["primeiro_nome"].value`. A propriedade `value` retorna o valor do campo. Um processo semelhante é feito para a

variável `segundo_nome` ;

- por fim, uma string é apresentada na tela, concatenando o primeiro e o segundo nomes.

Outra forma de acessar os campos do formulário é por meio da propriedade `elements` do objeto que representa o formulário. Por exemplo, o código a seguir seria similar ao código anterior:

```
var form = document.forms[0];
var primeiro_nome = form.elements[0].value;
var segundo_nome = form.elements[1].value;
```

Da mesma forma, o resultado seria o mesmo para o código a seguir, que utiliza os nomes dos campos ao invés do índice do array `elements` :

```
var form = document.forms[0];
var primeiro_nome = form.elements['primeiro_nome'].value;
var segundo_nome = form.elements['segundo_nome'].value;
```

Tipos de campos de formulário

O elemento `input` com o atributo `type` com valor `text` representa um campo de texto. Outros valores para o atributo `type` permitem representar outras formas de entrada de dados:

- `checkbox`
- `radio`
- `password`
- `color`
- `date`
- `datetime`
- `datetime-local`
- `email`
- `month`
- `number`
- `range`
- `search`
- `tel`
- `time`
- `url`
- `week`

Além do elemento `input` o formulário pode conter os seguintes elementos representando maneiras diferentes de entrada de dados:

- `select` : representa uma caixa de seleção (um "drop-down"). Contém elementos `option` , que representam as opções do `select` .
- `textarea` : representa uma caixa de texto com várias linhas;
- `datalist` : representa valores iniciais para um campo `text` ;
- `keygen` : gera um par de chaves pública e privada para que seja utilizado um processo de criptografia assimétrica.

Restrições em campos de formulário

A lista a seguir apresenta as restrições mais comuns em campos de formulário:

- `disabled` : o campo torna-se desabilitado;
- `max` : define o valor máximo para um campo (ex: em campo `number`);
- `maxlength` : define a quantidade máxima de caracteres em um campo (ex: campo `text`);
- `min` : define o valor mínimo para um campo (ex: em campo `number`);
- `pattern` : define uma expressão regular para validar a entrada de dados;
- `readonly` : o campo torna-se apenas para leitura;
- `required` : o campo é de preenchimento obrigatório;
- `size` : define a quantidade de caracteres do campo;
- `step` : define o intervalo, passo de incremento (ex: em campo `number`);
- `value` : define o valor do campo.

O exemplo a seguir ilustra a utilização de vários campos de formulários e restrições sobre eles.

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      function enviarDados() {
        var form = document.forms[0];
        var primeiro_nome = form["primeiro_nome"].value;
        var segundo_nome = form["segundo_nome"].value;
        var sexo = form["sexo"].value;
        var cor = form["cor"].value;
        var data = form["data_de_nascimento"].value;
        var mensagem = document.getElementById('mensagem');
        mensagem.innerHTML = primeiro_nome + ' ' + segundo_nome
        + '<br/>sexo: ' + sexo
        + '<br/>cor: ' + cor
        + '<br/>data: ' + data;
```



```
    }
  </script>
</head>
<body>

<form onsubmit="return false;">
  <div>
    <label for="primeiro_nome">Primeiro nome:</label>
    <input type="text" name="primeiro_nome" id="primeiro_nome"
      placeholder="Informe seu primeiro nome" required>
  </div>
  <div>
    <label for="segundo_nome">Segundo nome:</label>
    <input type="text" name="segundo_nome" id="segundo_nome"
      placeholder="Informe seu segundo nome">
  </div>
  <div>
    Sexo:
    <label>M <input type="radio" name="sexo" value="m" required></label>
    <label>F <input type="radio" name="sexo" value="f" required></label>
  </div>
  <div>
    <label>
      Data de nascimento:
      <input type="date" name="data_de_nascimento">
    </label>
  </div>
  <div>
    <label>Cor favorita:

    <select name="cor">
      <option>Vermelho</option>
      <option>Preto</option>
      <option>Azul</option>
      <option>Amarelo</option>
    </select>

    </label>

  </div>
  <button type="submit" onclick="enviarDados()">Enviar dados</button>
</form>

<p>Seu nome completo é:</p>
<div id="mensagem"></div>

</body>
</html>
```

Veja o exemplo em funcionamento [aqui](#).

Exercícios

Exercício 1: Crie um aplicativo web que permite cadastrar dados de pessoas: nome, idade e sexo. Quando os dados do formulário forem salvos eles devem ser armazenados para uso posterior e o formulário deve ser limpo (campos recebem o valor padrão). Um botão deve permitir que o usuário veja a lista de dados cadastrados. Um botão deve permitir que o usuário veja o nome da pessoa mais nova e mais velha para cada sexo.

Exercício 2: Incremente as funcionalidades do aplicativo desenvolvido na parte 1:

- a lista de pessoas deve ser apresentada automaticamente, após cada novo cadastro;
- deve ser possível excluir uma pessoa da lista.

Exercício 3: Incremente as funcionalidades do aplicativo desenvolvido na parte 2:

- deve ser possível editar os dados de uma pessoa da lista;
- apresente os dados na forma de uma tabela.

Exercício 4: Incremente as funcionalidades do aplicativo desenvolvido na parte 3:

- na tabela que apresenta os dados, as linhas que representam as pessoas mais velhas e mais novas devem ser apresentadas de forma diferente. Por exemplo, as pessoas mais velhas são apresentadas em linhas com fundo amarelo, enquanto as pessoas mais novas são apresentadas em linhas com fundo verde claro.
- a funcionalidade anterior deve ser apresentada sempre que uma pessoa for adicionada ou removida da lista, ou quando seus dados tiverem sido alterados.

Tecnologias Front-end

As tecnologias front-end integram, na verdade, um conceito (não tão novo) de desenvolvimento de software voltado para a web em que a camada de software executada no cliente (browser) recebe atenção diferenciada.

Em específico, as tecnologias utilizadas nesta seção do livro tratam de:

- Ferramentas
- Bootstrap
- AngularJS

As seções seguintes apresentarão estas tecnologias em detalhes.

Ferramentas

O desenvolvimento de software para web atualmente tem estabelecido como foco a utilização de ferramentas para front-end que permitem, por exemplo, utilizar pacotes e compilar o código (JavaScript, HTML e CSS, por exemplo).

Git

O [Git](#) é um systema distribuído de controle de versão gratuito, projetado para lidar com projetos grandes com velocidade e eficiência. Por meio do Git, você poderá gerenciar e acessar repositórios no [github.com](#).

Acesse o site do [Git](#) e faça download e instale o mesmo. Se preferir uma ferramenta gráfica, pode baixar o [GitHub Desktop](#).

NodeJS

O [Node.js](#) é um ambiente de execução do JavaScript construído sobre o motor [JavaScript V8 do Google Chrome](#). Node.js fornece também um ecossistema de pacotes, chamado **npm** (considerado atualmente o maior ecossistema para projetos open source).

Acesse o site do [Node.js](#) e faça download e instale o mesmo.

npm

O [npm](#) é o gerenciador de pacotes do NodeJS.

npm

O **npm** é o gerenciador de pacotes para JavaScript mais utilizado em desenvolvimento web moderno para front-end e back-end (ao utilizar o NodeJS). Para utilizá-lo, basta instalar o [NodeJS](#).

Inicializando um projeto

Além de permitir o gerenciamento de pacotes, que inclui funções de instalar e remover pacotes disponíveis em um repositório remoto chamado [npmjs](#), o **npm** permite o gerenciamento de um projeto local. Na verdade, um projeto local é entendido como um pacote.

Para inicializar um projeto, em um prompt no diretório escolhido execute:

```
npm init
```

A interface baseada em texto apresentará algumas perguntas, as quais, depois de respondidas, levarão à criação do arquivo `package.json`. Este arquivo contém informações sobre o pacote (projeto atual) e sobre as suas dependências, tanto de desenvolvimento quanto de produção.

Um atalho, que cria um arquivo `package.json` padrão, é utilizar a opção `--yes`:

```
npm init --yes
```

As dependências de desenvolvimento são aquelas que não serão disponibilizadas no servidor de produção. Em outras palavras, representam ferramentas, por exemplo, que executarão apenas localmente, durante o tempo de desenvolvimento (como o **Webpack**).

As dependências de produção, ao contrário das dependências de desenvolvimento, serão implantadas no servidor de produção. Exemplo deste tipo de dependência são o Bootstrap e o Angular.

Instalar dependência (pacote)

A instalação de dependências permite que o projeto atual reutilize pacotes que fornecem as mais variadas funcionalidades. Por exemplo, um projeto que utiliza Angular e Bootstrap depende deles.

Para instalar uma dependência é executado o comando `npm install` :

```
npm install angular bootstrap
```

O comando anterior instala os pacotes localmente e os salva no diretório `node_modules` (cada pacote também está armazenado localmente de forma individual, no seu próprio diretório).

Como dito anteriormente, as dependências do projeto local estão no arquivo `package.json` . O comando `npm install` possui duas opções que editam o arquivo `package.json` :

- `--save` : para dependências de produção
- `--save-dev` : para dependências de desenvolvimento

Exemplos:

```
npm install angular bootstrap --save
npm install http-server --save-dev
```

Scripts

O arquivo `package.json` de um projeto pode conter scripts, que permitem simplificar o processo de execução de códigos no prompt de comando. Exemplo:

```
{
  "name": "app",
  "version": "1.0.0",
  "scripts": {
    "clear": "rmdir /S /Q node_modules"
  }
}
```

O trecho de código anterior, que demonstra um arquivo `package.json` , o atributo `scripts` permite definir scripts que podem ser executados via npm. Neste caso, há um script `clear` , que executa a linha de comando `rmdir /S /Q node_modules` . Para executar o script, deve-se utilizar:

```
npm run clear
```

O script do exemplo é útil para limpar o projeto em questão, apagando o diretório

```
node_modules .
```

TypeScript

Este capítulo sobre o TypeScript é baseado em documentos oficiais, que podem ser obtidos em <https://www.typescriptlang.org/docs> e na [Especificação formal da linguagem](#).

Iniciando

TypeScript compila código em JavaScript. Na verdade, como o browser entende apenas JavaScript, é exatamente isso que ele terá para executar. Então, você vai precisar do seguinte:

- Compilador TypeScript (vamos usar o npm)
- Editor TypeScript (você pode usar qualquer editor de texto ou IDE, mas iremos usar o PHPStorm)

Instalação

A instalação do TypeScript é feita com o npm. Na prática, um programa será instalado globalmente, para poder ser usado de qualquer lugar. Execute no prompt:

```
npm install -g typescript
```

Isso vai instalar dois programas:

- **tsc**: o compilador do TypeScript
- **tsserver**: uma versão do compilador que também cria um servidor web

Para compilar um arquivo TypeScript, basta executar `tsc programa.ts`. Isso vai criar o arquivo *programa.js* (na mesma pasta onde está o arquivo *programa.ts*).

Outro programa bastante útil, que permite executar código TypeScript diretamente no prompt de comando é o [ts-node](#). Para instalá-lo com npm, use:

```
npm install -g ts-node
```

Depois disso, basta executar `ts-node arquivo.ts`.

IIFE

IIFE é a sigla para *Immediately Invoked Function Expression* (algo como expressão de função invocada imediatamente). Esse é um recurso geralmente em concepções *avançadas* de programação JavaScript.

```
var a = (function(i) {  
    return i * 2;  
})(4);  
console.log(a); // imprime 8
```

IIFE, portanto, é um recurso utilizado para representar uma expressão que é declarada e, imediatamente, chamada como uma função.

Acesse as seções deste capítulo

- [Quickstart](#)
- [Tipos de dados](#)
- [Declaração de variáveis](#)
- [Funções](#)
- [Classes](#)

Quickstart

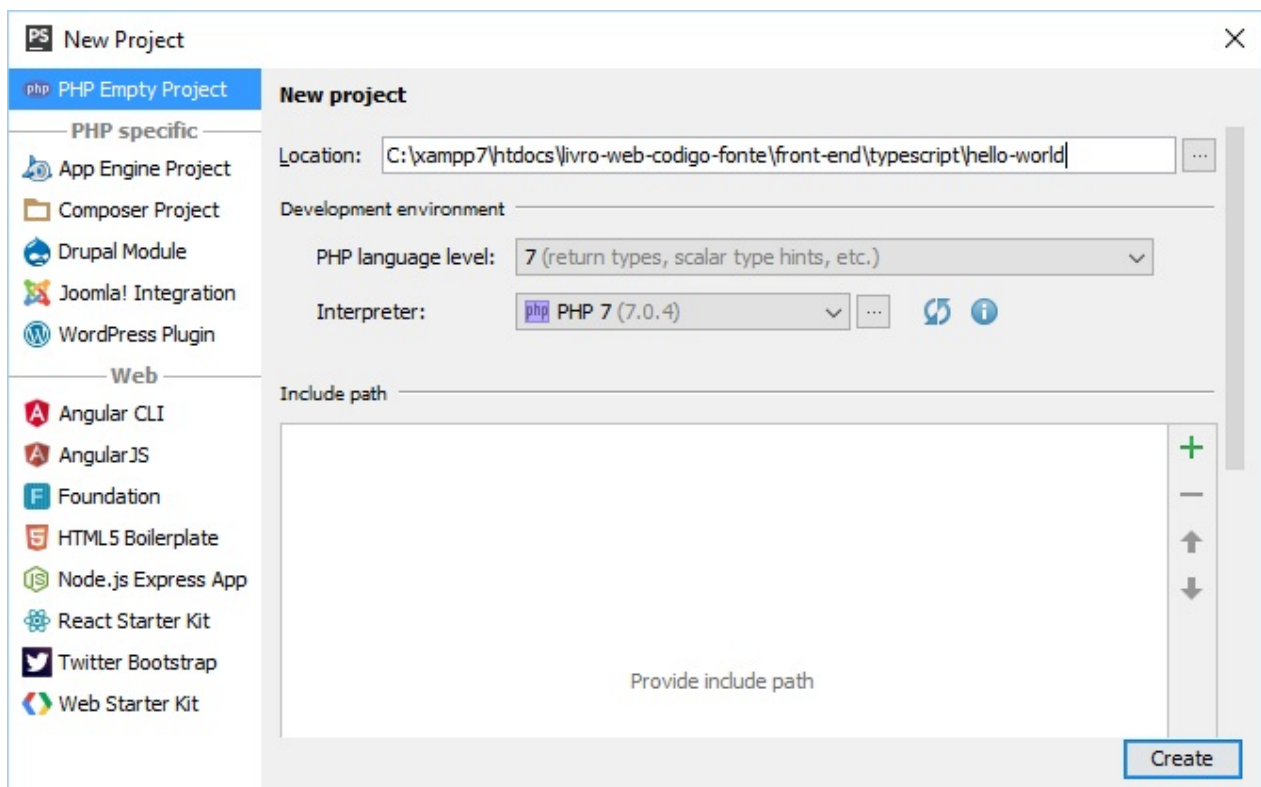
Este capítulo permite um início rápido à utilização do TypeScript e a configuração do PHPStorm como IDE de programação. As seções a seguir apresentam a criação de três projetos que demonstram os recursos do TypeScript de forma simples e objetiva.

Projeto Hello World!

Um projeto básico, do tipo "hello world", pode ser criado no PHPStorm seguindo os seguintes passos.

Criar o projeto

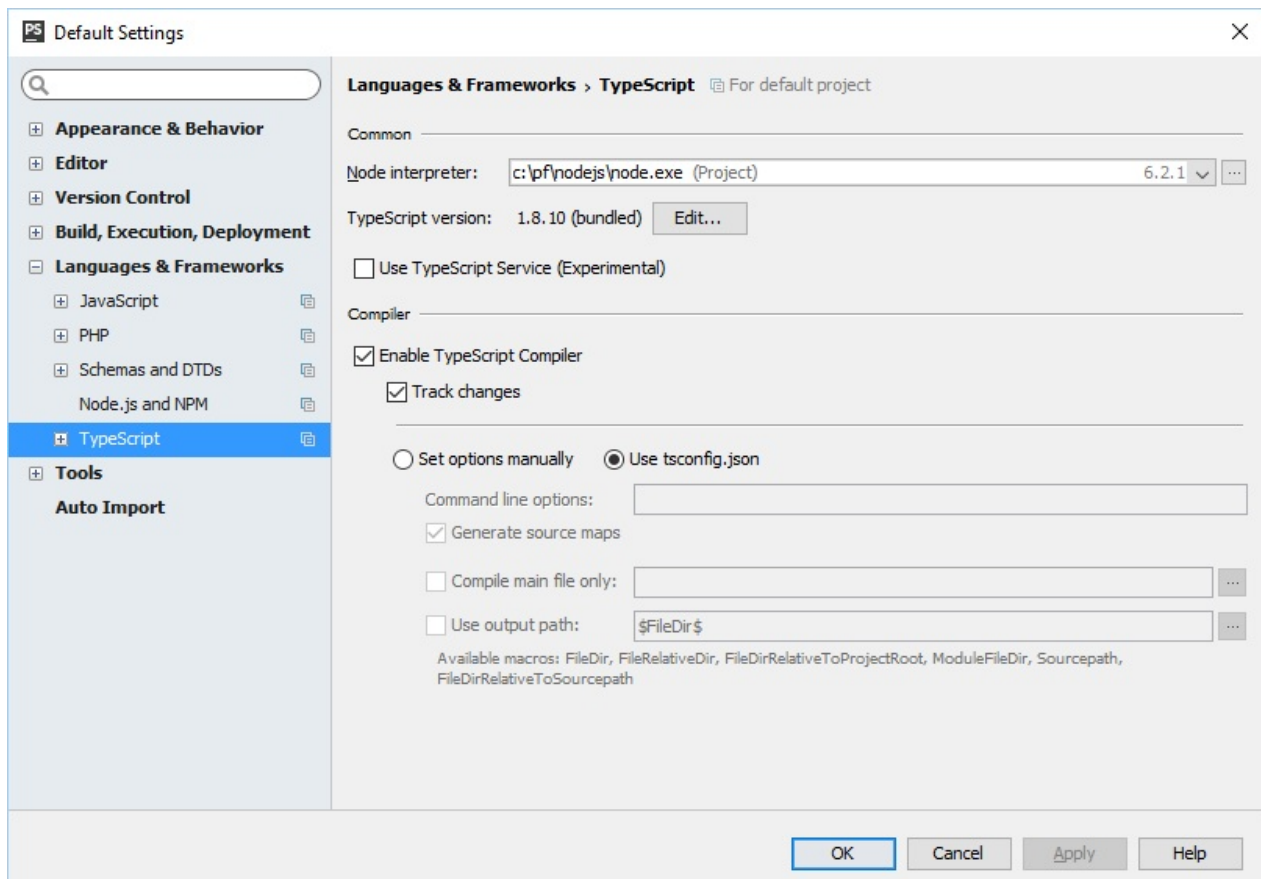
Por meio do menu *File -> New Project* crie um novo projeto. Na janela *New Project* escolha o tipo de projeto *PHP Empty Project* e informe onde os arquivos serão armazenados.



Configurar o suporte para o TypeScript

Por meio do menu *File -> Default Settings* (para configurar apenas o projeto atual use apenas *Settings*) na página *Languages & Frameworks -> TypeScript* informe a localização do NodeJS e marque as opções *Enable TypeScript Compiler* e *Track changes*. Por fim,

escolha a opção *Use tsconfig.json*.



Após fazer essa configuração para o modo padrão (*Default settings*) você não precisará fazer isso novamente para **novos** projetos.

Hora do código

Crie o arquivo *index.ts* com o conteúdo a seguir.

```
function mensagem(nome) {  
    return "Olá, " + nome + "! Seja bem-vindo(a)!";  
}  
var pessoa = "José";  
console.log(mensagem(pessoa));
```

Se você já conhece JavaScript certamente está pensando: mas... isso é JavaScript! E é isso mesmo. Veremos mais sobre isso daqui a pouco.

Execute o código

Por meio do PhpStorm é possível executar código TypeScript usando o NodeJS. Você também pode usar o formato padrão de incorporar o código em uma página HTML e executá-lo no browser. Entretanto, um procedimento anterior precisa ser feito. O NodeJS e

o browser não entendem TypeScript. Assim, será necessário usar o processo conhecido como "transpiling" (algo como "transpilar") para traduzir o código TypeScript para JavaScript.

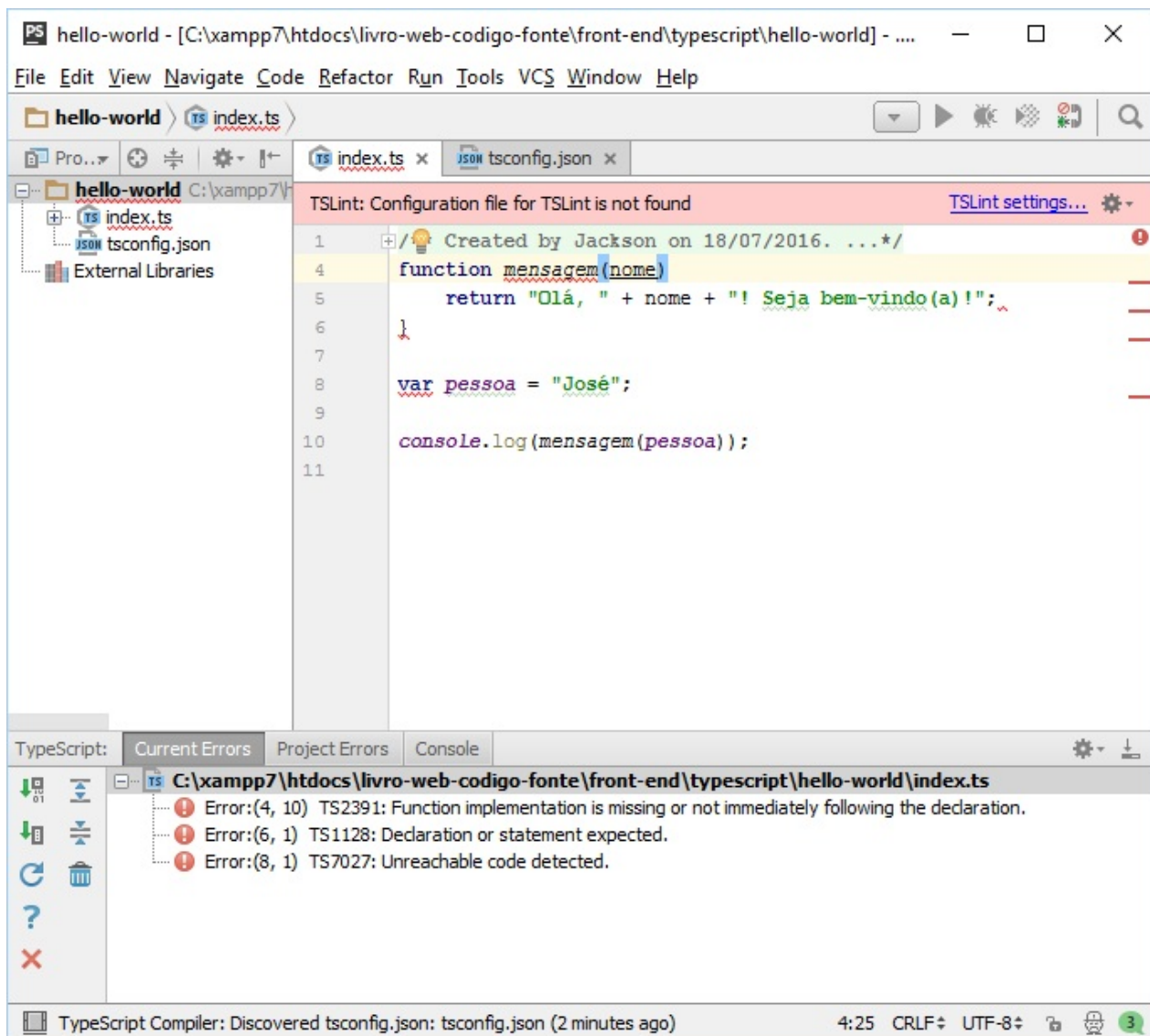
Antes de escolher um modo de execução, é necessário configurar o PHPStorm.

Crie o arquivo *tsconfig.json*. O PHPStorm já tem um template padrão. Basta usar o menu *File -> New* e escolher o template *tsconfig.js File*. O conteúdo desse arquivo será algo como o seguinte.

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

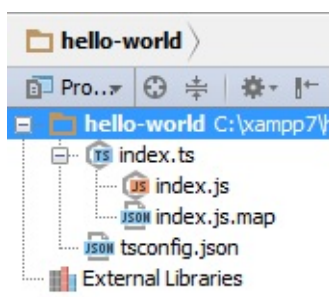
Não vou explicar o significado disso agora. Veremos isso mais adiante. Por enquanto, entenda que esse arquivo é parte do processo.

Depois disso, o painel do TypeScript mostrará erros de compilação em tempo-real. Por enquanto, com o código contendo, na verdade, conteúdo JavaScript, não será possível notar o poder desta ferramenta. Mas, para vê-la em ação, modifique o código, gerando um erro de sintaxe, por exemplo. A figura a seguir exemplifica esse processo.



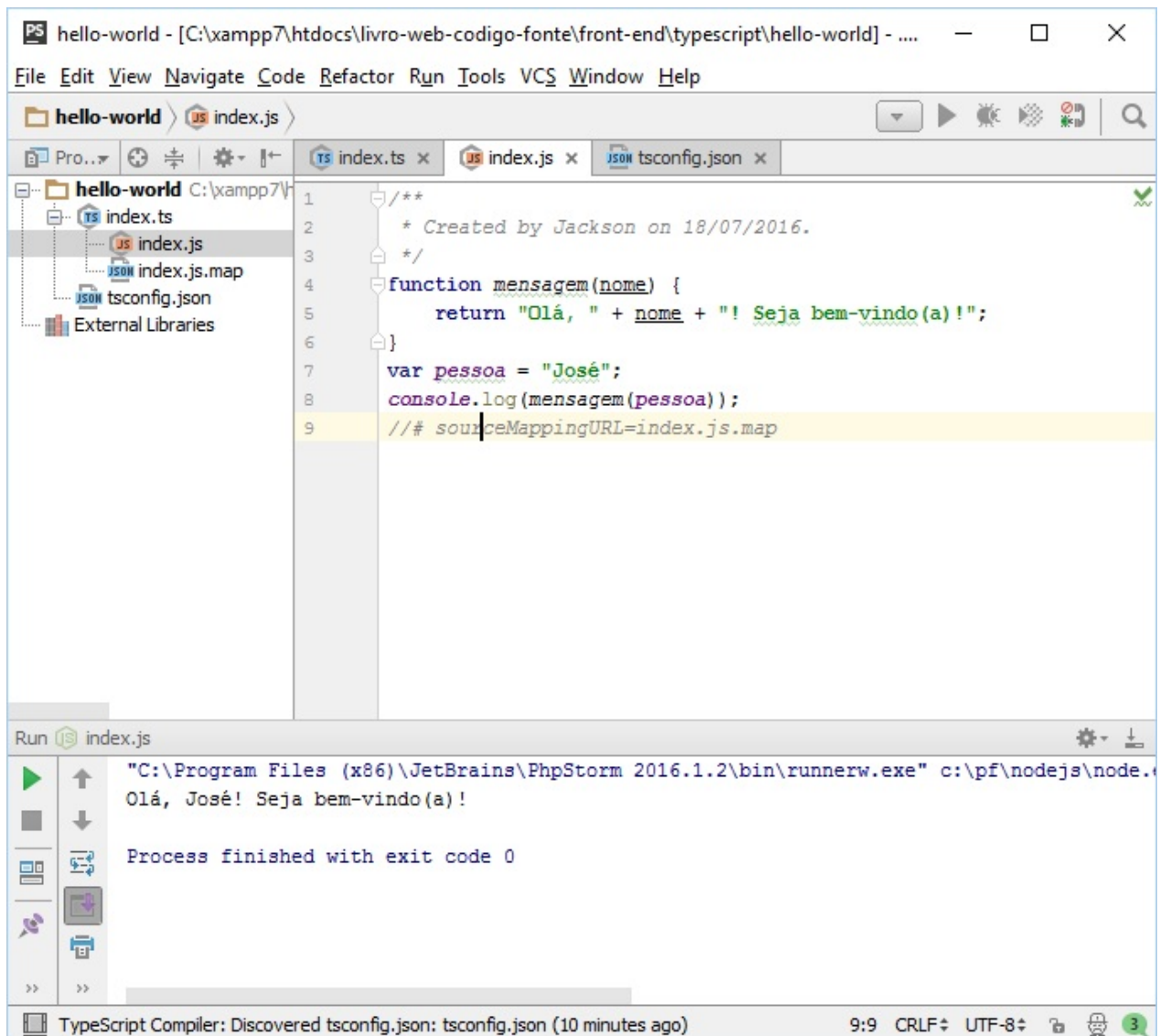
A figura mostra que o painel TypeScript indicou vários erros no arquivo. Isso é de grande ajuda. Esse processo de "compilação" ocorrerá automaticamente, após cada mudança no arquivo *index.ts*.

Também como resultado do processo de "compilação", foram gerados dois arquivos: *index.js* e *index.js.map*. O primeiro deles é um arquivo JavaScript, que tem o mesmo conteúdo do arquivo *index.ts*, como seria de se esperar. O segundo é um arquivo utilizado pelo PHPStorm, para identificar pontos de transformações entre os arquivos TypeScript e JavaScript.



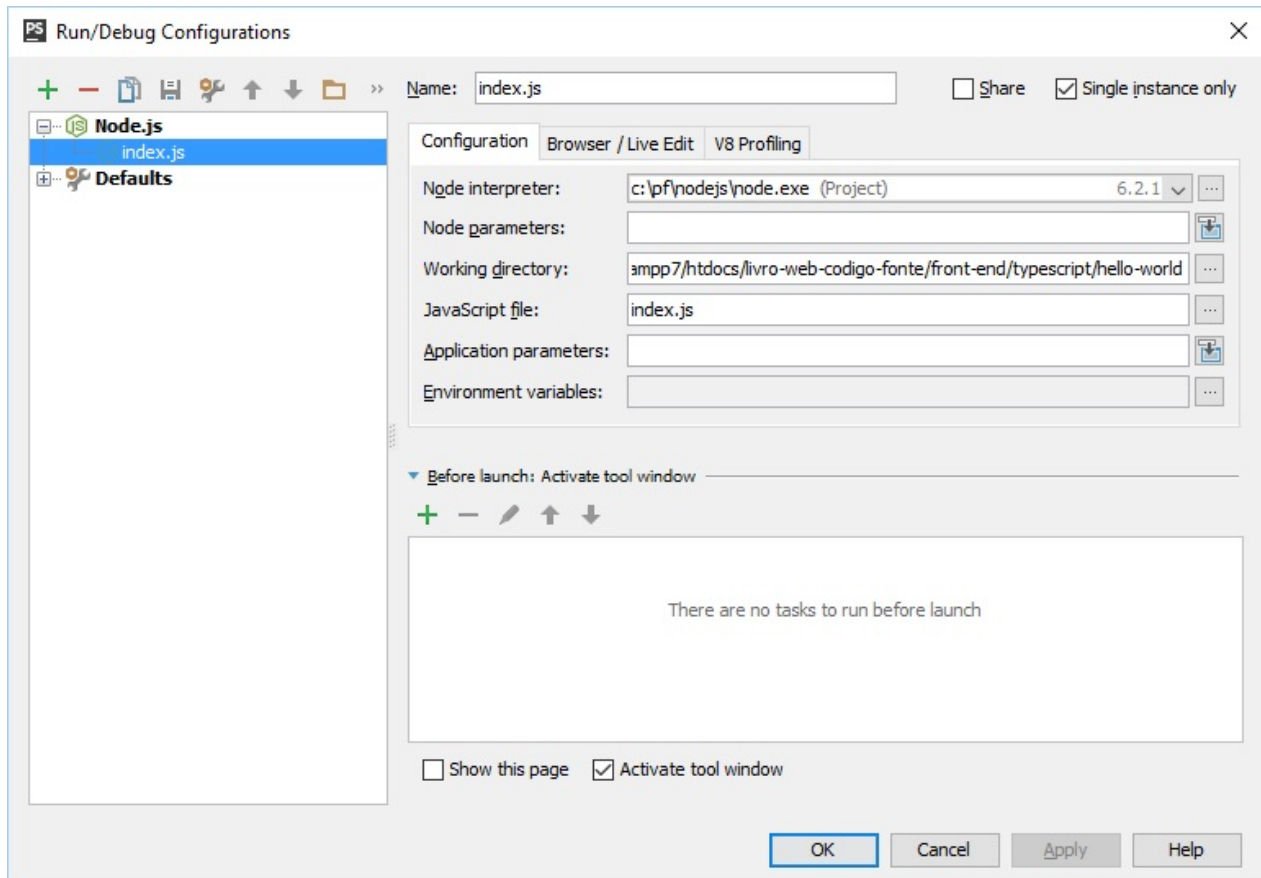
Executando no NodeJS

Abra o arquivo *index.js* e, por meio do menu *Run -> Run*, execute o arquivo. No menu flutuante escolha *index.js*. O painel Run apresentará o resultado da execução.

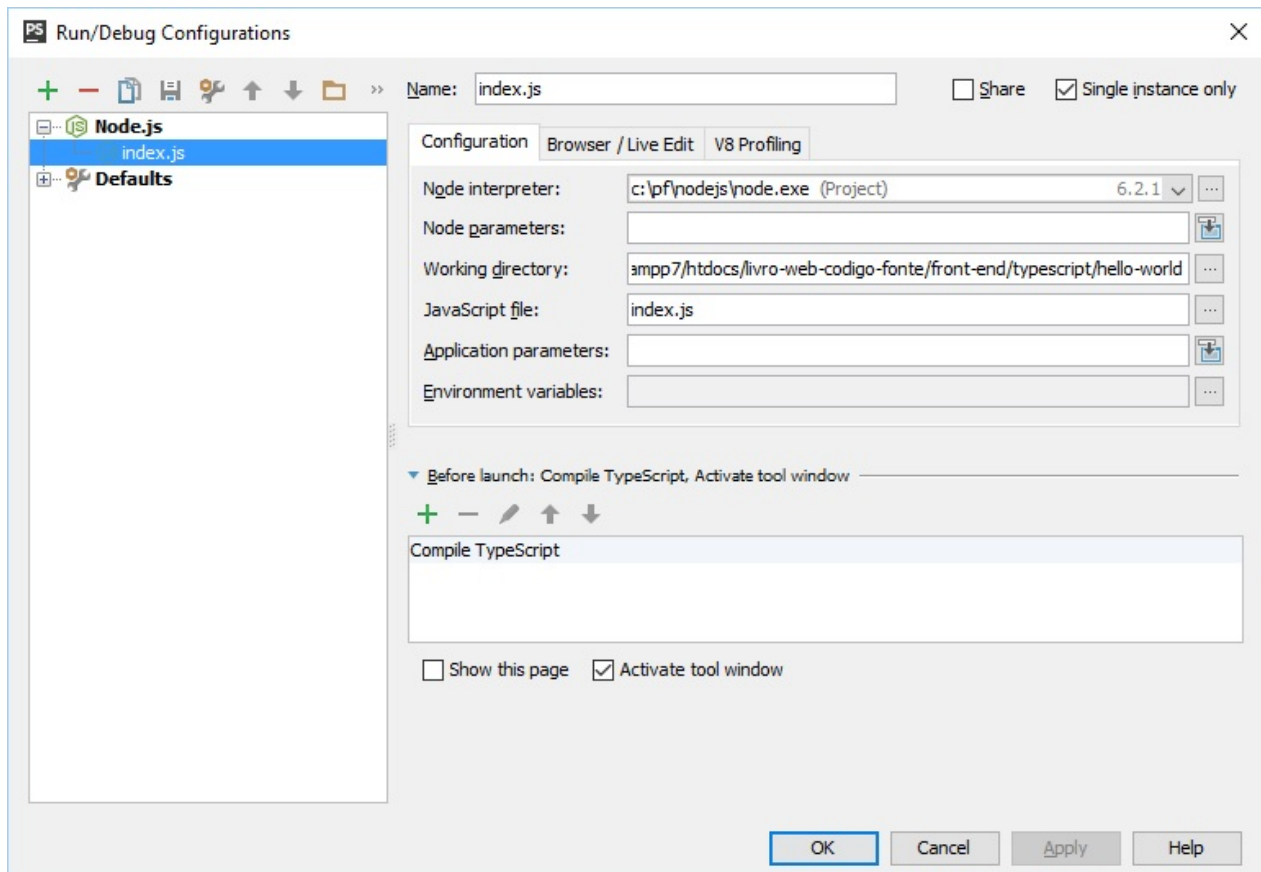


Efetivamente, o arquivo *index.js* está sendo executado pelo NodeJS, não o arquivo *index.ts*. Não se perca aqui. A configuração do PhpStorm indica que sempre que houver uma modificação no arquivo *index.ts* então ele será compilado, gerando o arquivo *index.js*. Por isso não há um problema com essa forma de trabalho. Se você quiser garantir adicionalmente (manualmente) a compilação do arquivo TypeScript pode seguir o passo seguinte.

Use o menu *Run -> Edit configurations* para abrir a janela *Run/Debug Configurations*. A figura a seguir ilustra a janela no momento.



Clique no sinal "+" e escolha *Node.js*. Na opção *JavaScript file* informe *index.js*. Na seção *Before launch: Activate tool window* clique no sinal "+" e escolha *Compile TypeScript*.

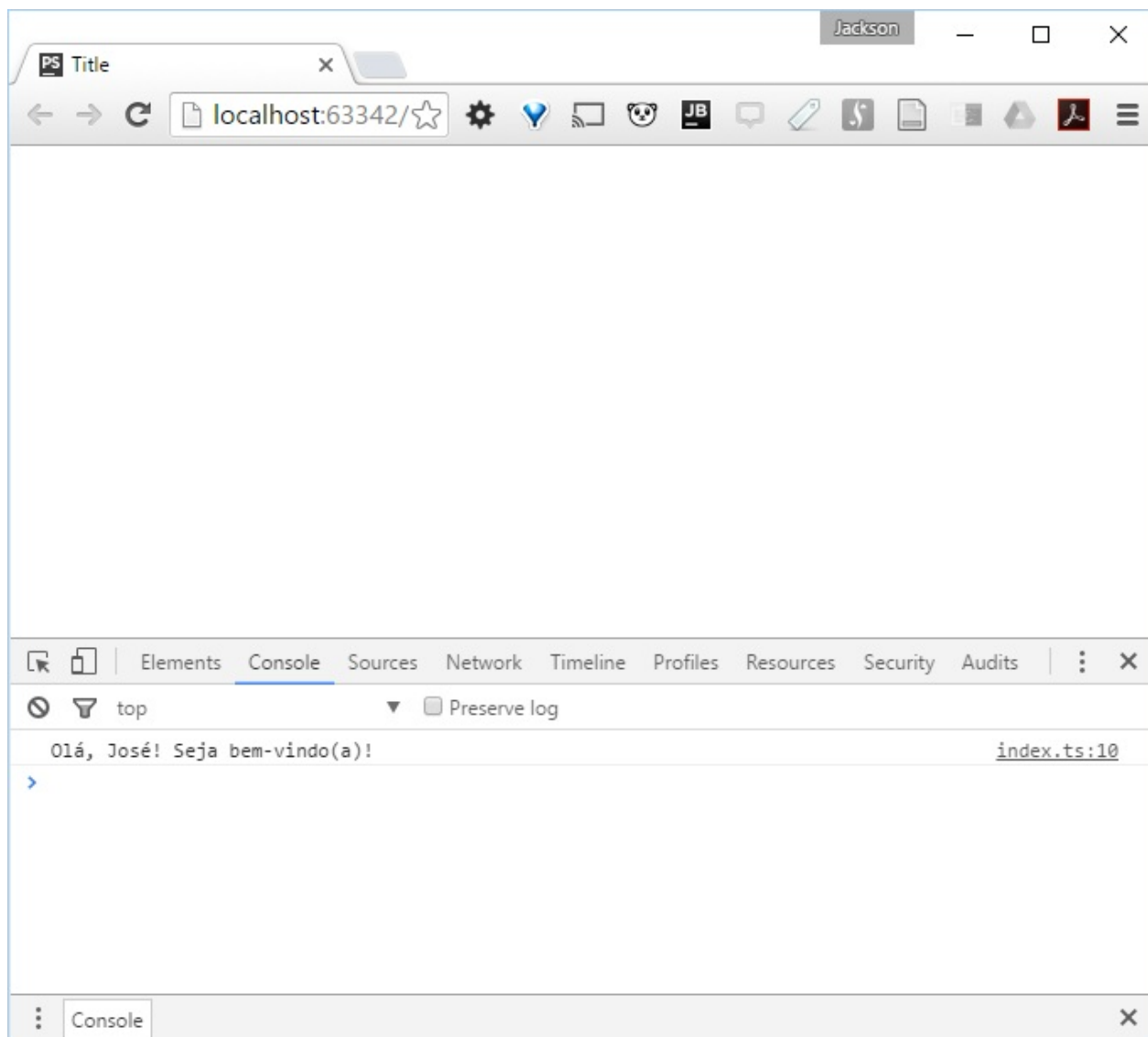


Executando no browser

Crie o arquivo *index.html*, com o conteúdo a seguir.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
  <script src="index.js"></script>
</head>
<body>
</body>
</html>
```

Clique sobre o arquivo com o botão direito do mouse e escolha *Run index.html*. Isso vai abrir uma janela do browser.



A página está em branco (sem conteúdo). Por isso, para ver a saída, ative o painel de desenvolvedor e escolha a aba *Console*. Um fluxo de trabalho interessante é não fechar a janela do browser. Mantenha-a aberta e, após alterações no TypeScript, atualize-a utilizando a tecla *F5*.

Esses modos de execução não são certamente úteis em qualquer tipo de trabalho. Para o momento inicial, entretanto, funcionam muito bem. Escolha um deles e bom trabalho!

Hello World, TypeScript!

Como já disse, o projeto Hello World não apresenta diferenças entre o TypeScript e o JavaScript. Então, vamos começar a fazer algumas modificações no código: adicionar tipos.

O TypeScript permite a definição de tipos de dados para variáveis. Com isso, o código do arquivo *index.ts* é modificado para o seguinte.

```
function mensagem(nome: string) {  
    return "Olá, " + nome + "! Seja bem-vindo(a)!";  
}  
var pessoa:string = "Maria";  
console.log(mensagem(pessoa));
```

O código adiciona duas modificações importantes:

1. A variável `pessoa` é do tipo `string`. A sintaxe `nome: tipo` é típica do TypeScript para indicar esse tipo de informação;
2. Na função `mensagem()` o parâmetro `nome` é do tipo `string`.

Verifique a comparação com o arquivo *index.js*.

index.ts	index.js
<pre>function mensagem(nome: string) { return "Olá, " + nome + " Seja bem-vindo(a)!"; } var pessoa:string = "Maria"; console.log(mensagem(pessoa)) ;</pre>	<pre>function mensagem(nome) { return "Olá, " + nome + " Seja bem-vindo(a)!"; } var pessoa = "Maria"; console.log(mensagem(pessoa))</pre>

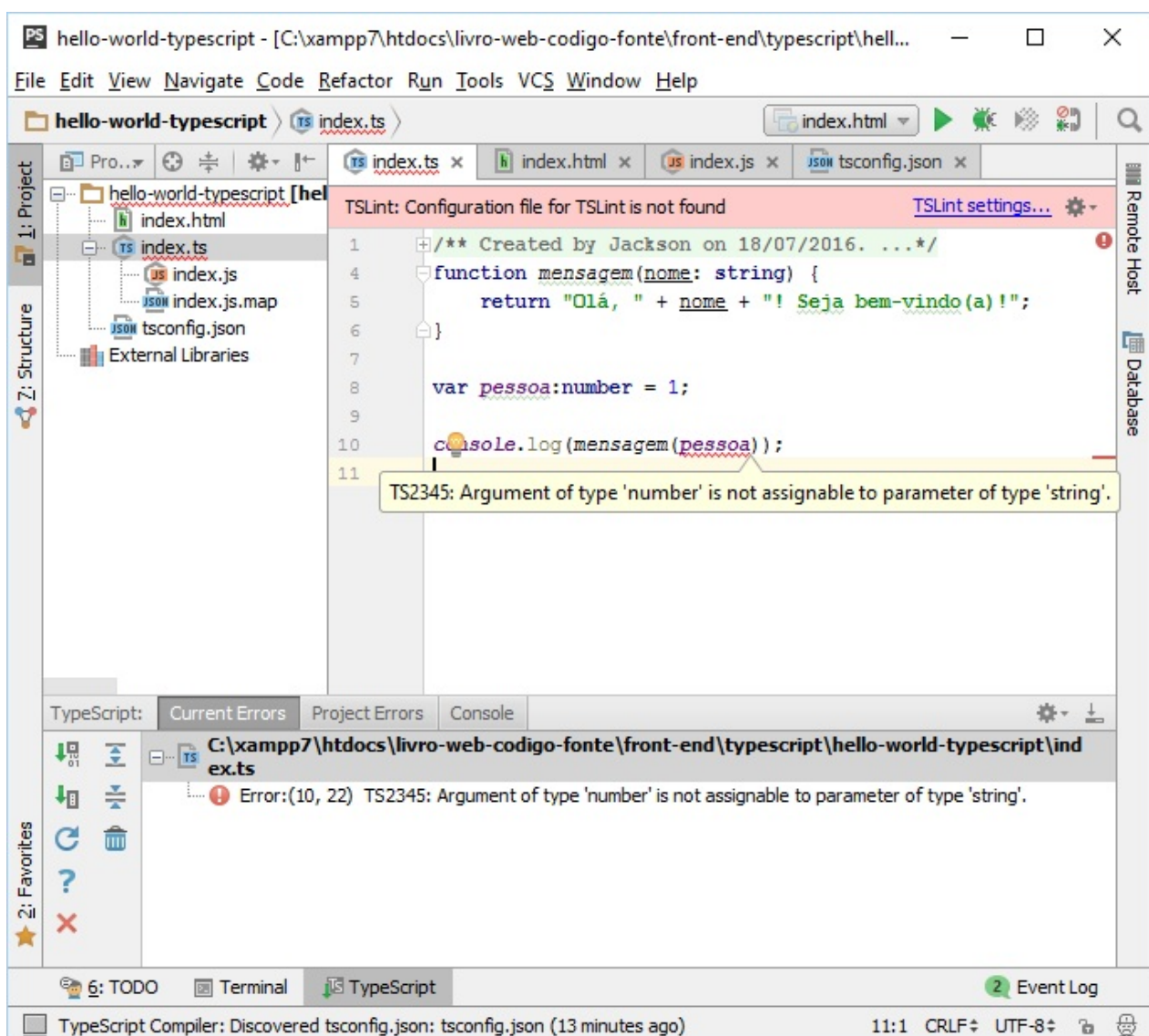
Como se poderia esperar, o código JavaScript continua idêntico, mesmo com a utilização de tipos de dados no arquivo TypeScript. Modifique o arquivo *index.ts* mais uma vez.

```
function mensagem(nome: string) {
    return "Olá, " + nome + "! Seja bem-vindo(a)!";
}
var pessoa:number = 1;
console.log(mensagem(pessoa));
```

Esse é um erro simplório, certamente, mas demonstra recursos interessantes.

A variável `pessoa` tem o tipo `number` (representa um valor numérico). Segundo, na chamada da função `mensagem()` há um erro: não se pode atribuir um valor do tipo `number` para um parâmetro do tipo `string`.

Tanto o editor quanto o painel TypeScript apresentam indicações do erro.



Isso é muito interessante do ponto-de-vista do desenvolvedor, principalmente para grandes projetos.

Os próximos dois projetos apresentarão recursos voltados para a programação orientada a objetos.

Hello World, Interfaces!

Interfaces são um recurso muito importante em linguagens fortemente tipadas, como Java e C#. Com TypeScript não é diferente, pois esse recurso também está disponível. Modifique o arquivo *index.ts* para o seguinte.

```
interface Pessoa {
    primeiroNome: string;
    ultimoNome: string;
}

function mensagem(pessoa: Pessoa) {
    return "Olá, " + pessoa.primeiroNome + " " + pessoa.ultimoNome + "! Seja bem-vindo(a)!";
}

var pessoa:Pessoa = {
    primeiroNome: "José",
    ultimoNome: "Silva"
}

console.log(mensagem(pessoa));
```

Vamos por partes. Primeiro, a definição da interface Pessoa.

```
interface Pessoa {
    primeiroNome: string;
    ultimoNome: string;
}
```

A sintaxe é `interface Nome { lista de atributos e métodos }`. Neste caso, o código define a interface `Pessoa`, que tem os atributos `primeiroNome` (do tipo `string`) e `ultimoNome` (também `string`).

Geralmente, as linguagens de programação mais conhecidas requerem que o código crie uma classe para implementar a interface. Com TypeScript isso não é necessário.

```
var pessoa:Pessoa = {
    primeiroNome: "José",
    ultimoNome: "Silva"
}
```

O código cria a variável `pessoa`, do tipo `Pessoa`. O valor atribuído a ela representa uma "estrutura" que combina com a definição da interface `Pessoa`. Isso é tudo. Não é necessário criar uma classe que implementa a interface para criar uma instância de objeto com a sua estrutura.

Agora é sua vez de brincar um pouco. Veja o código do arquivo *index.js*. O que mudou? Perceba como o PHPStorm ajuda na completção de código. Gere erros no código para ver o resultado da compilação.

Hello World, agora com classe!

Juntamente com o recurso de interfaces, a utilização de classes em programação orientada a objetos permite a definição de uma estrutura de dados. A seguir, o código do arquivo *index.ts*.

```
interface Pessoa {
    primeiroNome: string;
    ultimoNome: string;
}

class Aluno {
    nome: string;
    constructor(public primeiroNome:string, public ultimoNome:string) {
        this.nome = primeiroNome + " " + ultimoNome;
    }
}

function mensagem(pessoa: Pessoa) {
    return "Olá, " + pessoa.primeiroNome + " " + pessoa.ultimoNome + "! Seja bem-vindo (a)!";
}

var pessoa = new Aluno("José", "Silva");

console.log(mensagem(pessoa));
```

Mais uma vez, vamos por partes. Primeiro, a definição da classe `Aluno`.

```
class Aluno {
    nome: string;
    constructor(public primeiroNome:string, public ultimoNome:string) {
        this.nome = primeiroNome + " " + ultimoNome;
    }
}
```

A sintaxe é `class Nome { definições }`. Neste caso, ocorre o seguinte:

- A classe `Aluno` possui o atributo `nome`, do tipo `string`;
- O construtor da classe (definido pela função `constructor()` — perceba que não é utilizada a palavra `function`) aceita dois parâmetros: `primeiroNome`, do tipo `string`, e `ultimoNome`, também `string`.

Os parâmetros do construtor da classe `Aluno` também são marcados com `public`. Isso quer dizer que o código está utilizando um recurso da linguagem que cria, automaticamente, atributos públicos conforme esses parâmetros do construtor.

Agora... veja o que aconteceu com o arquivo *index.js*. Certamente, o código demonstra o emprego de um esforço maior para conseguir o mesmo recurso.

A variável `pessoa` recebe uma instância (objeto) de `Aluno`. Na sequência, algo chama a atenção. A definição da função `mensagem()` indica que ela continua aceitando um parâmetro do tipo `Pessoa`. Entretanto, a chamada da função indica que está sendo passado um valor do tipo `Aluno`. O que chama a atenção é que a compilação não indicou erros. O que acontece?

Estruturas de dados representadas por interfaces, classes e objetos têm uma relação bem próxima. O TypeScript interpreta, na verdade, a correspondência da estrutura do valor. Como a estrutura da variável `pessoa` (que é do tipo `Aluno`) combina com a estrutura do parâmetro `pessoa` da função `mensagem()` (que é do tipo `Pessoa`), não há erro. Aqui, "combinar" significa: ambos os tipos de dados contêm os atributos `primeiroNome` e `ultimoNome` (com seus tipos de dados correspondentes).

Esse é o mesmo recurso que torna válido o código a seguir.

```
var pessoa = { primeiroNome: "José", ultimoNome: "Silva" };
```

O objeto atribuído à variável `pessoa` contém a estrutura esperada para combinar com o tipo do parâmetro da função `mensagem()`.

Isso conclui o **Quickstart**.

Tipos de Dados

Os tipos de dados em TypeScript são:

- Boolean
- Number
- String
- Array
- Tuple
- Enum
- Any
- Void

Os três primeiros, Boolean, Number e String, são chamados "tipos primitivos". Os demais possuem especificações de tipos mais detalhadas, como será visto a seguir.

Boolean

Tipos de dados `boolean` podem conter valores `true` ou `false`.

```
let completo: boolean = false;
```

Number

Todos os números em TypeScript são valores em ponto flutuante. O tipo `number` representa qualquer valor numérico, inclusive em outros sistemas de numeração.

```
let d: number = 6;  
let f: number = 6.1;  
let h: number = 0xf00d; // 0x... número hexadecimal  
let b: number = 0b1010; // 0b... um número binário  
let o: number = 0o744;  // 0o... número octal
```

O tipo `string` representa literais (conjuntos de caracteres). Podem ser usadas aspas simples ou duplas.

```
let cor: string = 'azul';  
cor = 'preto';
```

TypeScript fornece o recurso de **template strings**, que podem conter várias linhas e incluir expressões. Esta representação de string usa o acento grave (``) e as expressões são representadas com a sintaxe `${ expr }`.

```
let nome: string = `João`;
let idade: number = 37;
let mensagem: string = `Olá, meu nome é ${ nome }.

Completarei ${ idade + 1 } anos de idade no próximo mês.`;
```

Isso seria o equivalente a declarar uma `string` como:

```
let mensagem: string = 'Olá, meu nome é ' + nome + '.\n\n' +
'Completarei ' + (idade + 1) + ' anos de idade no próximo mês.';
```

Array

Tipos de dados array podem ser declarados de duas formas. A primeira delas usa o tipo dos elementos do array seguido de `[]`.

```
let numeros: number[] = [1, 2, 3, 4, 5];
```

A outra forma utiliza a classe `Array` e o conceito de **generics** (veremos mais adiante).

```
let numeros: Array<number> = [1, 2, 3, 4, 5];
```

Nas duas formas, o código determina o tipo de dados dos elementos do array. Assim, a regra é que todos os elementos do array tenham o tipo usado na sua declaração.

O tipo `Array` é uma exceção aos outros tipos. `Number`, `Boolean` e `String`, pois é um tipo de instância. Na prática, `Array` é do tipo `object` (herda da classe `object` como todo tipo de instância).

Tuple

O tipo **tuple** (tupla) parece com uma especialização de um array, contendo um número finito de elementos.

```
// declara uma variável do tipo tupla com dois elementos
let x: [number, number];
// atribui valor para a variável
x = [10, 10];

// declara uma variável do tipo tupla com três elementos e a inicializa
let indice: [number, string, number] = [1, 'José', 40];
```

Para acessar os elementos de uma variável do tipo tupla utiliza-se a mesma sintaxe para acessar elementos de um array.

```
console.log(x[0]);
```

Ao acessar um índice fora dos limites da declaração da variável, o TypeScript emprega o conceito de **union types** (isso será visto mais adiante).

```
x[4] = 1; // ok, os elementos podem ser dos tipos number ou string
x[5] = false; // erro, boolean não pode ser atribuído a 'number | string'
```

Na prática, uma tupla é do tipo `Array`.

Enum

O tipo `enum` representa uma maneira de dar nomes mais amigáveis para um conjunto de valores.

```
enum Cor { Vermelho, Verde, Azul};
let c: Cor = Cor.Vermelho;
let c: Cor = 0; // é a mesma coisa que a linha de cima
```

Por padrão, os elementos de um `enum` começam com `0`, mas isso pode ser modificado manualmente.

```
enum Cor { Vermelho = 1, Verde = 3, Azul = 5};
let c: Cor = Cor.Vermelho;
```

Também é possível acessar um `enum` pelo índice e ter acesso ao nome do elemento.

```
enum Cor { Vermelho, Verde, Azul };
let v: string = Cor[0];
console.log(v); // imprime 'Vermelho'
```


Na prática, Enum é do tipo `number` .

Any

Quando não for possível conhecer um tipo de dados ou quando for necessário abrir mão da checagem de tipo em tempo de compilação, pode-se utilizar o tipo `any` .

```
let n: any = 4;
n = 'uma string';
n = false;
```

O tipo `any` também pode ser útil para declarar um variável do tipo `array` cujos tipos de dados dos elementos devem ser diferentes -- ou não são conhecidos.

```
let lista: any[] = [1, true, 'uma string'];
```

Void

O tipo `void` é o oposto de `any` : significa a ausência de um tipo. É mais comum utilizar esse tipo de dados em funções que não retornam um valor.

```
function log(mensagem: string): void {
  console.log(mensagem);
}
```

Além disso, uma variável do tipo `void` pode receber apenas os valores `undefined` ou `null` .

```
let n: void = undefined;
n = null;
```

Na prática, o valor `undefined` é do tipo `undefined` e o valor `null` é do tipo `object` .

Asserção de tipos

O recurso de asserção de tipos é utilizado para afirmar, no código, qual o tipo de dados que se deseja utilizar. Isso também é conhecido como **cast**. A sintaxe é utilizar o tipo de dados que se deseja entre `<>` antes do valor ou da variável.

```
let valor: any = 'uma string';  
let tamanho: number = (<string>valor).length;
```

Outra sintaxe disponível é utilizar o operador `as`.

```
let valor: any = 'uma string';  
let tamanho: number = (valor as string).length;
```

Para identificar o tipo de dados de uma variável ou de um valor podem ser utilizados os operadores:

- `typeof` : para tipos primitivos
- `instanceof` : para tipos de instância (objetos)

Declaração de variáveis

Declarações usando `var`

A palavra `var` é a maneira mais comum para declaração de uma variável. Há algumas questões ao utilizar `var` em JavaScript. Por exemplo:

```
function f() {  
  var a = 10;  
  return function g() {  
    var b = a + 1;  
    return b;  
  }  
}  
var g = f();  
g(); // retorna 11;
```

Nesse caso, a função `g()`, declarada dentro de `f()` tem acesso à variável `a`. Assim, entende-se que a declaração de `a` tem escopo de função (está visível dentro da função `f()` também para sub-funções declaradas dentro dela, como é o caso da função `g()`).

Outro exemplo.

```
function f(inicializar: boolean) {  
  if (inicializar) {  
    var x = 10;  
  }  
  
  return x;  
}  
  
f(true); // retorna 10  
f(false); // retorna undefined
```

A variável `x` foi declarada dentro do `if`, entretanto, está sendo acessada ao final da função `f()`.

Declarações usando `let`

Declarações `let` são similares a declarações `var`, com a diferença que utilizam escopo mais restrito. Por exemplo:

```
function f(inicializar: boolean) {  
  if (inicializar) {  
    let x = 10;  
  }  
  return x; // erro variável não acessível  
}
```

Nesse caso, a variável `x` foi declarada dentro do `if` utilizando `let`. Assim, ela só pode ser acessada dentro desse escopo.

O mesmo acontece em laços de repetição.

```
for(let i = 0; i < 10; i++) {  
  console.log(i);  
}
```

No caso de laços de repetição, é criado um escopo novo e exclusivo para cada passo da execução.

Declarações `const`

Declarações `const` são como `let`, mas o valor da variável não pode ser modificado depois da atribuição.

```
const ladosDoTriangulo: number = 3;
```

É importante diferenciar o conceito de `const` de variáveis *imutáveis*.

```
const pessoa = {  
  id : 1,  
  nome : 'José'  
};  
  
pessoa.id = 10;  
pessoa.nome = 'Maria';
```

Neste caso, não se pode atribuir novo valor para a variável `pessoa`, declara com `const`, mas, como é um objeto, é possível modificar os valores dos seus atributos.

Desconstrução

Desconstrução de Array

A desconstrução de `array` permite "expandir" os valores dos elementos de um array, atribuindo-os a variáveis independentes.

```
let ponto: number[] = [1, 5];
let [x, y] = ponto;
console.log(x); // imprime 1
console.log(y); // imprime 5
```

Esse recurso também é interessante para trocar os valores entre duas variáveis.

```
let [x, y] = [y, x];
```

Outro recurso de desconstrução de array é acessar os elementos restantes por meio da sintaxe `...nome`.

```
let [primeiro, segundo, ...outros] = [1, 2, 3, 4, 5];
console.log(primeiro); // imprime 1
console.log(segundo); // imprime 2
console.log(outros); // imprime [3, 4, 5]
```

Desconstrução de Object

Também é possível desconstruir objetos.

```
let o = {
  a: 'foo',
  b: 12,
  c: false
};
let {a, b} = o;
console.log(a); // imprime foo
console.log(b); // imprime 12
```

Neste caso, é feita uma correspondência entre os nomes das variáveis e os nomes dos atributos do objeto. O código a seguir teria, portanto, um erro.

```
let o = {  
  a: 'foo',  
  b: 12,  
  c: false  
};  
let {x, y} = o; // erro: objeto o não possui atributos x e y
```

O código a seguir também é válido e usa o mesmo princípio.

```
let {x, y} = {x: 1, y: 5};
```

A definição de tipos associada a esse recurso torna o código um pouco difícil de entender, na verdade.

```
let {x, y}: {x: number, y: number} = {x: 1, y: 5};
```

Funções

Funções podem ser criadas como funções nomeadas e funções anônimas.

```
// funcao nomeada
function adicionar(x, y) {
    return x + y;
}

// funcao anonima
let add = function(x, y) { return adicionar(x, y); }
```

Funções com tipos

TypeScript permite a definição de tipos em parâmetros de funções e no tipo de retorno da função.

```
function adicionar(x: number, y: number): number {
    return x + y;
}
```

A sintaxe para definição de tipos de variáveis continua válida aqui para a definição de tipos de parâmetros: `nome: tipo` . Além disso, a sintaxe permite a definição do tipo do retorno da função: `function nome(): tipo {}` .

O tipo de retorno da função pode ser além de um tipo primitivo.

```
function somar(p1: {x: number, y: number}, p2: {x: number, y: number}) : {x: number, y: number} {
    let p = {x: p1.x + p2.x, y: p1.y + p2.y};
    return p;
}

let ponto1 = {x: 1, y: 5};
let ponto2 = {x: 10, y: 20};

let ponto3 = somar(ponto1, ponto2); // retorna {x: 11, y: 25}
```

Neste exemplo, a função `somar()` aceita dois parâmetros (`p1` e `p2`) que compartilham a mesma estrutura (possuem os atributos `x` e `y` , ambos do tipo `number`) que também é a mesma do tipo de retorno da função.

Parâmetros opcionais e padrões

No TypeScript, cada parâmetro de função é obrigatório (diferente do JavaScript, que considera os parâmetros como opcionais). Para definir um parâmetro como opcional, adiciona-se o símbolo `?` logo após seu nome.

```
function nome(primeiro: string, ultimo?: string): string {  
    if (ultimo) {  
        return `${primeiro} ${ultimo}`;  
    } else {  
        return primeiro;  
    }  
}  
  
nome('José', 'Silva'); // retorna 'José Silva'  
nome('José'); // retorna 'José'
```

Como acontece em linguagens que fornecem o recurso de parâmetros opcionais, eles devem estar presentes na lista de parâmetros após os parâmetros obrigatórios.

Há também o recurso de valores padrões para parâmetros.

```
function inicializar(valor: number = 0) : number {  
    return valor;  
}  
  
inicializar(); // retorna 0  
inicializar(10); // retorna 10
```

A utilização de valores padrões para parâmetros também torna opcional a passagem de valor na chamada da função.

Parâmetros rest

Não confundir com REST, recurso para definição de serviços sobre HTTP

JavaScript fornece a variável `arguments` para funções. TypeScript, por outro lado, como trata parâmetros de funções de uma maneira mais estrita, fornece o recurso de "parâmetros rest", que cria um array com argumentos passados para a função.


```
function concatenar(primeiro: string, ...ultimos: string[]): string {  
    return primeiro + ' ' + ultimos.join(' ');  
}  
  
concatenar('a', 'b', 'c', 'd', 'e'); // retorna 'a b c d e';
```

A sintaxe é `...nome: tipo`. No caso do exemplo anterior, o parâmetro rest é `ultimos`, do tipo `string[]`.

Sobrecarga

O recurso de sobrecarga de métodos (ou funções) está presente no TypeScript.

```
let pessoas = ['josé', 'maria', 'pedro', 'joana'];  
  
function consultar(nome: string): number;  
function consultar(indice: number): string;  
function consultar(p: any): any {  
    if (typeof p == 'string') {  
        let idx: any = undefined;  
        let achou: boolean = false;  
        for(let i: number = 0; i < pessoas.length && !achou; i++) {  
            if (pessoas[i] == p) {  
                achou = true;  
                idx = i;  
            }  
        }  
        return idx;  
    }  
    if (typeof p == 'number') {  
        return pessoas[p];  
    }  
}  
  
consultar(1); // retorna 'maria'  
consultar(-1); // retorna undefined  
consultar('maria'); // retorna 1  
consultar('mariana'); // retorna undefined
```

Neste exemplo, a função `consultar()` possui três "versões":

- a primeira, `consultar(nome: string): number` recebe uma `string` como parâmetro (o nome da pessoa) e retorna o índice dela no array `pessoas`; *não há implementação (código)*;
- a segunda, `consultar(indice: number): string` recebe um `number` como parâmetro (o índice da pessoa no array) e retorna o nome da pessoa; *não há implementação*

(código);

- a terceira, `consultar(p: any): any`, é que realmente fornece alguma implementação (código para a função)

O operador `typeof` é utilizado para distinguir os tipos dos parâmetros e, assim, utilizar a lógica necessária.

A utilidade de utilizar sobrecarga de funções é adicionar recurso de validação de tipos para se certificar que a função aceite apenas `string` ou `number` como valor para seu parâmetro.

Outro exemplo:

```
function somar(p1: number[], p2x: number[]) : number[];
function somar(p1: {x: number, y: number}, p2: {x: number, y: number}) : {x: number, y: number};
function somar(p1: any, p2: any): any {
  if (p1 instanceof Array) {
    return [p1[0] + p2[0], p1[1] + p2[1]];
  } else {
    return {x: p1.x + p2.x, y: p1.y + p2.y};
  }
}

let ponto1 = {x: 1, y: 5};
let ponto2 = {x: 10, y: 20};

somar(ponto1, ponto2); // retorna {x: 11, y: 25}
somar([1, 1], [2, 2]); // retorna [3, 3]
```

Neste caso, o operador `typeof` não será útil pois `typeof [1, 2]` retorna `object`, da mesma forma que `typeof {x: 1, y: 2}`, assim, não seria possível distinguir os tipos dos parâmetros. O operador `instanceof`, por outro lado, identifica o tipo específico do objeto, assim, `p1 instanceof Array` permite checar se o parâmetro `p1` é do tipo (ou, uma instância de) `Array`.

Portanto, a regra é:

- para tipos primitivos (tipos de valor), usar `typeof`
- para tipos de instância, usar `instanceof`

Classes

TypeScript fornece recursos completos do paradigma de Programação Orientada a Objetos. Um destes recursos é a utilização de classes, para detinição de tipos de dados, bem como a utilização de herança.

Exemplo:

```
class Saudacao {
  mensagem: string;
  constructor(mensagem: string) {
    this.mensagem = mensagem;
  }
  saudar(pessoa: string): string {
    let m: string = this.mensagem;
    m = m.replace('{pessoa}', pessoa);
    return m;
  }
}

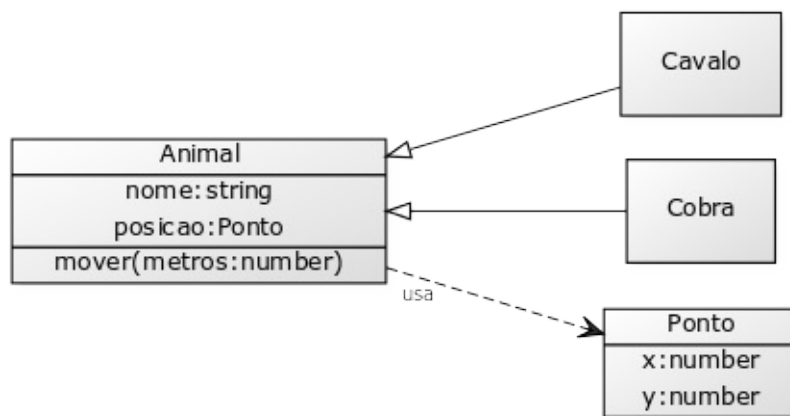
let s:Saudacao = new Saudacao('Olá, {pessoa}!'); // instância de Saudacao
s.saudar('José'); // retorna 'Olá, José!'
```

A sintaxe é similar à de outras linguagens orientadas a objeto, como C# e Java:

- A palavra `class` é utilizada para declarar a classe.
- A declaração de atributos da classe precede o construtor
- O construtor da classe é representado pela função `constructor()`
- Membros da classe são acessados por meio da palavra `this`
- Outras funções da classe são declaradas após o construtor
- A palavra `new` é utilizada para se criar uma instância da classe

Herança

Um dos padrões de projeto mais comuns em programação orientada a objeto é a utilização de herança, permitindo a extensão de tipos para criar novos. TypeScript também fornece esse recurso. O diagrama de classes a seguir apresenta a relação entre quatro classes: Animal, Cavalo, Cobra e Ponto.



A classe `Animal` usa a classe `Ponto` (por causa do atributo `posicao`). As classes `Cavalo` e `Cobra` herdam (ou estendem) a classe `Animal`. O código a seguir representa esse domínio em TypeScript.

```
class Ponto {
    constructor(public x: number = 0, public y: number = 0) {
    }
}

class Animal {
    nome: string;
    posicao: Ponto;
    constructor(nome: string) {
        this.nome = nome;
        this.posicao = new Ponto();
    }
    mover(metros: number = 0): void {
        this.posicao.x += metros;
    }
}

class Cobra extends Animal {
    constructor(nome: string) {
        super(nome);
    }
    mover(metros = 5): void {
        console.log("Deslizando...");
        super.mover(metros);
    }
}

class Cavalo extends Animal {
    constructor(nome: string) {
        super(nome);
    }
    mover(metros = 45): void {
        console.log("Galopando...");
        super.mover(metros);
    }
}

let nagini = new Cobra("Nagini");
let ajax: Animal = new Cavalo("Ajax");

nagini.mover();
ajax.mover(50);
```

A palavra `extends` é utilizada para criar subclasses `Cavalo` e `Cobra`, que herdam de `Animal`.

A palavra `super` é utilizada para acessar um membro da superclasse. Por exemplo, no construtor da classe `Cobra` há uma chamada ao construtor da classe `Animal` por meio de `super()`.

Importante destacar a diferença entre `this` e `super`. Enquanto `this` é usado para referência à classe em questão, `super` é utilizado para referência à superclasse. A diferença pode ser sutil (já que o mecanismo de herança realmente torna parte da subclasse tudo o que está definido na superclasse como `public` ou `protected`) e é fundamental para o recurso de *sobrescrita de métodos*.

O recurso de sobrescrita de métodos nas subclasses permite que uma subclasse modifique comportamento de um método herdado da superclasse. A classe `Cobra`, por exemplo, sobrescreve o método `mover()` da classe `Animal` adicionando funcionalidade e pode chamar o método de mesmo nome da superclasse por meio de `super.mover()`.

Um comportamento bastante interessante do código é que a variável `ajax` é declarada como sendo do tipo `Animal`, mas, por ser uma instância de `Cavalo`, é o método `mover()` de `Cavalo` que é chamado em tempo de execução, não de `Animal` -- obviamente, como já foi dito, o método na subclasse chama o da superclasse por meio de `super`.

Modificadores `public`, `private` e `protected`

`public` por padrão

Em algumas linguagens de programação orientadas a objetos o padrão para declaração de membros da classe é que sejam `private`. No caso do TypeScript, o padrão é que os membros são `public`.

```
class Animal {  
  public nome: string;  
  constructor(nome: string) {  
    this.nome = nome;  
  }  
}  
let garfield = new Animal('garfield');  
console.log(garfield.nome); // imprime 'garfield'
```

O modificador `public` permite que o membro da classe seja acessível de fora dela.

Por ser o modificador padrão, ele não precisa ser utilizado explicitamente.

Modificador `private`

Um membro marcado como `private` só pode ser acessado dentro da própria classe.

```
class Animal {
  private nome: string;
  construtor(nome: string) {
    this.nome = nome;
  }
}
let garfield = new Animal('garfield');
console.log(garfield.nome); // ERRO! membro privado inacessível
```

Uma característica importante do TypeScript ao tratar o modificador `private` precisa ser destacada. No momento de verificar a equivalência entre tipos, o TypeScript considera os atributos que não estejam marcados como `private`. Exemplo:

```
class Animal {
  private nome: string;
  constructor(nome: string) {
    this.nome = nome;
  }
}

class Cobra extends Animal {
  constructor(nome: string) {
    super(nome);
    console.log(this.nome); // ERRO! atributo privado na superclasse
    console.log(super.nome); // ERRO! atributo privado na superclasse
  }
}

class Pessoa {
  private nome: string;
  constructor(nome: string) {
    this.nome = nome;
  }
}

let animal = new Animal('gato');
let cobra = new Cobra('cobra');
let pessoa = new Pessoa('pessoa');

animal = cobra; // OK! estrutura de tipos equivalente
animal = pessoa; // ERRO! estrutura de tipos não equivalente
```

O código anterior declara as classes `Animal`, `Cobra` (subclasse de `Animal`) e `Pessoa`. A classe `Animal` possui o atributo `nome`, que está marcado como `private`, o que também acontece de forma similar com a classe `Pessoa`.

O primeiro erro é tentar acessar o atributo `nome` na classe `Cobra`. Como ele foi declarado como `private`, somente a classe em que foi declarado possui acesso a ele -- não importa se acessando como `this` ou `super`.

O outro erro é menos claro e depende do entendimento do mecanismo de equivalência de tipos (ou estrutura) do TypeScript. Antes, na verdade, é importante lembrar do mecanismo de tipos. Uma variável declarada sem um tipo explícito continua tendo um tipo (mesmo que `undefined`, se não tiver sido iniciada). Isso quer dizer que as variáveis no código têm seus tipos:

- `animal` é do tipo `Animal`
- `cobra` é do tipo `Cobra`
- `pessoa` é do tipo `Pessoa`

A atribuição `animal = cobra` funciona porque as estruturas (os tipos) destas duas variáveis são equivalentes (neste caso, por causa do mecanismo de herança).

A atribuição `animal = pessoa` não funciona porque as suas estruturas não são equivalentes. Embora ambas possuam um atributo `nome: string`, o fato de serem marcados como `private` implica na conclusão de que são, estruturalmente, diferentes. Por isso, o código falha nesse ponto.

Modificador `protected`

Membros de classe declarados como `protected` podem ser acessados apenas pela classe e por suas subclasses, continuam inacessíveis de fora da classe.

```
class Animal {
  protected nome: string;
  constructor(nome: string) {
    this.nome = nome;
  }
}

class Cobra extends Animal {
  constructor(nome: string) {
    super(nome);
    console.log(this.nome); // ok
  }
}

let gato = new Animal('gato');
console.log(gato.nome); // ERRO! atributo não acessível
```

Propriedades parâmetros

O TypeScript fornece o recurso de propriedades parâmetros, que fornecem um atalho para a declaração de atributos.

```
class Animal {  
    constructor(public nome: string) {  
    }  
}  
  
let animal = new Animal('gato');  
console.log(animal.nome);
```

A diferença é que os parâmetros do construtor podem vir acompanhados de um modificador de acesso. No caso do código anterior, o parâmetro `nome` é marcado com `public`, o que torna este parâmetro um atributo da classe. Assim, pode-se acessá-lo posteriormente e não é necessário atribuir valores de parâmetros do construtor para atributos da classe.

Acessores (`get` e `set`)

TypeScript fornece o recurso de "acessores", também conhecido como `get/set`.

```
class Animal {  
    constructor(private nome?: string) { }  
    get Nome(): string {  
        return this.nome;  
    }  
    set Nome(nome: string) {  
        if (nome.length > 5) {  
            this.nome = nome;  
        } else {  
            console.log('Valor inválido para o atributo `nome`');  
        }  
    }  
}  
  
let animal = new Animal();  
animal.Nome = 'gato';
```

O acessor `get` permite retornar o valor de um atributo marcado como `private`, enquanto o `set` permite definir o valor. A maior utilidade de acessores é na garantia de certas regras de negócio. No exemplo anterior, a regra é que a `string` a ser atribuída para o atributo `nome` precisa ter mais de cinco caracteres.

Membros `static`

O recurso de membros `static` permite acessar conteúdo de uma classe diretamente, sem a necessidade de uma instância.

```
class Animal {  
  private static instancias: Animal[] = [];  
  constructor(public nome?: string) {  
    Animal.instancias.push(this);  
  }  
  static get Instancias(): Animal[] {  
    return Animal.instancias;  
  }  
}  
  
let gato = new Animal('gato');  
let cachorro = new Animal('cachorro');  
console.log(Animal.Instancias);
```

Algumas linguagens de programação fornecem a palavra `self`, que dá acesso a membros estáticos. No caso do TypeScript, usa-se diretamente o nome da classe. No código anterior, há o atributo `instancias`, marcado como `private` e `static`. Dentro da classe, o atributo deve ser acessado como `Animal.instancias`, ou seja, usa-se o nome da própria classe (ao invés de `self`).

O segundo membro marcado como `static` é o acessor `get`. Nesse caso, a classe fornece uma espécie de *acesso apenas de leitura* para o atributo `instancias`, já que não disponibiliza o acessor `set`.

Classes abstratas

O recurso de classes abstratas permite definir a regra de uma classe não poder ser instanciada. Além disso, permite que métodos sejam definidos sem código, situação na qual subclasses são obrigadas a fornecer implementações para tais métodos.

```
abstract class Animal {  
  abstract emitirSom(): void;  
  mover(metros: number = 5): void {  
    console.log(`Movendo-se por ${metros}m`);  
  }  
}  
  
class Cobra extends Animal {  
  emitirSom(): void {  
    console.log('Barulho de cobra...');  
  }  
}  
  
let cobra = new Cobra();  
cobra.emitirSom();  
cobra.mover();  
cobra.emitirSom();  
  
let gato = new Animal(); // ERRO! não pode instanciar classe abstrata
```

A palavra `abstract` é utilizada para marcar a classe como abstrata e um método como não contendo código. No código anterior, a classe `Animal` é abstrata e contém o método `emitirSom()`, que também é abstrato. Como regra, se a classe possui um método abstrato, ela também precisa ser abstrata.

Visão Geral

Bootstrap é um framework front-end (HTML, CSS e JavaScript). Provavelmente, é um dos mais populares. Dentre seus principais recursos estão o suporte para o desenvolvimento de web sites e aplicações web responsivas e adaptadas para dispositivos móveis.

Para utilizar o Bootstrap, o primeiro passo é fazer o [download](#). Outras formas de utilização estão disponíveis, mas, por enquanto, vamos ter enfoque nesta (baixar os arquivos).

Após o download, descompacte o arquivo. Você verá uma estrutura de pastas semelhante ao seguinte:

```
bootstrap/
├── css/
│   ├── bootstrap.css
│   ├── bootstrap.css.map
│   ├── bootstrap.min.css
│   ├── bootstrap-theme.css
│   ├── bootstrap-theme.css.map
│   └── bootstrap-theme.min.css
├── js/
│   ├── bootstrap.js
│   └── bootstrap.min.js
└── fonts/
    ├── glyphsicons-halflings-regular.eot
    ├── glyphsicons-halflings-regular.svg
    ├── glyphsicons-halflings-regular.ttf
    ├── glyphsicons-halflings-regular.woff
    └── glyphsicons-halflings-regular.woff2
```

Os arquivos principais são `bootstrap.css` (e sua versão compactada `bootstrap.min.css`) e `bootstrap.js` (e sua versão compactada `bootstrap.min.js`). Opcionalmente, você pode utilizar o arquivo de tema (por exemplo: `bootstrap-theme.min.css`). O Bootstrap também usa a biblioteca jQuery, necessária para o funcionamento de alguns componentes.

Os arquivos também pode ser utilizados online, sem necessidade de download, direto da CDN.

Estrutura básica do HTML

O código a seguir apresenta a estrutura básica de um arquivo HTML com Bootstrap.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap 101 Template</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstr
ap.min.css" />

    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
    <![endif]-->
  </head>
  <body>
    <h1>Hello, Bootstrap!</h1>

    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></
script>
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></scri
pt>
  </body>
</html>
```

Veja o exemplo em funcionamento [aqui](#).

Importante notar a importação dos arquivos `bootstrap.css` , `jquery.min.js` e `bootstrap.min.js` .

Fundamentos

Contêineres

Bootstrap requer um elemento contêiner, que conterá outros elementos da página e o sistema de grid. Há dois contêineres, representados por classes CSS: `container` e `container-fluid`. A diferença entre os dois é que o primeiro tem largura fixa, enquanto o segundo tem largura dependente da janela de visualização (do browser). Exemplo:

```
<div class="container">
  conteúdo...
</div>
```

Podem existir vários elementos contêineres na mesma página. Além disso, eles podem estar aninhados (o que é assunto para outro momento).

Sistema de Grid

O Bootstrap inclui um sistema de grid que é fluido e responsivo, baseado em 12 colunas. Há um conjunto de classes que facilitam a criação de grid para vários contextos. No geral, os elementos do grid devem estar contidos em um elemento que tenha a classe `row`. Os nomes das classes do grid seguem um padrão: `col-<dispositivo>-<tamanho>`. Dispositivo é definido conforme a tabela a seguir.

Dispositivo	classe
Dispositivos com telas bem pequenas (<768px)	xs
Dispositivos com telas pequenas (>=768px)	sm
Dispositivos com telas médias (>=992px)	md
Dispositivos com telas grandes (>=1200px)	lg

O tamanho varia entre 1 e 12.

Por exemplo, para criar um grid com uma linha e duas colunas, uma com tamanho 8 e outra com tamanho 4, poderia ser utilizado o trecho de código a seguir:

```
<div class="container">
  <div class="row">
    <div class="col-md-8">
      Conteúdo do primeiro elemento do grid.
    </div>
    <div class="col-md-4">
      Conteúdo do segundo elemento do grid.
    </div>
  </div>
</div>
```

A classe `col-md-8` define uma coluna com tamanho 8, enquanto a classe `col-md-4` define uma coluna com tamanho 4. O importante é que a soma dos tamanhos das colunas em uma linha não seja superior a 12.

O Bootstrap (via CSS) calcula automaticamente o tamanho real da coluna conforme a tela do dispositivo de visualização. Por exemplo, conforme a tela do dispositivo, um grid com uma linha e duas colunas pode ser apresentado como uma coluna apenas. Um recurso interessante é utilizar várias classes no mesmo elemento, com enfoque em telas de tamanhos diferentes.

O exemplo a seguir demonstra a utilização do grid.

```
<!DOCTYPE html>
<html lang="pt-br">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap 101 Template</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css" />

    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
    <![endif]-->
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col-xs-8 col-md-8">
          Conteúdo do primeiro elemento do grid.
        </div>
        <div class="col-xs-4 col-md-4">
          Conteúdo do segundo elemento do grid.
        </div>
      </div>
    </div>

    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></script>
  </body>
</html>
```

Veja o exemplo em funcionamento [aqui](#).

Se você abrir o exemplo em telas de tamanho diferente (por exemplo, redimensionando a janela do browser) perceberá como o Bootstrap trata o grid

Formulários

O Bootstrap fornece classes para lidar com formulários. As classes CSS mais utilizadas neste processo são `form-control` (aplicadas a campos e elementos de formulário) e `form-group`, para melhor espaçamento dos elementos do formulário.

As seções a seguir demonstram a utilização dos principais elementos e configurações de formulário. Veja um exemplo completo em funcionamento [aqui](#).

Configurações do formulário

A princípio, não é necessário utilizar uma classe específica para que haja uma correta utilização de um formulário, apenas o elemento `form`. Entretanto, há duas aplicações que podem ser úteis em situações específicas:

- **formulários em linha (in-line):** utilizar a classe `form-inline` faz com que o formulário fique na mesma linha
- **formulários horizontais:** utilizar a classe `form-horizontal` faz com que os rótulos dos campos do formulário fiquem à esquerda, enquanto os campos ficam à direita, na mesma linha

Grupos de elementos

Os grupos de elementos representam um conceito que permite agrupar os elementos necessários para formar uma linha ou um grupo de elementos do formulário. A classe `form-group` é aplicada a um elemento `div` que contém pelo menos dois outros elementos:

- o elemento `label`, que representa o rótulo do campo do formulário
- o campo do formulário, em si

Exemplo:

```
<form>
  <div class="form-group">
    <label for="nome">Nome</label>
    <input type="text" class="form-control" id="nome" placeholder="Informe seu nome">
  </div>
</form>
```

Elemento `input`

O elemento `input` representa o tipo de entrada mais básica do formulário. A classe `form-control` deve ser aplicada ao elemento `input`. O código a seguir demonstra sua utilização.

```
<form>
  <div class="form-group">
    <label for="nome">Nome</label>
    <input type="text" class="form-control" id="nome" placeholder="Informe seu nome">
  </div>
</form>
```

Além de receberem texto (atributo `type` com valor `text`) o HTML5 permite que o elemento `input` valide a entrada, conforme o tipo:

- `password` : campo para entrada de senha
- `datetime` : campo para entrada de data e hora
- `date` : campo para entrada de data
- `month` : campo para entrada de mês
- `time` : campo para entrada de hora
- `number` : campo para entrada de número
- `email` : campo para entrada de endereço de e-mail
- `url` : campo para entrada de endereço de internet (URL)
- `tel` : campo para entrada de telefone
- `color` : campo para entrada de cor

Cada um destes tipos de campos pode ser tratado de uma forma diferente pelo browser, que pode utilizar uma interface específica para facilitar a entrada do usuário.

Elemento `textarea`

O elemento `textarea` permite uma entrada de texto de múltiplas linhas. Exemplo:

```
<form>
  <div class="form-group">
    <label for="endereco">Endereço</label>
    <textarea class="form-control" id="endereco" placeholder="Informe seu endereço com
pleto" rows="5"></textarea>
  </div>
</form>
```

O atributo `rows` define a quantidade de linhas.

Elemento `input` do tipo `radio` e `checkbox`

O elemento `input` pode ter dois tipos que fazem com que o comportamento da entrada seja bem diferente de uma entrada textual:

- `radio` : faz com que a entrada seja um "botão rádio", que é útil para entradas com opções mutuamente exclusivas
- `checkbox` : faz com que a entrada seja um "botão de marcar", que é útil para entradas com múltiplas opções

Além disso, para lidar com `radio` e `checkbox` é necessário utilizar as classes `radio` e `checkbox` , respectivamente. Exemplo:

```
<form>
  <div class="form-group">
    <label>Termos do contrato</label>
    <div class="checkbox">
      <label>
        <input type="checkbox" name="termos" value="aceito">Aceito os termos
      </label>
    </div>
  </div>

  <div class="form-group">
    <label>Sexo</label>
    <div class="radio">
      <label><input type="radio" name="sexo" value="m">M</label>
      <label><input type="radio" name="sexo" value="f">F</label>
    </div>
  </div>
</form>
```

Elemento `select`

O elemento `select` permite uma entrada em que o usuário precisa escolher uma opção dentre um grupo [de opções]. O grupo de opções é representado por um ou mais elementos `option` . Exemplo:

```
<form>
  <div class="form-group">
    <label for="estado">Estado</label>
    <select id="estado" class="form-control">
      <option>Tocantins</option>
      <option>Goiás</option>
    </select>
  </div>
</form>
```

O elemento `option` pode possuir o atributo `value`, que determina o valor da opção. Neste caso, o conteúdo do elemento `option` é o que o usuário vê, enquanto o atributo `value` representa o que é realmente o valor selecionado pelo usuário.

Controle estático

Um control estático é um conceito do Bootstrap para representar uma situação em que um elemento do formulário não recebe entrada. A classe a ser aplicada ao elemento que se tornará estático (que pode ser qualquer elemento do HTML) é `form-control-static`.

Exemplo:

```
<form>
  <div class="form-group">
    <label>Identificador</label>
    <p class="form-control-static">001</p>
  </div>
</form>
```

Configurações específicas dos campos do formulário

Os campos do formulário também podem assumir situações específicas, conforme apresentado a seguir:

- **Campo desabilitado:** não recebe entrada do usuário (funciona também como campo para leitura). Deve-se utilizar o atributo `disabled` (sem valor)
- **Campo somente leitura:** não recebe entrada para o usuário. Deve-se utilizar o atributo `readonly` (sem valor)
- **Campo requerido:** requer uma entrada do usuário. Deve-se utilizar o atributo `required` (sem valor)

Texto de ajuda

A classe `help-block` pode ser utilizada para fornecer ajuda para entrada em um campo. Exemplo:

```
<form>
  <div class="form-group">
    <label>Nome</label>
    <input type="text" class="form-control">
    <span class="help-block">Informe seu nome completo, sem abreviações</span>
  </div>
</form>
```

Tamanho dos controles

O tamanho dos controles pode ser definido por meio das classes `input-lg` (maior tamanho), `input-sm` (menor tamanho). O tamanho normal não necessita de uma classe.

AngularJS

Referências

O material desta seção é construído com base na documentação oficial do AngularJS (<https://docs.angularjs.org/guide/introduction>), no tutorial oficial do AngularJS (<https://docs.angularjs.org/tutorial>), no curso de Angular do W3Schools (<https://docs.angularjs.org/tutorial>) e no curso de Angular do Codeschool (<http://www.w3schools.com/angular/default.asp>).

O que é o Angular?

AngularJS é um framework estrutural para aplicações web. Diferentemente de uma **biblioteca**, que fornece um conjunto de funções reutilizáveis (como o jQuery) e de **frameworks** que implementam aplicações web de formas particulares, o Angular procura um meio termo, minimizando a distância entre uma abordagem orientada ao documento HTML e novas funcionalidades e construções necessárias pela aplicação web. De qualquer forma, nos referimos ao Angular como um **framework JavaScript**.

Um dos principais recursos do Angular são as **diretivas**, que estendem o HTML de diversas formas, como veremos neste material. A tabela a seguir apresenta estes conceitos e recursos.

Conceito	Descrição
Template	Fornece marcação adicional ao HTML
Diretivas	Estendem o HTML com atributos e elementos
Model	O dado apresentado para o usuário na view e com o qual o usuário interage
Escopo	Contexto no qual o model é armazenado para que controllers, diretivas e expressões possam acessá-lo
Expressões	Acessa variáveis e funções do escopo
Compilador	Analisa o template e instancia diretivas e expressões
Filtro	Formata o valor de uma expressão para apresentar ao usuário
View	O que o usuário vê (o DOM)
Data binding	Sincroniza dado entre o model e a view
Controller	A lógica de negócio entre views
Injector	Container de injeção de dependência
Módulo	Um container para diferentes partes de um aplicativo, incluindo controllers, serviços, filtros, diretivas que configuram o injetor
Serviço	Lógica de negócio reutilizável, independente de views

Primeiro exemplo

O exemplo a seguir ilustra alguns dos conceitos, que serão destacados posteriormente.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <title>Exemplo - Data binding</title>
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/1.4.5/angular.min.js"></script>
</head>
<body >
  <div ng-app ng-init="quantidade=1;custo=2">
    <b>Pedido:</b>
    <div>
      Quantidade: <input type="number" min="0" ng-model="quantidade">
    </div>
    <div>
      Custo: <input type="number" min="0" ng-model="custo">
    </div>
    <div>
      <b>Total:</b> {{quantidade * custo | currency}}
    </div>
  </div>
</body>
</html>
```

Você pode ver o exemplo em funcionamento [aqui](#).

Conceitos iniciais

A diretiva mais importante de um aplicativo angular é a `ng-app`. Ela deve estar presente no HTML, no elemento de mais alto nível, que conterá os demais elementos do aplicativo.

Diretivas são aplicadas no HTML de diversas formas. No primeiro exemplo, a diretiva `ng-app` foi aplicada a um elemento `div`, na linha 9, como um atributo. Se preferir utilizar um elemento de mais alto nível ainda, pode aplicar a diretiva `ng-app` no elemento `html`.

A diretiva `ng-init`, no elemento `div` da linha 9, permite utilizar código para inicializar elementos do model. Neste caso, dois elementos do model, chamados `quantidade` e `custo`, são inicializados com valores `1` e `2`, respectivamente. O conteúdo da diretiva, neste caso, é código JavaScript de atribuição de valores a variáveis.

Nas linhas 12 e 15, respectivamente, estão as diretivas `ng-model` que indicam para o Angular que os campos de texto estão vinculados a elementos do model. O valor da diretiva é o nome do model ao qual está associado o elemento `input`. O processo de vinculação de dados (*data binding*) é um dos elementos-chave do Angular. Por meio deste recurso,

qualquer atualização no model será automaticamente visualizada pelo usuário e qualquer atualização no `input` gerará uma atualização do model (isso é chamado de *two-way data binding*).

Na linha 18 há uma expressão baseada no model e um filtro. Expressões estão presentes entre chaves duplas. Neste caso, a primeira parte da expressão (`quantidade * custo`) corresponde à multiplicação entre dois elementos do model: `quantidade` e `custo` . Posteriormente, após o símbolo `|` está o filtro `currency` . Filtros formatam valores. Neste caso, o filtro `currency` formata um valor para apresentação em formato de moeda.

Iniciando

Há várias formas de utilizar o AngularJS. Algumas delas são as seguintes:

- fazer [download](#)
- usar uma CDN (como <https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js>)
- usar o [Plunker](#)

Independente da forma que você escolher, lembre-se que o angular estará disponível na forma de um arquivo JavaScript, portanto, basta incluí-lo na página HTML de forma apropriada.

Estrutura do aplicativo

Um aplicativo que usa Angular está baseado no padrão arquitetural **MVC** (*Model-View-Controller*). Por causa disso, mesmo um aplicativo mais simples terá uma estrutura natural, baseada no MVC. O exemplo a seguir apresenta este aplicativo simples e a estrutura baseada no MVC. O aplicativo possui dois arquivos:

- `index.html` (view)
- `app.js` (controller)

View

O arquivo `index.html` possui o seguinte conteúdo:

```
<!DOCTYPE html>
<html lang="pt-br" ng-app="appsimples">

  <head>
    <meta charset="utf-8" />
    <title>App Simples</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstr
ap.min.css" />
    <link rel="stylesheet" href="style.css" />

    <script>document.write('<base href="' + document.location + '" />');</script>
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></scri
pt>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.6/angular.min.j
s"></script>

    <script src="app.js"></script>
  </head>

  <body ng-controller="HomeController">
    <div class="container">
      <p>Hello {{nome}}!</p>
    </div>
  </body>

</html>
```

O arquivo `index.html` representa a **View**, ou seja, a parte visual do aplicativo. Na linha 2, a diretiva `ng-app` possui o valor `appsimples`, que é o nome do módulo que representa o aplicativo. Isso quer dizer que a lógica do aplicativo está em um módulo (um aplicativo pode possuir vários módulos e pode existir uma relação de dependência entre eles, o que será visto posteriormente).

Na linha 17, a diretiva `ng-controller` é utilizada para indicar que o elemento `body` em questão está no contexto do **Controller** chamado `HomeController`. Um **controller** é outro elemento do MVC, responsável por ligar a **view** ao **model** (e vice-versa) e representar a lógica de negócio. O restante do código HTML é responsável por apresentar uma mensagem na tela (linha 19) por meio de uma expressão.

Módulo e Controller

O arquivo `app.js` possui o seguinte conteúdo:

```
(function(){  
  
angular.module('appsimples', [])  
  .controller('HomeController', function($scope) {  
    $scope.nome = 'AngularJS';  
  });  
  
})();
```

Pode-se perceber, na linha 3, a chamada da função `angular.module()` passando como parâmetro: o nome do aplicativo (`appsimples`) e a coleção de dependências (neste caso, nenhuma dependência necessária).

A função `angular.module()` retorna um objeto do Angular, que possui função `controller()` . Esta função é chamada com dois parâmetros:

- o nome do controller (`HomeController`); e
- a função anônima que determina o controller.

A lista de parâmetros da função anônima define a lista de dependências do controller. Tanto para as dependências do módulo quanto para as dependências do controller, o Angular utiliza um recurso chamado **injeção de dependência**, que identifica as dependências e trata de instanciar os objetos, caso necessário.

A função anônima que define o controller possui um parâmetro chamado `$scope` , que tem um papel especial no Angular. Na view, o que está contido no elemento no qual o controller está aplicado é gerenciado por meio do objeto `$scope` . Isso quer dizer que um model ou funções utilizadas no controller são definidos por meio deste objeto. Para exemplificar, na linha 5 é criado um elemento `nome` no model. O valor atribuído a `nome` é apresentado na view por meio de uma expressão (linha 19 do arquivo `index.html`).

Veja o aplicativo em funcionamento [aqui](#).

Esta é a estrutura básica do aplicativo Angular baseado no MVC. Nas próximas seções, exploraremos mais dos recursos do Angular e aprenderemos sobre como incluir novos elementos nesta estrutura do aplicativo.

Diretiva ng-click

A diretiva `ng-click` é utilizada para representar um comportamento de clique do mouse em um elemento, por exemplo um botão. O conteúdo da diretiva (o atributo) é um código JavaScript. O exemplo a seguir demonstra a utilização desta diretiva.

```
<!DOCTYPE html>
<html lang="pt-br" ng-app="appsimples">

  <head>
    <meta charset="utf-8" />
    <title>App Smples</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstr
ap.min.css" />
    <link rel="stylesheet" href="style.css" />

    <script>document.write('<base href="' + document.location + '" />');</script>
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></scri
pt>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.6/angular.min.j
s"></script>

    <script src="app.js"></script>
  </head>

  <body ng-controller="HomeController">
    <div class="container">
      <h1>Cidades do Tocantins</h1>

      <div class="row">
        <div class="col-xs-6 col-md-6">
          <h2>Lista de cidades</h2>
          <p>
            <div class="input-group">
              <input type="text" class="form-control" ng-model="q" place
holder="Pesquisar...">
              <span class="input-group-addon">
                <span class="glyphicon glyphicon-search"></span>
              </span>
            </div>
          </p>
          <ul>
            <li ng-repeat="cidade in cidades | filter:q">{{cidade}}</li>
          </ul>
        </div>
        <div class="col-xs-6 col-md-6">
          <h2>Cadastrar cidade</h2>
        </div>
      </div>
    </div>
  </body>
</html>
```

```

        <div class="input-group">
            <input type="text" class="form-control" ng-model="nova_cidade"
placeholder="Informe o nome da cidade...">
            <span class="input-group-btn">
                <button class="btn btn-default" type="button" ng-click="sa
lvar(nova_cidade); nova_cidade=''>Salvar</button>
            </span>
        </div>

    </div>
</div>
</div>
</body>

</html>

```

```

(function(){

angular.module('appsimples', [])
.controller('HomeController', function($scope) {
    $scope.cidades = ['Araguaína', 'Gurupi', 'Palmas', 'Paraíso', 'Porto Nacional'];
    $scope.salvar = function(cidade) {
        $scope.cidades.push(cidade);
    }
});

})();

```

Você pode ver o exemplo em funcionamento [aqui](#).

No controller, no arquivo `app.js`, na linha 6 é definida a função `salvar()`, que aceita como parâmetro o nome da cidade a ser inserida no vetor `cidades`.

Na view, no arquivo `index.html`, na linha 42 está presente o elemento `button` e nele é aplicada a diretiva `ng-click`. Neste caso, a diretiva indica que será chamada a função `salvar()` passando como parâmetro o model `nova_cidade` (vinculado ao `input` da linha 40) e, posteriormente, o valor do model será uma string vazia.

Info Exercício

O exemplo anterior implementa parte das funcionalidades de um CRUD. Neste caso, está implementando a consulta (listagem dos dados) e o cadastro. Implemente a funcionalidade de excluir uma cidade.

Diretivas ng-show e ng-hide

As diretivas `ng-show` e `ng-hide` são utilizadas, respectivamente, para mostrar e ocultar conteúdo. Veja o aplicativo a seguir.

O arquivo `index.html` :

```
<!DOCTYPE html>
<html ng-app="app" lang="pt-br">

  <head>
    <meta charset="utf-8" />
    <title>Demonstração ng-hide e ng-show</title>
    <script>document.write('<base href="' + document.location + '" />');</script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.6/angular.min.js"></script>
    <script src="app.js"></script>
  </head>

  <body>
    <p>Você é um
      <label><input type="radio" ng-model="tipo" value="sith">sith</label> ou um
      <label><input type="radio" ng-model="tipo" value="jedi">jedi</label>?
    </p>
    <p ng-show="tipo=='sith'">"The Force shall set me free" (Sith Code)</p>
    <p ng-show="tipo=='jedi'">"A Jedi uses the Force for knowledge and defense, never for attack" (Yoda)</p>
  </body>

</html>
```

O arquivo `app.js` :

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

Veja o aplicativo em funcionamento [aqui](#).

No arquivo `index.html` , nas linhas 15 e 16, são criados elementos `input` para representar opções para uma pergunta. Cada elemento está vinculado ao modelo `tipo` (por meio da diretiva `ng-model`) e cada uma possui o atributo `value` , indicando o valor de `tipo` quando cada opção estiver selecionada.

Nas linhas 18 e 19 há dois elementos `p` (representando frases dos filmes "Star Wars") nos quais está aplicada a diretiva `ng-show`. O conteúdo da diretiva é uma expressão que deve ser avaliada como `true` ou `false`. Quando a expressão resultar em `true`, o conteúdo ao qual está aplicada a diretiva `ng-show` será apresentado. Caso contrário, será oculto. No caso do aplicativo em questão, o primeiro elemento `p` (linha 18) será apresentado se a primeira opção ("Sith") for selecionada. O mesmo acontece com o segundo elemento `p` (linha 19), que será apresentado se a segunda opção ("Jedi") for selecionada.

A diretiva `ng-hide` tem o comportamento complementar da diretiva `ng-show`, ou seja, oculta um conteúdo quando o valor da expressão for `true`.

Diretiva ng-repeat

A diretiva `ng-repeat` é utilizada para repetir uma parte da view (template) conforme uma expressão de repetição baseada no model. O exemplo a seguir demonstra a utilização desta diretiva.

O arquivo `index.html` :

```
<!DOCTYPE html>
<html lang="pt-br" ng-app="appsimples">

  <head>
    <meta charset="utf-8" />
    <title>App Simples</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstr
ap.min.css" />
    <link rel="stylesheet" href="style.css" />

    <script>document.write('<base href="' + document.location + '" />');</script>
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></scri
pt>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.6/angular.min.j
s"></script>

    <script src="app.js"></script>
  </head>

  <body ng-controller="HomeController">
    <div class="container">
      <h1>Cidades do Tocantins</h1>
      <p>A lista abaixo apresenta algumas cidades do Tocantins:</p>
      <ul>
        <li ng-repeat="cidade in cidades">{{cidade}}</li>
      </ul>
    </div>
  </body>

</html>
```

O arquivo `app.js` :

```
(function(){  
  
angular.module('appsimples', [])  
.controller('HomeController', function($scope) {  
    $scope.cidades = ['Araguaína', 'Gurupi', 'Palmas', 'Paraíso', 'Porto Nacional'];  
});  
  
})();
```

Você pode ver o exemplo em funcionamento [aqui](#).

No arquivo `index.html` (a view), na linha 22, a diretiva `ng-repeat` é aplicada ao elemento `li`. Neste caso, o uso da diretiva, baseada na expressão `cidade in cidades`, faz com que o elemento (e seu conteúdo) seja repetido conforme a quantidade de elementos em `cidades` (esperando-se que seja um vetor). No conteúdo do `li` está uma expressão que apresenta o nome de cada cidade da iteração.

No arquivo `app.js`, no controller `HomeController`, na linha 5, temos a definição do array `cidades` (com os elementos que são apresentados na view).

Filtrando o ng-repeat

Uma funcionalidade bastante interessante dos filtros em Angular é percebida ao serem aplicados à diretiva `ng-repeat`. O exemplo a seguir demonstra isso.

O arquivo `index.html` :

```
<!DOCTYPE html>
<html lang="pt-br" ng-app="appsimples">

  <head>
    <meta charset="utf-8" />
    <title>App Simples</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstr
ap.min.css" />
    <link rel="stylesheet" href="style.css" />

    <script>document.write('<base href="' + document.location + '" />');</script>
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></scri
pt>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.6/angular.min.j
s"></script>

    <script src="app.js"></script>
  </head>

  <body ng-controller="HomeController">
    <div class="container">
      <h1>Cidades do Tocantins</h1>
      <p>A lista abaixo apresenta algumas cidades do Tocantins:</p>
      <ul>
        <li ng-repeat="cidade in cidades | filter:'Ara'">{{cidade}}</li>
      </ul>
    </div>
  </body>

</html>
```

O arquivo `app.js` :

```
(function(){

angular.module('appsimples', [])
.controller('HomeController', function($scope) {
  $scope.cidades = ['Araguaína', 'Gurupi', 'Palmas', 'Paraíso', 'Porto Nacional'];
});

})();
```

Você pode ver o exemplo em funcionamento [aqui](#).

Na view, na linha 22, o conteúdo da diretiva `ng-repeat` foi modificado de modo a incluir o filtro `filter`. Neste caso, o array `cidades` está sendo filtrado, de forma que apenas serão apresentadas as cidades que têm "Ara" como parte do nome.

O exemplo a seguir é mais funcional: filtra os elementos do vetor `idades` com base em uma entrada do usuário.

O arquivo `index.html` :

```
<!DOCTYPE html>
<html lang="pt-br" ng-app="appsimples">

  <head>
    <meta charset="utf-8" />
    <title>App Smples</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstr
ap.min.css" />
    <link rel="stylesheet" href="style.css" />

    <script>document.write('<base href="' + document.location + '" />');</script>
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></scri
pt>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.6/angular.min.j
s"></script>

    <script src="app.js"></script>
  </head>

  <body ng-controller="HomeController">
    <div class="container">
      <h1>Cidades do Tocantins</h1>
      <p>A lista abaixo apresenta algumas cidades do Tocantins:</p>
      <p>
        <div class="input-group">
          <input type="text" class="form-control" ng-model="q" placeholder="
Pesquisar...">
          <span class="input-group-addon">
            <span class="glyphicon glyphicon-search"></span>
          </span>
        </div>
      </p>
      <ul>
        <li ng-repeat="cidade in cidades | filter:q">{{cidade}}</li>
      </ul>
    </div>
  </body>
</html>
```

O arquivo `app.js` :

```
(function(){  
  
angular.module('appsimples', [])  
.controller('HomeController', function($scope) {  
    $scope.cidades = ['Araguaína', 'Gurupi', 'Palmas', 'Paraíso', 'Porto Nacional'];  
});  
  
})();
```

Você pode ver o exemplo em funcionamento [aqui](#).

Na view, na linha 23, o elemento `input` está vinculado ao elemento do model `q`. Desta forma, na linha 28, o filtro é baseado neste elemento do model, não em um valor definido diretamente no código.

Propriedades

Propriedades especiais são expostas com a utilização da diretiva `ng-repeat` :

Propriedade	Tipo	Descrição
<code>\$index</code>	número	Índice da iteração
<code>\$first</code>	booleano	Indica se a iteração é a primeira
<code>\$middle</code>	booleano	Indica se a iteração não é a primeira e nem a última
<code>\$last</code>	booleano	Indica se a iteração é a última
<code>\$even</code>	booleano	Indica se o índice da iteração é par
<code>\$odd</code>	booleano	Indica se o índice da iteração é ímpar

Model complexo

Os exemplos apresentados até o momento lidaram com estruturas mais básicas para o model. O Angular também permite trabalhar com model com uma estrutura mais complexa, como será visto nesta seção. Por "estrutura mais complexa" e por "model complexo" quero dizer que o aplicativo é um CRUD completo e que o model possui uma formulação com duas entidades relacionadas entre si.

Estrutura do aplicativo

O aplicativo realiza o CRUD de cidades. As funcionalidades são:

- listar cidades (com filtro e ordenação)
- cadastrar cidade
- excluir cidade
- editar cidade

Há dois arquivos no aplicativo: `app.js` e `index.html`. O primeiro contém a definição do módulo e do controller, respectivamente: `cidades` e `CidadesController`. O `app.js` contém, ainda, a definição da estrutura de dados usada no model e o `CidadesController` contém a lógica para o CRUD e definir dados iniciais (estados e cidades).

O arquivo `index.html` fica com a responsabilidade de representar a view e o comportamento do aplicativo. Ele contém, essencialmente, código HTML e utiliza as diretivas do angular para que seja possível a comunicação com o módulo `cidades` e seu conteúdo.

Model

Há duas entidades no modelo de dados: `Cidade` e `Estado`. A figura a seguir apresenta um diagrama de classes UML.



A classe `Estado` possui `uf` e `nome`, enquanto a classe `Cidade` possui `estado` e `nome`. Assim, há um relacionamento entre as duas entidades. Estas entidades estão definidas no arquivo `app.js`.

```
function Cidade(estado, nome) {  
    this.estado = estado;  
    this.nome = nome;  
}  
  
function Estado(uf, nome) {  
    this.uf = uf;  
    this.nome = nome;  
}
```

Entretanto, é bem verdade que o JavaScript não requer que estas entidades estejam definidas como "classes", portanto, este tipo de procedimento não é estritamente necessário.

Lógica de negócio (CRUD)

Como já disse, o CRUD está definido no `CidadesController` (arquivo `app.js`) no qual está definida a lógica de negócio para a manipulação do model e a comunicação com a view. As seções a seguir fazem referência ao `CidadesController` e explicam seu funcionamento.

Definição dos dados

O aplicativo começa com alguns dados pré-definidos. Esta etapa poderia ser substituída por uma comunicação com um banco de dados (se estiver disponível).

```
$scope.estados = [  
    new Estado('TO', 'Tocantins'),  
    new Estado('SP', 'São Paulo'),  
    new Estado('MG', 'Minas Gerais')  
];  
  
$scope.cidades = [  
    new Cidade(encontrarEstado('TO'), 'Araguaína'),  
    new Cidade(encontrarEstado('TO'), 'Gurupi'),  
    new Cidade(encontrarEstado('TO'), 'Palmas'),  
    new Cidade(encontrarEstado('TO'), 'Porto Nacional'),  
    new Cidade(encontrarEstado('TO'), 'Paraíso do Tocantins'),  
    new Cidade(encontrarEstado('SP'), 'São Paulo'),  
    new Cidade(encontrarEstado('MG'), 'Belo Horizonte')  
];
```

Em resumo, o código cria dois elementos no `$scope : estados`, que representa um vetor de Estados, e `cidades`, que representa um vetor de Cidades. A função `encontrarEstado()` recebe como parâmetro uma UF, pesquisa a lista de Estados e retorna o objeto (Estado)

correspondente. Importante lembrar que é necessário fazer com que estes objetos estejam no `$scope` para que a view consiga acessar estes dados.

Uma parte importante do código que não está explícita é que o controller também trata de uma "cidade atual", que representa a cidade que está sendo cadastrada ou editada no momento. Isso ficará mais visível a seguir.

Excluir

A funcionalidade de exclusão é representada pela função `$scope.excluir()` :

```
$scope.excluir = function(index) {  
    $scope.cidades.splice(index, 1);  
};
```

A função recebe como parâmetro o índice da cidade a ser excluída e usa a função `splice()` para remover um elemento do vetor `$scope.cidades` .

Salvar

A funcionalidade de salvar dados representa dois aspectos: o cadastro de uma [nova] cidade e a atualização (edição) dos dados de uma cidade. Esta funcionalidade é representada pela função `$scope.salvar()` :

```
$scope.salvar = function(cidade) {  
    if (!$scope.cidade.edit) {  
        var c = angular.copy(cidade);  
        $scope.cidades.push(c);  
        cidade.nome = "";  
        cidade.estado = null;  
    } else {  
        $scope.cidade = null;  
    }  
};
```

A função recebe como parâmetro um objeto `cidade` e, como já disse, realiza duas tarefas. O código verifica se o objeto `$scope.cidade` (a cidade atual -- e seria o mesmo que usar o parâmetro `cidade`) não possui um atributo `edit` . Se for este o caso, o modo é o de *cadastro*. Neste caso, é feita uma cópia do objeto, a qual é inserida no vetor `$scope.cidades` por meio da função `push()` . Na sequência, os dados do objeto são limpos, para garantir que o formulário de cadastro apresente os valores padrões. O segundo caso

(se o `$scope.cidade` possuir o atributo `edit`) é o de *edição*. Neste caso, o código não realiza uma tarefa mais específica, senão redefinir a referência de `$scope.cidade`. Por que isso acontece?

O controller `CidadesController` expõe para a view a função `$scope.editar()` :

```
$scope.editar = function(cidade) {  
    $scope.cidade = cidade;  
    $scope.cidade.edit = true;  
}
```

Esta função recebe um objeto `cidade` como parâmetro e o atribui a `$scope.cidade` (a cidade atual). Neste caso, o código faz com que a cidade atual fique em *modo de edição*, por assim dizer. Como já visto, a função `$scope.salvar()` leva em consideração a existência de um atributo `edit` no objeto que representa a cidade que está sendo salva. Assim, a última linha da função `$scope.editar()` cria o atributo `edit` no objeto `cidade` atual e define seu valor como `true`.

Uma vez definida a lógica de negócio do aplicativo, a seção a seguir apresenta a definição da view.

View

A view, representada pelo arquivo `index.html`, contém a interface gráfica (o visual e o comportamento) do aplicativo. A estrutura da interface gráfica é representada por duas seções:

- à esquerda, a lista de cidades, com opção de filtro; e
- à direita, o formulário de cadastro e edição.

Para estruturar a interface desta maneira (em colunas) foram utilizados recursos do Bootstrap.

Lista de cidades

Para a lista de cidades, a parte principal está no trecho de código a seguir:

```

<tr ng-repeat="cidade in cidades | filter:q | orderBy:estado.uf">
  <td>{{ $index }}</td>
  <td>{{ cidade.estado.uf }}</td>
  <td>{{ cidade.nome }}</td>
  <td>
    <button type="button" class="btn btn-danger btn-xs" ng-click="excluir($index)">

      <span class="glyphicon glyphicon-remove" aria-hidden="true"></span>
    </button>
    <button type="button" class="btn btn-default btn-xs" ng-click="editar(cidade)">

      <span class="glyphicon glyphicon-pencil" aria-hidden="true"></span>
    </button>
  </td>
</tr>

```

O trecho de código em questão representa a criação das linhas da tabela que apresenta a lista de cidades. Primeiro, perceba a utilização da diretiva `ng-repeat`. Seu valor indica que o elemento `tr` será repetido conforme o vetor `cidades` (definido no `CidadesController`), que cada elemento do vetor será representado, na repetição, por um objeto chamado `cidade`, que o vetor é filtrado pelo elemento `q` (vinculado a um elemento `input`) e ordenado por `estado.uf`. Perceba que a ordenação leva em consideração a estrutura do conteúdo do vetor, ou seja, cada elemento é do tipo `Cidade`, que possui um atributo `estado`, que é do tipo `Estado`, que possui um atributo `uf`.

A primeira coluna apresenta o índice da lista (por meio de `$index`, uma das propriedades automáticas da diretiva `ng-repeat`). A segunda coluna apresenta a UF do Estado da Cidade (`cidade.estado.uf`) e a terceira coluna apresenta o nome da cidade (`cidade.nome`).

A última coluna apresenta dois botões de ação: excluir e editar. O clique no botão excluir chama a função `excluir()` (definida no `CidadesController`) passando como parâmetro o índice atual da iteração (`$index`). O clique no botão editar chama a função `editar()` (também definida no `CidadesController`) passando como parâmetro a cidade atual da iteração (o objeto, em si). Perceba que, no caso do botão editar, o comportamento esperado é o que acontece, ou seja, os dados da cidade em questão são apresentados no formulário, para edição. Além disso, quando você editar os dados (nome e estado) a lista de cidades já será atualizada, demonstrando a edição.

Formulário de cadastro e edição

O formulário à direita realiza a função de cadastrar e editar os dados de uma cidade. O trecho de código a seguir apresenta a parte principal desta seção do aplicativo:

```
<form class="form">
  <div class="form-group">
    <label for="nomeDaCidade">Nome
    </label>
    <input type="text" id="nomeDaCidade" class="form-control"
      ng-model="cidade.nome" placeholder="Nome da cidade" required>
  </div>
  <div class="form-group">
    <label for="estado">Estado</label>
    <select class="form-control" ng-model="cidade.estado.uf">
      <option ng-repeat="estado in estados" value="{{estado.uf}}">
        {{estado.nome}}
      </option>
    </select>
  </div>
  <div>
    <button class="btn btn-default" type="button"
      ng-click="salvar(cidade)">Salvar</button>
  </div>
</form>
```

Utilizando Bootstrap, o formulário possui dois campos: nome da cidade e Estado. O primeiro campo é representado por um elemento `input`, vinculado a `cidade.nome`. O segundo campo é um elemento `select`, vinculado a `cidade.estado.uf`. Importante notar que os elementos `option` do `select` são criados por meio da diretiva `ng-repeat`, com base no vetor de Estados definido no `CidadesController`. Ainda, o atributo `value` de cada elemento `option` está vinculado a `estado.uf`.

A última parte do formulário é o botão "Salvar". No clique do botão salvar, é chamada a função `salvar()`, definida no `CidadesController`.

Você pode ver o exemplo em funcionamento [aqui](#).

Validação de formulários

```
<!DOCTYPE html>
<html lang="pt-br" ng-app="appsimples">

  <head>
    <meta charset="utf-8" />
    <title>App Simples</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.1/css/bootstr
ap.min.css" />
    <link rel="stylesheet" href="style.css" />

    <script>document.write('<base href="' + document.location + '" />');</script>
    <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js"></scri
pt>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.6/angular.min.j
s"></script>

    <script src="app.js"></script>
  </head>

  <body ng-controller="HomeController">
    <div class="container">
      <h1>Cidades do Tocantins</h1>

      <div class="row">
        <div class="col-xs-6 col-md-6">
          <h2>Lista de cidades</h2>
          <p>
            <div class="input-group">
              <input type="text" class="form-control" ng-model="q" place
holder="Pesquisar...">
              <span class="input-group-addon">
                <span class="glyphicon glyphicon-search"></span>
              </span>
            </div>
          </p>

          <table class="table table-hover table-striped">
            <thead>
              <tr>
                <th>#</th>
                <th>UF</th>
                <th>Nome</th>
              </tr>
            </thead>
            <tbody>
              <tr ng-repeat="cidade in cidades | filter:q | orderBy:esta
```

```

do.uf">
    <td>{{ $index }}</td>
    <td>{{ cidade.estado.uf }}</td>
    <td>{{ cidade.nome }}</td>
</tr>
</tbody>
</table>
</div>
<div class="col-xs-6 col-md-6">
    <h2>Cadastrar cidade</h2>

    <form class="form" name="form" novalidate>
        <div class="form-group">
            <label for="nomeDaCidade">Nome
            </label>
            <input type="text" id="nomeDaCidade" name="nomeDaCidade" c
lass="form-control" ng-model="cidade.nome" placeholder="Nome da cidade" required="" va
lue="">

            <div ng-show="form.$submitted || form.uEmail.$touched">
                <div ng-show="form.nomeDaCidade.$error.required">Infor
me o nome da cidade</div>
            </div>
        </div>
        <div class="form-group">
            <label for="estado">Estado</label>
            <select class="form-control" id="estado" name="estado" ng-
model="cidade.estado.uf" required="" value="TO">
                <option ng-repeat="estado in estados" value="{{ estado.
uf }}">{{ estado.nome }}</option>
            </select>
            <div ng-show="form.$submitted || form.uEmail.$touched">
                <div ng-show="form.estado.$error.required">Escolha um
Estado da lista</div>
            </div>
        </div>
        <div>
            <button class="btn btn-default" type="submit" ng-click="sa
lvar(cidade)">Salvar</button>
        </div>
    </form>

</div>
</div>
</div>
</body>

</html>

```

```
(function(){

function Cidade(estado, nome) {
    this.estado = estado;
    this.nome = nome;
}

function Estado(uf, nome) {
    this.uf = uf;
    this.nome = nome;
}

angular.module('appsimples', []).controller('HomeController', function($scope) {

    function encontrarEstado(uf) {
        for(var i = 0; i < $scope.estados.length; i++) {
            if ($scope.estados[i].uf == uf) {
                return $scope.estados[i];
            }
        }
        return null;
    }

    $scope.estados = [
        new Estado('TO', 'Tocantins'),
        new Estado('SP', 'São Paulo'),
        new Estado('MG', 'Minas Gerais')
    ];

    $scope.cidades = [
        new Cidade(encontrarEstado('TO'), 'Araguaína'),
        new Cidade(encontrarEstado('TO'), 'Gurupi'),
        new Cidade(encontrarEstado('TO'), 'Palmas'),
        new Cidade(encontrarEstado('TO'), 'Porto Nacional'),
        new Cidade(encontrarEstado('TO'), 'Paraíso do Tocantins'),
        new Cidade(encontrarEstado('SP'), 'São Paulo'),
        new Cidade(encontrarEstado('MG'), 'Belo Horizonte')];

    $scope.salvar = function(cidade) {
        if ($scope.form.$valid) {
            $scope.cidades.push(angular.copy(cidade));
            $scope.form.$setPristine();
            $scope.cidade = {};
        }
    }
});

})();
```

Você pode ver o exemplo em funcionamento [aqui](#).

Serviço http

O serviço `http` é um componente do Angular que facilita a comunicação com serviços HTTP remotos utilizando o objeto `XMLHttpRequest` do browser. Este serviço fornece os seguintes métodos:

- `get()`
- `post()`
- `head()`
- `put()`
- `delete()`
- `jsonp()`
- `patch()`

Aplicativo app-cidades-http

O método `$http.get()` recebe como parâmetro a URL de requisição. O aplicativo `app-cidades-http` é uma versão modificada do aplicativo `app-cidades-crud` que utiliza dois arquivos para representar os dados de Estados e Cidades.

O arquivo `dados/estados.json` contém a seguinte estrutura:

```
[
  {
    "uf": "TO",
    "nome": "Tocantins"
  },
  ...
]
```

O arquivo `dados/cidades.json` contém a seguinte estrutura:

```
[
  {
    "estado" : {"uf": "TO"},
    "nome": "Araguaína"
  },
  ...
]
```

Ambos representam, respectivamente, os dados de Estados e Cidades.

Carregando os dados

O controller `CidadesController` usa o serviço `$http` para carregar os dados dos arquivos:

```
$http.get('dados/estados.json').success(function(dados){
    $scope.estados = dados;
});

$http.get('dados/cidades.json').success(function(dados){
    $scope.cidades = dados;
});
```

Assim, o método `$http.get()` retorna uma **Premissa**, um objeto que permite definir funções *callback* para situações de sucesso ou falha. No caso do trecho de código anterior, a função `success()` define a *callback* utilizada quando os dados são carregados com sucesso (que é o esperado). Quando os dados são carregados com sucesso, eles são atribuídos aos elementos correspondentes do `$scope`.

Você pode ver o exemplo em funcionamento [aqui](#).

Módulo ngRoute

O `ngRoute` é um módulo do Angular utilizado para fornecer o serviço de rota para aplicativos baseados no Angular. Uma rota é um recurso que permite interpretar a URL conforme certos padrões. Detalhes sobre isso serão vistos nesta seção.

Para utilizar o `ngRoute` é necessário instalá-lo, já que ele não é distribuído, por padrão, com o Angular. Para isso, esta seção introduz a utilização do `npm`, gerenciador de pacotes do NodeJS. Acesse a pasta do aplicativo via prompt (linha de comando) e execute o comando:

```
npm init -y
```

Isso fará com o que o projeto seja iniciado e esteja inicialmente configurado para utilizar o `npm` e os componentes do Angular.

Instale o Angular:

```
npm install angular --save
```

Instale o "Angular route", que fornece o módulo `ngRoute` :

```
npm install angular-route --save
```

A partir de agora, ao invés de utilizar o Angular de uma CDN (bem como outros componentes) utilize o que está na pasta `node_modules`.

Estrutura do aplicativo

O aplicativo `app-cidades-route` usa o `app-cidades-http` com base. As principais diferenças estão na arquitetura da solução e quanto aos recursos do Angular utilizados.

Em relação à arquitetura o aplicativo tem a seguinte estrutura de arquivos:

```
+ app-cidades-route
+--- dados
|   +--- cidades.json
|   +--- estados.json
+--- views
|   +--- cidades
|       +--- editor.html
|       +--- lista.html
+--- app.js
+--- index.html
```

A pasta `dados` contém os arquivos de dados do aplicativo (`cidades.json` e `estados.json`). A pasta `views` contém as views do aplicativo. Neste caso, o aplicativo está organizado por contexto, de modo que a pasta `cidades` contém os arquivos `editor.html` e `lista.html` que representam, respectivamente, o formulário de cadastro e a lista de cidades. Os demais arquivos são similares ao utilizado anteriormente (`app.js` e `index.html`).

Além da organização dos arquivos, o aplicativo, representado pelo módulo `cidades` , utiliza o módulo `ngRoute` como dependência.

Configurando as rotas

A configuração das rotas é feita na função `config()` do módulo:

```
angular.module('cidades', ['ngRoute'])
.config(
  function($routeProvider){
    $routeProvider
      .when('/cidades', {
        templateUrl: 'views/cidades/lista.html',
        controller: 'CidadesListaController'
      })
      .when('/cidades/editor', {
        templateUrl: 'views/cidades/editor.html',
        controller: 'CidadesEditorController'
      })
      .when('/cidades/:id/editor', {
        templateUrl: 'views/cidades/editor.html',
        controller: 'CidadesEditorController'
      })
      .otherwise({
        redirectTo: '/cidades'
      });
  }
);
```

O parâmetro da função `config()`, `$routeProvider`, representa o objeto que dá acesso às configurações de rotas. Uma rota é configurada na chamada do método `when()`, que recebe como parâmetros:

- a rota (representada por uma string); e
- um objeto que indica qual template será utilizado (atributo `templateUrl`) e qual controller (atributo `controller`).

Neste caso, o aplicativo trata três rotas:

- `/cidades` : tratada pelo controller `CidadesListaController` e pela view `/views/cidades/lista.html` ;
- `/cidades/editor` : tratada pelo controller `CidadesEditorController` e pela view `/views/cidades/editor.html` ;
- `/cidades/:id/editor` : tratada da mesma forma que a rota anterior, com a diferença que esta rota possui um **parâmetro** chamado `id` .

Além disso, se nenhuma destas rotas for atendida, a função `otherwise()` indica o que tem que ser feito. Neste caso, ela indica que será feito um redirecionamento para `/cidades` .

CidadesListaController

O controller `CidadesListaController` é responsável por listar as cidades e dar acesso às funções de exclusão e edição:

```
angular.module('cidades').controller('CidadesListaController',
function($scope, $http, $location){
    $http.get('dados/cidades.json').success(function(dados){
        $scope.cidades = dados;
    });

    $scope.excluir = function(index) {
        if (confirm('Tem certeza que deseja excluir esta cidade?')) {
            $scope.cidades.splice(index, 1);
        }
    };

    $scope.editar = function(index) {
        $location.path('/cidades/' + index + '/editor');
    }
});
```

Neste caso, na lista de dependências do controller, além do serviço `$http`, está sendo utilizado o serviço `$location`. `$location` é o serviço que permite manipular a barra de endereços do navegador. O código é praticamente o mesmo do que já foi visto no aplicativo

`app-cidades-http` , com a diferença de que a função `editar()` usa o serviço `$location` para navegar para o editor de cidades. Isso é feito com a chamada da função `path()` , que recebe como parâmetro uma string que representa a nova rota. Neste caso, a nova rota será `/cidades/<id>/editor` (onde `<id>` representa o parâmetro `index` -- o índice da cidade na lista de cidades).

CidadesEditorController

O controller `CidadesEditorController` realiza a função de cadastro (nova cidade e editar cidade). Como os dados estão sendo armazenados em um arquivo estático (`/dados/cidades.json`) o aplicativo não insere, altera ou exclui dados, especificamente (não que isso não seja possível):

```
angular.module('cidades').controller('CidadesEditorController',
function($scope, $http, $routeParams, $location){
    $http.get('dados/estados.json').success(function(dados){
        $scope.estados = dados;
    });

    $http.get('dados/cidades.json').success(function(dados){
        $scope.cidades = dados;

        if ($routeParams.id) {
            $scope.cidade = $scope.cidades[$routeParams.id];
        }

    });

    $scope.salvar = function(cidade) {
        $location.path('/cidades');
    };

    $scope.cancelar = function(){
        $location.path('/cidades');
    };
});
```

Na lista de dependências estão: `$scope` , o serviço `$http` , o serviço `$routeParams` (que dá acesso aos parâmetros da rota) e `$location` . Como o parâmetro da rota em questão chama-se `id` , o mesmo é acessado por meio de `$routeParams.id` . Neste caso, no momento em que os dados do arquivo `/dados/cidades.json` são carregados, a cidade atual recebe a cidade cujo índice do vetor de cidades corresponde ao `id` .

As funções `salvar()` e `cancelar()` modificam a rota para que o aplicativo apresenta a lista de cidades novamente, por meio da função `$location.path()` .

Para executar o aplicativo de exemplo que acompanha este material (`app-cidades-route`) clone o [repositório de códigos-fonte de exemplo](#) (branch `gh-pages`) e, no diretório `/angularjs/app-cidades-route` , execute o comando:

```
npm install
```

Isso fará com que os módulos necessários para o funcionamento do aplicativo sejam devidamente instalados.

Posteriormente, acesse o aplicativo via servidor HTTP local.

Angular - Tutorial

Esta seção apresenta um "Tutorial do Angular", que é baseada no [material correspondente](#) da distribuição oficial do Angular.

Este tutorial apresenta o **PhoneCat**, um aplicativo que é um catálogo de informações sobre dispositivos Android. Além disso, alguns recursos serão apresentados de maneira diferente, como na questão de testes e de utilização de ferramentas como NodeJS e NPM. O tutorial utiliza o framework [Bootstrap](#).

Cada passo do tutorial apresentará como utilizar novos recursos do Angular e, assim, será uma maneira guiada de aprendizagem:

- [Passo 0: Iniciando](#)
- [Passo 1: Template estático](#)
- [Passo 2: Templates do Angular](#)
- [Passo 3: Filtrando Repeaters](#)
- [Passo 4: Vinculação de Dados de Via dupla](#)
- [Passo 5: Telas de lista e detalhes](#)
- [Passo 6: XHR e Injeção de Dependência](#)
- [Passo 7: Mais detalhes do telefone](#)
- [Passo 8: Roteamento e múltiplas Views](#)
- [Passo 9: REST e Serviços personalizados](#)
- [Passo 13: Aplicando animações](#)
- Fim

Passo 0 - Iniciando

Ao abrir o arquivo `index.html` do **Passo 0** no browser, você irá ver a mensagem "Nada aqui ainda". O que você verá nesta seção do tutorial representa o básico da criação de um aplicativo com Angular e o código associado.

O código a seguir é o conteúdo do arquivo `index.html` :

```
<!doctype html>
<html lang="pt-br" ng-app>
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Angular Tutorial</title>

  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/boo
tstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></sc
ript>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"></
script>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.9/angular.min.js"></
script>
  <!--[if lt IE 9]>
    <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
    <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
  <![endif]-->
</head>
<body>
<div class="container">
  <p>Nada aqui {{ 'ainda' + '!' }}</p>
</div>
</body>
</html>
```

Este código é bastante padrão para qualquer aplicativo Angular que você criar daqui para a frente (inclusive os demais passos). Perceba o código utiliza alguns componentes:

- **Bootstrap** (inclui arquivos CSS e JavaScript a partir da CDN do Bootstrap)
- **jQuery** (inclui o arquivo JavaScript a partir da CDN do Google)
- **Angular** (inclui o arquivo JavaScript a partir da CDN do Google)

Até este momento, os arquivos necessários para o funcionamento destes componentes são carregados diretamente de fontes que estão disponíveis na internet, as chamadas CDN (*Content Delivery Network*). Também é possível baixar os arquivos para serem carregados localmente.

O que o código está fazendo?

Diretiva `ng-app`

A tag `html` possui o atributo `ng-app` :

```
<html lang="pt-br" ng-app>
```

O atributo `ng-app` representa uma **diretiva** do Angular chamada **ngApp**. Esta diretiva é usada para indicar ao Angular que deve usar a tag `html` como elemento raiz do aplicativo. Isso dá liberdade para os desenvolvedores indicarem ao Angular tratar como aplicativo o arquivo HTML inteiro ou apenas uma parte dele.

Tag `script`

O arquivo `index.html` possui uma tag `script` que carrega o arquivo JavaScript do Angular:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.9/angular.min.js"></script>
```

Novamente, neste caso, o arquivo está sendo carregando da CDN do Google.

Após carregar este arquivo JavaScript, o Angular entra em ação, procurando uma tag do código HTML que tenha a diretiva `ngApp` . Ao encontrar a diretiva, o Angular vai iniciar o aplicativo considerando a tag em questão como raiz do aplicativo.

A classe `container`

O Bootstrap fornece a classe `container` . O arquivo `index.html` , está sendo aplicada ao elemento `div` :

```
<div class="container">
...
</div>
```

Ao ser aplicada a um elemento (atributo `class`) isso faz com que o conteúdo seja formatado e esteja preparado para receber a estrutura de *grid*, com colunas e linhas.

Chaves duplas vinculadas a uma expressão

O conteúdo do elemento `body` é bastante simples, mas demonstra recursos interessantes do Angular:

```
<p>Nada aqui {{'ainda' + '!'}}</p>
```

A linha, conteúdo do elemento `p` demonstra duas características principais dos recursos de template do Angular:

- Uma vinculação (*binding*), entre por chaves duplas (`{}`)
- Uma expressão é usada na vinculação: `'ainda' + '!'`.

A vinculação diz ao Angular que deve avaliar uma expressão e inserir o resultado no DOM, no local da vinculação. Um recurso muito interessante é que atualizações na expressão implicam em atualizações automáticas do DOM.

Uma *expressão Angular* é um trecho de código JavaScript que é avaliado pelo Angular no contexto do *escopo de modelo* atual, ao invés do *escopo global* (*window*).

Como esperado, uma vez que o código é processado pelo Angular, a página HTML, contém o texto: "Nada aqui ainda!".

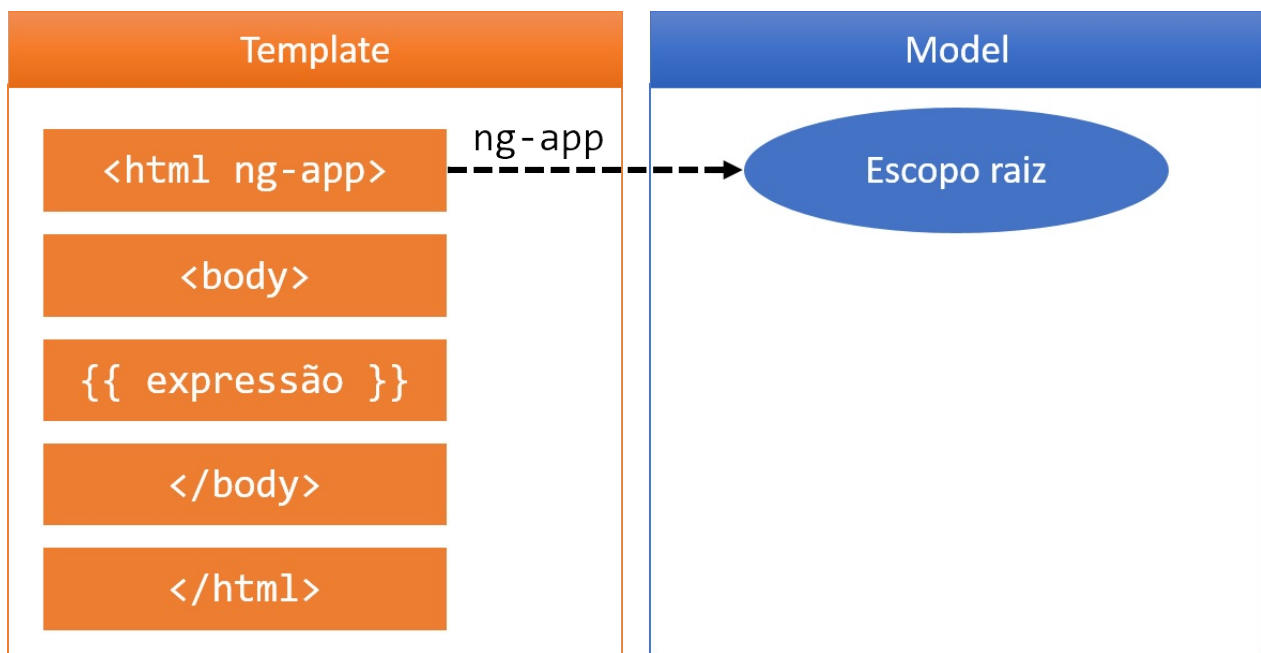
Carregando Aplicativos Angular

Há três coisas importantes acontecendo durante o momento de início de um aplicativo Angular:

- O *injetor*, que será usado para injeção de dependência, é criado;
- O injetor irá, então, criar o *escopo raiz* que se tornará o contexto do *model* do aplicativo;
- O Angular irá, então, "compilar" o DOM, iniciando com o elemento raiz que contém a diretiva **ngApp** e processando as demais diretivas e vinculações restantes.

Assim que o aplicativo é iniciado, ele irá aguardar por entrada de eventos do browser (como clique de mouse, pressionar de teclas ou respostas HTTP) que causem mudanças no *model*. Uma vez que um evento ocorre, o Angular detecta se ele causou mudanças no *model* e, se encontrar, atualizará as vinculações existentes.

A figura a seguir ilustra este processo.



-----> Declaração implícita do escopo

A estrutura do aplicativo é atualmente muito simples. O *template* contém apenas uma diretiva e uma *vinculação estática*. Além disso, o *model* está vazio. Isso mudará em breve! :)

Experimentos

Tente adicionar expressões matemáticas ao template e ver o resultado. Por exemplo, adicione o seguinte ao elemento `body` :

```
<p>1 + 2 = {{1 + 2}}</p>
```

O que acontece se você inserir no template uma expressão Angular com erro?

Resumo

Este passo do tutorial apresentou:

- Qual a estrutura básica de um aplicativo Angular e quais arquivos são necessários para funcionar com os componentes Bootstrap e jQuery, além do próprio Angular
- A diretiva **ngApp** e a sua importância para o início de um aplicativo Angular
- Os conceitos de *vinculação* e *expressão Angular* e a sua importância na criação de templates.

O **Passo 1** acrescentará mais detalhes ao aplicativo.

Carregando arquivos localmente

Se você preferir utilizar arquivos do Bootstrap, jQuery e do Angular localmente, pode fazer isso utilizando maneiras disponíveis nos sites destes componentes. O conteúdo das seções seguintes é bem resumido para que você não tenha a tendência de utilizar esta abordagem, pois é a menos indicada.

Bootstrap

Acesse a [página de download do Bootstrap](#), baixe os arquivos e descompacte na pasta do aplicativo.

jQuery

Acesse a [página de download do jQuery](#) e baixe os arquivos o arquivo da [versão "compactada"](#).

Angular

Acesse a [página inicial do Angular](#) e clique no botão "download". Siga as instruções na tela para baixar os arquivos necessários localmente.

Depois de baixar os arquivos, corrija as partes do código do arquivo `index.html` que os importam na página para que o aplicativo funcione normalmente, carregando os arquivos localmente.

Passo 1 - Template Estático

Um *template* em um aplicativo Angular representa código HTML, responsável pela interface gráfica. Um *template estático* é o primeiro passo na criação de um aplicativo mais complexo, pois representa um nível inicial de abstração em que o conteúdo é definido diretamente no código. Isso quer dizer que em passos seguintes será possível apresentar conteúdo dinamicamente, carregando dados, por exemplo, de uma fonte de dados.

Por enquanto, é interessante progredir na construção do aplicativo utilizando um template estático porque torna o desenvolvimento mais simples, uma vez que agora o interesse é na formatação (construção visual, interface gráfica) do aplicativo.

A princípio, a página inicial do aplicativo deve apresentar uma lista de telefones. A forma mais simples e direta de fazer isso é utilizar um template estático com elementos do HTML que permitam esse tipo de construção. O trecho de código a seguir representa o template estático do aplicativo, no arquivo `index-0.html`. O arquivo `index-0.html` é baseado no arquivo `index.html` do **Passo 0**.

```
<!doctype html>
<html lang="pt-br" ng-app>
...
<body>
<div class="container">
  <p>Lista de telefones</p>
  <ul>
    <li>
      <strong>Nexus S</strong>
      <p>Melhor desempenho com o Nexus S</p>
    </li>
    <li>
      <strong>Motorola Xoom com Wi-Fi</strong>
      <p>O tablet da próxima geração</p>
    </li>
  </ul>
</div>
</body>
</html>
```

As partes omitidas não são diferentes do arquivo `index.html` do **Passo 0**.

O arquivo `index-0.html` apresenta a estrutura básica para a listagem dos telefones, apresentando o nome e uma descrição. Neste momento, são utilizados os elementos `ul` e `li`, principalmente, para gerar a lista de telefones.

Uma vez que o processo de geração do template estático representa, na prática, a criação de uma página com conteúdo HTML, poderíamos encerrar o **Passo 1** por aqui. Entretanto, para apresentar mais recursos do Bootstrap, a seção a seguir fornece uma interface gráfica mais elaborada.

Melhorias no template estático

Ao invés de apresentar apenas o nome de uma descrição de cada telefone, pode ser mais interessante apresentar também a imagem. Utilizando recursos do Bootstrap e do CSS é possível projetar uma interface gráfica mais elaborada. O trecho de código a seguir apresenta parte do arquivo `index-1.html` :

```
...
<head>
...
  <link rel="stylesheet" href="app.css">
...
</head>
<body>
<ul id="listaDeTelefones">
  <li>
    <div class="panel panel-default">
      <div class="panel-body">
        
        <div class="caption text-center">
          <h4 class="ng-binding">Dell Streak 7</h4>
        </div>
      </div>
      <div class="panel-footer text-center">
        <a class="btn btn-default" role="button" href="#">
          <i class="glyphicon glyphicon-zoom-in"></i> Detalhes
        </a>
      </div>
    </div>
  </li>
...
</ul>
...
```

O código utiliza CSS, elementos `ul` e `li`, bem como classes do Bootstrap. O elemento `ul` possui o atributo `id` com valor `listaDeTelefones`. Este recurso será importante posteriormente. Os detalhes serão vistos a seguir.

CSS

O arquivo `index-1.html` faz referência ao arquivo `app.css` por meio do elemento `link`. O atributo `href` indica o caminho para o arquivo `app.css` (relativo ao local do arquivo `index-1.html`).

```
<link rel="stylesheet" href="app.css">
```

O arquivo `app.css` contém as regras de formatação utilizadas para definir certas características da interface gráfica.

```
#listaDeTelefones {  
    padding: 0px;  
    list-style: none;  
}  
  
#listaDeTelefones li {  
    float: left;  
    margin-right: 10px;  
}  
  
#listaDeTelefones li .panel-body {  
    height: 250px;  
}  
  
#listaDeTelefones li .panel {  
    width: 200px;  
}
```

O elemento `ul` do arquivo `index-1.html` possui o atributo `id` com valor `listaDeTelefones`. Isso permite que sejam criadas regras no CSS que façam referência a este elemento especificamente. No arquivo `app.css` estão presentes **seletores** e **regras** (conjunto de propriedades).

Um seletor CSS representa a maneira de encontrar elementos no documento HTML e pode ser feita das seguintes formas:

- **seletor de elemento**: encontra elementos pelo nome da tag
- **seletor de id**: encontra elementos com base no valor do atributo `id`
- **seletor de classe**: encontra elementos com base no valor do atributo `class`
- **seletor de hierarquia**: encontra elementos com base em uma hierarquia
- **seletor de grupo**: permite a utilização de mais de um seletor ao mesmo tempo

O **seletor de elemento** é apenas o nome da tag. Exemplos:

```
body { ... }  
p { ... }
```


A primeira regra aplica-se ao elemento `body` (que é único no documento HTML). Enquanto a segunda, a todos os elementos `p`.

O **seletor de id** é o símbolo `#` seguido do valor do atributo `id` do elemento ao qual se deseja aplicar a regra. Exemplos:

```
#listaDeTarefas { ... }  
#titulo { ... }
```

A primeira regra aplica-se ao elemento com atributo `id` com valor `listaDeTarefas`, enquanto a segunda àquele que tiver valor `titulo`.

De preferência, um documento HTML considera os valores dos atributos `id` como únicos. Se houver mais de um elemento com o mesmo valor para o atributo `id`, a regra CSS será aplicada ao primeiro elemento encontrado.

O **seletor de classe** é o símbolo `.` seguido do nome da classe. O conceito de *classe* refere-se ao valor do atributo `class`. Enquanto o atributo `id` tem apenas um único valor e deve ser único no documento HTML, o atributo `class` pode ter mais de um valor e não é considerado de forma exclusiva. Exemplos:

```
.paragrafo { ... }  
.texto { ... }  
.externo { ... }
```

Cada regra é aplicada ao elemento que possuir, no atributo `class`, a classe correspondente. Exemplos:

```
<p class="paragrafo">...</p>  
<p class="paragrafo texto">...</p>  
<p class="paragrafo texto externo">...</p>
```

O primeiro elemento `p` tem o atributo `class` com o valor `paragrafo`. Isso quer dizer que será aplicada a regra CSS `.paragrafo`. O segundo elemento `p` tem o atributo `class` com o valor `paragrafo texto`. Isso quer dizer que serão aplicadas as classes `paragrafo` e `texto`. Para o último elemento `p` serão aplicadas as classes `paragrafo`, `texto` e `externo`.

No arquivo `app.css` a primeira regra CSS utiliza o seletor `#listaDeTarefas`. Como há um elemento `ul` que se enquadra nesta situação, a regra será aplicada a ele, como esperado.

A regra possui uma lista de propriedades. As propriedades do CSS são como "chaves" para situações específicas (em termos de formatação e representação visual). A lista de propriedades está contida entre chaves e está separada por ponto-e-vírgula.

```
padding: 0px;  
list-style: none;
```

As propriedades utilizadas são:

- **padding**: controla o espaçamento interno do elemento. Neste caso, o valor `0px` indica que o espaçamento interno será de "zero pixels"
- **list-style**: determina como serão apresentados os marcadores para os elementos do `ul`. Neste caso, o valor "none" indica que não serão apresentados marcadores.

Há mais três regras e seus seletores são:

- `#listaDeTelefones li` : aplica-se aos elementos `li` filhos de `#listaDeTelefones`
- `#listaDeTelefones li .panel-body` : aplica-se aos elementos filhos de `li` (que são filhos de `#listaDeTelefones`) que possuem a classe `panel-body`
- `#listaDeTelefones li .panel` : aplica-se aos elementos filhos de `li` (que são filhos de `#listaDeTelefones`) que possuem a classe `panel`

Exercícios

1. Quais são os outros valores para a propriedade `list-style` ?
2. O que significam as propriedades a seguir e seus valores?
3. `float`
4. `margin-right`
5. `height`
6. `width`

Componente Panel, do Bootstrap

O **Panel** é um componente do Bootstrap que permite a criação de um painel. Geralmente, painéis são utilizados em interfaces que precisem organizar parte do conteúdo, contextualizando-o. Neste caso, cada telefone da lista de telefones é apresentado em um painel.

Para utilizar um **Panel**, deve-se utilizar a classe `panel` e uma das classes auxiliares que determinam a cor do painel. Neste caso, está também sendo utilizada a classe `panel-default`.

```
<div class="panel panel-default">
```

Experimento #1 Utilize outras classes de formatação do `panel`.

Um **Panel** possui três partes, duas das quais podem ser utilizadas opcionalmente:

- **Cabeçalho** (opcional): representada pela classe `panel-heading`
- **Corpo**: representado pela classe `panel-body`
- **Rodapé** (opcional): representado pela classe `panel-footer`

```
<div class="panel panel-default">
  <div class="panel-body">
    ...
  </div>
  <div class="panel-footer text-center">
    ...
  </div>
</div>
```

No caso do arquivo `index-1.html`, o painel que apresenta as informações de cada telefone mostra:

- No corpo: foto e nome do telefone
- No rodapé: botão para acessar mais detalhes do telefone

O rodapé utiliza componentes do Bootstrap e requer um pouco mais de atenção.

```
<div class="panel-footer text-center">
  <a class="btn btn-default" role="button" href="#">
    <i class="glyphicon glyphicon-zoom-in"></i> Detalhes
  </a>
</div>
```

A classe `text-center`, do Bootstrap, alinha o conteúdo ao centro.

Dentro do rodapé está um elemento `a`, que contém as classes `btn` e `btn-default`, que é auxiliar da primeira para formatação do botão (aplica o visual padrão de um botão).

Experimento #2 Utilize outras classes de formatação do `btn` (botão).

Dentro do elemento `a` está o elemento `i`, que tem as classes `glyphicon` e a auxiliar `glyphicon-zoom-in`. A segunda define o ícone a ser apresentado (o ícone de "zoom").

Experimento #3

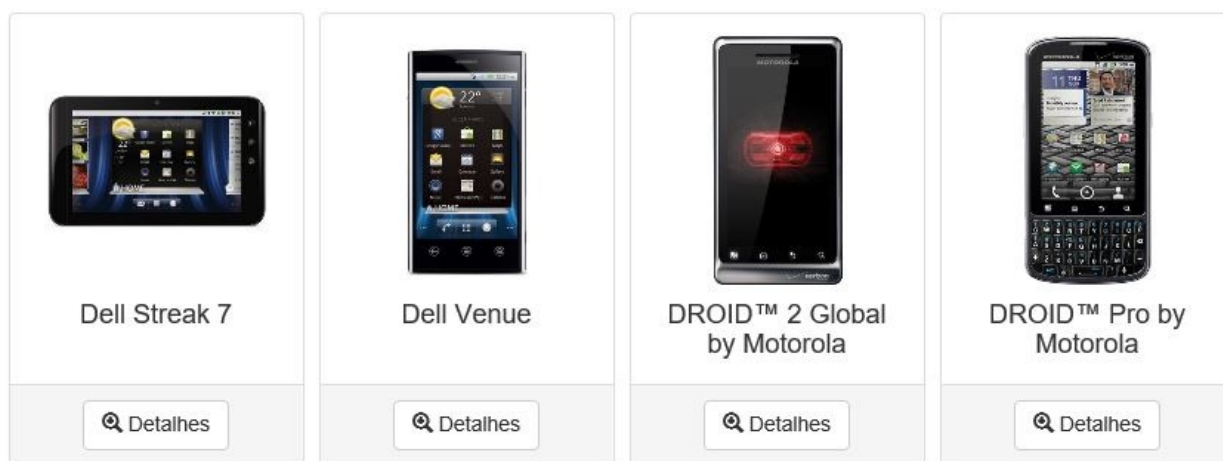
1. Utilize outros ícones do Bootstrap.
2. Abaixo do nome do telefone, apresente seu preço (em moeda corrente).

Passo 2 - Templates do Angular

Uma vez que o **Passo 1** definiu a estrutura da interface gráfica, o *template estático*, o objetivo do **Passo 2** é utilizar os recursos de template do Angular para que o aplicativo se torne dinâmico, carregando dados dos telefones de uma fonte de dados.

O resultado final deste passo é ilustrado pela figura a seguir.

Telefones



O aplicativo apresenta uma lista de telefones, com foto, nome e um botão para visualizar detalhes de cada telefone (que ainda não está funcionando).

Uma das maneiras preferidas de criar aplicativos Angular é utilizar o padrão de projeto arquitetural **MVC (Model-View-Controller)**. Ao utilizar este padrão, a estrutura do aplicativo permitirá separar código e responsabilidades, trazendo maior organização e facilidade de manutenção. Em resumo, cada um dos componentes é assim definido:

- **Controller**: representa a lógica de negócio (onde ficará o código JavaScript)
- **Model**: é uma abstração dos dados (a lista de telefones, por exemplo)
- **View**: é a interface gráfica; sob domínio do **controller**, apresenta o **model** (os dados) e efetua a interação com o usuário.

View e Template

No Angular, uma **view** é uma projeção do **model** por meio do template HTML, ou seja, é a apresentação dos dados. Na verdade, além de apresentar os dados, a **view** serve como receptora da interação com o usuário e repassa dados dessa interação para o **controller**.

Quando o **model** for alterado, por exemplo, o Angular irá procurar pelas vinculações que usam o **model** e fará as atualizações necessárias.

No **Passo 1**, o arquivo `index-1.html` apresenta a lista de telefones utilizando um template estático. Entretanto, não é interessante que o aplicativo seja criado desta forma, pois quaisquer alterações nos dados implicariam em alterações manuais na página (no template estático). A melhor abordagem, portanto, é separar conteúdo (dados) da apresentação (interface gráfica).

O código a seguir apresenta parte do arquivo `index.html` :

```
<!doctype html>
<html lang="pt-br" ng-app="phonecat">
<head>
...
  <link rel="stylesheet" href="app.css">
...
  <script src="app.js"></script>
</head>
<body>
<div class="container" ng-controller="Home">
  <h1>Telefones</h1>
  <ul id="listaDeTelefones">
    <li ng-repeat="telefone in telefones">
      <div class="panel panel-default">
        <div class="panel-body">
          
          <div class="caption text-center">
            <h4>{{telefone.name}}</h4>
          </div>
        </div>
        <div class="panel-footer text-center">
          <a href="#" class="btn btn-default" role="button">
            <i class="glyphicon glyphicon-zoom-in"></i> Detalhes
          </a>
        </div>
      </div>
    </li>
  </ul>
</div> <!--/container-->
</body>
</html>
```

Há várias coisas interessantes acontecendo aqui. As seções a seguir apresentam detalhes.

Identificando a raiz do aplicativo com a diretiva `ngApp`

Anteriormente, o código indicava que o elemento `html` era marcado com a diretiva `ngApp` (atributo `ng-app`). Entretanto, não identificava qual *módulo do Angular* era responsável por servir como ponto de partida do aplicativo. Um *módulo* é uma das formas principais de organização de código no Angular. Além disso, é também uma das maneiras de modularização de código, permitindo que usuários criem seus próprios módulos e os distribuam.

O código, agora, indica que o módulo se chama `phonecat` :

```
<html lang="pt-br" ng-app="phonecat">
```

Arquivo JavaScript do aplicativo

Parte importante do aplicativo está separada no arquivo `app.js`. O conteúdo deste arquivo será visto posteriormente, mas importante é considerar que ele contém a lógica de negócio da definição do módulo `phonecat`.

Diretiva `ngRepeat`

O elemento `li` do arquivo `index.html` tem o atributo `ng-repeat`. Este atributo representa a diretiva `ngRepeater`, que faz com que o elemento seja repetido conforme uma determinada situação.

O valor do atributo `ng-repeat` é `telefone in telefones`. Assim, a diretiva `ngRepeater` indica ao Angular para criar um elemento `li` para cada `telefone` da lista `telefones` utilizando o conteúdo do `li` como um template. Assim, as expressões contidas no `li` serão devidamente interpretadas pelo Angular, como em:

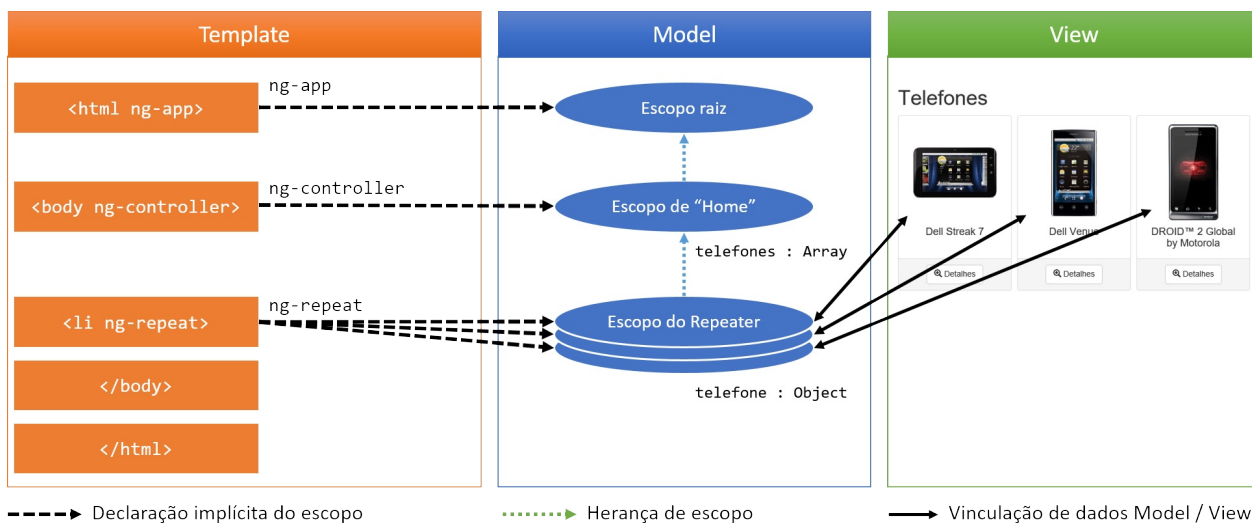
```
<h4>{{telefone.name}}</h4>
```

Neste caso, a expressão indica que será substituída pelo valor do atributo `name` de `telefone` (sim, valem todos os princípios da Programação Orientada a Objetos aqui também).

Diretiva `ngController`

O elemento `body` tem o atributo `ng-controller`, que representa a diretiva `ngController`. Seu valor é `Home`, o que indica que o Angular irá procurar, no módulo `phonecat`, um **controller** com este nome. O conteúdo do elemento `body` serve como `template` para o **controller** `Home`.

A diretiva `ng-repeat` está utilizando um objeto `telefones` (um `array`) que está definido no **controller** `Home`. A imagem a seguir ajuda a ilustrar esse comportamento.



O **escopo raiz** é criado para o aplicativo. O **escopo de "Home"** pertence ao **controller** `Home`, que herda do **escopo raiz**. No `ng-repeat` são criados **escopos locais**, que herdam do **escopo de "Home"**. A herança de escopo é um conceito que permite a um escopo "filho" acessar elementos (como objetos e funções) do escopo "pai".

Diretiva `ngSrc`

O código do template para apresentar a foto do produto é:

```

```

No elemento `img` é utilizado o atributo `ng-src`, que representa a diretiva `ngSrc`. Esta diretiva é utilizada como substituta da diretiva `src`. O motivo de utilizá-la é que utilizar simplesmente `src` não fará com o que o Angular interprete o valor da expressão. Neste caso, o valor `{{telefone.images[0]}}` é interpretado pelo Angular como o primeiro elemento de `telefone.images` (um `array`).

Módulo `phonecat`, Model e Controller

O módulo `phonecat` está definido no arquivo `app.js`.

```
'use strict';

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


Na função `angular.module()` o primeiro parâmetro é o nome do módulo, enquanto o segundo é a lista de dependências do módulo (mais sobre isso depois).

Exercício #1 Qual a finalidade de utilizar `'use strict';` na primeira linha do arquivo JavaScript?

A partir da definição do módulo, são criados os **controllers**. Na prática, a função `angular.module()` retorna um objeto que pode ser atribuído a uma variável.

```
var phonecat = angular.module('phonecat', []);
```

O objeto fornece a função `controller()` que aceita dois parâmetros:

- o nome do **controller**
- uma função que define o código do **controller**

O trecho de código:

```
angular.module('phonecat', []).  
  controller('Home', function() {  
    ...  
  });
```

é similar a este:

```
var phonecat = angular.module('phonecat', []);  
  
phonecat.controller('Home', function() {  
  ...  
});
```

Fica a critério do programador a maneira preferida.

Os argumentos da função (anônima ou não) que é passada como segundo argumento da função `controller()` são tratados pelo **injetor de dependência** do Angular. O trabalho do **injetor de dependência** é instanciar classes de modo que o código do **controller** funcione adequadamente.

```
angular.module('phonecat', []).  
  .controller('Home', function($scope) {  
    ...  
  });
```

Neste caso, a função do **controller** possui um parâmetro `$scope`.

Importante: Certos objetos cujos nomes começam com `$` são particulares do Angular. Por isso, não use `$` nos nomes de suas variáveis ou seus objetos.

O **injetor de dependência** identifica o parâmetro `$scope` e trata de instanciá-lo para que ele esteja disponível para uso.

Objeto `$scope`

O objeto `$scope` é utilizado para fornecer acesso ao **escopo do controller**. Voltando à figura anterior, por meio de `$scope` é que acontecerá uma interação entre os três componentes do modelo **MVC**. Esta interação é chamada *two-way data-binding* (ou vinculação de via dupla). Isso significa que uma alteração no **model** realizada na **view** será sincronizada no **controller**, e vice-versa. Além disso, o objeto `$scope` também pode conter funções. Por exemplo:

```
angular.module('phonecat', []).
  controller('Home', function($scope) {
    $scope.detalhes = function(telefone) {
      ...
    };
  });
```

Esta função poderá ser chamada na **view**, por exemplo, no clique de um botão. Mais sobre isso será visto posteriormente.

O modelo de dados (**model**) utilizado no aplicativo contém um `array` de objetos (telefones). Sua definição ocorre no `$scope` do **controller** `Home` e é isso que permite que `telefones` esteja disponível na **view** (*template dinâmico*, como visto antes).

```
angular.module('phonecat', []).
  controller('Home', function($scope) {
    $scope.telefones = [
      ...
    ];
  });
```

O conteúdo do `array` `$scope.telefones` é composto de objetos que representam dados de telefones. Por enquanto, a **view** precisa (espera) cada um destes objetos tenha os atributos:

- `name` : o nome do telefone
- `images` : um `array` de imagens (`array` de `string` cujos elementos representam nomes de arquivos de fotos do telefone)

O trecho de código a seguir ilustra a definição de `$scope.telefones`.

```
'use strict';

angular.module('phonecat', [])
  .controller('Home', function($scope) {
    $scope.telefones = [
      {
        "id": "dell-streak-7",
        "images": [
          "img/phones/dell-streak-7.0.jpg",
          "img/phones/dell-streak-7.1.jpg",
          "img/phones/dell-streak-7.2.jpg",
          "img/phones/dell-streak-7.3.jpg",
          "img/phones/dell-streak-7.4.jpg"
        ],
        "name": "Dell Streak 7"
      }
    ];
  });
```

Experimento #1

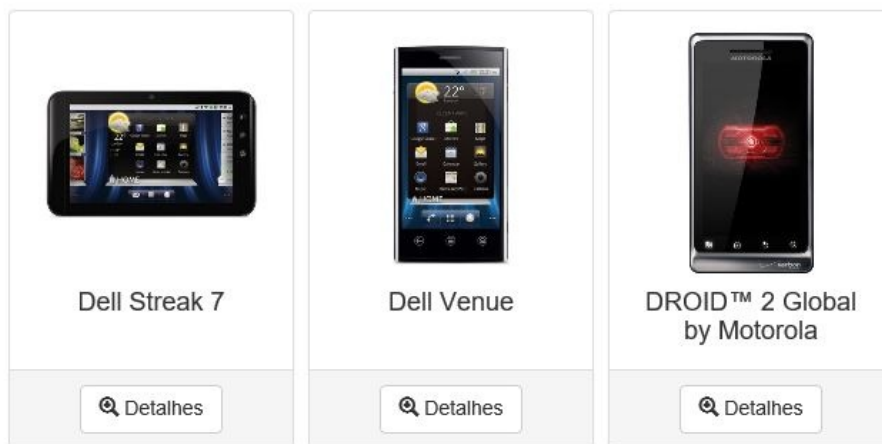
1. Apresente a quantidade total de telefones na **view**.
2. Adicione a propriedade `brand` (marca) nos telefones e apresente-a na **view** abaixo do nome do telefone

Passo 3 - Filtrando Repeaters

Enquanto o **Passo 2** tratou de definir a base do aplicativo, é hora de adicionar novas funcionalidades para facilitar a vida do usuário. O resultado do **Passo 3** é ilustrado pela figura a seguir.

Telefones

Pesquisar



Desta vez o aplicativo fornece, à esquerda da lista de telefones, uma entrada de texto que permite ao aplicativo filtrar a lista de telefones.

O **controller** deste passo é idêntico ao anterior, por isso não serão apresentados mudanças ou detalhes.

O que muda mesmo é no template.

Template

O código a seguir destaca as alterações no template.

```
...  
<div class="container" ng-controller="Home">  
  <h1>Telefones</h1>  
  <div class="row">  
    <div class="col-md-3">  
      <h3>Pesquisar</h3>  
      <input type="text" ng-model="query" placeholder="Pesquisar...">  
    </div>  
    <div class="col-md-9">  
      <ul id="listaDeTelefones">  
        <li ng-repeat="telefone in telefones | filter:query">  
          ...  
        </li>  
      </ul>  
    </div>  
  </div> <!--/row-->  
</div> <!--/container-->
```

Os detalhes do código serão apresentados a seguir.

Grid do Bootstrap

Um "grid" permite uma interface estilo matriz (linhas e colunas). O Bootstrap fornece classes para realizar esta tarefa em interfaces de aplicativos web.

A classe `row` é aplicada ao elemento (geralmente `div`) que servirá como uma linha do grid.

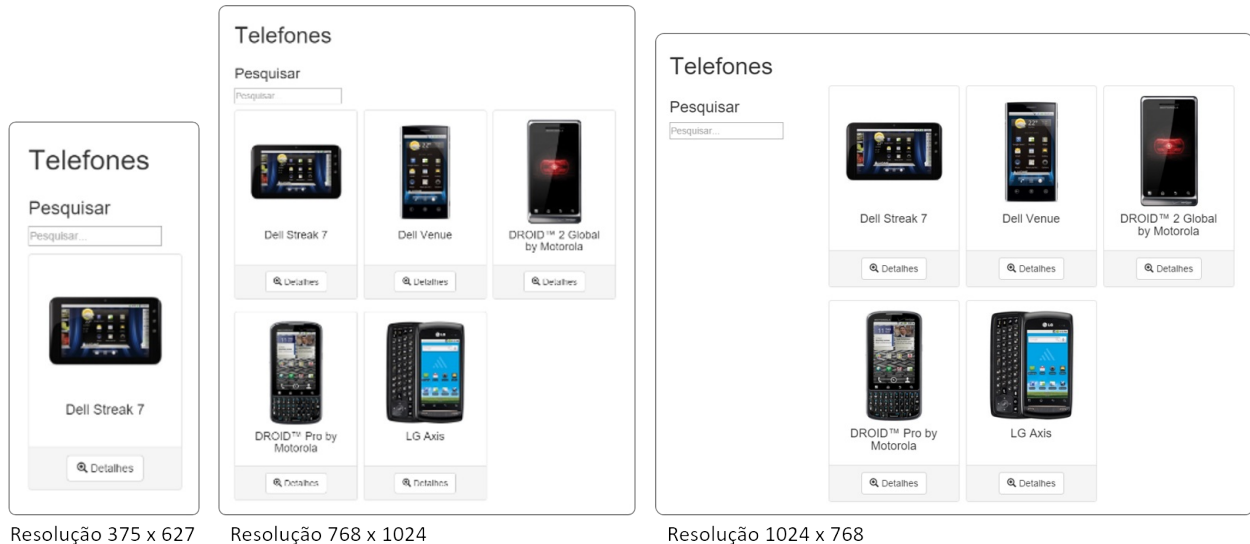
Para representar as colunas, existem diversas classes. Neste caso, estão sendo utilizadas duas delas:

- `col-md-3` : indica que a coluna tem tamanho 3
- `col-md-9` : indica que a coluna tem tamanho 9

Assim, o template possui duas colunas, uma com tamanho 3 e outra com tamanho 9.

A noção de tamanho de uma coluna é abstrata. Seu tamanho exato (em pixels) é proporcional ao tamanho da janela de visualização e é tratado pelo Bootstrap. O tamanho máximo de uma coluna é 12. O importante é que a soma dos tamanhos das colunas de uma linha não pode ser superior a 12. Além disso, é importante também a percepção proporcional, já que não se está trabalhando com pixels. Neste caso, como as duas colunas possuem tamanho 3 e 9, respectivamente, a proporção utilizada é 3/12 e 9/12 ou seja, 25% e 75% do tamanho da janela.

Uma característica também importante do grid do Bootstrap é que ele é, por padrão, **responsivo**. Isso significa que a interface do aplicativo será ajustada pelo Bootstrap ao tamanho da tela de visualização. Com a popularização de dispositivos móveis e seus diversos modelos, esse recurso é bastante útil. A imagem a seguir ilustra a visualização do aplicativo em telas com diferentes tamanhos.



O Bootstrap ajusta a largura das colunas conforme a resolução disponível no dispositivo de visualização. Na resolução menor, a coluna tem largura proporcional a 100% da tela, não importa qual classe do grid está sendo utilizada. Isso varia conforme a dimensão da tela.

Exercício #1

1. Qual a diferença entre as classes `col-xs-3`, `col-sm-3`, `col-md-3` e `col-lg-3` ?
2. O que acontece se um elemento tiver como valor do atributo `class` as classes `col-xs-12` e `col-sm-6` e `col-lg-3` ? Como o Bootstrap trata essas diferenças de tamanhos de colunas em relação ao tamanho da tela?
3. Qual recurso do Bootstrap permite ocultar ou mostrar um elemento com base no tamanho da tela?

Vinculando o `input` ao `Model`

O elemento `input` está vinculado ao `model` .

```
<input type="text" ng-model="query" placeholder="Pesquisar...">
```

A "vinculação" é feita por meio do atributo `ng-model` . O valor deste atributo indica qual propriedade (ou elemento) do **model** está sendo vinculada. No caso do elemento `input` isso significa que o que o usuário digitar será atribuído a `query` . Por causa do *two-way data-*

binding, o **controller** tem à disposição `$scope.query` e qualquer alteração no controller estará visível no template.

Filtro `filter`

O Angular disponibiliza vários "filtros", aplicados ao `ng-repeat`. Um deles é o `filter`. Desconsiderando a redundância, o filtro `filter` realiza a filtragem de uma lista com base em uma condição. Neste caso, ele está vinculado a `query`.

```
<li ng-repeat="telefone in telefones | filter:query">
```

Isso significa que qualquer alteração em `query` fará com que o `ng-repeat` seja atualizado, mostrando apenas os objetos que "casem" com o filtro. O "casamento" (a busca, se preferir) é realizada, neste caso, com base em todos os atributos do objeto, ou seja, o Angular realiza uma busca em todos os objetos de `telefones`; aqueles que tiverem qualquer propriedade cujo valor contenha o valor de `query` são retornados.

O resultado é imediato: a lista de telefones é filtrada com base na entrada do usuário. A sensação é de que está ocorrendo uma busca com resultados em tempo real.

Exercício #2

1. Apresente, no título da página, uma indicação de que o conteúdo está sendo filtrado com base no valor do `input`.
2. O que são as diretivas `ngBind` e `ngBindTemplate`? Para que são utilizadas?

Passo 4 - Two-way data binding

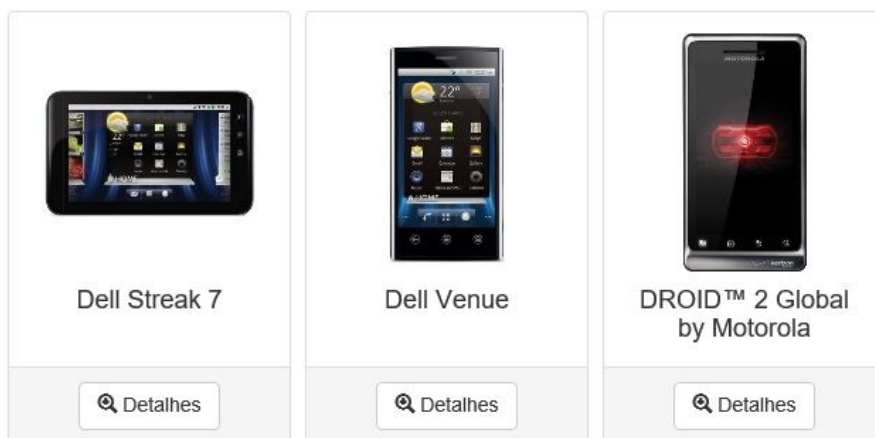
O **Passo 4** adiciona a funcionalidade que permite ao usuário escolher qual a forma de ordenação da lista de telefones. A figura a seguir ilustra o resultado final deste passo.

Telefones

Pesquisar

Ordenar por

Nome
Descrição



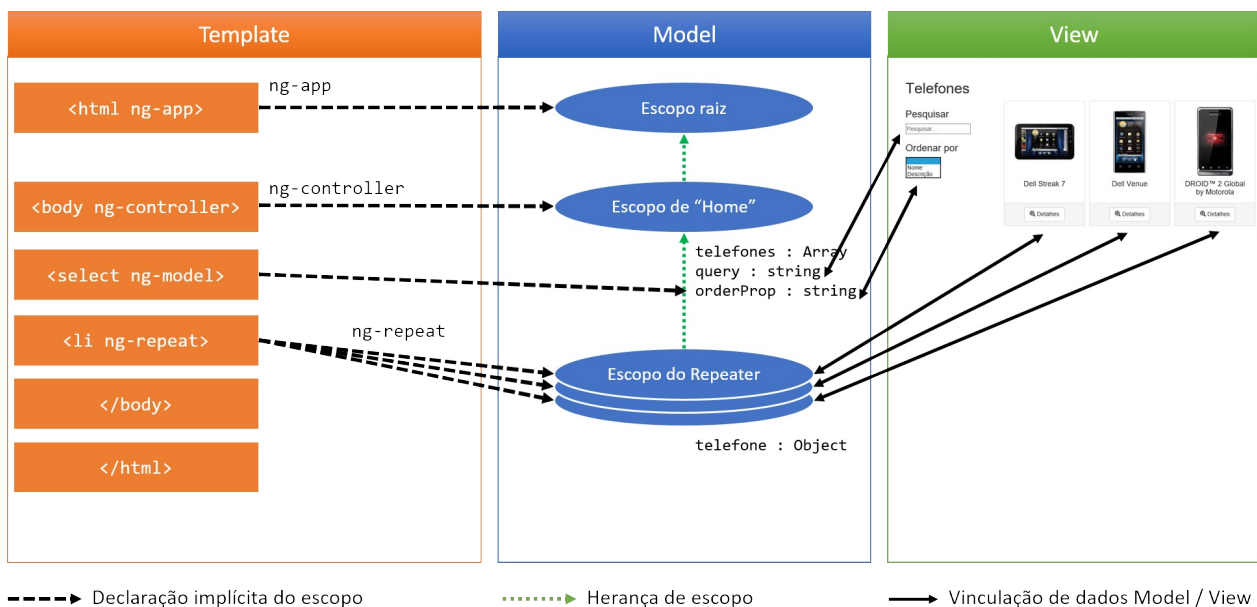
Abaixo da caixa de pesquisa está uma interface que permite ao usuário escolher se deseja ordenar a lista de telefones por nome ou por descrição.

Template

O arquivo `index.html` contém o trecho de código a seguir:


```
<!doctype html>
<html lang="pt-br" ng-app="phonecat">
<head>
...
</head>
<body>
<div class="container" ng-controller="Home">
  <h1>Telefones</h1>
  <div class="row">
    <div class="col-md-3">
      <h3>Pesquisar</h3>
      <input type="text" ng-model="query" placeholder="Pesquisar...">
      <h3>Ordenar por</h3>
      <select ng-model="orderProp">
        <option value="name">Nome</option>
        <option value="description">Descrição</option>
      </select>
    </div>
    <div class="col-md-9">
      <ul id="listaDeTelefones">
        <li ng-repeat="telefone in telefones | filter:query | orderBy:orderProp">
          <div class="panel panel-default">
            ...
          </div>
        </li>
      </ul>
    </div>
  </div> <!--/row-->
</div> <!--/container-->
</body>
</html>
```

O elemento `select` possui a diretiva `ngModel` (representada pelo atributo `ng-model`) que faz com que ocorra uma vinculação de dados de duas vias (*two-way data binding*). A figura a seguir ilustra e ajuda a explicar esse conceito.



O **model** (definido no escopo **controller** `Home`) contém três elementos:

- `telephones : Array` que representa a lista de telefones
- `query : string` que representa a consulta (**Passo 3**)
- `orderProp : string` que representa a ordenação da lista de telefones

O fato de afirmar "definido no escopo do controller `Home`" não implica em ter os elementos do **model** definidos explicitamente no controller. Isso quer dizer que o model pode ser definido implicitamente no escopo do **controller** por meio do **template** ao ser utilizada a diretiva `ng-model` .

Neste caso, o elemento `select` está vinculado à propriedade `orderProp` do **model**. A partir de então, esta propriedade está disponível no escopo do **controller**. Qualquer modificação no `select` provocará uma alteração de `orderProp` . Assim, a lista de telefones será ordenada como esperado.

A lista de telefones é filtrada (**Passo 3**) e também ordenada dinamicamente. O responsável por isso é o filtro **orderBy**. O elemento `li` contém `ng-repeat="telephone in telephones | filter:query | orderBy:orderProp"` . Isso significa que os filtros estão sendo utilizados *em cadeia*. Esse é um recurso importante do Angular: aplicar filtros encadeados.

A sintaxe do filtro `orderBy` é similar à do `filter` :

```
orderBy: expressão
```

Neste caso, `expressão` é a propriedade `orderProp` do **model** (definida no `select`).

Exercícios

1. Modifique o código para que a ordenação possa ser feita considerando outros atributos dos objetos da lista de telefones.
2. Como definir um "valor padrão" para a propriedade `orderProp` diretamente no **controller**? Qual o efeito disso no comportamento do aplicativo?

Passo 5 - Apresentando detalhes de um telefone

O **Passo 5** acrescenta a funcionalidade que permite ao usuário ver detalhes (os dados) de um telefone. Este passo também introduz o conceito de gerenciamento de estado do aplicativo para que seja possível alternar "telas".

Template

O arquivo `index.html` possui o trecho de código a seguir.

```
<!doctype html>
<html lang="pt-br" ng-app="phonecat">
...
<body>
<div class="container" ng-controller="Home">
  <div ng-show="ui_estado == 'lista'">
    ...
  </div>
  <div ng-show="ui_estado == 'detalhes'">
    ...
  </div>
</div> <!--/container-->
</body>
</html>
```

Neste **Passo 5** o aplicativo possui duas telas:

- Lista de telefones
- Detalhes de um telefone

O conceito de "tela" é utilizado aqui para não confundir com "página". O aplicativo em questão (até mesmo por não ser um "web site" convencional, mas um aplicativo) não possui páginas, mas telas.

Assim sendo, é necessária uma maneira de mudar de telas. A primeira forma de resolver isso, a usada neste passo, é bem simples: mostrar uma tela e ocultar a outra. Para isso, é utilizada a diretiva `ngShow` (atributo `ng-show`).

Diretiva `ng-show`

A diretiva `ng-show` permite mostrar conteúdo com base em uma expressão. Neste caso, há uma expressão diferente para cada painel:

- Lista de telefones: expressão é `ui_estado == 'lista'`
- Detalhes de um telefone: expressão é `ui_estado == 'detalhes'`

Assim, cada diretiva (aplicada aos elementos `div` correspondentes) está associada à propriedade `ui_estado` do **model**. Se o seu valor for `lista`, então é apresentada a primeira tela. Se for `detalhes` é apresentada a segunda tela.

Com isso obtém-se o comportamento de "ocultar-e-mostrar" as telas, e o aplicativo fornece as funcionalidade esperadas.

Diretiva `ng-click`

É necessária uma forma de mudar o valor da propriedade `ui_estado` do **model**. Uma maneira de fazer isso, respondendo a um comportamento do usuário, é permitir que o usuário clique em um botão e mostre os detalhes do telefone desejado. Este comportamento é conseguido por meio do uso da diretiva `ngClick` (atributo `ng-show`). O trecho de código a seguir apresenta a parte do arquivo `index.html` em que a diretiva `ng-click` está sendo utilizada.

```
<ul id="listaDeTelefones">
  <li ng-repeat="...">
    <div class="panel panel-default">
      ...
      <div class="panel-footer text-center">
        <a href="#" class="btn btn-default" role="button"
          ng-click="mostrarDetalhes(telefone)">
          <i class="glyphicon glyphicon-zoom-in"></i>
          Detalhes
        </a>
      </div>
    </div>
  </li>
</ul>
```

O elemento `a` (para cada telefone) possui o atributo `ng-click`. Seu valor representa a chamada da função `mostrarDetalhes()` passando como parâmetro `telefone` (ou seja, o telefone em questão, na lista de telefones).

Isso significa que quando o usuário clicar no botão "Detalhes", será chamada uma função que está definida no **controller** (arquivo `app.js`).

Tela de detalhes do telefone

A figura a seguir ilustra a tela de detalhes do telefone.

Dell Streak 7

 Lista

Introducing Dell™ Streak 7. Share photos, videos and movies together. It's small enough to carry around, big enough to gather around. Android™ 2.2-based tablet with over-the-air upgrade capability for future OS releases. A vibrant 7-inch, multitouch display with full Adobe® Flash 10.1 pre-installed. Includes a 1.3 MP front-facing camera for face-to-face chats on popular services such as Qik or Skype. 16 GB of internal storage, plus Wi-Fi, Bluetooth and built-in GPS keeps you in touch with the world around you. Connect on your terms. Save with 2-year contract or flexibility with prepaid pay-as-you-go plans

A tela apresenta o nome do telefone, a descrição completa e um botão que permite retornar à lista de telefones.

O trecho de código a seguir apresenta a parte do arquivo `index.html` que está relacionada à tela de detalhes de um telefone.

```
<div ng-show="ui_estado == 'detalhes'">
  <h1>
    <a href="" class="btn btn-default pull-right" role="button" ng-click="mostrarLista()">
      <i class="glyphicon glyphicon-th-large"></i> Lista
    </a>
    {{telefone.name}}
  </h1>
  <p>{{telefone.description}}</p>
</div>
```

O código indica que o elemento `a` possui a diretiva `ng-click` com valor `mostrarLista()`. Isso demonstra que deve existir uma função com este nome no **controller**. Além disso, o conteúdo também indica que está vinculado à propriedade `telefone` do **model**. É importante lembrar do conceito de *escopo* para entender que `telefone` não é o mesmo objeto da lista de telefones (da diretiva `ng-repeat`).

Controller

O **controller**, ou seja, o arquivo `app.js`, é modificado para incluir o trecho a seguir.

```
$scope.ui_estado = 'lista';
$scope.telefone = null;

$scope.mostrarDetalhes = function(telefone) {
    $scope.ui_estado = 'detalhes';
    $scope.telefone = telefone;
};

$scope.mostrarLista = function() {
    $scope.ui_estado = 'lista';
};
```

O código indica que o **model** possui quatro propriedades:

- `ui_estado` : é utilizada para controlar qual tela do aplicativo está visível (seu valor inicial indica que a primeira tela visível é a lista dos telefones)
- `telefone` : representa o "telefone atual" e é utilizada na tela de detalhes do telefone
- `mostrarDetalhes` : é uma função que aceita como parâmetro um objeto que representa um telefone e será utilizada para mostrar a tela de detalhes de um telefone
- `mostrarLista` : é uma função que será utilizada para mostrar a tela de lista de telefones

Função `mostrarDetalhes()`

A função `mostrarDetalhes()` realiza duas operações:

- modifica o valor da propriedade `ui_estado` para `"detalhes"` (indicando que a tela de detalhes se tornará visível)
- modifica o valor da propriedade `telefone` para que seja possível apresentar informações do telefone em questão (o passado como parâmetro)

Função `mostrarLista()`

A função `mostrarLista` realiza apenas uma operação: modifica o valor da propriedade `ui_estado` para `"lista"`, indicando que a tela de lista de telefones se tornará visível.

O **Passo 5** adicionou funcionalidades importantes ao aplicativo e aumentou, certamente, a necessidade de atenção aos detalhes do código que tornam viáveis essas funcionalidades:

- Apresenta a lista de telefones (como nos passos anteriores)
- Apresenta detalhes de um telefone a partir do clique em um botão de "detalhes" na lista de telefones
- Permite retornar à tela de lista de telefones a partir da tela de detalhes de um telefone
- Apresenta a noção de "tela" como forma de representar "navegação" no aplicativo. Em web sites convencionais, essa mudança seria entendida como "mudança de página" ou

"navegação de página".

Os passos seguintes apresentarão melhorias no aplicativo, principalmente em relação ao carregamento dos dados dos telefones e à navegação.

Passo 6 - Carregando dados via AJAX

O **Passo 6** adiciona a funcionalidade de carregar dados dos telefones via AJAX (requisições assíncronas ou XHR), ao invés de manter os dados deles em um `array` diretamente no código.

Antes de continuar neste passo, é necessário que o projeto utilize um servidor web. Como sugestão, em tempo de desenvolvimento, pode ser utilizado o [http-server](#).

npm e dependências

Na prática, o projeto referente a este passo que é disponibilizado no repositório do código-fonte do livro já está devidamente configurado para funcionar com esta e outras dependências. Na sua cópia local do repositório, no diretório referente ao **Passo 6**, no prompt de comando, digite:

```
npm install
```

Este comando instala as dependências:

- de **produção**
 - `jquery`
 - `bootstrap`
 - `angular`
- de **desenvolvimento**
 - `http-server`

Se você iniciar o projeto, então terá que adicionar as dependências. Para isso, é necessário utilizar o **npm**. Você pode acompanhar uma breve referência do **npm** [aqui mesmo neste livro](#).

Dados dos telefones

Até o **Passo 5** os dados dos telefones eram definidos diretamente no código do **controller**. Embora isso não esteja necessariamente incorreto, é interessante utilizar uma abordagem mais adequada. Neste caso, uma abordagem favorável ao projeto é armazenar os dados

dos telefones em arquivos `json`. No diretório `data\phones` estão vários arquivos `.json`. Um deles, em especial o arquivo `phones.json`, contém um `array` de objetos que representam os telefones. O código a seguir apresenta um trecho desse arquivo.

```
[
  {
    "age": 0,
    "id": "motorola-xoom-with-wi-fi",
    "imageUrl": "img/phones/motorola-xoom-with-wi-fi.0.jpg",
    "name": "Motorola XOOM\u2122 with Wi-Fi",
    "snippet": "The Next, Next Generation\r\n\r\nExperience the future with Motorola XOOM with Wi-Fi, the world's first tablet powered by Android 3.0 (Honeycomb)."
  },
  ...
]
```

Os atributos do objeto que representa um telefone são: `age`, `id`, `imageUrl`, `name` e `snippet`. Dentre estes atributos, o atributo `id` é particularmente interessante, pois serve como **chave** e também é utilizado para acessar o arquivo de dados do telefone, que terá o mesmo nome do atributo `id` para o telefone em questão. Por exemplo, o arquivo `motorola-xoom-with-wi-fi.json` corresponde ao telefone descrito no trecho de código acima.

Template

Por causa da estrutura dos dados, há uma pequena modificação no template: o código para apresentar a imagem do telefone é alterado para:

```

```

Em outras palavras, o objeto `telefone` possui o atributo `imageUrl`, que representa a URL da foto do telefone. Até o **Passo 5**, o mesmo dado era obtido ao acessar um `array` que representava um conjunto de fotos do telefone.

Carregando dados via XHR (AJAX)

O **controller** é bastante modificado neste passo:

```
'use strict';

angular.module('phonecat', [])
  .controller('Home', function($scope, $http) {
    $http.get('data/phones/phones.json').then(function(response){
      $scope.telefones = response.data;
    });

    $scope.ui_estado = 'lista';
    $scope.telefone = null;

    $scope.mostrarDetalhes = function(telefone) {
      $scope.ui_estado = 'detalhes';
      $http.get('data/phones/' + telefone.id + '.json').then(
        function(response){
          $scope.telefone = response.data;
        }
      );
    };

    $scope.mostrarLista = function() {
      $scope.ui_estado = 'lista';
    };
  });
```

Como já informado, os dados dos telefones não são definidos diretamente no código. Portanto, para ter acesso aos dados que serão apresentados na **view**, é utilizado o módulo `$http` do angular. Importante notar que a declaração do **controller** indica a dependência desse módulo e do `$scope` :

```
controller('Home', function($scope, $http) { ... });
```

O processo de *injeção de dependência* entrará em ação para tornar disponível o objeto `$http` , que permitirá carregar os dados dos telefones. Na prática, este objeto permite gerar requisições HTTP via código. O método `get()` gera uma requisição GET:

```
$http.get('data/phones/phones.json').then(function(response){
  $scope.telefones = response.data;
});
```

A requisição GET busca obter o arquivo `data/phones/phones.json` , ou seja, o arquivo que contém a lista dos telefones, como já informado. O método `get()` retorna um objeto que permite executar código no momento em que houver um retorno do servidor (o que é chamado de **callback**). Isso é possível por meio da função `then()` . Esta função pode receber dois parâmetros:

1. uma função que será executada quando a requisição ocorrer com sucesso
2. uma função que será executada quando a requisição ocorrer com erro

No caso do trecho do código em questão, apenas a primeira função (que executa no sucesso da requisição) está sendo definida.

A função **callback** possui um parâmetro (`response`): um objeto que representa a requisição XHR (AJAX). Ele possui o atributo `data`, que contém os dados interpretados a partir da requisição. Neste caso, a lista de telefones (conforme a estrutura já apresentada). O código da função faz com que a propriedade `telefones` do **model** receba `response.data`.

Função `mostrarDetalhes()`

A função `mostrarDetalhes()` também passa por uma modificação. Uma vez que os dados completos dos telefones estão em arquivos `json` separados, será necessário carregar o arquivo correspondente.

A exemplo do que ocorreu para carregar os dados do arquivo `data/phones/phones.json` aqui também é utilizado o módulo `$http`:

```
$scope.mostrarDetalhes = function(telefone) {  
    $scope.ui_estado = 'detalhes';  
    $http.get('data/phones/' + telefone.id + '.json').then(  
        function(response){  
            $scope.telefone = response.data;  
        });  
};
```

A função **callback** de sucesso da requisição GET que busca o arquivo associado ao telefone em questão faz com que o "telefone atual" receba os dados do arquivo do telefone, ou seja:

```
$scope.telefone = response.data;
```


Executando o aplicativo

A execução do aplicativo a partir do **Passo 66** não pode ser feita abrindo-se o arquivo `index.html` diretamente no browser. Isso acontece porque os recursos utilizados requerem a presença de um servidor web. Neste caso, o aplicativo utiliza o `http-server`.

Para iniciar o aplicativo, execute o comando a seguir:

```
hs -c-1
```

A animação a seguir ilustra esse processo.

 Prompt de Comando

```
C:\xampp\htdocs\livro-web-codigo-fonte\angularjs\angular-tutorial\passo-6>
```

Ao ser executado, o `http-server` inicia um processo que representa um servidor web. Por padrão, a execução utiliza todos os endereços IPs disponíveis no computador atual e a porta `8080`. A opção `-c-1` indica que o `http-server` instruirá o browser a não fazer cache do conteúdo, o que é útil para garantir que o aplicativo, ao ser acessado pelo browser, esteja sempre atualizado.

Para utilizar o aplicativo, no browser, acesse `http://localhost:8080`.

O **Passo 6** demonstrou como utilizar o módulo `$http` para gerar requisições XHR (AJAX) para um servidor e carregar dados dos telefones. Assim, os dados não são definidos diretamente no código do **controller** e é utilizada uma abordagem mais adequada para este problema (já que os dados são apenas de leitura).

Enxergando a solução empregada aqui neste passo de outro prisma, ela se aproxima do formato de *consumo de API* ou *API REST* que é, inclusive, utilizado neste livro na parte sobre **Back-end**.

Note Exercício

O formato de acesso a dados utilizado aqui no **Passo 6** é, na verdade, bastante utilizado em outras situações. Na prática, é um acesso a dados que funciona também com fontes remotas, ou seja, o aplicativo pode *consumir dados* de sites externos. Vários sites fornecem esse tipo de serviço, como Facebook e Google. Um dos serviços mais conhecidos é o [Rotten Tomatoes](#), que fornece informações sobre filmes e avaliações de usuários.

Crie um aplicativo angular para consumir dados de um serviço ou site externo. Fica a seu critério escolher uma fonte de dados e utilizá-la da forma apropriada. Algumas delas requerem um cadastro para obtenção de uma *chave de API*.

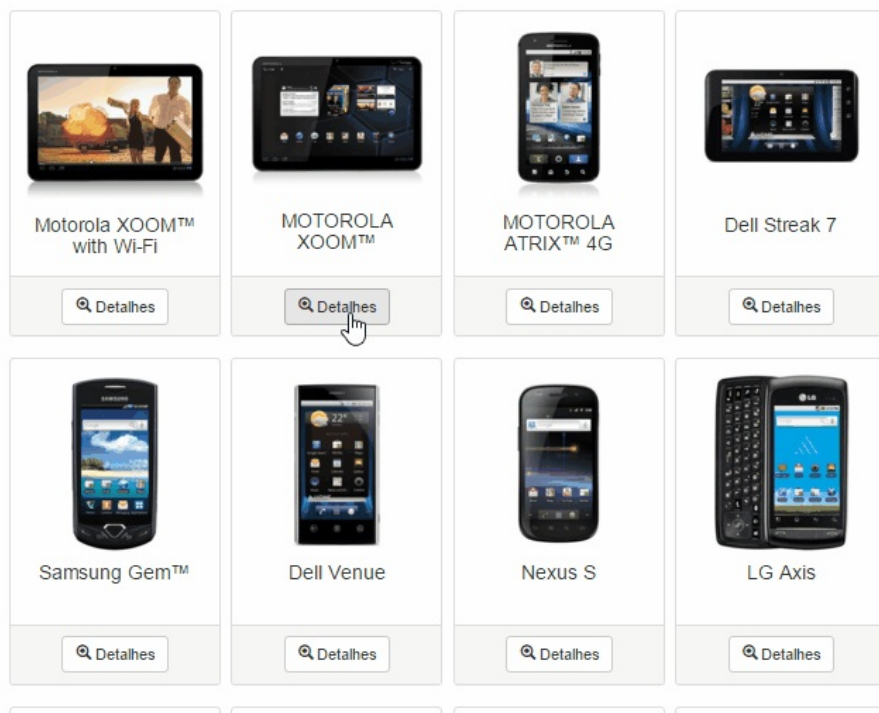
Passo 7 - Melhorias no template

O **Passo 7** aplica melhorias ao template, apresentando mais informações na tela de detalhes do telefone. A animação a seguir ilustra o resultado final.

Telefones

Pesquisar

Ordenar por



A tela de detalhes do telefone apresenta informações adicionais:

- fotos do telefone
- especificações técnicas como detalhes da bateria e conectividade

Template

Em relação ao **Passo 6** as principais mudanças do passo atual são na tela de detalhes do telefone. O arquivo `index.html` contém o trecho de código a seguir.

```

<h1>
<a href="" class="btn btn-default pull-right" role="button" ng-click="mostrarLista()">
  <i class="glyphicon glyphicon-th-large"></i> Lista
</a>
{{telefone.name}}
</h1>
<div class="row">
  <div class="col-md-5">
    
  </div>
  <div class="col-md-7">
    <p>{{telefone.description}}</p>
    <h3>Mais fotos do telefone (clique para ampliar)</h3>
    <div class="thumbs">
      ...
    </div>
  </div>
</div>
<div class="row">
  <h2>Detalhes e mais especificações técnicas</h2>
  <p>{{telefone.additionalFeatures}}</p>
  <div class="col-md-3">
    <h3>Disponibilidade e redes</h3>
    ...
  </div>
  <div class="col-md-3">
    <h3>Bateria</h3>
    ...
  </div>
  <div class="col-md-3">
    <h3>Armazenamento e memória</h3>
    ...
  </div>
  <div class="col-md-3">
    <h3>Conectividade</h3>
    ...
  </div>
</div>
</div>

```

Fotos do telefone

A estrutura do código demonstra que a tela de detalhes contém seções que apresentam os dados do telefone em questão. A primeira seção é a que apresenta as fotos do telefone.

```

<div class="row">
  <div class="col-md-5">
    
  </div>
  <div class="col-md-7">
    <p>{{telefone.description}}</p>
    <h3>Mais fotos do telefone (clique para ampliar)</h3>
    <div class="thumbs">
      <ul class='listaDeMiniaturas'>
        <li ng-repeat="image in telefone.images">
          <a href="#" ng-click="mostrarImagem(image)">
            
          </a>
        </li>
      </ul>
    </div>
  </div>
</div>

```

A tela apresenta uma foto grande, à esquerda da descrição e miniaturas do telefone, abaixo da descrição.

A foto grande está vinculada à propriedade `telefone.imageUrl` do **model**.

Utilizando a diretiva `ng-repeat` sobre o `telefone.images` (um `array`) o template apresenta elementos `li` com conteúdo para apresentar as fotos do telefone. Importante notar que ao clicar em uma foto pequena ela se torna a foto grande, pois é chamada a função `mostrarImagem()`, definida no **controller**.

Detalhes técnicos

Os detalhes técnicos do telefone são apresentados em quatro colunas. A primeira coluna apresenta *disponibilidade e redes* do telefone.

```

<div class="col-md-3">
  <h3>Disponibilidade e redes</h3>
  <ul>
    <li ng-repeat="availability in telefone.availability">{{availability}}</li>
  </ul>
</div>

```

Como pode ser notado, a diretiva `ng-repeat` está vinculada a `telefone.availability`.

A segunda coluna apresenta informações da bateria do telefone.


```
<div class="col-md-3">
  <h3>Bateria</h3>
  <ul>
    <li><strong>Standby</strong>: {{telefone.battery.standbyTime}}</li>
    <li><strong>Conversação</strong>: {{telefone.battery.talkTime}}</li>
    <li><strong>Tipo</strong>: {{telefone.battery.type}}</li>
  </ul>
</div>
```

Neste caso, não é necessário utilizar a diretiva `ng-repeat`. Os dados apresentados advêm das propriedades do objeto `telefone.battery`: `standbyTime`, `talkTime` e `type`.

A terceira coluna apresenta informações sobre armazenamento e memória.

```
<div class="col-md-3">
  <h3>Armazenamento e memória</h3>
  <ul>
    <li><strong>RAM</strong>: {{telefone.storage.ram}}</li>
    <li><strong>Interna</strong>: {{telefone.storage.flash}}</li>
  </ul>
</div>
```

A exemplo da coluna *bateria*, aqui são utilizadas propriedades do objeto `telefone.storage`: `ram` e `flash`.

A quarta e última coluna apresenta informações sobre *conectividade*.

```
<div class="col-md-3">
  <h3>Conectividade</h3>
  <ul>
    <li><strong>Bluetooth</strong>: {{telefone.connectivity.bluetooth}}</li>
    <li><strong>Rede</strong>: {{telefone.connectivity.cell}}</li>
    <li><strong>GPS</strong>: {{telefone.connectivity.gps}}</li>
    <li><strong>Infravermelho</strong>: {{telefone.connectivity.infrared}}</li>
    <li><strong>Wi-fi</strong>: {{telefone.connectivity.wifi}}</li>
  </ul>
</div>
```

O template apresenta os valores das propriedades do objeto `telefone.connectivity`: `bluetooth`, `cell`, `gps`, `infrared` e `wifi`.

Controller

Em relação ao **controller** há poucas mudanças em relação ao **Passo 6**:

- alteração na função `mostrarDetalhes()`
- inclusão da função `mostrarImagem()`

Função `mostrarDetalhes()`

A função `mostrarDetalhes()` passa a ser definida como:

```
$scope.mostrarDetalhes = function(telefone) {  
    $scope.ui_estado = 'detalhes';  
    $http.get('data/phones/' + telefone.id + '.json').then(  
        function(response){  
            $scope.telefone = response.data;  
            $scope.telefone.imageUrl = $scope.telefone.images[0];  
        });  
};
```

Ou seja, a única alteração em relação ao **Passo 6** é a criação do atributo `imageUrl` no objeto `$scope.telefone`. Este atributo é utilizado para apresentar a foto maior e, por isso, recebe o endereço da primeira foto do array `$scope.telefone.images`.

Função `mostrarImagem()`

A função `mostrarImagem()` é responsável por alternar a foto grande da página de detalhes do telefone. Seu código é:

```
$scope.mostrarImagem = function(imagem) {  
    $scope.telefone.imageUrl = imagem;  
}
```

Como informado, o atributo `$scope.telefone.imageUrl` representa a imagem utilizada no momento como a foto grande do telefone. Assim, a função `mostrarImagem()` recebe como parâmetro o caminho da nova imagem que se tornará a foto grande.

Conclusão

O **Passo 7** adicionou ao aplicativo **PhoneCat** a funcionalidade da apresentação de mais informações na tela de detalhes do telefone. Sendo o **Passo 6** o que mais causou modificações estruturais no aplicativo, até o momento, o passo atual apenas utilizou o conhecimento já fundamentado para chegar ao objetivo proposto.

Note Exercício

1. O Bootstrap fornece um componente interessante para apresentar fotos (imagens) de uma forma diferente: o **Carousel** (ou **slider**). Modifique a tela de detalhes do telefone utilizando o componente **Carousel** para apresentar as fotos do telefone. Neste caso, não será necessário utilizar as miniaturas.
2. Ainda utilizando o componente **Carousel**, modifique a tela de lista de telefones para que, antes da lista, sejam apresentados três telefones aleatoriamente. Basta apresentar a foto e o nome do telefone (como *caption*) no carousel.

Passo 8 - Roteamento e múltiplas Views

Nos passos anteriores, foi utilizado o conceito de "tela", que permite uma troca de contexto da interface gráfica. Entretanto, a abordagem utilizada (mostrar e ocultar conteúdo) não é a maneira mais adequada de realizar este procedimento, principalmente com o aumento da quantidade de funcionalidades (e de telas) do aplicativo.

Um aplicativo de única página (SPA) (do inglês *Single Page Application*) é um recurso de programação front-end que faz com que o aplicativo web não utilize o formato tradicional de troca de página, ou seja, há apenas uma única página e ocorrem trocas de telas. Esse conceito foi utilizado até o **passo 7** e continuará sendo utilizado no restante deste tutorial do Angular.

Para saber mais

Se quiser saber mais sobre o conceito de SPA, pode começar lendo esse artigo da wikipedia (em inglês): https://en.wikipedia.org/wiki/Single-page_application.

Até o passo 7, quando o usuário clica no botão "Detalhes", a tela de lista de telefones é ocultada e é apresentada a tela de detalhes do telefone. O Pass 8 utiliza o módulo `angular-route` para fornecer uma alternativa mais adequada.

Módulo `angular-route`

O módulo `angular-route` fornece serviços necessários para que o aplicativo utilize o conceito de múltiplas views.

O arquivo `package.json` precisa incluir o módulo `angular-route` nas suas dependências.

Múltiplas views, Rotas e Template de Layout

Uma "**rota**" é um recurso que permite ao navegador mudar o endereço (a URL) atual sem, efetivamente, mudar de página. A princípio, isso pode soar estranho, mas é um recurso amplamente utilizado em desenvolvimento web moderno. O módulo `angular-route` fornece o serviço `$route`, que permite relacionar controllers, views e a URL atual do navegador.

Uma rota é **padrão**, e é representada no navegador por uma URL, por exemplo:

```
http://localhost:8080/#/telefones
```

Importante observar a presença do caractere `#` (chamado de **hash**) na URL. Uma rota, portanto, é apresentada na URL a partir de `#`.

Um "**template de layout**" é responsável por definir um template que é comum à todas as views do aplicativo. As views são chamadas de "**templates parciais**" porque incluem somente a parte do template que é necessária para cada tela.

Estrutura do aplicativo

A partir de então, o aplicativo será organizado por "módulos". A estrutura de arquivos é a seguinte:

```
passo-8
|   app.css
|   app.js
|   index.html
|   package.json
|
|---data
|   |---phones
|       dell-streak-7.json
|       ...
|
|---img
|   |---phones
|       dell-streak-7.0.jpg
|       ...
|
|---node_modules
|   ...
|
|---telefones
|   detalhes.html
|   lista.html
|   modulo.js
```

O diretório `telefones` representa o módulo **Telefones**, que apresenta a lista e os detalhes de telefones. A ideia de separar o aplicativo em módulos representa uma proposta de arquitetura para o software que pretende isolar ou separar partes do software em módulos, isto é, se consegue, com isso, modularização.

Template de layout

O arquivo `index.html`, utilizado como **template de layout** é bastante modificado em relação ao **Passo 7**, como mostra o trecho de código a seguir:

```
<!doctype html>
<html lang="pt-br" ng-app="phonecat">
<head>
  ...
  <title ng-bind="pageTitle"></title>
  ...
  <script src="node_modules/angular/angular.min.js"></script>
  <script src="node_modules/angular-route/angular-route.min.js"></script>
  <script src="telefones/modulo.js"></script>
  <script src="app.js"></script>
  ...
</head>
<body>
<div class="container">
  <div ng-view></div>
</div>
</body>
</html>
```

A diretiva `ng-bind` está sendo aplicada ao elemento `title` para que o título da janela seja definido dinamicamente, no controller. O valor do atributo representa uma propriedade do model. Neste caso, a propriedade é `pageTitle`. Os passos anteriores apresentaram como interagir com o escopo de um controller. Como será visto posteriormente, o **Passo 8** demonstra como interagir com o escopo raiz do aplicativo.

Na seção de arquivos JavaScript importados no arquivo `index.html` estão os arquivos:

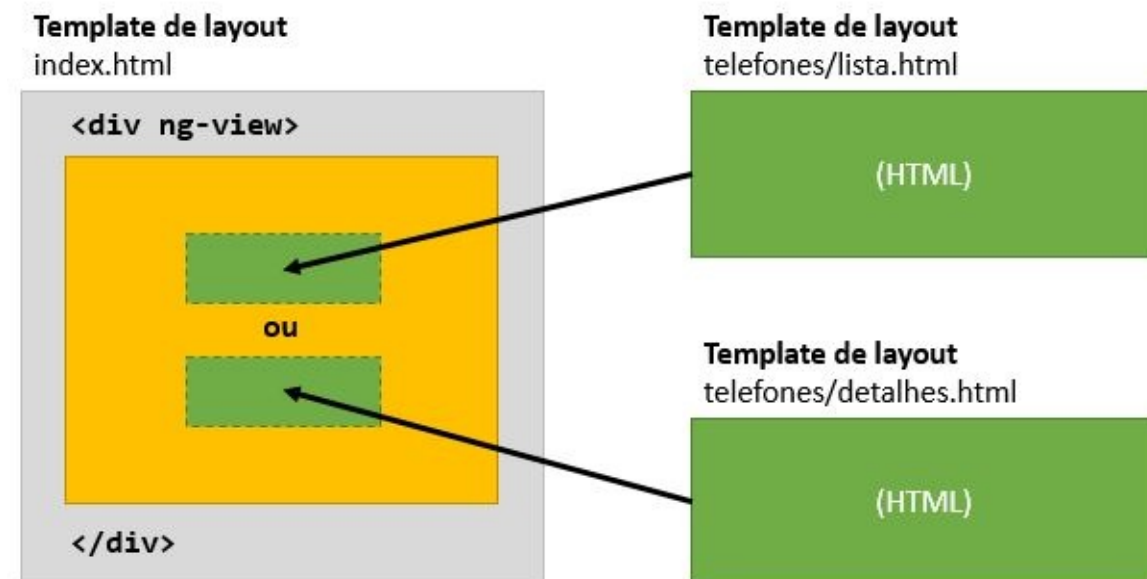
- `node_modules/angular-route/angular-route.min.js` (do módulo `angular-route`)
- `telefones/modulo.js` (que implementa o módulo telefone)

A ordem de inclusão dos arquivos JavaScript é importante, uma vez que o arquivo `app.js` depende do módulo telefone.

Diretiva `ng-view`

O módulo `angular-route` fornece a diretiva `ng-view`. Por meio dessa diretiva os **templates parciais** são dinamicamente embutidos neste local (dentro de `<div class="container">`).

A figura a seguir ajuda a ilustrar este conceito.



A figura demonstra que os **templates parciais** são incluídos de modo exclusivo dentro da `div` que está com a diretiva `ng-view`.

O módulo `angular-route` permite a utilização de apenas uma diretiva `ng-view` no **template de layout**.

Código JavaScript do aplicativo

O código JavaScript do aplicativo PhoneCat muda bastante em relação ao **Passo 7**.

O arquivo `app.js` passa a ter o seguinte conteúdo:

```
'use strict';

angular.module('phonecat', ['ngRoute', 'moduloTelefone'])
  .config(function($routeProvider){
    $routeProvider
      .when('/telefones', {
        templateUrl: 'telefones/lista.html',
        controller: 'TelefonesListaController'
      })
      .when('/telefones/:id', {
        templateUrl: 'telefones/detalhes.html',
        controller: 'TelefonesDetalhesController'
      })
      .otherwise({
        redirectTo: '/telefones'
      });
  });
```

Dependências

A primeira novidade está em relação às dependências (segundo parâmetro da função `module()`):

```
angular.module('phonecat', ['ngRoute', 'moduloTelefone'])
```

Anteriormente, o aplicativo não possuía dependências, agora, são duas:

- `ngRoute` (pacote/módulo `angular-route`)
- `moduloTelefone` (definido no arquivo `telefones/modulo.js`)

Função `config()` e rotas

O objeto criado pela função `angular.module()` possui a função `config()`, que é utilizada para realizar tarefas de configuração do módulo, que executam no momento em que o módulo é carregado. Neste caso, é injetado na função `config()` o objeto `$routeProvider`, que representa uma API de acesso ao módulo `angular-route` para, principalmente, definir **rotas**.

No código da função `config()` o objeto `$routeProvider` é utilizado para criar rotas. Há duas rotas:

- `/telefones`
- `/telefones/:id`

Como já informado, uma rota é um padrão. Assim, quando o padrão definido por uma rota estiver presente na URL, o módulo `angular-route` entrará em ação para definir o que acontecerá. Por exemplo, a URL:

```
http://localhost/#/telefones
```

atende a rota `/telefones`. De forma semelhante, a URL:

```
http://localhost/#/telefones/galaxy
```

atende a rota `/telefones/:id`.

Por definição, isso é semelhante a um **evento de troca da URL**, ou seja, . Por isso o objeto `$routeProvider` fornece a função `when()` que recebe dois parâmetros:

1. o padrão da rota na URL
2. um objeto que configura a vinculação entre a rota, template parcial e controller

A rota `/telefones` é definida por:

```
.when('/telefones', {  
  templateUrl: 'telefones/lista.html',  
  controller: 'TelefonesListaController'  
})
```

O segundo parâmetro da função `when()` indica que, quando a rota atual for `/telefones`, será utilizado o template parcial definido no arquivo `telefones/lista.html` e o controller `TelefonesListaController`.

A rota `/telefones/:id` é definida por:

```
when('/telefones/:id', {  
  templateUrl: 'telefones/detalhes.html',  
  controller: 'TelefonesDetalhesController'  
})
```

A rota `/telefones/:id` inclui um **parâmetro de rota**. Um parâmetro de rota representa uma parte da rota que pode ser substituída por um valor, o qual poderá ser tratado posteriormente no controller. O template parcial está definido no arquivo `telefones/detalhes.html` e o controller é `TelefonesDetalhesController`.

Por fim, o objeto `$routeProvider` fornece a função `otherwise()`:

```
otherwise({  
  redirectTo: '/telefones'  
})
```

A função `otherwise()` recebe como parâmetro um objeto que determina o que acontece quando nenhuma rota for encontrada. Neste caso, o módulo `angular-route` redirecionará para a rota `/telefones`.

Importante lembrar que esses templates parciais e controllers estão definidos no módulo **Telefones**, que será apresentado a seguir.

Módulo Telefones

O módulo Telefones é definido no arquivo `/telefones/modulo.js` e seu código é apresentado a seguir.

```
'use strict';

angular.module('moduloTelefone', [])
  .controller('TelefonesListaController',
    function($rootScope, $scope, $http, $location){
      $rootScope.pageTitle = 'Telefones - PhoneCat';
      $http.get('data/phones/phones.json').then(function(response){
        $scope.telefones = response.data;
      });
    })
  .controller('TelefonesDetalhesController',
    function($rootScope, $scope, $http, $routeParams){
      $http.get('data/phones/' + $routeParams.id + '.json').then(
        function(response){
          $scope.telefone = response.data;
          $scope.telefone.imageUrl = $scope.telefone.images[0];
          $rootScope.pageTitle = $scope.telefone.name + ' - Phonecat';
        });
      $scope.mostrarImagem = function(imagem) {
        $scope.telefone.imageUrl = imagem;
      };
    });
```

O módulo declara dois controllers:

- `TelefonesListaController` : implementa a funcionalidade de apresentar a lista de telefones; e
- `TelefonesDetalhesController` : implementa a funcionalidade de apresentar os detalhes de um telefone.

Estes módulos e os templates parciais associados serão apresentados a seguir.

Lista de telefones

Template parcial da lista de telefones

O template parcial da lista de telefones está definido no arquivo `telefones/lista.html` . Seu código é exatamente o mesmo de parte do template principal do **Passo 7** (arquivo `index.html`) que representava a "tela" de lista de telefones. Por este motivo, o código não será apresentado novamente aqui.

Controller `TelefonesListaController`

O controller `TelefonesListaController` é definido da seguinte forma:

```
.controller('TelefonesListaController',
function($rootScope, $scope, $http){
    $rootScope.pageTitle = 'Telefones - PhoneCat';
    $http.get('data/phones/phones.json').then(function(response){
        $scope.telefones = response.data;
    });
})
```

Na função que define o controller são injetados três objetos:

- `$rootScope`
- `$scope`
- `$http`

Dentre estes, os objetos `$scope` e `$http` já são conhecidos. O objeto `$rootScope` permite acessar o escopo raiz do aplicativo. Na prática, é semelhante ao `$scope` (ou seja, permite acessar o model) tendo o escopo como única diferença. Neste caso, `$rootScope` é utilizado para alterar o título da página, modificando a propriedade `pageTitle` do model:

```
$rootScope.pageTitle = 'Telefones - PhoneCat';
```

O restante do controller é idêntico ao definido no **Passo 7** e não será detalhado aqui.

Detalhes de um telefone

Template parcial dos detalhes de um telefone

O template parcial dos detalhes de um telefone está definido no arquivo `telefones/detalhes.html`. Seu código é exatamente o mesmo de parte do template principal do **Passo 7** (arquivo `index.html`) que representava a "tela" de detalhes de um telefone. Por este motivo, o código não será apresentado novamente aqui.

Controller `TelefoneDetalhesController`

O controller `TelefonesDetalhesController` é definido da seguinte forma:

```
.controller('TelefonesDetalhesController',
function($rootScope, $scope, $http, $routeParams){
    $http.get('data/phones/' + $routeParams.id + '.json').then(
        function(response){
            $scope.telefone = response.data;
            $scope.telefone.imageUrl = $scope.telefone.images[0];
            $rootScope.pageTitle = $scope.telefone.name + ' - Phonecat';
        });
    $scope.mostrarImagem = function(imagem) {
        $scope.telefone.imageUrl = imagem;
    };
});
```

Na função que define o controller são injetados quatro objetos:

- `$rootScope`
- `$scope`
- `$http`
- `$routeParams`

Dentre estes objetos, o que precisa de destaque é `$routeParams`, que é fornecido pelo módulo `angular-route`.

Como já informado, uma rota pode possuir um parâmetro de rota, que é definido seguindo a sintaxe: sinal de dois pontos seguido pelo nome do parâmetro. Assim, a rota

`/telefones/:id` possui um parâmetro chamado `id`. Para ter acesso ao parâmetro de rota, é utilizado o objeto `$routeParams`. O controller `TelefonesDetalhesController` o utiliza para essa função:

```
$http.get('data/phones/' + $routeParams.id + '.json')
```

O objeto `$rootScope` é utilizado novamente para definir o valor da propriedade `pageTitle`. Desta vez, o objetivo é fazer com o que o título da janela apresente o nome do telefone. Isso está presente na função callback da função `$http.get()`:

```
$http.get('data/phones/' + $routeParams.id + '.json').then(
    function(response){
        $scope.telefone = response.data;
        $scope.telefone.imageUrl = $scope.telefone.images[0];
        $rootScope.pageTitle = $scope.telefone.name + ' - Phonecat';
    });
```

O restante do controller é semelhante ao já apresentado, por isso os detalhes do código, bem como sua reprodução, serão omitidos.

Conclusões

O **Passo 8** implementa várias mudanças no aplicativo. Primeiro, a arquitetura do software foi modificada por meio da modularização, a qual impacta também a organização dos arquivos do projeto. Segundo, o recurso de rotas permitiu uma nova visão sobre como mudar "telas" do aplicativo.

Note Exercício

Estenda o **Passo 8**, criando funcionalidades que, implementando o módulo **Fabricantes**, permitam ao usuário:

- Ver a lista de fabricantes de telefones celulares (ex: rota `/fabricantes`)
- Ver os detalhes de um fabricante (tela de detalhes do fabricante) com nome, descrição e lista de telefones [do fabricante em questão]
- Filtrar a lista de telefones por fabricante (módulo **Telefones**)

Utilize arquivos `.json` para representar os dados dos fabricantes. Pode ser necessário alterar arquivos `.json` dos telefones para representar o "relacionamento" entre fabricante e telefone.

REST e Serviços

Este passo apresenta uma abordagem diferente para usar o objeto XHR (e o recurso de AJAX) introduzido no **passo 8**. De outro ponto de vista, a abordagem está mais adequadamente relacionada com a forma como dados de APIs podem ser consultados utilizando o recurso REST.

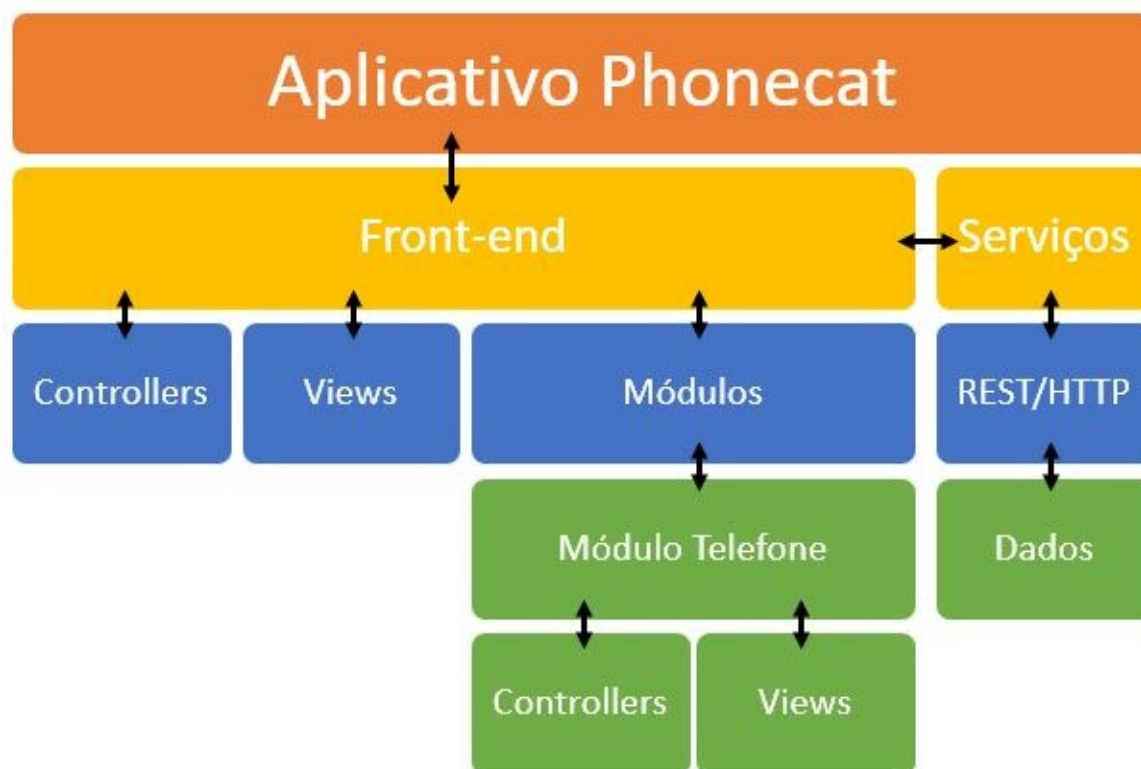
Dependências

O módulo `angular-resource` fornece os serviços necessários para consultar serviços no formato REST. A dependência pode ser adicionada ao projeto usando `npm` :

```
npm install angular-resource --save
```

Arquitetura do aplicativo

O **passo 8** adicionou várias mudanças na arquitetura e na estrutura do aplicativo. O **passo 9** acrescenta uma **camada de serviços** na arquitetura. A figura a seguir ilustra a definição atual da arquitetura.



A camada **Front-end** utiliza a camada **Serviços**, que fornece os serviços de comunicação com os dados por meio da camada **REST**. Esta é uma abordagem interessante com o propósito principal de modularização da arquitetura.

Estrutura de arquivos

A estrutura de arquivos do projeto continua semelhante à do **passo 8**. A principal diferença é a presença do arquivo `/telefones/servico.js`. Em relação ao conteúdo dos arquivos, apenas são modificados os arquivos `index.html` e `/telefones/modulo.js`.

Template

O arquivo `index.html` é modificado apenas para incluir os arquivos do módulo `angular-resource` e `telefones/servico.js`:

```
<!doctype html>
<html lang="pt-br" ng-app="phonecat">
<head>
...
  <script src="node_modules/angular/angular.min.js"></script>
  <script src="node_modules/angular-route/angular-route.min.js"></script>
  <script src="node_modules/angular-resource/angular-resource.min.js"></script>
  <script src="telefones/servico.js"></script>
  <script src="telefones/modulo.js"></script>
  <script src="app.js"></script>
...
</head>
<body>
...
</body>
</html>
```

Módulo Telefones

O módulo **Telefones** é modificado para implementar o conceito de serviço e para alterar os controllers deste módulo para usarem o serviço criado.

Serviço

Um **serviço** é outro mecanismo de modularização do Angular, sendo fornecido por um módulo. Assim, primeiro tem-se um módulo e, então, os serviços que ele fornece.

O arquivo `telefones/servico.js` define o módulo `phonecatServices` e o serviço `Telefones` :

```
'use strict';

angular.module('phonecatServices', ['ngResource'])
  .factory('Telefones', function($resource) {
    return $resource('data/phones/:phoneId.json', {}, {
      query: {
        method: 'GET',
        params: { phoneId: 'phones' },
        isArray: true
      }
    });
  });
```

O módulo `phonecatServices` depende do módulo `ngResource` (`angular-resource`).

A função `factory()` representa o conceito **factory** do Angular, e recebe dois parâmetros:

1. O nome do serviço (`Telefones` , neste caso)
2. A função que define o serviço. Neste caso, é injetado o objeto `$resource` (fornecido pelo módulo `angular-resource`).

O objeto `$resource` é utilizado como uma função para a qual são passadas três parâmetros:

1. Uma rota na qual o serviço é baseado
2. Um objeto sem definição/vazio
3. Um objeto que fornece ações para o serviço REST

O objeto vazio permite definir valores-padrões para os parâmetros da rota.

Rota do serviço

A rota do serviço é `data/phones/:phoneId.json` . Como já visto no **passo 8** uma rota pode possuir parâmetros. Desta forma, a rota possui o parâmetro `phoneId` .

Ações

As ações para os serviços REST remetem aos verbos do HTTP. As ações-padrões incluem GET, SAVE e DELETE.

Além das ações-padrões é possível fornecer ações personalizadas, o que é feito por meio do terceiro parâmetro para `$resource()` .


```
{
  query: {
    method: 'GET',
    params: { phoneId: 'phones' },
    isArray: true
  }
}
```

O objeto possui o atributo `query` (que se torna o nome da ação personalizada). O conteúdo do atributo é um objeto que possui:

- atributo `method`, com valor `GET`, indicando que esta ação é uma ação que usa o verbo GET do HTTP
- atributo `params`, com valor `{phoneId: 'phones'}`, indicando que o valor padrão para o parâmetro de rota `phoneId` é `phones`
- atributo `isArray`, com valor `true`, indicando que o retorno da requisição HTTP deve ser tratado como array (lista).

Controllers

Uma vez definido o serviço `Telefones` no módulo `phonecatServices`, este pode ser utilizado por outros módulos. Assim, o arquivo `telefones/modulo.js` tem uma alteração na declaração do módulo `moduloTelefone` para incluir uma dependência para o módulo

`phonecatServices` :

```
angular.module('moduloTelefone', ['phonecatServices'])
```

Por causa da dependência, o serviço `Telefones` pode ser injetado nos controllers e, assim, suas ações podem ser utilizadas como uma camada intermediária entre os controllers e a API REST.

TelefonesListaController

O controller `TelefonesListaController` injeta o serviço `Telefones` e utiliza a ação `query()` para consultar os dados da lista de telefones.

```
.controller('TelefonesListaController', function($rootScope, $scope, $http, Telefones)
{
  $rootScope.pageTitle = 'Telefones - PhoneCat';
  $scope.telefones = Telefones.query();
})
```

O objeto `$scope.telefones` recebe o retorno da função `query()`.

Um conceito importante aqui é chamado de **promise**. A maneira mais simples de entender este conceito é olhar para a linha a seguir:

```
$scope.telefones = Telefones.query();
```

Anteriormente, ao utilizar o serviço `$http`, foi colocado em prática o conceito de **funções callback** que executam código de forma assíncrona permitem executar código ao final da execução, por exemplo.

Ao utilizar **promise** o código continua utilizando uma chamada assíncrona, entretanto, ela é gerenciada pelo próprio Angular, que também dá uma garantia de que o que se espera como retorno da função seja atribuído a uma variável ou utilizado de outra forma. Em outras palavras, o Angular garante a expectativa futura. Neste caso, a expectativa futura é que a função `query()` retorne uma lista (array) de telefones. Assim, uma **promise** é uma maneira de o Angular "prometer" para o código que a sua expectativa será cumprida.

A mudança na arquitetura do software ajudou, também, a ocultar detalhes do código. O código do controller `TelefonesListaController` não tem acesso ao endereço do arquivo dos dados dos telefones. Isso fica isolado no serviço `Telefones`.

TelefonesDetalhesController

O controller `TelefonesDetalhesController` também injeta o serviço `Telefones`:

```
.controller('TelefonesDetalhesController', function($rootScope, $scope, $http, $routeParams, Telefones){
    $scope.telefone = Telefones.get({phoneId: $routeParams.id}, function(telefone){
        $scope.telefone.imageUrl = telefone.images[0];
        $rootScope.pageTitle = telefone.name + ' - Phonecat';
    });
    $scope.mostrarImagem = function(imagem) {
        $scope.telefone.imageUrl = imagem;
    };
});
```

O código do controller utiliza uma ação padrão `get()` para consultar os dados de um telefone. A função `get()` também retorna uma **promise**. A diferença para o código do controller anterior é que uma função **callback** (segundo parâmetro) é utilizada para realizar um determinado trecho do código (definir a imagem padrão e o título da janela do browser com base nos dados do telefone em questão).

Resumo

O **passo 9** realiza modificações na arquitetura do software ao utilizar o conceito de **serviços personalizados**. Por meio dos serviços, que criam uma camada para acessar os dados via REST, os módulos e controllers do software não têm acesso a certos detalhes, como o endereço do arquivo de dados. Ainda, o código atual utiliza o conceito de **promise**, como maneira alternativa para acessar dados dos serviços.

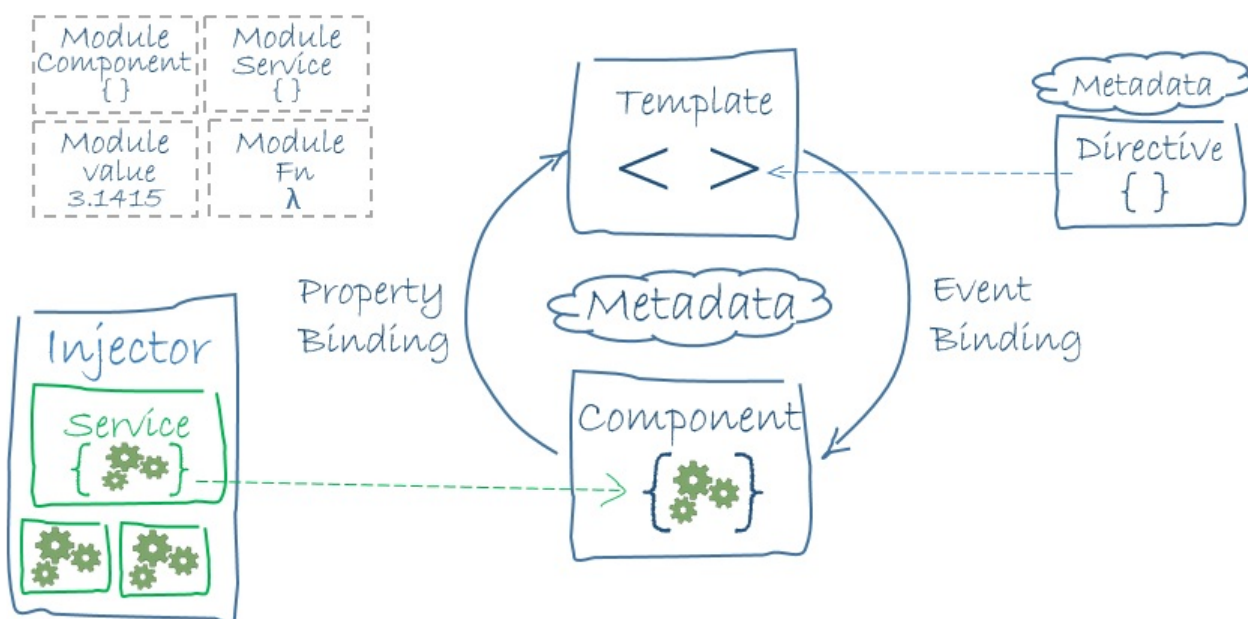
Angular

O conteúdo deste capítulo é baseado na [documentação oficial](#) e no *Angular Guide*.

Arquitetura

Angular é um **framework** para o desenvolvimento de aplicativos em HTML utilizando JavaScript ou outra linguagem que compila para JavaScript. A documentação do Angular, bem como a maioria das referências, utiliza TypeScript como linguagem de programação.

A figura a seguir apresenta as interações entre os principais componentes da arquitetura do Angular.



Os componentes dessa arquitetura são os seguintes e são detalhados nas seções em sequência:

- **Modules** (módulos)
- **Components** (componentes)
- **Templates** (templates)
- **Metadata** (metadados)
- **Data binding** (vinculação de dados)
- **Directives** (diretivas)
- **Services** (serviços)
- **Dependency Injection** (injeção de dependência)

Os nomes dos componentes da arquitetura serão mantidos no original, em inglês.

Modules

Aplicativos Angular são modulares, seguindo um sistema chamado *Angular Modules* ou *NgModules*.

Cada aplicativo Angular tem pelo menos um módulo, o *módulo raiz*. Geralmente, o *módulo raiz* é um componente visual, que hospeda outros componentes visuais ou não.

Independentemente de ser o módulo raiz, cada módulo é definido em uma classe marcada com `@NgModule` (uma *decorator function*).

`NgModule` é uma *decorator function* que recebe um objeto com os seguintes atributos:

- `declarations` : um array contendo a lista de *view classes* que pertencem ao módulo
- `exports` : um subconjunto de `declarations` que estarão visíveis e utilizáveis nos componentes *templates* de outros módulos
- `imports` : array contendo a lista de outros módulos cujas classes exportadas são requeridas por *templates* do módulo em questão
- `providers` : array com uma lista de serviços disponibilizados pelo módulo atual que se tornam disponíveis globalmente para todos os módulos do aplicativo
- `bootstrap` : array contendo o *módulo raiz*.

Exemplo (arquivo `app/app.module.ts`):

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

O aplicativo é executado por meio do processo de *bootstrap* do módulo raiz.

Exemplo (arquivo `app/main.ts`):

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
```

Bibliotecas

O Angular fornece vários módulos, cada um deles com nomes tendo o prefixo `@angular`.

Os módulos podem ser instalados utilizando `npm` e importados utilizando a instrução

`import`. Exemplo:

```
import { Component } from '@angular/core';
```

O código importa a decorator function `Component` , definida no módulo `@angular/core` .

As bibliotecas podem ser:

- Components (componentes)
- Directives (diretivas)
- Services (serviços)
- Values (valores)
- Functions (funções)

Components

Um **component** controla parte do estado real da tela que é chamada de `view` . A lógica de um componente é definida em uma classe.

Exemplo (arquivo `app/hero-list.component.ts`):

```
export class HeroListComponent implements OnInit {  
  heroes: Hero[];  
  selectedHero: Hero;  
  
  constructor(private service: HeroService) { }  
  
  ngOnInit() {  
    this.heroes = this.service.getHeroes();  
  }  
  
  selectHero(hero: Hero) { this.selectedHero = hero; }  
}
```

O Angular cria, atualiza e destrói componentes conforme o usuário utiliza o aplicativo. O desenvolvedor tem acesso aos momentos do ciclo de vida do aplicativo por meio dos **eventos do ciclo de vida**, como o método `ngOnInit()` utilizado no código anterior.

Templates

A view de um componente é definida no **template**, que é uma espécie de HTML que indica ao Angular como apresentar visualmente o componente.

Exemplo (arquivo `app/hero-list.component.html`):

```
{%raw%}  
<h2>Hero List</h2>  
<p><i>Pick a hero from the list</i></p>  
<ul>  
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">  
    {{hero.name}}  
  </li>  
</ul>  
<hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>  
{%endraw%}
```

O template utiliza elementos HTML em conjunto com a **sintaxe de template** do Angular.

O último elemento do código, `<hero-detail>`, é criado pelo componente

`HeroDetailComponent`.

Metadata

Os metadados dizem ao Angular como processar uma classe. Uma classe, na verdade, só se torna um componente do Angular ao utilizar metadados. No TypeScript é utilizada uma *decorator function* para informar metadados para o Angular.

Exemplo (arquivo `app/hero-list.component.ts`):

```
@Component({  
  selector:    'hero-list',  
  templateUrl: 'app/hero-list.component.html',  
  providers:   [ HeroService ]  
})  
export class HeroListComponent implements OnInit {  
  /* ... */  
}
```

O decorator `@Component` identifica uma classe como um componente. O parâmetro é um objeto que contém atributos como:

- `selector`: o seletor CSS que diz ao Angular como criar e inserir uma instância deste componente. No caso do exemplo, o seletor `hero-list` indica ao Angular que deve procurar por um elemento (está usando um seletor de elemento)
- `templateUrl`: o caminho para o arquivo do template
- `directives`: array de componentes ou diretivas que este componente requer
- `providers`: array de serviços que o componente requer. No exemplo, o componente `HeroListComponent` depende do componente `HeroService`, que é um serviço.

Outros decorators são `@Injectable`, `@Input` e `@Output`.

Data binding

O mecanismo de **data binding** é usado pelo Angular para coordenar a sincronia entre partes do template e partes de um componente.

Exemplo (arquivo `app/hero-list.component.html`):

```
{%raw%}  
<li>{{hero.name}}</li>  
<hero-detail [hero]="selectedHero"></hero-detail>  
<li (click)="selectHero(hero)"></li>  
{%endraw%}
```

Esse código é interpretado da seguinte forma:

- O código entre `{{ }}` usa o recurso chamado de **interpolação** e faz com que o valor de `hero.name` (um tipo de expressão TypeScript) seja apresentado dentro do elemento `li`.
- A **property binding** expressada por `[hero]="selectedHero"` passa o valor de `selectedHero` (atributo do componente `HeroListComponent`) para a propriedade `hero` do componente filho `HeroDetailComponent` (representado pelo elemento `hero-detail`).
- O **event binding** expressado por `(click)="selectHero(hero)"` chama o método `selectHero()` quando o usuário clicar no elemento `li`.

Um tipo especial de **data binding** chamado **two-way data binding** combina **property binding** e **event binding**, usando a diretiva `ngModel`. Exemplo:

```
<input [(ngModel)]="hero.name">
```

Usando **two-way data binding** o valor de uma propriedade (`hero.name`) passa do componente atual para o elemento `input`. Quando o valor do `input` é modificado, ele é atribuído de volta para a propriedade.

Directives

Os templates do Angular são dinâmicos, e são modificados quando o Angular transforma o DOM de acordo com as instruções dadas pelas **directives**.

Uma **directive** é uma classe com metadados de diretiva. No TypeScript é utilizada a *decorator function* `@Directive` para adicionar metadados a uma classe.

Directives podem ser de dois tipos: **de estruturas** ou **de atributos**.

As **diretivas de estrutura** alteram o DOM adicionando, removendo ou substituindo elementos. Exemplo:

```
<li *ngFor="let hero of heroes"></li>
<hero-detail *ngIf="selectedHero"></hero-detail>
```

São usadas duas diretivas:

- `*ngFor` indica ao Angular que deve repetir o elemento `li` para cada item do array `heroes`
- `*ngIf` indica ao Angular que só deve incluir o componente `HeroDetailComponent` se o valor de `selectedHero` estiver definido.

Diretivas de atributo alteram a aparência ou o comportamento de um elemento já existente no DOM. A diretiva `ngModel` modifica o comportamento de um elemento existente (como um `input`) definindo seu valor e respondendo ao evento de alteração do seu valor. Exemplo:

```
<input [(ngModel)]="hero.name">
```

Outras diretivas são `ngSwitch` (de estrutura), `ngStyle` e `ngClass` (de atributo).

Services

Um **service** é, tipicamente, uma classe com um propósito muito específico. Na verdade, não há uma definição especial do Angular para um **service**, pois ele é, basicamente, uma classe.

Exemplo (arquivo `app/logger.service.ts`):

```
export class Logger {
  log(msg: any) { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any) { console.warn(msg); }
}
```

O serviço `Logger` faz log de mensagens no console do browser.

Outro exemplo (arquivo `app/hero.service.ts`):

```
export class HeroService {
  private heroes: Hero[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes;
  }
}
```

Neste exemplo, o serviço `HeroService` consulta dados e depende dos serviços `BackendService` e `Logger`.

No formato de arquitetura do Angular, é preferível que as classes de componentes não façam tarefas como buscar dados de um servidor, validar entradas do usuário ou fazer log diretamente no console do browser. Essas tarefas são delegadas para serviços. A tarefa de um componente é lidar com a experiência (interação) com o usuário, e nada mais. O componente intermedia a interação entre a view e a lógica do aplicativo. Um bom componente apresenta propriedades e métodos para data binding. Por fim, delega tudo não trivial para serviços.

Dependency Injection

Dependency Injection (DI) é uma forma de suprir uma nova instância de uma classe com as dependências que ela requer. Exemplo do construtor do componente `HeroListComponent` (arquivo `app/hero-list.component.ts`):

```
constructor(private service: HeroService) { }
```

Quando o Angular cria um componente, primeiro consulta o **injector** sobre serviços requeridos pelo componente. O injetor mantém uma lista de instâncias de serviços criada anteriormente. Se uma instância de serviço não estiver na lista, o injetor cria uma instância e adiciona na lista antes de retorna o serviço ao Angular. Por fim, o Angular chama o construtor do componente informando os serviços como argumentos. Esse processo é chamado **injeção de dependência**.

O injetor identifica os serviços necessários por meio dos metadados do módulo ou do componente. Geralmente, isso é feito no módulo raiz. Por exemplo (trecho do arquivo `app/app.module.ts`):

```
@NgModule({
  ...
  providers: [ BackendService, HeroService, Logger ],
  ...
})
export class AppModule { }
```

Por meio da *decorator* `@NgModule` a lista de **providers** indica as dependências de serviços do `AppModule`. Desta forma, ao registrar os serviços como dependência do módulo raiz, elas ficam disponíveis em todo o aplicativo.

Uma alternativa é registrar as dependências de serviços em um componente. Exemplo (trecho do arquivo `app/hero-list.component.ts`):

```
@Component({
  selector: 'hero-list',
  templateUrl: 'app/hero-list.component.html',
  providers: [ HeroService ]
})
```

Registrar uma dependência de serviço no nível do componente significa que será criada uma instância do serviço para cada instância do componente em questão.

Resumo

Estes são os principais blocos de construção (elementos) da arquitetura do Angular:

- Modules
- Components
- Templates
- Metadata
- Data binding
- Directives
- Services
- Dependency Injection

Estes elementos formam a base tudo em um aplicativo Angular. Além deles, há recursos como animação, detecção de mudanças, eventos, formulários, XHR (ajax), elementos do ciclo de vida e roteamento.

Estrutura do projeto

Este capítulo apresenta a estrutura do projeto de um software para organização de eventos, com funções de gerenciamento de eventos e de inscrições.

O projeto utilizado neste capítulo tem a seguinte estrutura:

```
-- /raiz-do-projeto
+-- /config
    +-- helpers.js
    +-- webpack.common.js
    +-- webpack.dev.js
    +-- webpack.prod.js
+-- /public
+-- /src
    +-- /app
        +-- app.component.css
        +-- app.component.html
        +-- app.component.ts
        +-- app.module.ts
    +-- index.html
    +-- main.ts
    +-- polyfills.ts
    +-- vendor.ts
+-- package.json
+-- tsconfig.json
+-- typings.json
+-- webpack.config.js
```

Os três principais diretórios são:

- `config` : contém arquivos de configuração do Webpack
- `public` : contém arquivos comuns para todo o projeto (ex.: imagens, arquivos CSS)
- `src` : contém o "código-fonte" do aplicativo

Dependências

O arquivo `package.json` contém as dependências do projeto. Há as dependências de desenvolvimento e as dependências de produção. Veja mais sobre isso no capítulo sobre o `npm`.

Algumas dependências são:

- De produção: angular, jquery, bootstrap

- De desenvolvimento: webpack, webpack-dev-server, file-loader, css-loader

Webpack

A documentação do Angular utiliza **System.js** para compilação do código TypeScript em JavaScript. Entretanto, este capítulo utiliza **Webpack**.

O arquivo `webpack.config.js` contém o código:

```
module.exports = require('./config/webpack.dev.js');
```

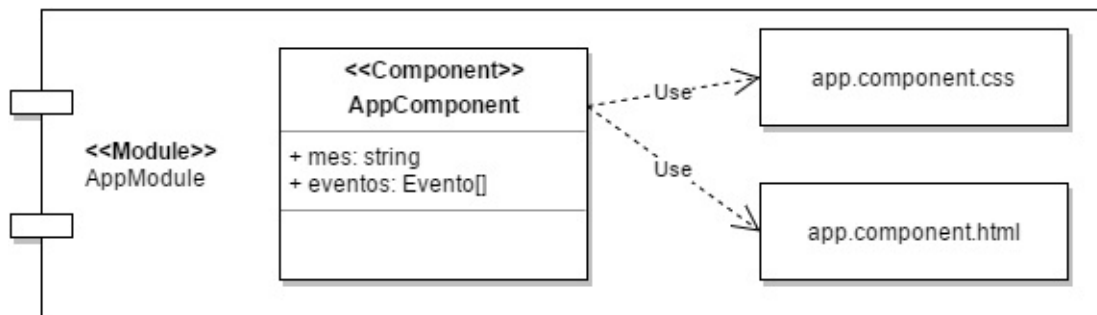
Os arquivos da pasta `config` contém três arquivos de configuração do Webpack:

- `config/webpack.common.js` : contém configurações comuns
- `config/webpack.dev.js` : contém configurações para o ambiente de desenvolvimento
- `config/webpack.prod.js` : contém configurações para o ambiente de produção

Veja mais informações sobre isso no capítulo do Webpack.

Diagrama

A figura a seguir apresenta o diagrama do aplicativo.



A classe `AppModule` fornece o módulo, enquanto a classe `AppComponent` fornece o componente (que usa os arquivos `app.component.css` e `app.component.html`).

Código-fonte do projeto

O diretório `src` contém o código-fonte do projeto. Há "arquivos de configuração" do aplicativo ou que criam uma estrutura padrão para receber o módulo `AppModule` :

- `src/index.html`
- `src/main.ts`

- `src/vendor.ts`

As seções seguintes apresentam estes arquivos em detalhes.

Arquivo `index.html`

O arquivo `src/index.html` é o template principal do projeto; é o primeiro a ser carregado pelo browser. Seu conteúdo:

```
<html lang="pt-br">
  <head>
    <title>Angular 2 QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
    <my-app>Aguarde, carregando...</my-app>
  </body>
</html>
```

O template contém código HTML para carregar o restante do aplicativo. Importante notar que não há uso de elementos `script` ou `link`, para carregar arquivos JavaScript ou CSS, pois eles são embutidos automaticamente pelo Webpack.

A parte diferenciada do template é a utilização do elemento `my-app`. Este elemento representa o local onde o Angular irá inserir o componente `AppComponent` (arquivo `src/app/app.component.ts`).

Arquivo `main.ts`

O arquivo `src/main.ts` contém o código TypeScript necessário para carregar o aplicativo ("carregar" no jargão do Angular é fazer o "bootstrap"). Seu conteúdo:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

if (process.env.ENV === 'production') {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```


Os usos da instrução `import` indicam os módulos ou bibliotecas sendo importados. O condicional baseado no valor de `process.env.ENV` é utilizado para habilitar o chamado "modo de produção do aplicativo" (que adiciona certos comportamentos quando o aplicativo estiver em modo de produção).

A última linha é que, efetivamente, faz o "bootstrap" do aplicativo, ao chamar o método `bootstrapModule()` do objeto criado pela chamada ao método `platformBrowserDynamic()`, informando como parâmetro o `AppModule` (que representa o módulo raiz do aplicativo).

Arquivo `vendor.ts`

O arquivo `src/vendor.ts` é responsável por importar módulos ou bibliotecas globalmente (de forma que possam ser usadas por todos os módulos do aplicativo). Seu conteúdo:

```
// Angular 2
import '@angular/platform-browser';
import '@angular/platform-browser-dynamic';
import '@angular/core';
import '@angular/common';
import '@angular/http';
import '@angular/router';

// RxJS
import 'rxjs';
```

As importações são, praticamente, direcionadas para carregar os módulos do Angular no aplicativo.

O diretório `app` contém a definição do módulo raiz, utilizado no aplicativo. O módulo possui quatro arquivos:

- `src/app/app.module.ts`
- `src/app/app.component.ts`
- `src/app/app.component.html`
- `src/app/app.component.css`

As seções a seguir apresentam os arquivos em detalhes.

Arquivo `app.module.ts`

O arquivo `src/app/app.module.ts` contém a definição da classe `AppModule`, ou seja, o módulo raiz do aplicativo. Seu conteúdo:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule, FormsModule ],
  declarations: [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

O arquivo começa importando alguns módulos (como `@angular/core` e `@angular/forms`) e importa a classe `AppComponent`. Utilizando a *anotator function* `@NgModule()` os metadados da classe `AppModule` são definidos:

- atributo `import` : importa os módulos `BrowserModule` e `FormsModule`
- atributo `declarations` : fornece (declara) o módulo `AppComponent`
- atributo `bootstrap` : indica ao Angular que deve utilizar o módulo `AppComponent` durante o bootstrap

Arquivo `app.component.ts`

O arquivo `src/app/app.component.ts` contém a declaração do componente `AppComponent`. Na prática, como o `AppModule` não possui visual, `AppComponent` representa o componente visual do aplicativo. Trecho do seu código:

```
import { Component } from '@angular/core';
import '../public/css/styles.css';
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  ...
}
```

O código importa a anotator function `Component`, definida em `@angular/core`. A função é utilizada para definir os metadados do componente:

- `selector` : indica o seletor usado para indicar ao Angular onde deve apresentar o componente
- `templateUrl` : indica o arquivo usado para o template do componente (neste caso, o arquivo `src/app/app.component.html`)

- `styleUrls` : um array que indica os arquivos CSS utilizados no componente (neste caso, o arquivo `src/app/app.component.css`)

O seletor `my-app` instrui o Angular que deve procurar o elemento `my-app` no template do aplicativo e, dentro dele, modificar o DOM para inserir o conteúdo. Como já visto, este elemento está no arquivo `src/index.html` .

A utilização da instrução `import` é bastante flexível. No caso de um arquivo `.css` , isso instrui o Webpack a inserir o arquivo em questão na lista de arquivos CSS que devem ser reunidos para gerar a versão de produção do aplicativo.

Arquivo `app.component.html`

O arquivo `src/app/app.component.html` contém o template do módulo. Seu conteúdo:

```
{%raw%}  
<h1>Eventos do mês de {{mes}}</h1>  
<ul>  
  <li *ngFor="let evento of eventos">{{evento}}</li>  
</ul>  
{%endraw%}
```

O template é informado nos metadados do componente (trecho do arquivo `src/app/app.component.ts`):

```
import { Component } from '@angular/core';  
import '../public/css/styles.css';  
@Component({  
  selector: 'my-app',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  ...  
}
```

O template apresenta o valor do atributo `mes` e utiliza a diretiva `ngFor` para apresentar uma lista de valores (definidos no atributo `evento` , que é um array).

Arquivo `app.component.css`

O arquivo `src/app/app.component.css` contém as definições de estilos CSS para o componente `AppComponent`. Um recurso importante do Angular é dar um contexto para o CSS: o próprio componente. Isso significa que o conceito de modularização também funciona para os estilos CSS.

O arquivo CSS é especificado nos metadados do componente (trecho do arquivo

`src/app/app.component.ts`):

```
import { Component } from '@angular/core';
import '.../public/css/styles.css';
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  ...
}
```

Observables

Cada método da classe `Http` (módulo `@angular/http`) retorna um `Observable` de objetos `Response`.

Um "observable" representa uma cadeia de eventos que podem ser processados como operadores de arrays. O Angular core tem suporte básico para observables, mas outros podem ser obtidos do pacote `rxjs`. Nesse contexto, um "observable" fornece uma maneira assíncrona para o processamento de dados retornados por `Http`.

Utilizando `Http.get()`

O código a seguir (trecho do arquivo `src/app/eventos.service.ts`) utiliza o método `Http.get()`:

```
getEventos(filtro: string) : Observable<Evento[]> {  
    return this.http.get(this.apiUrl + '?q=' + filtro)  
        .map(this.extractData)  
        .catch(this.handleError);  
}
```

Neste trecho de código, o atributo `apiUrl` é uma string que representa o endereço de um serviço `Http` (API).

O método `http.get()` retorna um `Observable` e os encadeamentos que se seguem, usando `map()` e `catch()`, são funções que tratam o `Observable` de maneiras específicas:

- `map()` : nesse contexto, é usada para converter o objeto `Response` para o formato `Json`
- `catch()` : nesse contexto, é usada para tratamento de erros.

Cada uma destas funções recebe uma função como parâmetro. A função `extractData()`:

```
private extractData(res: Response) {  
    let body = res.json();  
    return body.data || {};  
}
```

Nesse contexto, a API HTTP que está sendo utilizada adota o padrão de "encapsular" os dados de retorno em um atributo `data` (um array ou um objeto). Se ele não estiver presente, o método retorna `{}` (um objeto "vazio").

A função `handleError()` :

```
private handleError(error: any) {  
    let errMsg = (error.message) ? error.message :  
        error.status ? `${error.status} - ${error.statusText}` : 'Server error';  
    console.error(errMsg);  
    return Observable.throw(errMsg);  
}
```

Utilizando um `observable`

O `observable` é encapsulado em um Service (no caso, `EventosService`). Para ser utilizado em um componente, o retorno de `getEventos()` é tratado com a função `subscribe()` .

Exemplo (trecho do arquivo `src/app/eventos-list.component.ts`):

```
getEventos() {  
    this.eventosService.getEventos(this.filtro)  
        .subscribe(  
            eventos => this.eventos = eventos,  
            error => this.errorMessage = <any>error);  
}
```

A função `subscribe()` recebe dois parâmetros, seguindo o encadeamento definido no código de `EventosService.getEventos()` :

- o primeiro parâmetro é uma função que recebe o parâmetro `eventos` , do tipo `Evento[]` (relacionada com a função `map()`)
- o segundo parâmetro é uma função que recebe o parâmetro `error` , que representa uma indicação de erro da comunicação Http (relacionada com a função `catch()`)

Serviços

Conforme as recomendações do Angular, a utilização de serviços traz recursos como a separação de interesses. Como já informado, um service é basicamente uma classe, e não tem um tratamento diferenciado do Angular, como acontece com os componentes.

Exemplo (trecho do arquivo `src/app/eventos.service.ts`):

```
import { Injectable } from '@angular/core';
import { Http, Response, Headers, RequestOptions } from '@angular/http';
import { Evento } from '../evento';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class EventosService {
  constructor(private http: Http) { }

  getEventos(filtro: string) : Observable<Evento[]> { }

  private extractData(res: Response) { }

  private handleError(error: any) { }

  save(evento: Evento) : Observable<Evento> { }

  delete(evento: Evento) : Observable<number> { }
}
```

Note a presença da função `@Injectable()` (do pacote `@angular/core`). Essa anotação da classe é importante para informar a outras classes que `EventosService` deve ser utilizada como um serviço.

<<Injectable>> Eventos Service
+ getEventos(filtro: string): Observable<Evento[]> - extractData(res: Response) - handleError(error: any) + save(evento: Evento) : Observable<Evento> + delete(evento: Evento) : Observable<number>

No construtor é utilizado o recurso de injeção de dependência para que a classe `EventosService` possa usar a classe `Http` (fornecida pelo pacote `@angular/http`), que fornece acesso a recursos de comunicação sobre HTTP utilizando XHR (XmlHttpRequest).

Considera-se neste capítulo que a API utilizada tem a seguinte estrutura:

API
<ul style="list-style-type: none"> + GET /eventos + GET /eventos/{id} + POST /eventos/{id} + DELETE /eventos/{id}

As rotas são:

- `GET /eventos` : retorna um objeto `{data:[]}` representando os eventos existentes. Essa rota aceita o parâmetro de URL `q` , que contém uma string usada para filtrar o conjunto de dados
- `GET /eventos/{id}` : retorna um objeto `{data: {}}` representando um evento cujo identificador é representado pelo parâmetro de rota `id`
- `POST /eventos/{id}` : salva (cadastra ou atualiza) os dados de um evento cujo identificador é representado pelo parâmetro de rota `id` e os dados estão no corpo do protocolo HTTP. O retorno é um objeto `{data:{}}` com os dados atualizados do evento salvo
- `DELETE /eventos/{id}` : exclui um evento cujo identificador é representado pelo parâmetro de rota `id` . O retorno é um objeto `{data:number}` (com `1` indicando que o evento foi excluído e `0` , que não foi incluído)

Consultando eventos

O método `getEventos(filtro: string): Observable<Evento[]>` utiliza o conceito de "Observable" para consultar dados via HTTP. Pela definição da API, utiliza a rota `GET /eventos` :

```
getEventos(filtro: string) : Observable<Evento[]> {
    return this.http.get(this.apiUrl + '?q=' + filtro)
        .map(this.extractData)
        .catch(this.handleError);
}
```

Salvando eventos

O método `save(evento: Evento) : Observable<Evento>` utiliza `http.post()` e envia os dados de um evento para a rota `POST /eventos/{id}` da API:


```
save(evento: Evento) : Observable<Evento> {
    let body = JSON.stringify(evento);
    let headers = new Headers({ 'Content-Type': 'application/json' });
    let options = new RequestOptions({ headers: headers });

    return this.http.post(this.apiUrl + evento.id, body, options)
        .map(this.extractData)
        .catch(this.handleError);
}
```

Requisições POST precisam de mais informações do que requisições GET. Por isso, o método `http.post()` aceita mais parâmetros:

- `this.apiUrl + evento.id` : representa a URL da requisição
- `body` : representa os dados que estão sendo enviados. O objeto `body` é uma string criada a partir da chamada de `JSON.stringify()` recebendo como parâmetro o objeto `evento` (o objeto que tem os dados que devem ser salvos), que é convertido para string
- `options` : é um objeto do tipo `RequestOptions` que é usado para indicar que a requisição utiliza o tipo `application/json` .

Excluindo eventos

O método `delete(evento: Evento) : Observable<number>` utiliza `http.delete()` e envia o identificador de um evento para a rota `DELETE /eventos/{id}` da API:

```
delete(evento: Evento) : Observable<number> {
    return this.http.delete(this.apiUrl + evento.id)
        .map(this.extractData)
        .catch(this.handleError);
}
```

A estrutura de `http.delete()` é bastante similar à de `http.get()` .

Múltiplos componentes

Um recurso muito importante em desenvolvimento de software é o princípio da responsabilidade única. Basicamente, o princípio indica que cada parte do software (por exemplo, um componente) deve ter um ciclo de vida próprio e deve estar direcionado para uma tarefa específica. Por exemplo, poderia ser interessante ter um componente para apresentar a lista de eventos, outro para permitir a edição/cadastro de eventos e ainda outro para apresentar os detalhes de um evento. Este capítulo apresenta como utilizar o princípio da responsabilidade única no Angular.

Componente Detalhes do Evento

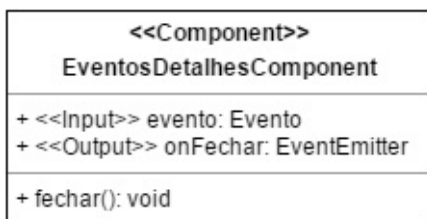
O componente Detalhes do evento está definido no arquivo `src/app/eventos-detalhes.component.ts` cujo trecho é apresentado a seguir:

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { EventosService } from './eventos.service';

import './rxjs-operators';
import { Evento } from './evento';

@Component({
  selector: 'eventos-detalhes',
  templateUrl: './eventos-detalhes.component.html',
  styleUrls: [],
  providers: [EventosService]
})
export class EventosDetalhesComponent {
  ...
}
```

A figura a seguir ilustra o component em formato UML.



Metadados

A classe `EventosDetalhesComponent` é decorada com `Component`, o que indica que é tratada pelo Angular como um componente:

- `selector` indica que o componente usa o elemento `eventos-detalhes`
- `templateUrl` indica que o arquivo `eventos-detalhes.component.html` é usado como `templateUrl`
- `providers` indica que `EventosService` é requerido pelo component

O início do código importa as funções `Component`, `Input`, `Output` e `EventEmitter`, do pacote `@angular/core`. Importa também `FormsModule`, do pacote `@angular/forms` e `EventosService`, definida no arquivo `src/app/eventos.service.ts`.

Dependência do serviço `EventosService`

O construtor da classe indica que é utilizada injeção de dependência para incluir o serviço `EventosService` como dependência do módulo:

```
constructor(private eventosService: EventosService) { }
```

Comunicação entre componentes

O Angular fornece o recurso de comunicação entre componentes por meio de **entradas** (decorador `Input()`) e **saídas** (decorador `Output()`).

As entradas permitem que um "componente hospedeiro" (o componente `AppComponent` hospeda o `EventosDetalhesComponent`) envie informações para o componente hospedado.

O componente `EventosDetalhesComponent` declara que o atributo `evento: Evento` (decorado com `Input()`) é um dado de entrada. Em outras palavras, como o componente mostra detalhes de um evento, ele precisa receber o evento (objeto do tipo `Evento`) [como entrada]. A comunicação entre hospedeiro e hospedado é feita no template (o trecho a seguir é do arquivo `app.component.html`):

```
<eventos-detalhes *ngIf="modo == 'detalhar'" #detalhes [evento]='eventoSelecioneado' (o
nFechar)="onFechar()"></eventos-detalhes>
```

O elemento `eventos-detalhes` tem como atributo `[evento]="eventoSelecioneado"`, isso faz com que o atributo `evento` de `EventosDetalhesComponent` receba `eventoSelecioneado` (um atributo de `AppComponent`, o hospedeiro).

O componente só apresenta dados se estiver recebendo um evento como entrada. Por isso, há uma verificação no template (arquivo `src/app/eventos-detalhes.component.html`):

```
{%raw%}  
<div *ngIf="evento">  
  <h2>{{evento.nome}}</h2>  
  <p>Local: {{evento.cidade}} - {{evento.estado}}</p>  
  <button (click)="fechar()" class="btn btn-default">Fechar</button>  
</div>  
{%endraw%}
```

As saídas permitem que um componente hospedeiro seja informado de um "evento" ocorrido no componente hospedado. No caso, o componente `EventosDetalhesComponent` informa seu hospedeiro quando o botão "Fechar" for clicado. O atributo `onFechar` : `EventEmitter` é decorado com `output()` , ou seja, declara um evento.

O evento é disparado na função `fechar()` , chamada no clique do botão "Fechar":

```
fechar() {  
  this.evento = null;  
  this.onFechar.emit();  
}
```

O tipo `EventEmitter` fornece o método `emit()` , que representa o "disparo" do evento. Em outras palavras, ao ser executada a função `fechar()` , o hospedeiro é informado disso. Isso é feito no template do hospedeiro:

```
<eventos-detalhes *ngIf="modo == 'detalhar'" #detalhes [evento]='eventoSelecioneado' (o  
nFechar)="onFechar()"></eventos-detalhes>
```

O elemento `eventos-detalhes` tem como atributo `(onFechar)="onFechar()"` , isso faz com que a função `onFechar()` (de `AppComponent`) seja chamada quando ocorrer o evento `onFechar` de `EventosDetalhesComponent` .

Tecnologias Back-end

As tecnologias Back-end já foram chamadas de "programação no lado servidor". Desde os primórdios da programação web, o lado cliente e o lado servidor divergiram entre si não apenas em relação ao local da computação mas na forma de utilização dela.

Esta seção do livro trata do PHP como linguagem de programação e de outras tecnologias e ferramentas, como o Silex, um micro-framework MVC voltado para a criação de aplicações web modernas e construção de API REST.

PHP - Hypertext Preprocessor

PHP é uma linguagem de programação de script, interpretada e multiplataforma para o desenvolvimento de aplicações Web. O site oficial do PHP é <http://php.net/> - lá você ter acesso às definições do PHP e sua documentação (acesso online ou download).

Para que o desenvolvimento de scripts PHP possa ser iniciado, antes de mais nada, é necessário ter um Servidor Web Apache e um interpretador de PHP instalado na máquina. Depois que estiver um servidor devidamente instalado e configurado, pronto, já podemos começar a desenvolver scripts PHP.

História do PHP

Tim Berners-Lee, em 1990, foi o idealizador da World Wide Web, que tornava possível o funcionamento de página HTML através do protocolo HTTP. O protocolo HTTP permite que páginas disponibilizadas em um determinado servidor sejam enviadas requisitadas/enviadas para os clientes que requererem.

Em 1994, Rasmus Lerdorf, teve curiosidade em saber quantas pessoas estavam visitando o seu site pessoal. Isso o motivou a desenvolver um mecanismo um mecanismo que permitisse esta contagem, que à época foi intitulada como Personal Home Pages Tools, através da linguagem Perl. Naquele momento, o PHP ainda não era considerado uma linguagem, pois se tratava apenas de uma interface que permitia as pessoas verificarem quantos clientes estavam acessando um determinado site em um instante de tempo qualquer.

Dado o sucesso que esta funcionalidade trouxe à época, Rasmus se inspirou a desenvolver a segunda versão do Personal Home Page Tools, em que adicionou novas funcionalidades, por exemplo, a possibilidade dos visitantes deixarem mensagens para os proprietários dos sites através de formulários. A possibilidade desta interpretação de formulários motivou Rasmus a renomear o seu projeto para Personal Home Page Forms Interpreter (PHP/FI). Mesmo com as alterações realizadas, o PHP ainda não era considerado uma linguagem de programação, ainda era considerado uma CGI (interface) que auxiliava na criação de funcionalidades para páginas pessoais.

Depois da segunda versão, Rasmus liberou o acesso ao código fonte do PHP/FI a outros programadores, que desenvolveram uma série de outras funcionalidades, como por exemplo, o acesso a banco de dados. Esta estratégia popularizou ainda mais o PHP/FI. A popularização foi tão grande que em 1997 já existiam em torno de 60.000 páginas utilizando PHP/FI. Após este marco de 1997, em 1998, dois programadores israelitas se juntaram e

reescreveram o PHP: Andi Gutmans e Zeev Suraski, e neste momento, na versão 3.0, o PHP passou a ser chamada de PHP Hypertext Preprocessor e considerado uma linguagem de programação. A linguagem se popularizou tanto que já em 1999 estimava-se que aproximadamente 10% dos sites já utilizavam o PHP. Do seu surgimento até hoje, várias versões surgiram, com o acréscimo de diversas funcionalidades com o propósito de torna-la uma linguagem mais robusta e que atenda com maior eficácia as necessidades dos usuários. Atualmente já está versão 5.6.

Referências da linguagem: sintaxe básica e visão geral

Sempre que existirem códigos dentro das super-tags `<?php` e `?>`, então estes serão tratados como códigos PHP. A seguir é apresentado um exemplo simples em PHP, que imprime uma simples mensagem:

```
<?php
    echo "Isso é um teste!";
?>
```

`echo` é uma definição que permite imprimir uma mensagem. `print` também pode ser usado, ao invés de `echo`.

Para comentar uma linha de um código PHP basta utilizar `//`.

```
// Esta linha é um comentário e não será interpretada pelo PHP
```

Caso deseje comentar múltiplas linhas, basta coloca-las entre os símbolos `/*...*/`.

```
/*
Esta é a linha 1 comentada
Esta é a linha 2 comentada
Esta é a linha 3 comentada
...
*/
```

O símbolo `$` indica uma variável, por exemplo:

```
$variavel_exemplo ="teste variável";
```

No exemplo apresentado anteriormente, a variável `$variavel_teste` recebe a string `"teste variável"`.

Estruturas de Controle

- `if...else`
- `while`

Tipos de Dados

São 8 os tipos primitivos suportados pelo PHP, dentre eles estão: boolean, que permite trabalhar com valores lógicos; integer, que permite trabalhar com números inteiros; e string, para textos. A referência completa da linguagem pode ser encontrada em <http://php.net/>. Aqui nós apresentaremos apenas alguns deles.

- [String](#)
- [Array](#)

String

A maneira mais simples para especificar uma string é colocá-la entre apóstrofes (caracter ').

```
<?php
echo 'forma mais simples de especificar uma string';
echo 'é possível imprimir "aspas" assim';
?>
```

Com o uso das aspas simples, o valor `{ $valor }` contido na string será interpretado como texto (`string`), diferente do uso das aspas duplas (`"`).

```
<?php
$valor=5;
echo "utilizando as aspas { $valor } duplas";
echo "</br>";
echo 'utilizando as aspas { $valor } simples';
?>
```

Array

Um `array` permite representar um conjunto de dados em uma única variável. Para criar um `array` em PHP é necessário utilizar a função `array()`. São três os tipos de arrays disponíveis no PHP, são eles:

- **Arrays indexados:** arrays com índices numéricos;
- **Arrays associativos:** arrays onde cada índice é referenciado por uma chave;
- **Arrays multidimensionais:** são matrizes, isto é, arrays que contém um ou mais arrays.

Arrays Indexados

Quando o array é indexado por índices. No código apresentado a seguir, é criado um array `$arr` que contém 4 elementos (3, 5, 7 e 9).

```
<?php
$arr = array (3, 5, 7, 9);
?>
```

A atribuição também pode ser feita maneira manual, índice por índice:

```
<?php
$arr[0]=3;
$arr[1]=5;
$arr[2]=7;
$arr[3]=9;
?>
```

Acrescentando um novo valor ao `array`

Se um `array` foi previamente definido, é possível modificá-lo acrescentando um novo elemento a ele, conforme exemplificado a seguir:

```
<?php
$arr = array (3, 5, 7, 9);
$arr[4]=11;

// esta sintáxe também funcionaria.
// $arr[]=11;
?>
```

Se o índice for omitido (conforme linha comentada no código acima), o PHP identifica automaticamente o último índice utilizado e associa o novo valor ao índice seguinte (no exemplo supracitado, seria o índice de valor **4** que receberia o valor **11**).

Percorrendo todos os valores do array

A seguir é apresentado um exemplo em que o array é percorrido e todos os seus elementos são apresentados.

```
<?php
$arr = array (3, 5, 7, 9);
for ($i=0;$i<4;$i++){
    echo $arr[$i];
    echo "<br>"; // quebra de linha
}
?>
```

Arrays associativos

Os **arrays associativos** são mapas que permitem relacionar valores às chaves. Assim, para cada valor representado no array há uma chave equivalente que permite acessá-lo. A estrutura básica de um array associativos é:

```
array( chave => valor
    , ...
)
// chave pode ser tanto string ou um integer
// valor pode ser qualquer coisa
```

Um array associativo pode ser criado conforme código apresentado a seguir:

```
<?php
$arr = array("salvador" => "Jesus Cristo", 10 => true);
?>
```

Os arrays associativos permitem a criação de chaves e valores heterogêneos (tipos diferentes). No exemplo apresentado anteriormente, existem dois tipos de chave, uma `string` ("salvador") , com uma `string` como valor ("Jesus Cristo"); e outra chave do tipo `inteiro` (10) com um valor `booleano` (true) .

Da mesma forma que o **array indexado**, os elementos de um **array associativo** também podem ser criados um a um.

```
<?php
$arr["salvador"] ="Jesus Cristo";
$arr [10]=true;
?>
```

Um exemplo que mostra como é realizada a impressão dos elementos de um **array associativo** é mostrado a seguir:

```
<?php
$arr = array("salvador" => "Jesus Cristo", 10 => true);
echo $arr["salvador"]; // saída: Jesus Cristo
echo "<br>"; // quebra de linha em HTML
echo $arr[10]; // saída: 1
//echo $arr[12]; #dará erro porque não existe esta chave
?>
```

O acesso a uma chave que não tenha sido previamente definida gerará uma **exceção** (*tire o comentário da linha que tenta imprimir o array co a chave 12 para testar*).

Acrescentando um novo valor ao array

É possível adicionar novo par de **chave-valor** a um `array` previamente definido.

```
<?php
$arr = array("num1"=>2, "num2"=>4);
$arr["num3"]=8;
echo $arr["num3"];
?>
```

Neste exemplo anterior, foi acrescentada a **chave** `num3` associando ao **valor** `8` .

Percorrendo todos os valores do array

Para percorrer todos os elementos pode ser utilizada a estrutura de controle `foreach` , da seguinte forma:

```
<?php
$arr = array("num1"=>2, "num2"=>4);
$arr["num3"]=8;
echo "Imprimindo os valores do array";
foreach ($arr as $i=>$valor)
{
    echo "<br>";
    echo ($arr[$i]);
}
?>
```

Arrays Multidimensionais

Um **array multidimensional** é um `array` com várias dimensões, que pode conter um ou mais arrays.

Para exemplificação de um **array multidimensional** utilizaremos a tabela apresentada a seguir:

Nome	Sexo	Idade
João	Masculino	18
Paula	Feminino	19
Francisca	Masculino	20

Para representar os dados da tabela anterior em um **array multidimensional**, o seguinte código poderia ser utilizado:

```
<?php
$dados = array
(
    array("João", "Masculino", 18),
    array("Paula", "Feminino", 19),
    array("Francisco", "Masculino", 20),
);
?>
```

Para que cada elemento deste `array` possa ser acessado, é necessário fazer referência a dois índices, a linha e a coluna. Por exemplo:

```
<?php
echo "Nome: ".$dados[0]{0}." , Sexo: ".$dados[0][1]." , Idade: ".$dados[0]{2}."</br>";
echo "Nome: ".$dados[1]{0}." , Sexo: ".$dados[1][1]." , Idade: ".$dados[1]{2}."</br>";
echo "Nome: ".$dados[2]{0}." , Sexo: ".$dados[2][1]." , Idade: ".$dados[2]{2};
?>
```

Acrescentando um novo valor ao array

Conforme sintaxes previamente apresentadas, é possível acrescentar novos valores a um array previamente definido. A seguir é apresentado um exemplo de código com várias formas possíveis de fazer esta modificação.

```
<?php
$dados = array
(
    array("João", "Masculino", 18),
    array("Paula", "Feminino", 19),
    array("Francisco", "Masculino", 20),
);

$dados[3] = array ("Maria", "Feminino",21);

// OU

//$dados[] = array ("Maria", "Feminino",21);

// OU

/*
$dados[3][0]="Maria";
$dados[3][1]="Feminino";
$dados[3][2]=21;
*/

// OU

/*
$dados[3][]="Maria";
$dados[3][]="Feminino";
$dados[3][]=21;
*/
?>
```

Para testar as demais alternativas, basta **retirar os comentários** das outras abordagens e comentar as demais.

Percorrendo todos os valores do array

Também é possível percorrer todos os elementos do `array` através de uma **estrutura de repetição**. O código a seguir exemplifica o uso de um laço de repetição:

```
<?php
for ($linha=0; $linha<3; $linha++){
    echo "linha n.º $linha";
    echo "<ul>";
    for ($coluna=0; $coluna<3; $coluna++){
        // echo "<li>. $dados[$linha][$coluna].</li>";
        echo "<li> {$dados[$linha][$coluna]}</li>";
    }
    echo "</ul>";
}
?>
```

Exercícios

Exercício 1: Utilizando *array indexado*, faça um script PHP que armazene uma lista de pessoas, em que cada uma é identificada por um código. Deve ser armazenado somente o nome de cada pessoa. O código que identifica cada pessoa é o próprio índice do vetor. Apresente os dados das pessoas (nome e código) em uma tabela formatada em HTML.

Exercício 2: Utilizando *array multidimensional*, altere o exercício anterior acrescentando mais dados para cada pessoa, que são Sobrenome e RG.

Exercício 3: Agora, utilizando *array associativo*, incremente os exercícios anteriores, acrescentando *Endereço* para cada pessoa. O *Endereço* deve ser representado em outro array e deve conter: Logradouro, Bairro, número e CEP. Os nomes das chaves devem estar coerentes com a sua representação, por exemplo, para identificar o nome de uma pessoa, utilize a chave `nome`.

Operadores Aritméticos

São aqueles operadores utilizados para realização de operações matemáticas.

Operador	Nome
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto da divisão)
**	Exponencial

A seguir é apresentado um exemplo de utilização destes operadores:

```
<?php

$a=10;
$b=5;

echo "A soma de A e B é: " . ($a+$b);
echo "</br> A subtração de A e B é: " . ($a-$b);
echo '</br> A multiplicação de A e B é: ' . $a*$b;
echo '</br> A divisão de A por B é: ' . ($a/$b);
echo '</br> O resto da divisão de A por B é: ' . ($a%$b);
echo '</br> A potência de A elevado a B é: ' . ($a**$b) ;
?>
```

Exercícios

Exercício 1: Faça um programa que gere como saída as tabuadas de adição, subtração, multiplicação e divisão dos números 1, 2, 3, 4, 5, 6, 7, 8, 9 e 10. A saída deve ser em HTML e cada linha da tabuada deve ser gerada automaticamente (através do uso do laço de repetição).

Multiplicação: Tabuada do 1	Multiplicação: Tabuada do 2
$1 \times 1 = 1$	$2 \times 1 = 2$
$2 \times 1 = 2$	$2 \times 2 = 4$
$3 \times 1 = 3$	$3 \times 2 = 6$
$4 \times 1 = 4$	$4 \times 2 = 8$
$5 \times 1 = 5$	$5 \times 2 = 10$
$6 \times 1 = 6$	$6 \times 2 = 12$
$7 \times 1 = 7$	$7 \times 2 = 14$
$8 \times 1 = 8$	$8 \times 2 = 16$
$8 \times 1 = 8$	$9 \times 2 = 18$
$10 \times 1 = 10$	$10 \times 2 = 20$

Resolução do Exercício 1 por um aluno.

Estruturas de Controle

Assim como em outras linguagens, na PHP, um script é desenvolvido por uma série de instruções. São vários os tipos de instruções, tais como, atribuições, uma condição, ou um laço de repetição. O fim de uma instrução é determinado por um ponto e vírgula (;). Estas instruções podem ser agrupadas em um grupo de comandos com o uso das chaves { } . Cada grupo de instruções também é denominado uma instrução. Estes grupos de comandos denominados, Estruturas de Controle, tem dois tipos de controle: de Seleção ou de Repetição. O propósito deste capítulo é apresentar os diferentes tipos de instruções disponíveis.

if...else

O `if` habilita o uso de uma seleção (expressão condicional). A sintaxe básica utilizada é:

```
if (expressao==true){  
    comando1;  
    comando2;  
}else{  
    comando3;  
    comando4;  
}
```

A seguir é apresentado um exemplo simples de utilização da sintaxe da expressão condicional:

```
<?php  
$media=7;  
if ($media>6)  
{  
    echo "aprovado";  
}  
else  
{  
    echo "reprovado";  
}  
?>
```

Conforme já apresentado anteriormente, para determinar blocos de códigos (instruções), são utilizadas `{}` .

Um **condicional** também pode ser representado de uma **maneira alterantiva**, através de uma sintaxe que não faz uso das chaves `{}` , conforme apresentado a seguir:

```
``php <?php if (expressao == true): comando 1; comando 2; else: comando 3; comando 4;  
endif; ?>
```

while

O `while` é uma estrutura de controle que permite a repetição de blocos de códigos (sequências de comandos). A estrutura básica de uma estrutura `while` é dada da seguinte forma:

```
<?php
while (expressao==true)
{
    comando1;
    comando2;
    ...
}
?>
```

Um exemplo de utilização da **estrutura de iteração** `while` é apresentado a seguir:

```
<?php
$i=0;
while ($i<10)
{
    echo "i: ".$i;
    echo "<br>"; // inserindo uma quebra de linha em HTML
    $i++; // é a mesma coisa que $i = $i+1
}
?>
```

Um exemplo de utilização da **estrutura de iteração** `while` alternativa é apresentado a seguir:

```
<?php
// estrutura alternativa para o While
$i=0;
while ($i<10):
    echo "i: ".$i;
    echo "<br>"; // inserindo uma quebra de linha em HTML
    $i++; // é a mesma coisa que $i = $i+1
endwhile
?>
```

Da mesma forma como na **estrutura condicional** que possui uma sintaxe alternativa para o seu uso, assim também funciona a estrutura `while`.

Exercícios

Exercício 1: Desenvolva um código PHP que crie automaticamente uma tabela em HTML como mostrado a seguir. O número **N** de linha e o **M** de colunas devem ser ajustáveis, ou seja, devem existir variáveis que controlam suas respectivas quantidades.

Cabeçalho	Cabeçalho	Cabeçalho	Cabeçalho
Linha 1, Coluna 1	Linha 1, Coluna 2	Linha 1, Coluna 3	Linha 1, Coluna 4
Linha 2, Coluna 1	Linha 2, Coluna 2	Linha 2, Coluna 3	Linha 2, Coluna 4
Linha 3, Coluna 1	Linha 3, Coluna 2	Linha 3, Coluna 3	Linha 3, Coluna 4

Funções

Uma função pode ser definida a partir da utilização da seguinte estrutura:

```
function nome_da_funcao ()
{
    $valor_retorno="Exemplo de função retornando este valor.";
    return $valor_retorno;
}
```

Uma **função** pode ou não ter parâmetros. A seguir é apresentada uma função com 1 parâmetro. Caso necessite passar mais parâmetros para uma função, basta inserí-los e separá-los por vírgula (,).

```
function primeira_funcao_param ($arg_1)
{
    return $arg_1;
}
```

Conforme apresentado, as funções podem ser definidas pelos usuários. Existem também uma série de funções definidas na linguagem PHP, por exemplo, funções para manipular strings ou `arrays`.

Exercícios

Exercício 1: Faça um programa que gere como saída as tabuadas de adição, subtração, multiplicação e divisão dos números 1, 2, 3, 4, 5, 6, 7, 8, 9 e 10. A saída deve ser em HTML e cada linha da tabuada deve ser gerada automaticamente (através do uso do laço de repetição). Seu código deve ser organizado em funções. Devem conter, no mínimo, 4 funções, uma para cada operação matemática: soma, subtração, multiplicação, divisão.

Funções para manipulação de variáveis

Há um conjunto de funções pré-definidas para manipulação de variáveis. Algumas delas serão apresentadas neste capítulo.

Função `empty`

A função `empty()` verifica se uma determinada variável é ou não vazia. Se for vazia, retorna `True`, caso contrário retorna `False`.

Os seguintes valores são considerados vazios: 1.

1. `""` (uma string vazia)
2. `0` (um inteiro vazio)
3. `"0"` (uma string vazia)
4. `NULL`
5. `FALSE` (um booleano vazio)
6. `array` (um array vazio)
7. `var $var;` (um variável declarada, mas sem atribuição de valor)

```
<?php
$varInteira = 0;
$varString = "0";
$varString="";
$varBooleana=False;
$varArray = array();
$varNula= NULL;

if (empty($varInteira) && empty($varString) && empty($varString)
    && empty($varBooleana) && empty($varArray) && empty($varNula)){
    echo "Todas as variáveis são vazias!";
}
?>
```

Função `gettype()`

A função `gettype()` retorna o tipo de uma variável.

```
<?php
$dados = array(1, 1.0, NULL, True, "texto");
for ($i=0; $i<count($dados); $i++){
    echo (gettype($dados[$i]))."</br>";
}
?>
```

Função `is_array()`

A função `is_array()` verifica se uma variável é um array. Se for, retorna `True`, caso contrário, retorna `False`.

```
<?php
$dados = array(1, 1.0, NULL, True, "texto");
if (is_array($dados)){
    echo 'a variável $dados é um array </br>';
}
if (!is_array($dados[0]))
    echo 'a variável $dados[0] não é um array';
?>
```

De maneira similar, existem funções para verificar outros tipos, dentre elas: `is_bool()`, `is_float()`, `is_int()`, `is_integer()`, `is_null()`. A relação completa pode ser encontrada na [documentação oficial do PHP](#).

Função `isset()`

A função `isset()` verifica se uma variável foi iniciada. Se sim, retorna `True`, caso contrário, `False`.

```
<?php
$variavel=NULL;
if (!isset($variavel)){
    echo "A variável não foi iniciada </br>";
}

$var = "";
if (isset($var)) {
    echo '$var está "setado" apesar de vazia!';
}
?>
```

Função `unset()`

A função `unset()` destrói uma variável especificada.

```
<?php
$variavel = "";
if (!isset($variavel)){
    echo "A variável foi inciada </br>";
}
unset($variavel);
if (!isset($variavel)){
    echo "A não variável não existe </br>";
}
?>
```

A seguir é apresentado um exemplo de aplicação da função `unset()` aplicado a um **array associativo**.

```
<?php
echo "</br> </br> removendo ";
$arr = array("num1"=>2, "num2"=>4);
$arr["num3"]=8;
unset($arr["num2"]); // removendo a chave "num2"
foreach ($arr as $i=>$valor)
{
    echo "</br>";
    echo ($i."=>".$arr[$i]);
}
?>
```

No trecho de código mostrado acima, foi removida a chave `"num3"` do **array** `$arr`.

Função `var_dump()`

A função `var_dump()` mostra informações de uma variável de maneira estruturada, inclusive o seu tipo. Se a variável for um `array`, mostra a quantidade de elementos, os próprios elementos e seus tipos.

```
<?php
$arr= array (1, 2, 3, 4);
var_dump($arr);
/*
  SAÍDA:
    array (size=4)
      0 => int 1
      1 => int 2
      2 => int 3
      3 => int 4
  */
?>
```

Função `print_r()`

A função `print_r()` imprime informações de uma variável. Se a variável for do tipo `array`, então todos seus valores serão impressos, no formato de chave e valor.

```
<?php
$arr= array (1, 2, 3, 4);
print_r($arr); // Saída: Array ( [0] => 1 [1] => 2 [2] => 3 [3] => 4 )
?>
```

Esta função tem um segundo parâmetro, `return`, que se definido como `True`, retornará o resultado, ao invés de imprimi-lo.

Função `strval()`

A função `strval()` retorna de uma variável convertido em `string`.

```
<?php
$v1 = 5.0;
echo $v1. "é: ".gettype($v1);
echo "<br>"; // quebra de linha
$v1 = strval($v1);
echo $v1. "é: ".gettype($v1);
?>
```

Funções para string

Existem funções específicas para manipulação de string. Algumas delas são apresentadas nesta seção.

Função `echo()`

A função `echo()` é utilizada para imprimir uma ou mais strings. `echo`, atualmente, não é considerada uma função, logo, não é obrigatório a utilização do parênteses “`()`”.

```
<?php
echo "Esta é uma mensagem simples </br>";
echo ("esta é uma mensagem "."concatenada! </br>");
?>
```

Função `print()`

A função `print()` é utilizada para imprimir uma ou mais strings, assim como a função `echo`. `print`, atualmente, não é considerada uma função, logo, não é obrigatório a utilização do parênteses “`()`”

```
<?php
print ("Esta é uma mensagem simples </br>");
print "esta é uma mensagem "."concatenada! </br>";
?>
```

Função `explode()`

A função `explode()` transforma uma `string` em um `array` de strings. Para isso, é necessário passar por parâmetro um *caracter delimitador*, em que que, sempre que for identificado na *string base*, será gerada uma nova *posição no array*.

```
</br> explode </br>

<?php
$str = "Esta é uma explicação sobre a função explode()";
$arr = explode(" ", $str);

for ($i=0; $i<count($arr); $i++){
    echo ($arr[$i])."</br>";
}
?>
```

A aplicação da função `explode()` no código mostrado, gera um array `$arr` com 8 posições, uma vez que o caracter delimitador é o espaço em branco (" ").

Função `implode()`

A função `implode()` forma uma string única a partir dos elementos de um determinado *array*. Para isso, é necessário informar o caracter que irá separar um elemento do outro.

```
<?php
$arr = array("Esta", "é", "uma", "explicação", "sobre", "a", "função", "implode()");
$str = implode(" ", $arr);
echo $str;
?>
```

Conforme código apresentado anteriormente, a função `implode()` utiliza os elementos do array `$arr` e os une em uma só string com o caracter delimitador espaço em branco " ", formando uma só frase, que foi atribuída à variável `$str`.

Função `trim()`

A função `trim()` retira os espaços em branco de uma string tanto do final quanto do início.

```
<?php
$str = " exemplo da função trim ";
if ($str == "exemplo da função trim")
    echo 'As strings NÃO são iguais iguais!';
$str = trim($str);
echo "</br>"; // imprime quebra de linha em HTML
if ($str == "exemplo da função trim")
    echo 'As strings agora são iguais!';
?>
```


Funções para array

Existe um conjunto de funções que permitem a manipulação de `arrays`. Aqui será apresentada apenas algumas. Para ter acesso a todas, procure na documentação oficial do PHP.

`count()`

A função `count` retorna a quantidade de elementos de um vetor.

```
<?php
$arr = array (2, 4, 6, 8);
$quant_itens = count($arr);
echo $quant_itens;
?>
```

`array_push()`

A função `array_push()` permite adicionar um ou vários elementos no final de um array.

```
<?php
$arr = array (2, 4, 6, 8);
array_push($arr, 10);
for ($i=0; $i<count($arr);$i++){
    echo $arr[$i]."<br>";
}
?>
```

`array_pop()`

A função `array_pop()` retorna o último elemento de um array, removendo-o do array passado por parâmetro.

```
<?php
$arr = array (2, 4, 6, 8);
$ultimo_elemento = array_pop($arr);
echo "Último elemento: ".$ultimo_elemento;
echo "</br> Novo Array: </br>" ;
for ($i=0; $i<count($arr);$i++){
    echo $arr[$i]."</br>";
}
?>
```

array_shift()

A função `array_shift()` retorna o primeiro elemento de um array, removendo-o do array passado por parâmetro.

```
<?php
$arr = array (2, 4, 6, 8);
$primeiro_elemento = array_shift($arr);
echo "Primeiro elemento: ".$primeiro_elemento;
echo "</br> Novo Array: </br>" ;
for ($i=0; $i<count($arr);$i++){
    echo $arr[$i]."</br>";
}
?>
```

array_unshift()

A função `array_unshift()` adiciona um elemento na primeira posição de um array.

```
<?php
$arr = array (2, 4, 6, 8);
array_unshift($arr,1);
echo "</br> Novo Array: </br>" ;
for ($i=0; $i<count($arr);$i++){
    echo $arr[$i]."</br>";
}
?>
```

array_pad()

A função `array_pad()` permite expandir (aumentar) o tamanho de um array para um tamanho N e inserir um determinado valor em cada uma das novas posições do array.

No exemplo seguinte, a função `array_pad` recebe por parâmetro: um array `$arr` com 4 elementos; o limite de expansão do novo array que será gerado; e o elemento que será inserido em cada nova posição do novo array. Assim, o novo array passará a ter 7 elementos, com mais 3 elementos de valor 10.

```
<?php
$arr = array (2, 4, 6, 8);
$arr = array_pad($arr, 7, 10);
echo "</br> Novo Array: </br>" ;
for ($i=0; $i<count($arr);$i++){
    echo $arr[$i]."</br>";
}
?>
```

O array só será expandido se o tamanho *N* for maior que o tamanho atual do array. No exemplo anterior, se o novo tamanho `N` passado por parâmetro for 2, então nada acontecerá, porque o tamanho atual do array é 4.

O tamanho `N` pode ser negativo. Neste caso, os novos elementos serão inseridos nas posições iniciais do novo array.

```
<?php
$arr = array (2, 4, 6, 8);
$arr = array_pad($arr, -7, -10);
echo "</br> Novo Array: </br>" ;
for ($i=0; $i<count($arr);$i++){
    echo $arr[$i]."</br>";
}
?>
```

reverse()

A função `reverse()` inverte a ordem dos elementos de um array passado por parâmetro.

```
<?php
$arr = array (2, 4, 6, 8);
$arr = array_reverse($arr);
echo "</br> Novo Array: </br>" ;
for ($i=0; $i<count($arr);$i++){
    echo $arr[$i]."</br>";
}
?>
```


Orientação a Objetos em PHP

O propósito aqui não é discutir a literatura de Orientação a Objetos, por isso, não vamos abordar com detalhes seus conceitos. A discussão sempre será direcionada para a definição de determinados conceitos da OO na linguagem PHP.

Classes e Objetos

Construtores

Propriedades

Herança

EXERCÍCIOS

Exercício 1: Utilizando conceitos de Orientação a Objetos, considere o domínio apresentado a seguir para desenvolvimento de uma solução em PHP.

Descrição do problema:

Um Serviço de Entrega Expresso oferece serviços de entregas de pacotes. Cada pacote possui código, remetente, destinatário e peso (em quilos). O custo de envio de cada pacote é de acordo com o seu peso: R\$ 5,00 (cinco reais) por quilo. Os remetentes e os destinatários são clientes cadastrados na empresa. Os dados dos clientes são: código, nome e endereço completo. Este tipo de envio de pacote está enquadrado na categoria Pacote Normal.

Também há outra opção de entrega **Pacote24Horas**, com os seguintes dados: código, remetente, destinatário, peso (em quilos) e um adicional. O adicional é calculado com base em um percentual informado no momento do cadastro de um pacote. O custo do envio é de R\$ 5.00 por quilo mais o percentual adicional sobre o valor.

De acordo com a descrição mostrada previamente, implementa um conjunto de classes baseando-se nos conceitos de OO. Sempre que possível, utilize Herança, Polimorfismo e encapsulamento. As classes devem oferecer construtor parametrizado (inicializa todos os atributos).

Além disso, as seguintes instruções devem ser atendidas:

- A implementação deve oferecer também o método **calculaCusto()**, que retorna o valor a ser pago com o envio do pacote.
- Deve existir um método para Excluir pacote (pelo código). Se o pacote for excluído com sucesso, então o método deve True, caso contrário, False. Caso a exclusão não

ocorra, deve ser impressa uma mensagem de erro no navegador.

- Cadastre diretamente no código, pelo menos: quatro clientes (lista de clientes); e dois pacotes (lista de pacotes), sendo um deles **Pacote24Horas**.
- Deve existir um método que pesquise pacote (pelo código). Se o código for inválido, deve ser apresentada uma mensagem de aviso.
- Deve existir um método que imprima um relatório de pacotes. Este relatório deve conter: código do pacote, nome do remetente e do destinatário, tipo de pacote (Normal ou 24Horas) e valor de cada pacote, e um valor total com todos os pacotes enviados(cadastrados).

Classes

Uma classe permite representar uma abstração genérica de um determinado universo, isto é, permite representar um conjunto de itens (objetos) com características similares em uma única representação. Assim, considerando que toda e qualquer possui um *Cpf* e um *Nome*, a seguir é apresentado um exemplo da definição de uma classe Pessoa.

```
<?php
class Pessoa
{
    private $cpf;
    private $nome;

    public function getCpf()
    {
        return $this->cpf;
    }

    public function setCpf($cpf)
    {
        $this->cpf = $cpf;
    }

    public function getNome()
    {
        return $this->nome;
    }

    public function setNome($nome)
    {
        $this->nome = $nome;
    }
}
```

Na classe `Pessoa` definida anteriormente (`pessoa.php`), foram definidos os métodos **Getters** e **Setters**, que possibilitam acessar e definir os atributos da classe. Os seus atributos são `$cpf` e `$nome`.

Objetos

Os objetos são instâncias de uma determinada classe. Por exemplo, considerando a classe `Pessoa` previamente definida, poderiam existir vários objetos desta classe, tais como, `pessoa1`, `pessoa2` e `pessoa3`, cada um assumindo seus respectivos valores. A seguir é

apresentado um exemplo de um código PHP exemplifica a instanciação de um objeto Pessoa.

```
<?php
require ("Pessoa.php");
$objPessoa = new Pessoa();
$objPessoa->setCpf("000-000-000-000");
$objPessoa->setNome("Fulano da Silva");
echo "CPF: ".$objPessoa->getCpf()."<br>NOME:".$objPessoa->getNome();
?>
```

A instrução `require("Pessoa.php")` habilita o uso de todas as definições disponíveis no arquivo **"Pessoa.php"** (neste caso, a definição da classe `Pessoa`) no arquivo atual. Assim, é possível instanciar objetos da referida classe e, naturalmente, realizar ações sobre estes através dos seus métodos.

No exemplo mostrado anteriormente, é instanciado o objeto `objPessoa` da classe `Pessoa`, através da instrução `new Pessoa()`, logo em seguida são realizadas ações através dos métodos *Setters* para definições dos valores dos atributos Nome e Cpf. Por fim, os valores são apresentados através da instrução `echo`.

Construtores

Os *métodos construtores* são aqueles métodos que são invocados sempre que uma instância de uma classe (um objeto) é criada. Um *método construtor* pode ser construído conforme a sintaxe mostrada abaixo:

```
<?php
class Pessoa
{
    public $Cpf;
    public $Nome;

    public function __construct()
    {
        echo "Chamndo o construtor de Pessoa...";
    }
}
```

O uso da palavra reservada `__construct` diz que se trata de um *método construtor*. Outra forma de definir um *método construtor* é criá-lo com o mesmo nome da classe, conforme segue exemplificação a seguir:

```
<?php
class Pessoa
{
    public $Cpf;
    public $Nome;

    public function Pessoa()
    {
        echo "Chamndo o construtor de Pessoa...";
    }
}
```

Propriedades

Variáveis membros das classes são chamadas de propriedades. Elas podem ser definidas utilizando as palavras reservadas `public` .

```
<?php
class Pessoa
{
    public $Cpf;
    public $Nome;
}
?>
```

No código definido acima, foi definida a classe Pessoa com as propriedades \$Cpf e \$Nome. Esta classe e respectivas propriedades podem ser utilizadas da seguinte forma:

```
<?php
$p2 = new Pessoa2();
$p2->Nome="João";
echo $p2->Nome;
?>
```

Herança

A Herança em Orientação a Objetos permita que uma determinada classe herde definições de outras classes. Para utilizar este conceito em PHP, deve ser utilizada a palavra reservada `extends`. A Herança Múltipla, que permite que uma classe herde definições de mais de uma classe, não é disponibilizada no PHP.

A seguir é apresentado um exemplo de uma classe `Funcionario` que herda as definições da classe `Pessoa`.

```
<?php
require_once ("Pessoa.php");

class Funcionario extends Pessoa
{
    public $CodFuncionario;
    public $Salario;

    function Funcionario(){
        parent::Pessoa();
        echo "Chamando o método construtor de Funcionario..";
    }

    public function mostrarDados(){
        echo "</br> Subclasse Funcionario </br> ";
        echo "Cpf: ".$this->Cpf;
        echo "</br>";
        echo "Cód. Funcionário: ".$this->CodFuncionario;
    }
}
?>
```

Na classe `Funcionario` foram definidas duas propriedades, `CodFuncionario` e `Salario`, além do método `mostrarDados` e o método construtor (`Funcionario`). O método `mostrarDados` imprime o *Cpf*, propriedade definida na classe base (superclasse) e o *Código do Funcionário*, que é uma propriedade da classe `Funcionario`. No método construtor é usada a instrução `parent::`, que é utilizada para acessar definições (por exemplo, métodos e propriedades) da classe base (superclasse). Neste exemplo específico, a instrução `parent::Pessoa()` invoca o método construtor da classe `Pessoa`,

A instrução `require_once ("Pessoa.php")` requer/importa as definições da classe `Pessoa`, que estão em um arquivo *"Pessoa.php"* para que possam ser utilizadas. A instrução `require_once` só importa o arquivo solicitado se este já não tiver sido importado anteriormente.

Agora, é mostrado um script PHP que faz uso das definições anteriores da classe `Pessoa` e `Funcionario` .

```
<?php

$f1 = new Funcionario();
$f1->Cpf = "000-000-000-00";
$f1->CodFuncionario="010101";
x$f1->mostrarDados();
?>
```

No código mostrado, foi instanciado um objeto da classe `Funcionário` , logo em seguida foi definido um valor para propriedade *Cpf*, que pode ser acessada porque a classe `Funcionario` herda todas as definições de `Pessoa` e, por fim, foi invocado o método `mostrarDados` , definido na classe `Funcionario` .

Formulários HTML e PHP

A criação de um formulário é realizada através das definições (marcações) da linguagem HTML. A seguir é apresentado um exemplo de criação de um formulário em HTML.

```
<html>
<body>
  <form name="form1">
    <input name="txtNome" type="text">
    <input name="btnEnviar" type="submit" value="Enviar">
  </form>
</body>
</html>
```

No código HTML apresentado anteriormente, foi criado um formulário com nome `form1` ; uma caixa de texto `txtNome` e um botão para envio dos dados de nome `btnEnviar` . O atributo `type` define o tipo do controle HTML que está sendo criado. Para a definição da caixa de texto, foi definido o valor `text` para o atributo `type` , enquanto que para um botão, responsável pelo envio dos dados, foi definido o valor `submit` .

Para que os dados digitados nos formulários possam ser enviados para um script PHP, são utilizados os métodos (verbos) **GET** e **POST**.

[Método GET](#)

[Método POST*](#)

[Validação de Formulários](#)

Método GET

O método GET cria um **array** de chaves e valores (*exemplo, chave1=>valor1, chave2=>valor2, ..., chaven=>valorn*), em que as chaves correspondem aos nomes dos controles dos formulários e os valores correspondem aos dados de entradas fornecidos pelo usuário.

O atributo `method` define o método de envio dos dados do formulário, conforme exemplificação a seguir:

```
<html>
<body>
  <form name="form1" method="GET">
    <input name="textNome" type="text">
    <input name="btnEnviar" type="submit" value="Enviar">
  </form>
</body>
</html>
```

Uma vez que o usuário informar algum valor no campo de texto do formulário e clicar no botão **Enviar**, um array de **chave=>valor** será criado e passado via URL. Este array é também chamado de **Query String**.

Supondo que o usuário digite **John** no campo de texto e clique no botão **Enviar**, o seguinte array será criado e enviado pela URL: **?textNome=John&btnEnviar=Enviar**. O carácter **?** separa a URL base do array com os valores. Cada par *chave=>valor* é separado pelo carácter **&**. Por exemplo, se página carregada fosse *index.php*, e estivesse dentro da pasta exemplo, então a URL base poderia ser <http://localhost/exemplo/index.php>.

Acessando os dados do formulário em PHP

Quando os valores são enviados via *método GET*, automaticamente é criada um array chamado `$_GET` que contém todos os valores passados para o corrente *script* via URL. O array definido em `$_GET` é global para o script corrente e pode ser acessado a qualquer momento, independente do escopo (e.g. em uma função ou em uma classe), sem necessidade de nenhum tratamento especial.

Vamos usar o código HTML definido anteriormente para acrescentar um script PHP que acessa os dados enviados via *método GET* e mostra-los no *browser*.

```
<html>
<body>
    <?php
        if (isset($_GET["textNome"])){
            $nome = $_GET["textNome"];
            echo "Nome: " . $nome;
        }
    ?>
    <form name="form1" method="GET">
        <input name="textNome" type="text">
        <input name="btnEnviar" type="submit" value="Enviar">
    </form>
</body>
</html>
```

Na primeira vez que a página acima for carregada, um erro PHP será gerado porque o array `$_GET` e seus valores ainda **não** existem. Uma vez que o formulário for preenchido pelo usuário e o botão *Enviar* for acionado, o *array* será criado, os dados serão mostrados no *browser* e os erros deixarão de existir.

Para resolvermos o problema do erro gerado no primeiro acesso, pode ser realizada uma verificação com o intuito de identificarmos se os valores de `$_GET` não estão nulos, conforme código mostrado a seguir:

```
<?php
    if (isset($_GET["textNome"])){
        $nome = $_GET["textNome"];
        echo "Nome: " . $nome;
    }
?>
```

No código mostrado anteriormente foi realizada uma verificação com o auxílio da função `isset` para identificarmos se a chave `textNome` existe. Mais informações sobre a função `isset` estão disponíveis no capítulo de [Funções](#).

As exemplificações mostradas anteriormente sempre enviam os dados para a página PHP corrente. Caso necessite enviar os dados para outra página (outro script PHP), pode ser utilizado o atributo `action` do formulário. Assim, deve ser atribuído ao atributo `action` o nome da página destino, por exemplo, `action="pagina_destino.php"`. A seguir você pode fazer [download](#) de um projeto PHP que mostra a utilização do envio de dados para uma página diferente utilizando um formulário com método GET.

DICA IMPORTANTE: Caso necessite enviar parâmetros via URL é possível criar esse *array* manualmente e colocá-lo como parte do endereço. Assim, os dados também podem ser enviados via *GET* diretamente pela URL.

```
<a href="mostrar-dados.php?Cod=2&Nome='John'"> Passando CÓDIGO e NOME via link </a>  
>
```

O código mostrado acima cria um link em HTML definindo o *array* de parâmetros que serão enviados para a página `mostrar-dados.php`. O projeto completo para que você possa testar está disponível para download [aqui](#).

Método POST

Similar ao método *GET*, o **método POST** também cria um array de chaves e valores (exemplo, chave1=>valor1, chave2=>valor2, ..., chaven=>valorn), em que as chaves correspondem aos nomes dos controles dos formulários e os valores correspondem aos dados de entradas fornecidos pelo usuário.

O atributo `method` define o método de envio dos dados do formulário, conforme exemplificação a seguir:

```
<html>
<body>
  <form name="form1" method="POST">
    <input name="textNome" type="text">
    <input name="btnEnviar" type="submit" value="Enviar">
  </form>
</body>
</html>
```

Uma vez que o usuário informar algum valor no campo de texto do formulário e clicar no botão *Enviar*, um *array* de pares chave=>valor será criado e passado via método *HTTP POST*. Diferentemente do método *GET*, o *array* não fica visível na URL.

Acessando os dados do formulário em PHP

Quando os valores são enviados via método *POST*, automaticamente é criada um *array* chamado `$_POST` que contém todos os valores preenchidos no formulário e que agora poderão ser acessados via código PHP. O *array* definido em `$_POST` é global para o script (página) corrente e pode ser acessado a qualquer momento, independente do escopo (e.g. em uma função ou em uma classe), sem necessidade de nenhum tratamento especial.

Vamos usar o código HTML definido anteriormente para acrescentar um script PHP que acessa os dados enviados via método *POST* e mostra-los no *browser*.

```
<html>
<body>
    <?php
        $nome = $_POST["textNome"];
        echo "Nome: " . $nome;
    ?>
    <form name="form1" method="POST">
        <input name="textNome" type="text">
        <input name="btnEnviar" type="submit" value="Enviar">
    </form>
</body>
</html>
```

Na primeira vez que a página acima for carregada, um erro será gerado porque o array `$_POST` e seus valores ainda não existem. Uma vez que o formulário for preenchido pelo usuário e o botão *Enviar* for acionado, o *array* será criado, o nome informado será mostrado no *browser* e o erro deixará de existir.

Para resolvermos o problema do erro gerado no primeiro acesso, pode ser realizada uma verificação com o intuito de identificarmos se os valores de `$_POST` não estão nulos. A seguir é apresentado o código PHP com esta verificação, que deve substituir o código PHP mostrado anteriormente:

```
<?php
    if (isset($_POST["textNome"])){
        $nome = $_POST["textNome"];
        echo "Nome: " . $nome;
    }
?>
```

No código mostrado anteriormente foi realizada uma verificação com o auxílio da função `isset` para identificarmos se a chave `textNome` existe. Mais informações sobre a função `isset` estão disponíveis no capítulo de [Funções](#).

As exemplificações mostradas anteriormente sempre enviam os dados para a página PHP corrente. Caso necessite enviar os dados para outra página (outro script PHP), pode ser utilizado o atributo `action` do formulário. Assim, deve ser atribuído ao atributo `action` o nome da página destino, por exemplo, `action="pagina_destino.php"`. A seguir você pode fazer [download](#) de um projeto PHP que mostra, na prática a utilização do envio de dados para uma página diferente.

Validação de Formulários

Neste capítulo, é apresentado o exemplo de criação de um formulário que contém diferentes campos de entrada: *campos de texto* (`text`), *radio button* (`radio`), *caixa de texto para textos maiores* (`textarea`), *botão para envio dos dados* (`submit`), e um para *limpar o formulário* (`reset`). Os campos do formulário e as respectivas especificações são mostrados na tabela a seguir:

Campo	Tipo	Requerido
Nome	text	Sim
CPF	text	Sim
E-mail	text	Sim
Site	text	Não
Observações	textarea	Não
Sexo	radio	Sim

Campos de texto

Os campos *Nome*, *Cpf*, *E-mail*, *Site* são todos campos de texto definidos através do atributo `type` com valor `text` (`type="text"`). O campo *Observações* também é um campo de texto, porém é um `textarea`, que segue uma declaração diferente. O campo `textarea` permite múltiplas linhas e colunas. A seguir o código HTML para definição dos respectivos campos é apresentado.

```
Nome: <input type="text" name="txtNome">
CPF: <input type="text" name="txtCpf">
E-mail: <input type="text" name="txtEmail">
Site: <input type="text" name="txtSite">
Observações: <textarea name="textAreaObservacoes" rows="7" cols="40"> </textarea>
```

Campo para seleção exclusiva (`radio`)

O campo para escolha do sexo oferece duas opções possíveis: *Masculino* e *Feminino*. Para isso, é utilizado um campo do tipo `radio`, conforme código apresentado a seguir:

```
<input type="radio" name="radioSexo" value="Masculino"> Masculino  
<input type="radio" name="radioSexo" value="Feminino"> Feminino
```

Se o usuário escolher a opção `Feminino`, a opção `Masculino` deverá estar desmarcada, e vice-versa. Para garantir esta escolha única, o nome de ambos os campos (`Feminino` e `Masculino`) devem ser o mesmo. Neste caso específico, o nome definido foi `radioSexo`.

Botões para ações no formulário

A qualquer momento, o usuário pode limpar o formulário ou enviar os dados para serem processados. Para isso, existem específicos botões: `reset` (limpar) e `submit` (submete). A seguir é apresentado o código HTML que os define:

```
<input type="submit" name="btnEnviar" value="Enviar">  
<input type="reset" name="resetBtn" value="Limpar">
```

Código PHP para validação

Para que os dados possam ser validados, inicialmente deve-se definir todos os elementos apresentados anteriormente como elementos de um formulário. O método de envio dos dados no formulário deve ser `POST`. Depois disso, podemos passar para a etapa de codificação do código PHP que receberá os dados para verificação. A verificação que será realizada é a obrigatoriedade dos campos: *Nome*, *Cpf*, *E-mail* e *Sexo*. A seguir é apresentado um exemplo de codificação:

```
<?php  
$nomeError=$cpfError=$emailError= $sexoError="";  
if ($_SERVER["REQUEST_METHOD"]=="POST"){  
    if (empty($_POST["txtNome"])){  
        $nomeError = "Nome é obrigatório!";  
    }  
    if (empty($_POST["txtCpf"]))  
        $cpfError = "CPF é obrigatório!";  
    if (empty($_POST["txtEmail"]))  
        $emailError = "O e-mail é obrigatório!";  
    if (empty($_POST["radioSexo"]))  
        $sexoError = "O sexo é obrigatório!";  
}
```

Dentre as instruções apresentadas no código anterior, destaca-se o condicional que verifica se o método de envio dos dados é POST. Caso seja, os passos subsequentes fazem verificações se os campos estão ou não vazios. Caso algum campo esteja vazio, uma variável específica recebe a mensagem de erro que será mostrada para o usuário.

O código PHP completo com o funcionamento do exemplo mostrado anteriormente, pode ser encontrado [aqui](#).

Sessão

Sessão permite que dados sejam armazenados em variáveis que possam ser usadas em múltiplas páginas. Assim, é possível armazenar informações dos perfis dos usuários que podem ser utilizadas em diferentes páginas (e.g. username, hora do login, cor favorita, etc).

Para iniciar uma sessão, basta usar a função `session_start()`. Uma vez iniciada a sessão, é possível adicionar variáveis ao array global `$_SESSION`. Este *array* que contém todas as variáveis definidas na sessão de um usuário.

Para exemplificar a utilização da sessão, desenvolveremos um projeto simples para exemplificar uma situação de login em um sistema. Por isso, inicialmente será criada uma página `frmLogin.php`, que conterá um formulário com campos para que o usuário informe o login e senha, de acordo com trecho de código mostrado a seguir:

```
<form name="frmLogin" method="POST" action="login.php">
  <input type="text" name="textLogin"> </br>
  <input type="password" name="textSenha" value=""> </br>
  <input type="submit" name="btnLogin" value="Login" >
</form>
```

O atributo `action` do formulário tem o valor `login.php`, que indica que os dados do presentes no formulário serão enviados para a referida página (mais informações sobre o assunto no capítulo sobre [Formulários](#)). Na página `login.php`, coloque o código que segue:

```
<?php
if ($_POST["textLogin"]=="usuario" && $_POST["textSenha"]=="senhaacesso" ){
    session_start();
    date_default_timezone_set("America/Sao_Paulo");
    $_SESSION["user_name"] = $_POST["textLogin"];
    $_SESSION["login_time"] = date("d-m-Y h:i:sa");
    header("Location: profile.php");
}else{
    header('Location: frmLogin.php?erroLogin=1');
}
?>
```

No código apresentado anteriormente, inicialmente é realizada uma verificação se os dados de autenticação fornecidos pelo usuário são `usuario` e `senhaacesso`. Se sim, a sessão é iniciada e o nome de usuário e a data e hora de acesso são armazenados na sessão. O *login de usuário* é associado à chave `user_name`, já a *hora de login* é associada à chave

`login_time` . Depois o usuário é direcionado para uma nova página, `profile.php` , através da função `header` . Caso os dados fornecidos pelo usuário não sejam conforme os esperados, este é direcionado para a página de login (`frmLogin.php`) passando pela QueryString um parâmetro `erroLogin` com valor `1` . Este parâmetro é importante para podermos verificar a podermos mostrar uma mensagem de erro ao usuário informando que os dados fornecidos estão incorretos.

Na página `profile.php` os dados previamente definidos na sessão são acessados e mostrados no browser, conforme código apresentado a seguir:

```
<?php
session_start();

$username = $_SESSION["user_name"];
$hora_login = $_SESSION["login_time"];

echo "Hora login: ".$hora_login."</br>";
echo "Username: ".$username."</br>";

echo "</br> </br> <a href='frmLogin.php'> Página de Login </a>"
?>
```

A cada nova página que se deseja acessar ou definir uma variável na sessão, inicialmente é necessário iniciá-la, através da função `session_start()` . A última linha do código supracitado redireciona o usuário para a página do formulário de login.

O projeto completo para download pode ser baixado [aqui](#).

Para baixar outro projeto exemplo de manipulação de sessão, clique [aqui](#).

Acesso a Banco de dados via PDO (*PHP Data Objects*)

PHP Data Object (PDO) é uma extensão do PHP para acesso a banco de dados, isto é, dispõe de uma interface que simplifica a conexão com banco de dados e a realização de operações sobre ele. PDO está disponível a partir da versão 5 do PHP.

- [Conexão com o banco de dados](#)
- [Inserção de dados](#)
- [Seleção de dados](#)
- [Alteração de dados](#)
- [Exclusão de dados](#)

Alguns exemplos de projetos com uso de PDO são mostrados [aqui](#);

Conexão via PDO

Para estabelecer uma conexão com um banco de dados utilizando PDO é necessário definir a string de conexão e passá-la para uma instância PDO, conforme exemplificação a seguir:

```
<?php
$servername = "localhost";
$username = "usuario";
$password = "senha";
$db = "nome-do-banco";

try {
    $conn = new PDO("mysql:host=$servername;dbname=$db", $username, $password);
    // set o modo de erro do PDO para gerar exceções
    set the PDO error mode to exception
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    echo "Conexão realizada com sucesso!";
}
catch(PDOException $e)
{
    echo "Conexão falhou!: " . $e->getMessage();
}
?>
```

Na definição da string de conexão com o referido banco de dados, deve-se definir inicialmente o tipo do banco (neste caso, MySQL), os dados para autenticação no banco de dados (usuário e senha), além do banco de dados ao qual deseja-se estabelecer conexão (neste caso “nome-do-banco”). No script apresentado anteriormente, também foi definido o modo de erro do PDO para gerações de exceções sempre que quaisquer erros ocorrerem com as execuções das consultas no banco de dados.

Sempre que um script for finalizado, a conexão é automaticamente finalizada. Caso necessite finalizar a conexão antes, basta utilizar a seguinte instrução:

```
$conn = null;
```

Inserção de dados

Uma vez que o banco de dados tenha sido criado, é possível fazer operações sobre ele. A primeira operação que será apresentada aqui é a de inserção. Existem diferentes alternativas para realizar tal procedimento via PDO. Serão mostradas aqui apenas duas alternativas possíveis:

Inserindo Dados: Alternativa I

Uma das maneiras de realizar as operações no banco de dados é diretamente através de instruções SQL (*Structured Query Language*). Assim, define-se uma string contendo a instrução SQL com os respectivos parâmetros e logo em seguida executa-se esta instrução.

Para realizar inserção via PDO com instrução SQL, a seguinte sintaxe pode ser utilizada:

```
$conn = //abrir conexão com o banco de dados
$sql = //definir a instrução SQL utilizando a cláusula INSERT
$conn->exec($sql);
```

De acordo com o código apresentado anteriormente, primeiramente estabelece-se uma conexão com o banco de dados, logo em seguida pode ser definida uma instrução SQL que irá inserir um conjunto de dados em uma específica tabela e, por fim, basta executar a referida consulta a partir do método `exec`. O método `exec` retorna o número de linhas afetadas com a instrução SQL. Com o método `exec` é possível também executar instruções de **UPDATE** e **DELETE**, mas **não SELECT**(recuperação de dados - seção [Seleção de dados](#)).

Inserindo Dados: Alternativa II

Outra alternativa para realizar operações de inserções é utilizar uma instância de `PDOStatement`, que permite a criação de sentenças que possibilita a mesma (ou uma similar) instrução SQL ser executada várias vezes.

Com o uso de um objeto `PDOStatement`, uma instrução SQL pode ter zero ou mais parâmetros que podem ser definidos de duas maneira distintas: através do seu próprio nome `":nome_do_parâmetro"` ou através do caracter `"?"`. Os valores reais dos parâmetros são substituídos quando a instrução SQL é executada.

Sintaxe com parâmetros nomeados

```
$conn = //abrir conexão com o banco de dados
$sql = "INSERT INTO `nome_da_tabela`(`coluna1`, `coluna2`)
      VALUES (:coluna1, :coluna2)";
$stmt = $conn->prepare($sql);
$valor1=// valor da coluna 1
$valor2=// valor da coluna 2
$stmt->bindParam(":coluna1",$valor1);
$stmt->bindParam(":coluna2",$valor2);
$stmt->execute();
```

Sintaxe com parâmetros definidos pela "?"

```
$conn = //abrir conexão com o banco de dados
$sql = "INSERT INTO `nome_da_tabela`(`coluna1`, `coluna2`)
      VALUES (?, ?)";
$stmt = $conn->prepare($sql);
$valor1=// valor da coluna 1
$valor2=// valor da coluna 2
$stmt->bindParam(1,$valor1);
$stmt->bindParam(2,$valor2);
$stmt->execute();
```

Executando várias instruções com um mesmo objeto PDOStatement

Conforme mencionado anteriormente, um mesma instância de um objeto `PDOStatement` permite executar várias instruções SQL, conforme mostrado a seguir:

```
$conn = //abrir conexão com o banco de dados
$sql = "INSERT INTO `nome_da_tabela`(`coluna1`)
      VALUES (:coluna1)";
$stmt= $conn->prepare($sql);
$stmt->bindParam(":coluna1",$coluna1);

$coluna1=4;
$stmt->execute();

$coluna1=5;
$stmt->execute();
```

No código apresentado acima, é possível observar que uma mesma instrução SQL foi executada duas vezes, através da chamada ao método `execute`. Em cada execução, foi passado por parâmetro um valor diferente para o campo `":coluna1"`, `4` e `5`. O método

`bindParam` , que define o valor para o respectivo parâmetro, recebe a referência da variável, por isso que é possível apenas alterar o valor da variável passada por parâmetro, neste caso `$coluna1` e executar novamente a instrução SQL, através do método `execute()` do objeto `PDOStatement` .

Seleção de dados

A seleção de dados também pode ser realizada a partir da linguagem SQL e PDO. A seguir é apresentado um exemplo de sintaxe que pode ser utilizado.

```
$conn = //abrir conexão com o banco de dados  
$sql = //definir a instrução SQL utilizando a cláusula SELECT  
$objPDO = $conn->query($sql);
```

De maneira similar a inserção, inicialmente deve-se estabelecer uma conexão com o específico banco de dados, a posteriori deve ser definida a instrução SQL, e logo em seguida executar o método `query` que retorna um objeto `PDOStatement`.

Para ter acesso aos valores retornados no objeto `PDOStatement`, a seguinte sintaxe pode ser utilizada:

```
foreach ($objPDOStatement as $linha):  
    $linha["nome_da_coluna"]
```

Pode-se percorrer os valores do objeto `PDOStatement` linha-a-linha e acessar os valores de cada coluna através de uma chave, que é equivalente ao nome da coluna definido no banco de dados.

Alteração de dados

A sintaxe para alteração de dados é similar as outras operações. A diferença está no uso da cláusula **UPDATE** da linguagem SQL.

```
$conn = //abrir conexão com o banco de dados  
$sql = //definir a instrução SQL utilizando UPDATE  
$conn->exec($sql)
```

Exclusão de dados

A sintaxe para exclusão de dados é similar as outras operações. A diferença está no uso da cláusula **DELETE** da linguagem SQL.

```
$conn = //abrir conexão com o banco de dados  
$sql = //definir a instrução SQL utilizando DELETE  
$conn->exec($sql)
```

Exemplos de projetos com PDO

Esta seção trará alguns exemplos de projetos com o PDO, tais como, projetos que mostra na prática a conexão com um respectivo banco de dados, operações de inserção, consulta, exclusão e alteração.

- [PROJETO 1: Conexão com o banco de dados](#)
- [PROJETO 2: operações DAO \(`Data Access Object` \) com 1 tabela](#)

PROJETO 1: Conexão com um banco de dados

Para que possa ser possível realizar a conexão com um determinado banco de dados, inicialmente faz-se necessária a sua criação. Portanto, **crie um banco de dados** intitulado `"db-facul"`, logo em seguida, **crie uma classe** `Connection`, com um método `open`, que terá a implementação do código que realiza a conexão com o respectivo banco. O código é mostrado a seguir:

```
<?php
define("SERVERNAME", "localhost");
define ("DATABASE_NAME", "db-facul");
define ("USERNAME", "root");
define ("PASSWORD", "");

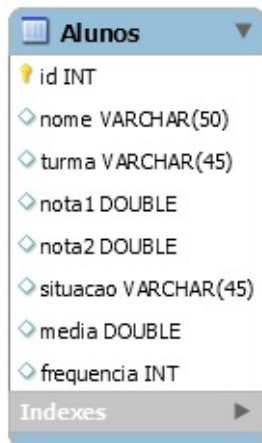
class Connection
{
    public static function Open(){
        try {
            $conn = new PDO("mysql:host=".SERVERNAME.";dbname=".DATABASE_NAME, USERNAME, PASSWORD);
            // set the PDO error mode to exception
            $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
            return $conn;
        }
        catch(PDOException $e)
        {
            throw new Exception($e->getMessage());
        }
    }
}
```

Com a definição desta classe em seu projeto, sempre que necessário é possível utilizá-la para estabelecer conexão com banco de dados.

A implementação deste projeto, inclusive, com o projeto de banco de dados desenvolvimento na ferramenta MYSQL WorkBench pode ser encontrado [aqui](#).

PROJETO 2: operações DAO (Data Access Object) com 1 tabela

Passo 1: No banco `db-facul`, crie a tabela `Aluno`, conforme figura apresentada a seguir:



Passo 2: agora crie uma *pasta* para o seu projeto, dentro dela, crie uma pasta `classes` e dentro dela uma classe `Connection`, com um método estático `open`, que retorna a conexão com o banco **db-facul** (idem ao **Projeto 1** mostrado anteriormente).

Passo 3: Crie uma classe `Aluno`, dentro da pasta `classes`, e defina todas as **propriedades** de aluno, conforme a tabela **Alunos** previamente definida. Além disso, defina também um **construtor** para a classe.

```
<?php

class Aluno
{
    public $Id;
    public $Nome;
    public $Turma;
    public $Nota1;
    public $Nota2;
    public $Media;
    public $Frequencia;
    public $Situacao;

    // faça também o construtor
}
```

Passo 4: na pasta `classes`, crie a classe `AlunoDAO`. Esta classe conterá todas as operações de acesso a dados, as operações **CRUD (Create-Retrieve-Update-Delete)**

Operações *Create*

Passo 5: agora, considerando os conceitos que foram apresentados na seção [Acesso a Banco de dados via PDO](#), pode ser realizada a implementação do método **add(Create)**, que recebe por parâmetro um objeto Aluno e o adiciona no banco de dados.

Exemplo 1 para o método `add`

```
public static function add($aluno){
    $conn = Connection::Open();
    $sql = "INSERT INTO `alunos`(`turma`, `nome`, `nota1`,
        `nota2`, `situacao`, `media`, `frequencia`)
        VALUES ('$aluno->Turma', '$aluno->Nome',
            $aluno->Nota1, $aluno->Nota2, '$aluno->Situacao',
            $aluno->Media, $aluno->Frequencia)";
    $conn->exec($sql);
    return $conn->lastInsertId();
}
```

Exemplo 2 para o método `add`

```
public static function add2($aluno){
    $conn = Connection::Open();
    $sql = "INSERT INTO `alunos`(`turma`, `nome`, `nota1`,
        `nota2`, `situacao`, `media`, `frequencia`)
        VALUES (:turma, :nome, :nota1,
            :nota2, :situacao, :media, :frequencia)";
    $stmt = $conn->prepare($sql);
    $stmt->bindParam(":turma", $aluno->Turma);
    $stmt->bindParam(":nome", $aluno->Nome);
    $stmt->bindParam(":nota1", $aluno->Nota1);
    $stmt->bindParam(":nota2", $aluno->Nota2);
    $stmt->bindParam(":situacao", $aluno->Situacao);
    $stmt->bindParam(":media", $aluno->Media);
    $stmt->bindParam(":frequencia", $aluno->Frequencia);
    $stmt->execute();
}
```

Operações *Retrieve*

Passo 6: para exemplificar as operações *Retrieve*, na classe `AlunoDAO` devem ser implementados **dois** métodos: o `getById`, que recebe um `id` por parâmetro e retorna um objeto do tipo `Aluno`; e o `getAll`, que retorna todos os alunos cadastrados (um `array` de alunos).

Exemplo 1 para o método `getById`

```
public static function getById($idAluno){
    $conn = Connection::Open();
    $sql = "SELECT *FROM Alunos Where id=".$idAluno;
    $objPDOStatement = $conn->query($sql);
    return AlunoDAO::retorno($objPDOStatement)[0];
}
```

Após executar a consulta SQL, através do método `query`, é retornado um objeto `PDOStatement`. O objeto `PDOStatement` deve ser **mapeado** para um objeto `Aluno`. Este mapeamento é realizado através do método estático e privado nomeado `retorno`. A implementação deste método é mostrada a seguir:

```
private function retorno($objResult){
    $alunos = array();
    if ($objResult instanceof PDOStatement)
    {
        if ($objResult->rowCount()==0)
            return null;
        else{
            foreach ($objResult as $linha) {
                $aluno = AlunoDAO::fillObject($linha);
                array_push($alunos, $aluno);
            }
        }
    }else
        if (is_array($objResult)){
            for ($i=0; $i< count($objResult); $i++){
                $aluno = AlunoDAO::fillObject($objResult[$i]);
                array_push($alunos, $aluno);
            }
        }
    return $alunos;
}
```

O método `retorno` está preparado também para o caso do parâmetro passado para ela ser um `array`, que deverá ser percorrido e **mapeado para um array de objetos** `Aluno`.

O método `fillObject` recebe por parâmetro os valores correspondentes ao retorno de uma determinada consulta SQL e mapeia para um objeto `Aluno`, conforme código apresentado a seguir:

```

private static function fillObject($obj){
    $aluno = new Aluno();
    $aluno->Id = $obj["id"];
    $aluno->Nome=$obj["nome"];
    $aluno->Turma=$obj["turma"];
    $aluno->Nota1=$obj["nota1"];
    $aluno->Nota2=$obj["nota2"];
    $aluno->Media=$obj["media"];
    $aluno->Frequencia=$obj["frequencia"];
    $aluno->Situacao=$obj["situacao"];
    return $aluno;
}

```

Exemplo 2 para o método getById

```

public static function getById($idAluno){
    $conn = Connection::Open();
    $sql = "SELECT *FROM Alunos Where id= ?";
    $stmt= $conn->prepare($sql);
    $stmt->bindParam(1, $idAluno);
    $stmt->execute();
    $result = $stmt->fetchAll();
    return AlunoDAO::retorno($result)[0];
}

```

Exemplo 3 para o método getById

O uso desta sintaxe permite que o mapeamento do retorno para um objeto de uma determinada classe possa ser realizado a partir do método `fetchAll`, através `PDO::FETCH_CLASS`. O uso deste argumento para o método `fetchAll` mapeia o resultado para um *array* de instâncias de uma classe, nomeando cada uma das propriedades da classe de acordo com os nomes das colunas do resultado (nomes das colunas da tabela definida no banco de dados). O uso desta sintaxe evitaria o uso do método retorno mostrado anteriormente.

```

public static function getById($idAluno){
    $conn = Connection::Open();
    $sql = "SELECT *FROM Alunos Where id= ?";
    $stmt= $conn->prepare($sql);
    $stmt->bindParam(1, $idAluno);
    $stmt->execute();
    $retorno = $stmt->fetchAll(PDO::FETCH_CLASS, "Aluno");
    return $retorno[0];
}

```

De acordo com o código acima, a constante `PDO::FETCH_CLASS` e o nome da classe `Aluno` foram passados por parâmetro para o método `fetchAll`, assim o resultado após a execução da instrução SQL será mapeado para um objeto `Aluno`.

Operações *Update*

Passo 7: para exemplificar a operação de *Update*, na classe `AlunoDAO` deve ser implementado um método `update`, que recebe um *objeto aluno* por parâmetro, localiza-o no banco a partir do seu identificador (`id`), e só então é realizada a operação de alteração.

```
public static function update ($aluno){
    $conn = Connection::Open();
    $sql = "UPDATE `alunos` SET `nome`='$aluno->Nome', `turma`='$aluno->Turma',
        `nota1`='$aluno->Nota1', `nota2`='$aluno->Nota2', `situacao`='$aluno->Situacao',
        `media`='$aluno->Media', `frequencia`='$aluno->Frequencia' WHERE `id`='$aluno->Id";
    return $conn->exec($sql);
}
```

Operações *Delete*

Passo 8: por fim, para exemplificar a operação de *Delete*, na classe `AlunoDAO` deve ser implementado um método `delete`, que recebe por parâmetro o `id` do aluno a ser removido.

```
public static function delete($idAluno){
    $conn = Connection::Open();
    $sql = "DELETE FROM `alunos` WHERE id=".$idAluno;
    return $conn->exec($sql);
}
```

Para fazer download do projeto completo, clique [aqui](#).

Backend

O **Backend** representa outro componente importante do modelo de desenvolvimento para web que está sendo adotado neste livro. Como já informado, a comunicação ocorre entre **Frontend** e **Backend**. O primeiro, utilizando tecnologias como HTML, CSS, JavaScript e AngularJS. O segundo, utilizando tecnologias como PHP e Silex.

Esta seção apresenta o conteúdo sobre backend necessário para a criação da lógica do lado do servidor, utilizando um modelo de **API REST**. Para isso, será utilizada uma combinação de PHP+Silex.

API REST

Silex

O conteúdo desta seção é baseado na [documentação oficial do Silex](#).

Silex é um microframework para PHP. É construído com base em Symfony2 e Pimple, além de ser inspirado pelo Sinatra.

A instalação do Silex pode ser feita utilizando Composer, bastando executar o comando a seguir no prompt de comando, estando na pasta do projeto PHP em questão:

```
composer require silex/silex
```

Estrutura básica

A estrutura básica de um programa com Silex é a seguinte:

```
require_once 'vendor/autoload.php';  
$app = new Silex\Application();  
$app->run();
```

O código importa o arquivo `vendor/autoload.php` (do composer), cria uma instância de `Silex\Application` e executa o método `run()`. Esta é a estrutura padrão. Entre a instanciação de `Silex\Application` e a chamada do método `run()` são definidas as rotas

e os controllers que representam a primeira forma de tratar requisições e representar lógica de negócio.

Em desenvolvimento, pode ser interessante visualizar mensagens de erro de processamento do Silex de forma mais detalhada. Para isso, utilize o seguinte:

```
$app['debug'] = true;
```

Roteamento

Silex, como um framework REST, é baseado em rotas. Por meio de uma rota, pode-se definir o controller que será chamado quando a rota for encontrada. Em alto nível, a sintaxe é a seguinte:

```
$app-><metodo>(<rota>, <controller>);
```

Onde:

- `<metodo>` : representa o método HTTP associado à rota;
- `<rota>` : representa o padrão da rota em questão (uma string);
- `<controller>` : representa a função que trata a rota, o que é chamado de *controller*.

Exemplo:

```
$app->get('/home', function() {  
    return '<h1>Página inicial</h1>';  
});
```

O código define:

- a rota utiliza o verbo **GET** do HTTP;
- a rota é baseada no padrão `/home` e informa o controller (a função anônima, neste caso), que trata a rota.

O Silex trata as rotas registradas no momento em que recebe uma requisição. Para ver o resultado da rota definida no exemplo anterior, é necessário acessar, no browser, a URL

`/home` (exemplo: `http://localhost/home`).

Configuração do servidor web

Por padrão, o Silex trata rotas a partir do arquivo `index.php` . Por exemplo, uma requisição para `http://localhost/index.php/produtos/1` é tratada pela rota `/produtos/{id}` .

Entretanto, pode ser interessante fazer com que as URLs sejam melhor apresentáveis e limpas. Por exemplo, ao invés de a URL ser `http://localhost/index.php/produtos/1` poderia ser `http://localhost/produtos/1` .

Para fazer isso, considerando o servidor web **Apache**, pode ser utilizado o `mod_rewrite` . Crie o arquivo `.htaccess` no diretório raiz do aplicativo e informe o seguinte conteúdo:

```
<IfModule mod_rewrite.c>
    Options -MultiViews

    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [QSA,L]
</IfModule>
```

Se o aplicativo não estiver no diretório raiz do servidor web, adicione a linha a seguir logo após `Options ... :`

```
RewriteBase /caminho/para/o/aplicativo
```

Por exemplo, se o diretório raiz do servidor web for `c:\xampp\htdocs` e o diretório raiz do aplicativo for `c:\xampp\htdocs\aplicativo` então a linha que usa `RewriteBase` deve ser:

```
RewriteBase /aplicativo
```

Roteamento

Esta seção apresenta mais conteúdo sobre a maneira de tratar rotas e definir controllers.

Roteamento dinâmico

O roteamento dinâmico é um recurso que permite definir *parâmetros de rota*. O exemplo a seguir demonstra como criar uma rota para acessar um post de um blog com base no seu identificador:

```
$app->get('/home/{id}', function($id) use ($app) {  
    if (!existePost($id)) {  
        $app->abort(404, 'Post não encontrado');  
    }  
    $post = consultarPost($id);  
    return '<h1>' . $post['titulo'] . '</h1>';  
});
```

O parâmetro de rota é definido entre chaves. Neste caso a rota é `/home/{id}` e o parâmetro é `id`, o que significa que a URL deve ser chamada como `/home/1`. Perceba que o parâmetro da rota torna-se disponível como parâmetro do controller, em outras palavras, o Silex *injeta* o parâmetro de rota `id` no controller.

A utilização de parâmetros de rota é a primeira forma de passar informações para um controller.

Convertendo Parâmetros de Rota

Pode ser necessário tratar o valor de um parâmetro de rota antes de injetá-lo no controller. Para fazer isso, utilize o método `convert()`. Exemplo:

```
$app->get('/usuarios/{id}', function($id) {  
    // ... procurar o usuário pelo id  
})->convert('id', function($id) {  
    return (int) $id;  
});
```

O primeiro parâmetro do método `convert()` é o nome do parâmetro de rota, neste caso, `id`. O segundo parâmetro é uma função que aceita como parâmetro o `$id`. Neste caso, a função anônima recebe o parâmetro `id` e retorna seu valor representado como número inteiro.

Restrições

Outra forma de tratar valores de parâmetros de rota é por meio de restrições. Para isso, é utilizado o método `assert()`. Exemplo:

```
$app->get('/usuarios/{id}', function($id) {  
    // ... procurar o usuário pelo id  
})->assert('id', '\d+');
```

A restrição representada por `assert('id', '\d+')` indica que o parâmetro `id` precisa combinar com a expressão regular `\d+`, ou seja, precisa ser um número de um ou mais dígitos.

Request

Além de ser possível informar dados para o controller por meio de parâmetros de rota, é possível utilizar a classe `Request` para acessar variáveis informadas via GET ou POST, por exemplo. A classe `Request` pertence ao **Symfony**, pacote requerido pelo Silex, e está disponível em `Symfony\Component\HttpFoundation\Request`. Algumas das principais propriedades são:

- `$attributes` : retorna parâmetros personalizados
- `$request` : retorna parâmetros de requisições POST (objeto `$_POST` do PHP)
- `$query` : retorna parâmetros de URL (objeto `$_GET` do PHP)
- `$server` : retorna informações do servidor web (objeto `$_SERVER` do PHP)
- `$files` : retorna arquivos enviados via upload (objeto `$_FILES` do PHP)
- `$cookies` : retorna variáveis armazenadas em cookies (objeto `$_COOKIE` do PHP)
- `$headers` : retorna cabeçalhos da requisição (extraídos de `$_SERVER`)

As propriedades `$attributes`, `$request`, `$query` e `$cookies` são do tipo `ParameterBag`, cujos principais métodos são apresentados a seguir.

Tipo de retorno	Método	Descrição
array	<code>all()</code>	Retorna os parâmetros (como <code>array</code>)
array	<code>keys()</code>	Retorna as chaves (nomes dos parâmetros)
misto	<code>get(\$chave, \$default, \$deep)</code>	Retorna um parâmetro pelo nome (chave) com valor padrão igual a <code>\$default</code> (padrão é <code>null</code>)
bool	<code>has(\$chave)</code>	Retorna <code>true</code> se o parâmetro estiver definido
int	<code>count()</code>	Retorna a quantidade de parâmetros

Além destes, há outros métodos que retornam valores convertendo-os para tipos específicos:

- `getInt()` : retorna o parâmetro com valor convertido para inteiro
- `getBoolean()` : retorna o parâmetro com valor convertido para booleano

As seções a seguir apresentam a utilização destas propriedades.

Parâmetros de URL

Parâmetros de URL estão presentes na URL, seguindo o formato: `nome=valor` (separados por `&`, no caso de mais de um parâmetro). Por exemplo, a URL

`http://localhost/pessoas/?s=Jose&i=1&o=nome` possui os seguintes parâmetros:

- `s` com valor `Jose`
- `i` com valor `1`
- `o` com valor `nome`

Para ter acesso a estes parâmetros de URL deve ser utilizada a propriedade `$query`, como mostra o exemplo:

```
require_once 'vendor/autoload.php';

use Silex\Application;
use Symfony\Component\HttpFoundation\Request;

$app = new Silex\Application();

$app->get('/', function(Application $app, Request $request) {
    return $request->query->get('id');
});

$app->run();
```

Para usar as classes `Silex\Application` e `Symfony\Component\HttpFoundation\Request`, é necessário incluí-las no código. Isso é feito por meio da instrução `use`, utilizada nas linhas 3 e 4.

No controller, estão sendo injetados dois objetos: `$app` (do tipo `Application`) e `$request` (do tipo `Request`). Para acessar o parâmetro de URL `id` é utilizada a propriedade `$query` e o método `get()`.

Para acessar o aplicativo, deve ser utilizada a URL `http://localhost/?id=1`. Assim, o controller acessa o parâmetro `id`, com valor `1`.

Integração com Angular

O capítulo sobre Angular demonstrou que o serviço `$http` está disponível e permite realizar requisições HTTP. A primeira forma de integração entre o aplicativo Angular e o Silex é por meio da utilização de parâmetros de URL. Para isso, o aplicativo precisa do frontend e do backend.

Backend

Do lado do backend, o aplicativo contém um controller que trata a rota `/`. Este controller espera por dois números via dois parâmetros de URL: `numero1` e `numero2`:

```
$app->get('/', function(Application $app, Request $request) {  
    $numero1 = $request->query->get('numero1', 0);  
    $numero2 = $request->query->get('numero2', 0);  
    return ($numero1 + $numero2);  
});
```

Como os dados serão transmitidos via GET, a rota `/` é tratada via este protocolo. O código acessa os parâmetros de URL por meio de `$request->query->get()`, calcula a soma e retorna o resultado.

Frontend

Do lado do frontend, o código de exemplo desta seção está nos arquivos `index.html` e `app.js`. No arquivo `index.html` há o trecho:

```
<body ng-controller="Home">  
    <p>Número A: <input type="text" ng-model="numero1"></p>  
    <p>Número B: <input type="text" ng-model="numero2"></p>  
    <p><button ng-click="somar()">Somar</button>  
    <p>Resultado da soma: {{resultado}}</p>  
</body>
```

O trecho de código refere-se à view do controller `Home`.

O código a seguir apresenta o arquivo `app.js`:

```
angular.module('silex-request-get', [])  
.controller('Home', function($scope, $http){  
    $scope.somar = function() {  
        $http.get('/livro-web-codigo-fonte/backend/request-angular-get/', {  
            params: {numero1: $scope.numero1, numero2: $scope.numero2}  
        }).then(function(response){  
            $scope.resultado = response.data;  
        });  
    };  
});
```

No controller `Home` o serviço `$http` está sendo utilizado para criar uma requisição GET para a URL `/livro-web-codigo-fonte/backend/request-angular-get/` e informa os parâmetros que representam os dois números a serem enviados para o backend por meio do segundo parâmetro: um objeto que possui o atributo `params` que, efetivamente, define quais são os parâmetros enviados para o backend. Aqui, perceba que o valor de `params` é um objeto

com dois atributos `numero1` e `numero2`. Não por acaso, os nomes destes atributos referem-se aos nomes dos parâmetros de URL esperados no backend. Os valores dos atributos estão vinculados ao model correspondente.

Uma vez que a requisição GET tenha ocorrido com sucesso, a variável `$scope.resultado` recebe o valor de `response.data`. O objeto `response`, neste caso, parâmetro da função anônima que trata o sucesso da requisição GET, contém informações sobre a resposta enviada pelo backend. O atributo `data` contém o conteúdo da resposta.

Requisições POST

O que se pode perceber com as requisições GET é que os dados (variáveis ou parâmetros de URL) ficam visíveis na URL. Entretanto, pode ser que isso não seja a maneira mais adequada. Uma vez que estes dados ficam visíveis inclusive em logs de acesso, dados com requisitos de segurança não podem ser transmitidos desta forma. Assim, a segunda forma, requisições POST, fornece uma maneira de enviar dados diretamente no cabeçalho da requisição HTTP.

Para ter acesso aos dados via deve ser utilizada a propriedade `$request` (do objeto `$request`), como mostra o exemplo:

```
require_once 'vendor/autoload.php';

use Silex\Application;
use Symfony\Component\HttpFoundation\Request;

$app = new Silex\Application();

$app->post('/', function(Application $app, Request $request) {
    return $request->request->get('id');
});

$app->run();
```

Enquanto os parâmetros de URL estão presentes na URL da requisição, isso não acontece com os dados enviados via POST. Assim, não é possível (ou não tão fácil) criar requisições POST diretamente no navegador. Por isso, esta seção apresenta duas formas de fazer isso. A primeira delas, apresentada a seguir, usa o mesmo formato de requisição integrada com o Angular. A segunda, utiliza uma extensão para o Google Chrome, que pode ser usada para qualquer tipo de requisição, inclusive, GET ou POST.

Integração com Angular

Backend

A integração com o Angular, no caso de requisições POST, implica em um pequeno problema: o formato como os dados estão codificados no cabeçalho da requisição HTTP. Por padrão, o Silex espera que as requisições estejam codificadas com `application/x-www-form-urlencoded`, mas o Angular as codifica com `application/json` (uma vez que os dados são sempre tratados em formato JSON). Por causa disso, é necessário tratar a requisição, convertendo os dados para o formato adequado do Silex.

Para facilitar este trabalho, uma classe chamada `AngularPostRequestServiceProvider` é utilizada como um serviço do aplicativo Silex, de modo a fazer as conversões necessárias entre o formato do Angular e o formato do Silex.

O código do backend é o seguinte:

```
require_once 'vendor/autoload.php';

use Silex\Application;
use Symfony\Component\HttpFoundation\Request;
use Lpweb\AngularPostRequestServiceProvider;

$app = new Silex\Application();
$app['debug'] = true;
$app->register(new AngularPostRequestServiceProvider());

$app->post('/', function(Application $app, Request $request) {
    $data = $request->request->get('data');
    $numero1 = $data['numero1'];
    $numero2 = $data['numero2'];
    return ($numero1 + $numero2);
});

$app->run();
```

Perceba que é utilizada a coleção `$request` (do objeto `$request`) e o nome da chave é `data`. Esta é a chave utilizada pelo Angular para enviar os dados para o backend. Assim, a primeira linha do controller acessa a chave `data` e as linhas seguintes acessam os valores `numero1` e `numero2` já como elementos do vetor `data`. O restante do código não muda em relação ao backend para as requisições GET.

Frontend

O Frontend muda pouco em relação às requisições GET: ao invés de utilizar a função `get()` do serviço `$http`, é usada a função `post()`; ao invés de utilizar o atributo `params` do objeto de configuração, usa-se o atributo `data`:


```
angular.module('silex-request-get', [])
.controller('Home', function($scope, $http){
  $scope.somar = function() {
    $http.post('/livro-web-codigo-fonte/backend/request-angular-post/', {
      data: {numero1 : $scope.numero1, numero2 : $scope.numero2}
    }).then(function(response){
      $scope.resultado = response.data;
    });
  };
});
```

Aplicativo Advanced REST Client

O aplicativo *Advanced REST Client* para o Google Chrome (<https://chrome.google.com/webstore/detail/advanced-rest-client/hgmlloofddffdnphfgcellkdfbfjeloo>) é outra forma de gerar requisições via navegador e testar a comunicação com o backend.

Após instalar o aplicativo realize os seguintes passos:

- Informe a URL (não se esqueça de indicar `/` ao final, pois representa a rota)
- Escolha "POST" como o tipo da requisição
- Na seção *Payload*, na aba *Raw*, informe o conteúdo da requisição (em formato JSON)
- Escolha "application/json" como codificação da requisição
- Clique no botão "Send"

Sobre o conteúdo da requisição, deve ser informado da mesma forma com o Angular o faria (já que o backend está esperando este formato). Assim, o conteúdo deve ser JSON, como o exemplo:

```
{"data":{"numero1":1,"numero2":3}}
```

A figura a seguir ilustra este processo (*animação GIF disponível apenas na versão online deste livro*).

Advanced Rest Client

Request

Socket

Projects
empty

Saved

History

Settings

About

Rate this application ▼

Donate

[Unnamed] Save Open

▶ URL

☐ GET ☒ POST ☐ PUT ☐ PATCH ☐ DELETE ☐ HEAD ☐ OPTIONS ☐ Other

application/json

⚙

Raw **Form** Headers

Add new header

Raw **Form** Files (0) Payload

Encode payload Decode payload

application/x-www-form-urlencoded ▼ Set "Content-Type" header to overwrite this value.

Clear Send

Assim, o aplicativo *Advanced REST Client* para o Google Chrome pode ser usado como alternativa para criar clientes de backends construídos com Silex e outros frameworks para o PHP ou outras plataformas de programação.

Banco de Dados

Um banco de dados, representado por meio de um SGBD-R (Servidor de Bancos de Dados Relacional), permite o armazenamento persistente de dados. Em aplicativos web, armazenar os dados de forma persistente é uma tarefa bastante comum.

Em termos de SGBD, o contexto open-source apoia-se sobre opções como *MySQL* e *PostgreSQL*. Esta seção utilizará como base o SGBD *Microsoft SQL Server*, que possui uma versão gratuita, a *Microsoft SQL Server Express* -- e assume que o mesmo estará disponível para utilização.

Configurando o ambiente

Por padrão, o PHP não se comunica com o SQL Server diretamente (embora possa fazer isso em "baixo nível", utilizando a ODBC). Entretanto, a maneira mais prática de acessar bancos de dados com PHP é via *PDO*, uma API de acesso a dados que permite um nível mais alto de comunicação, convertendo valores e estruturas do banco de dados para representações em PHP. Por exemplo, um registro de uma tabela é convertido para um objeto ou um array.

Para configurar o ambiente de modo que o PHP possa comunicar-se com um SGBD SQL Server é necessário instalar os [drivers do SQL Server para PHP](#). Os drivers estão distribuídos conforme versões do PHP e características do servidor web, como mostra a tabela a seguir.

Versão(ões) do PHP	Versão do driver
5.4 a 5.6	3.2
5.4 e 5.5	3.1
5.4	3.0

Após baixar a versão desejada, conforme a versão do PHP, execute o instalador (que, na verdade, apenas irá descompactar os arquivos em uma pasta escolhida por você). Por padrão, o PHP armazena módulos de extensão (como este da Microsoft) no diretório `ext` (no diretório de instalação do PHP. Ex: `c:\xampp\php`).

No diretório de instalação do PHP há o arquivo `php.ini` , que contém configurações padrões do PHP. Edite o arquivo e adicione as linhas a seguir:

```
extension=php_sqlsrv_55_ts.dll  
extension=php_pdo_sqlsrv_55_ts.dll
```

Neste caso, o arquivo indica que está sendo utilizado para a versão 5.5 do PHP (que é a utilizada como base de código no momento). Os arquivos possuem, no nome, a parte `ts`, que indica que são os arquivos usados para o servidor web *Apache*. No caso de estar utilizando outro servidor, verifique na documentação da Microsoft como fazer a configuração de forma correta.

Depois de editar o arquivo reinicie o servidor web.

O acesso ao banco de dados SQL Server pode ser feito, a partir de então, via PDO ou outra biblioteca. Este livro está utilizando Silex e, portanto, utilizará outras bibliotecas complementares para acessar o banco de dados.

Embora o Silex não seja dependente de um framework de acesso a dados específico, é interessante, por causa do Symfony, utilizar o [Doctrine](#). Doctrine fornece duas maneiras básicas de acesso a dados:

- DBAL; e
- ORM.

As seções seguintes mostram como utilizar estes componentes.

Doctrine DBAL - Database Abstraction Layer

Para instalar a DBAL utilize composer:

```
composer require doctrine/dbal
```

Criando uma conexão

Uma conexão com o banco de dados é criada por meio da classe

```
\Doctrine\DBAL\DriverManager .
```

```
$config = new \Doctrine\DBAL\Configuration();
$connectionParams = array(
    'dbname' => 'cidades',
    'user' => 'sa',
    'password' => 'sa',
    'host' => 'localhost',
    'driver' => 'pdo_sqlsrv',
    'charset' => 'utf-8'
);
$db = \Doctrine\DBAL\DriverManager::getConnection($connectionParams, $config);
```

No trecho de código acima, o método `getConnection()` retorna uma conexão com o banco de dados, conforme os parâmetros da conexão (`$connectionParams`). Nos parâmetros da conexão, as chaves do array são importantes.

Chave	Descrição
<code>dbname</code>	O nome do banco de dados
<code>user</code>	O nome do usuário da conexão
<code>password</code>	A senha do usuário da conexão
<code>host</code>	O nome do host do banco de dados
<code>driver</code>	O nome do driver da conexão
<code>charset</code>	(opcional) A codificação dos dados

No caso do trecho de código:

- O banco de dados chama-se `cidades`

- O nome do usuário e sua senha são `sa` e `sa` , respectivamente
- O nome do host é `localhost`
- O nome do driver é `pdo_sqlsrv`
- A codificação de caracteres é `utf-8`

Se você encontrar problemas utilizando o driver `pdo_sqlsrv` , utilize o driver `sqlsrv` .

Recuperação e manipulação de dados

Consulta (SELECT)

Uma consulta SQL pode ser executada por meio de uma conexão e do método `query()` :

```
$sql = "SELECT * FROM Cidades";
$query = $db->executeQuery($sql);
$cidades = $query->fetchAll();
```

A consulta SQL é representada pela variável `$sql` (uma `string`). O método `executeQuery()` cria uma instância de `Doctrine\DBAL\Statement` .

O método `fetchAll()` retorna todos os registros encontrados, na forma de um `array` . Se for necessário iterar pelos resultados, ou se a consulta fornecer apenas um resultado, você pode utilizar o método `fetch()` :

```
$sql = "SELECT * FROM Cidades";
$query = $db->executeQuery($sql);
while ($linha = $query->fetch()) {
    echo $linha['nome'];
}
```

Parâmetros

Embora a consulta SQL seja representada por uma string, por questões de segurança, não é interessante concatenar valores informados pelo usuário, por exemplo, para filtrar um conjunto de resultados com base em um critério. Assim, deve ser usado o método

`executeQuery()` de `Doctrine\DBAL\Connection` para criar um `Statement` :

```
$sql = "SELECT * FROM Cidades WHERE id = ?";
$query = $db->executeQuery($sql, array(1));
$cidade = $query->fetch();
```

A instrução SQL possui o símbolo `?` para indicar que deve se recuperado o registro cuja coluna `id` corresponde a um valor (ainda não informado).

O método `executeQuery()` aceita como parâmetros a instrução SQL e um `array`, que contém os valores para os parâmetros da instrução SQL.

Inserindo valores (UPDATE, DELETE, INSERT)

Enquanto o método `Connection->executeQuery()` é útil para instruções SELECT, o método `Connection->executeUpdate()`, que funciona de maneira idêntica, é indicado para instruções UPDATE, DELETE e INSERT. O método `executeUpdate()` retorna a quantidade de linhas afetadas pela execução da instrução.

```
$sql = "INSERT INTO Estados(nome, uf) VALUES(?, ?)";  
$r = $db->executeUpdate($sql, array("Tocantins", "TO"));
```

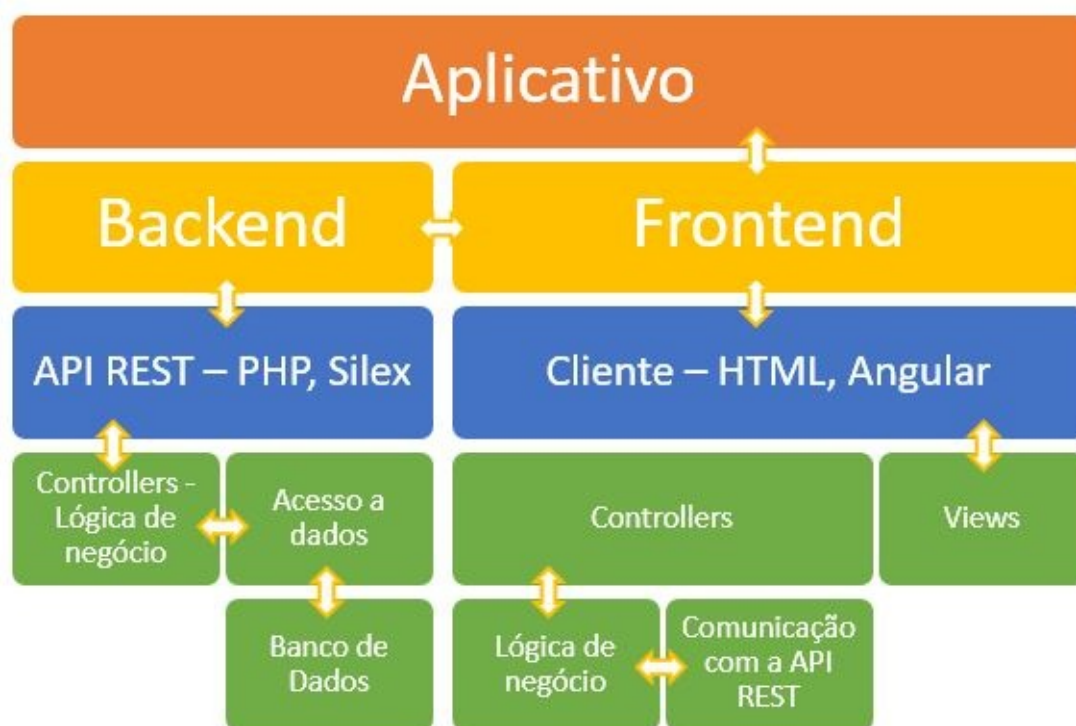
A instrução SQL é um INSERT. A variável `$r` contém a quantidade de linhas afetadas pela execução da instrução SQL. No caso de uma instrução INSERT isso quer dizer que se `$r > 0`, então os dados foram inseridos. Caso contrário, `$r == 0`.

Modelo de Arquitetura API - Cliente

Com base no que já foi visto até o momento, pode-se evoluir em sentido de arquitetura do software. Esta seção apresenta informações sobre a arquitetura de uma solução baseada em API REST e o consumo da mesma por um cliente. Para demonstrar o software criado sobre esta arquitetura, o contexto do software é um gerenciador de dados de cidades (com CRUD para Estados e Cidades).

Arquitetura da solução

A figura a seguir apresenta a arquitetura da solução proposta nesta seção do livro.



Os componentes da arquitetura são:

Componente	Descrição
Backend	Visão geral do lado servidor. Acessado pelo Frontend
API REST	Software construído sobre PHP e Silex. Contém: Controllers, Acesso a Dados e o Banco de dados
Cliente	Software construído sobre HTML e Angular. Contém: Controllers, Lógica de Negócio, Comunicação com a API REST e Views

A figura também apresenta os fluxos de comunicação entre os componentes:

- O usuário acessa o **Aplicativo**, que acessa mais diretamente o **Frontend**
- O usuário acessa mais diretamente o **Frontend (Views)**, que comunica-se com o **Backend** por meio da **Comunicação com a API REST** (componente utilizado pela **Lógica de Negócio - Controllers**)
- O **Backend**, construído sobre PHP e Silex, contém os **Controllers**, que contêm a **Lógica de negócio** e acessam o **Banco de Dados** por meio do **Acesso a Dados**

A seguir, as seções seguintes apresentam cada componente de forma detalhada.

Backend

O Backend representa o *lado servidor*. Utilizando Silex, o Backend fornece uma API REST que acessa o Banco de Dados para realizar as operações CRUD.

Banco de Dados

O Banco de Dados armazena dados de cidades e Estados. A figura a seguir ilustra a estrutura do banco de dados.



API REST

A API REST, desenvolvida sobre Silex, atende as seguintes rotas:

Rota	Método	Descrição
/estados	GET	Retorna a lista de Estados. Um array cujos elementos são objetos com três atributos: id, nome e uf
/estados	POST	Salva um Estado (cadastra ou edita) no banco de dados
/estados/{id}	GET	Retorna o Estado com identificador igual ao parâmetro id. Um objeto com os atributos: id, nome e uf
/estados/{id}	DELETE	Exclui o Estado com identificador igual ao parâmetro id
/cidades	GET	Retorna a lista de Cidades. Um array cujos elementos são objetos com os atributos: id, nome, idEstado, ufEstado e nomeEstado
/cidades	POST	Salva uma Cidade (cadastra ou edita) no banco de dados
/cidades/{id}	GET	Retorna a Cidade com identificador igual ao parâmetro id. Um objeto com os atributos: id, nome, idEstado, ufEstado e nomeEstado
/cidades/{id}	DELETE	Exclui a Cidade com identificador igual ao parâmetro id

Frontend

O Frontend representa o *lado cliente*. Utilizando Angular, o Frontend possui uma estrutura que permite o gerenciamento dos dados. As telas são gerenciadas por meio do módulo `ngRoute`. Os controllers utilizam o serviço `$http` para criar requisições HTTP à API. O trecho de código a seguir apresenta um trecho de código com o `EstadosListaController`, um controller que gerencia a tela de lista de Estados.

```
.controller('EstadosListaController', function($scope, $http){
    $http.get(API_BASE + '/estados')
    .then(function(response){
        $scope.estados = response.data;
    });

    $scope.excluir = function(estado, index) {
        if (confirm('Tem certeza que deseja excluir o Estado ' + estado.nome + '?')) {
            $http.delete(API_BASE + '/estados/' + estado.id)
            .then(function(response){
                $scope.estados.splice(index, 1);
            });
        }
    };
});
```

No início do controller, é feita uma requisição GET para a API, rota `/estados`. Quando o resultado for retornado, o `array` de Estados, ele é atribuído a `$scope.estados`, que está vinculado à view para apresentação da lista de Estados.

A função `excluir()` realiza uma requisição DELETE para a API, rota `/estados/{id}`, informando o identificador do Estado a ser excluído. Neste caso, quando a requisição for atendida com sucesso, o objeto `$scope.estados`, que está vinculado à view que lista os Estados, é modificado, excluindo o índice do Estado na lista. Uma alternativa seria realizar uma nova requisição GET para a API, retornando a lista de Estados.