PUC Minas
Virtual

TECNOLOGIAS DE DESENVOLVIMENTO DE APLICAÇÕES HÍBRIDAS

Tópicos Especiais em Sistemas de Informação: Aplicações Híbridas

Unidade 2

Prof. Marcos André Silveira Kutova

1. NODE.JS E NPM

- Node.js é uma plataforma do lado do servidor baseada no motor de JavaScript do Google Chrome (V8 Engine). É uma plataforma aberta, que roda em qualquer sistema operacional.

- Node.js reduz o esforço dos desenvolvedores no aprendizado de linguagens de programação, pois usa a mesma linguagem do lado do cliente. No entanto, Node.js estende a JavaScript com um conjunto de módulos e recursos necessários à criação do lado do servidor de uma aplicação web.

- Algumas características do Node.js são:
  - As bibliotecas do Node.js são assíncronas e orientadas a eventos. Assim, a aplicação não fica bloqueada enquanto uma requisição é processada.
  - O interpretador JavaScript V8 é leve e rápido, suportando a construção de aplicações complexas com um grande número de requisições.
  - Toda a programação é feita em JavaScript e a troca de dados entre cliente e servidor é feita com JSON.
  - Node está mais para uma infraestrutura para construção de aplicações web do que para um framework. Há vários frameworks que podem ser usados com Node.js, como Hapi.js, Socket.io e Express.js.

NPM

- Instalar NodeJS e npm: (LTS = Long Term Support)
  - https://nodejs.org/en/download/

- npm (*Node Package Manager*) é uma aplicação que permite que desenvolvedores JavaScript compartilhem seu código e, assim, que outros desenvolvedores usem esse código nas suas aplicações. Quando você usa um código por meio do npm, fica mais fácil ver se há atualizações e atualizar esses códigos.
  - https://www.npmjs.com/

- Esses trechos de códigos são chamados de *pacotes* (ou de módulos) e são apenas um diretório com um ou mais arquivos nele, incluindo um arquivo chamado `package.json` que contém metadados sobre o pacote.
  - Um site pode usar dezenas ou centenas de pacotes, pois geralmente eles são pequenos e focados na solução de um problema.

- O npm usa o terminal (*command prompt*) para entrada de comandos.
  - Para ver quais pacotes estão instalados no sistema, digite:

    ```
    npm list
    ```

    - A instalação de pacotes é por diretório. Assim, se você ainda não tiver instalado nenhum pacote, a lista aparecerá vazia.

VISUAL STUDIO CODE

- A construção das aplicações pode ser feita com qualquer editor de textos. No entanto, alguns editores são mais adequados para o desenvolvimento de aplicações com JavaScript (Node, Angular, Ionic, ...). Entre eles, destaca-se o Visual Studio Code

- Instalação do Visual Studio Code
  - Disponível no site https://code.visualstudio.com/
  - Editor de código gratuito da Microsoft
  - IntelliSense
  - Emmet - https://www.smashingmagazine.com/2013/03/goodbye-zen-coding-hello-emmet/
  - Depuração de código JavaScript

2. USO DO NODE.JS

- O primeiro teste que podemos fazer para testar a instalação do Node.js é solicitar a sua versão, por meio do seguinte comando no terminal:

```
node –v
```

A informação retornada será a versão da sua instalação do Node.js.

- Em seguida, é possível testar o funcionamento do interpretador, por meio da criação de uma simples aplicação em JavaScript, em um arquivo `app.js`.

```
console.log('Olá mundo');
```

Agora, basta rodar a aplicação, em uma janela do terminal, a partir do diretório em que se encontra o arquivo:

```
node app.js
```

- Também podemos usar o Node.js por meio de sua interface de comandos (*Command Line Interface – CLI*). Essa interface de comandos é apresentada quando digitamos apenas:

```
node
```

- Nessa interface de comandos, podemos digitar comandos JavaScript. Por exemplo:

```
> 1+1;                                    // retorna 2
> var a = 1;                              // retorna undefined
> a * 2;                                  // retorna 2
> (function(n){return (n*n)})(4)          // IIFE: retorna 16
> (n=>n*n)(4)                             // Exp.Lamba: retorna 16
```

MÓDULOS

- Node.js foi criado antes dos módulos da ES2015, como usado em Angular. Na época, a opção foi pela adoção da especificação CommonJS, que é ligeiramente diferente da ES2015. Há a previsão do suporte aos módulos da ES2015 em um futuro breve em Node.js, mas o suporte a CommonJS será mantido. Veremos, aqui, como criar módulos usando a CommonJS.

- Em Node.js, um arquivo exporta um objeto ou função, por meio do objeto `module.exports`. Por exemplo, podemos criar um objeto `ola`, que contém uma função anônima, no arquivo `ola.js` e exportá-lo.

```
var ola = function() {
  console.log('Olá mundo!');
}
module.exports = ola;
```

- No arquivo `app.js`, podemos importar esse objeto (ou qualquer outra coisa exportada), por meio da função `require()`.

```
var ola = require('./ola');
ola();
```

- A função `require()` realiza várias operações. As mais importantes são a leitura do arquivo que contém o módulo sendo importado e o encapsulamento desse módulo importado em uma função expressão. Essa função expressão é, em seguida, executada e a propriedade `exports` do objeto `module` é retornada. No entanto, isso são detalhes da implementação do Node.js que não precisam ser aprofundados para que se possa desenvolver aplicações web com Node.js.

- A função `require()` pode ser usada de forma hierárquica, isto é, um módulo pode incorporar outros módulos.

- A função `require()` também pode importar arquivos JSON.

- Se nenhum arquivo for especificado, mas apenas um diretório (ou pasta), então a função `require()` procurará por um arquivo `index.js` nesse diretório.

- A função `require()` pode receber qualquer tipo de dado: uma variável, um objeto, uma função, uma função construtora, etc. Obviamente, isso depende do que for associado ao `module.exports`.

- Até que receba algum valor, o objeto `module.exports` é criado como um objeto vazio, mas é, ainda assim, um objeto. Assim, nada impede que ele receba propriedades da seguinte forma:

```
module.exports.ola = function() {
    console.log('Olá mundo!');
}
```

no programa principal, essa propriedade poderia ser acessada:

```
var obj = require('./ola');
obj.ola();
```

- A função `require()` também é usada para a importação de módulos nativos do Node.js. Nesse caso, porém, não especificamos o caminho (`./`). Os módulos nativos podem ser encontrados na documentação do Node.js:
  - https://nodejs.org/dist/latest-v6.x/docs/api/

- Assim, podemos formatar uma string usando o método `format()` do módulo *Utilities* e escrevê-la no console usando o método `log()`.

```
const util = require('util');
util.log( util.format('PI = %d', 3.14159)
```

3. IONIC

- O Ionic é um *framework front-end* para o desenvolvimento de aplicações híbridas. Isso quer dizer que o Ionic se apoia em tecnologias web (HTML, CSS e JavaScript) para desenvolver uma aplicação, que rodará em *smartphones*, *tablets* ou outros dispositivos portáteis.

- O Ionic é uma extensão do *framework* Angular, o que significa que depende do mesmo para funcionar. Assim, uma aplicação Ionic é também uma aplicação Angular.
  - A criação de uma aplicação Ionic é ligeiramente diferente da criação de uma aplicação Angular, mas apenas porque muitas coisas estão sendo feitas automaticamente quando usamos Ionic.

- Uma aplicação Ionic é também uma aplicação Cordova, isto é, o que exportamos para as lojas virtuais (ou para os próprios dispositivos) é uma aplicação Cordova. Essa aplicação executará o código Ionic (HTML, CSS e JavaScript) em um container WebView.

- Quando criamos uma aplicação Ionic, criamos, automaticamente, uma aplicação Cordova para cada plataforma instalada. Essas aplicações Cordova são aplicações nativas. Serão elas que, de fato, serão exportadas para os dispositivos (e rodando o nosso código Ionic no container WebView).

- O Cordova também oferece uma API em JavaScript para acessos aos recursos nativos dos dispositivos por meio do Ionic. O que acontece, por trás dos panos, é que quando usamos essa API, a nossa aplicação Cordova traduz as nossas requisições em requisições para a API nativa de cada dispositivo.

- Assim, o Cordova é um *middleware* que faz a conexão entre as plataformas (iOS, Android, ...) e a aplicação Ionic.

- Voltando ao Ionic, o *framework* oferece uma série de componentes de interface familiares aos dispositivos portáteis, como barras de tarefas, menus, listas, campos de formulário, etc.

- As vantagens da criação de aplicações híbridas com Ionic são:
  - Aproveita as habilidades dos desenvolvedores web, isto é, os desenvolvedores web não precisam mais se especializar em todas as linguagens/plataformas de desenvolvimento.
  - Um único código para qualquer plataforma, o que não só acelera a publicação em todas elas, como também facilita a manutenção da aplicação.

- A documentação do Ionic 2 está disponível em: http://ionicframework.com/

- Instalar o Ionic e Cordova

  ```
  npm install -g cordova ionic
  ```

  - No Mac, será necessário usar o comando `sudo`.
  - No Windows, é importante conferir se os caminhos dos aplicativos estão incluídos na variável de ambiente `Path`.

APLICAÇÃO DE EXEMPLO

- Instalar uma aplicação de exemplo

  ```
  ionic start tesi --v2
  ```

  - O nome da aplicação, aqui, será `tesi`. Qualquer nome é válido (sem espaços e símbolos especiais).
  - Ao término da instalação, o Ionic perguntará se você deseja criar uma conta. Não é necessário, porém o cadastro permite acesso a recursos extras.
  - Os projetos em Ionic são sempre baseados em algum *template* (modelo). Se nenhum template for especificado, então o template *tabs* será usado. A lista de *templates* do Ionic pode ser encontrada em https://github.com/driftyco?utf8=%E2%9C%93&query=ionic2.
    - `ionic start --v2 myApp blank`
    - `ionic start --v2 myApp tabs`
    - `ionic start --v2 myApp sidemenu`

- Iniciar a aplicação

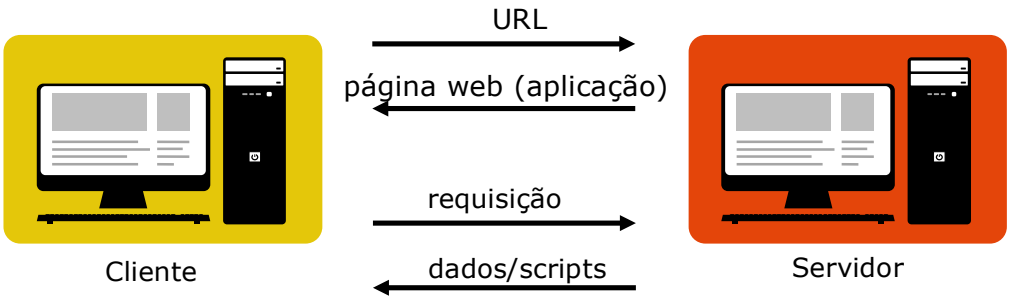
  ```
  cd tesi
  tesi serve
  ```

  - Se o parâmetro `--lab` for usado, então as visões para iOS, Android e Windows serão automaticamente apresentadas.

    ```
    ionic serve --lab
    ```

4. ANGULAR

- Angular é um *framework* JavaScript da Google, usado para a criação de aplicações web de página única (SPA – *Single Page App*).
  - Uma aplicação de página única é uma aplicação em que todos os recursos são acessados por meio de um único URL.
  - Em outras palavras, os dados e scripts necessários à aplicação são carregados de forma transparente para o usuário.



- Angular permite a criação de aplicações sofisticadas de forma rápida.
- Angular é modular. Isso não só organiza o código, como torna os componentes de uma aplicação possíveis de serem reaproveitados em diversas outras aplicações.
- Angular se integra bem com diversas outras tecnologias, como jQuery, Bootstrap, Node, Ionic, ...
- O desenvolvimento em Angular pode ser feito com JavaScript ou com TypeScript.
  - TypeScript é um superconjunto tipado de JavaScript que é compilado para JavaScript simples.
  - TypeScript traz uma série de novidades da ES2015
  - TypeScript é mantido pela Microsoft (https://www.typescriptlang.org/).

5. PRIMEIRA APLICAÇÃO ANGULAR

- Toda aplicação Angular é criada a partir de um projeto genérico, isto é, um conjunto de arquivos que contém a *semente* de uma aplicação, mesmo que ainda sem funcionalidades implementadas. Você deve baixar a aplicação semente (QuickStart) a partir do seguinte URL:
  - https://github.com/angular/quickstart/archive/master.zip
  - Os arquivos devem ser descompactados na pasta da sua aplicação.

- Os três arquivos principais da aplicação são:
  - `src/app/app.component.ts` Define o componente `AppComponent` como o componente raiz da sua aplicação. Uma aplicação funciona como uma árvore de componentes e este é, portanto, a raiz dessa árvore.
  - `src/app/app.module.ts` Define o módulo `AppModule` como o módulo raiz da sua aplicação. Uma aplicação pode possuir vários módulos, mas este exemplo contém apenas um módulo, em que é definido o `AppComponent`.
  - `src/main.ts` O programa principal inicia a aplicação por meio do seu módulo raiz `AppModule`. Algumas outras operações podem ser realizadas aqui, porém neste exemplo há apenas a inicialização da aplicação.

- Ainda há vários outros arquivos importantes, que serão apresentados de acordo com a necessidade. No entanto, já é importante conhecer o arquivo `package.json` que contém alguns metadados do projeto (como o nome e a versão) e todas as dependências, isto é, todos os pacotes que podem ser necessários à aplicação. É um conjunto bem completo de pacotes, além do que as aplicações básicas necessitam. Os comandos do npm estão definidos dentro desse arquivo, no objeto `scripts`.

- Agora basta instalar todos os pacotes. Para isso, na pasta da sua aplicação, digite:

  ```
  npm install
  ```

- Após a instalação de tudo, já é possível executar a aplicação, por meio do comando:

  ```
  npm start
  ```

  - Se tudo der certo, você verá uma página com a mensagem "Hello Angular".

6. CÓDIGO DA APLICAÇÃO ANGULAR

- No diretório da sua aplicação, estão todos os arquivos necessários para sua execução. O código fonte fica no diretório `src` e o código específico da sua aplicação fica no diretório `src/app`.

- Nesse diretório, você encontrará o componente raiz da aplicação, chamado de `AppComponent`: `src/app/app.component.ts`. Observe que há uma convenção de incluir, nos componentes, uma extensão que especifica que é um componente TypeScript (`component.ts`).

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`,
})
export class AppComponent  { name = 'Angular'; }
```

  - O elemento mais importante desse código é a classe `AppComponent`, que será o componente principal da nossa aplicação. Toda aplicação Angular precisa de um componente principal (componente raiz) e, geralmente, esse componente é o `AppComponent`.
  - No entanto, a classe está praticamente vazia e apenas declara uma variável `name`. Por meio dos atributos e métodos dessa classe é que criaremos o comportamento do componente.
  - A ES2015 (e a TypeScript) trouxeram para a JavaScript o padrão de projeto de *Decorators*. Um *decorator* nada mais é do que uma função que assume alguma responsabilidade de outra. Basicamente, é uma função que toma outra e modifica parte do seu comportamento ou da sua resposta (isso parece um bocado com uma subclasse). Os *decorators* são precedidos pelo símbolo @.
    - Sugestão de leitura: http://javascript.info/tutorial/decorators
  - No código do exemplo, o *decorator* `Component`, que recebe um objeto de metadados como parâmetro, modifica a classe `AppComponent`, inserindo nela os elementos básicos de um componente Angular.
    - Esse objeto passado como parâmetro contém algumas especificações necessárias para a modificação da classe.
    - Por exemplo, o atributo `template` define a visão do componente (de acordo com o padrão de projeto MVC)

  - Outra característica importante do código é que ele implementa o conceito de módulos da ES2015. Todo arquivo pode ter seu código próprio, mas, dentro de uma lógica modular, esse arquivo pode usar código de outro módulo (por meio do `import`) e disponibilizar código para outros módulos (por meio do `export`).
    - Sugestão de leitura: https://hacks.mozilla.org/2015/08/es6-in-depth-modules/
  - A importação é do *decorator* (ou função) `Component`, que faz parte do pacote `@angular/core` (que está instalado na sua pasta `node_modules`).

- Os parâmetros recebidos como metadados do *decorator* Component são:
  - `selector` Especifica um seletor CSS para um elemento no documento, que representará o componente. Esse componente pode ser algo personalizado como `<my-app>`.
  - `template` Especifica um *template* (marcação HTML) que servirá como conteúdo para o elemento selecionado. Nesse *template*, há uma interpolação de uma variável por meio das chaves duplas `{{` e `}}`.

- As aplicações em Angular são organizadas em módulos de funcionalidades. Toda aplicação precisa ter pelo menos um módulo que, por convenção, geralmente é chamado de `AppModule` e fica no arquivo `src/app/app.module.ts`.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

- É possível colocarmos toda a nossa aplicação dentro de um único módulo ou criarmos vários módulos. Basicamente, a ideia é pensar em blocos de funcionalidades *exportáveis*, isto é, que podem ser usados em outras aplicações. Se não for o caso da sua aplicação, então pode usar um único módulo.

- O *decorator* `@NgModule`, usado na criação do módulo, recebe vários parâmetros importantes, que eram especificados em outros lugares. Esses parâmetros são:
  - `imports` Lista dos módulos que serão usados em nossa aplicação, caso você use algum. O módulo `BrowserModule` é essencial em quase todas as aplicações e é usado para permitir que a aplicação rode no navegador. Outros módulos bastante úteis são `FormsModule`, `RouterModule` e `HttpModule`.
  - `declarations` Componentes e diretivas usadas neste módulo. Todos os componentes e diretivas usados em cada *decorator* dos seus componentes devem ser especificados aqui, inclusive as diretrizes usadas para roteamento.
  - `bootstrap` Componente raiz que, geralmente, é o `AppComponent`.

- Observe, em seguida, o código principal: `src/main.ts`. Esse arquivo deve ter o seguinte conteúdo:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

  - O método `bootstrapModule()` do objeto retornado por `platformBrowserDynamic()` inicializa a sua aplicação, por meio daquilo que estiver descrito no `AppModule`.
    - *Boostrap* significa inicializar. É daí que vem a expressão "*dar um boot no computador*".

- Observe, agora, a página `src/index.html`, no diretório raiz do projeto, com o seguinte código

```
<!DOCTYPE html>
<html>
  <head>
    <title>Angular QuickStart</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <base href="/">
    <link rel="stylesheet" href="styles.css">

    <!-- Polyfill(s) for older browsers -->
    <script src="node_modules/core-js/client/shim.min.js"></script>

    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="systemjs.config.js"></script>
    <script>
      System.import('main.js').catch(function(err){ console.error(err); });
    </script>
  </head>

  <body>
    <my-app>Loading AppComponent content here ...</my-app>
  </body>
</html>
```

  - As bibliotecas importam algumas operações para o navegador. A biblioteca `shim.min.js` "instala" algumas funções básicas da ES2015 em navegadores antigos. A biblioteca `zone.js` é responsável pela atualização da visão, quando houver alterações nos dados. A biblioteca `system.src.js` é o carregador de módulos usado pelo Angular.
  - Em seguida, é carregado o *script* `systemjs.config.js` com a configuração para a biblioteca SystemJS. Entre outras configurações, esse *script* diz de onde os arquivos devem ser carregados.
  - Finalmente, observe o elemento `<my-app>` inserido no corpo da página. O conteúdo desse elemento será substituído de acordo com o *template* especificado no *decorator* do `AppComponent`.

- O arquivo `src/styles.css` contém a formatação básica da aplicação.

```
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
```

- Rode a aplicação com o seguinte comando na janela do terminal:

```
npm start
```

- Os arquivos TypeScript serão transpilados (compilados para outra linguagem) em arquivos JavaScript e transpilador ficará rodando, observando qualquer eventual alteração nos seus arquivos.
- Um servidor web chamado **lite-server** carrega a página `index.html`. A página poderá ser acessada no endereço http://localhost:3000/. Qualquer alteração feita no código automaticamente atualizará a página.

7. COMPONENTES EM ANGULAR

- Os componentes normalmente representam instâncias de informações e controlam a apresentação dessas informações na página.

- Por exemplo, em um site de filmes de cinema, um componente poderia representar uma lista de filmes, outro poderia representar um filme propriamente dito. Um terceiro componente poderia ser o que representa um ator desse filme. Ainda um outro componente poderia ser usado para uma cena do filme.

- Vamos criar, então, uma aplicação para exibir dados de filmes. A primeira etapa para isso, é a criação de um novo componente (`filmes.component.ts`) que apresentará a lista de filmes.

```
import { Component } from '@angular/core';

@Component({
    selector: 'my-app',
    template: '<h2>Lista de filmes</h2>'
})
export class FilmesComponent {

}
```

- Esse componente implementa a classe `FilmesComponent`, ainda vazia. O *decorator* Component define dois metadados para a classe: `selector`, que indica qual elemento terá seu comportamento associado ao componente, e `template`, que indica o conteúdo propriamente dito desse elemento.

- O componente de filmes será renderizado pelo componente principal `AppComponent`, por meio do elemento `<filmes>`.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <h1>Filmoteca Online</h1>
    <filmes></filmes>
    `
})

export class AppComponent { }
```

- O uso do acento grave ` no `template`, ao invés das aspas simples, permite que ele seja criado usando várias linhas.

- O *template* também pode ficar em um arquivo externo (.html), bastando, para isso, usarmos a propriedade `templateUrl` que conterá o nome do arquivo como valor.

- O elemento <filmes> poderia ser substituído por qualquer outro como, por exemplo:

```
<section id="filmes"></section>
```

desde que, no `FilmesComponent`, o seletor fosse alterado para `#filmes`.

- A inserção do elemento `<filmes>` no `template` do `AppComponent`, não é suficiente para a execução do componente `FilmesComponent`. Para que ele seja incluído na aplicação, ele deve ser inserido na lista `declarations` do módulo `NgModule` (`app.module.ts`).

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';
import { FilmesComponent } from './filmes.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, FilmesComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

- A lista `declarations` contém todos os componentes e diretivas usadas no módulo. Diretivas (`directives`) são classes que alteram a visão (apresentação) do componente. Essas diretivas podem ser elementos ou atributos de elementos. Tecnicamente falando, um componente é também uma diretiva.

- O `template` de um componente pode usar dados do próprio componente. Isso é feito por meio de algo chamado de interpolação.

```
import { Component } from '@angular/core';

@Component({
    selector: 'filmes',
    template: '<h2>{{titulo}}</h2>'
})
export class FilmesComponent {
    titulo: string = 'Lista de Filmes';
}
```

- As interpolações são feitas por meio das chaves duplas `{{` e `}}`.

- Um componente também pode ter um construtor. Um construtor é declarado por meio do método `constructor()` e serve, obviamente, para inicializar o componente.

```
import { Component } from '@angular/core';

@Component({
    selector: 'filmes',
    template: `
      <h2>{{titulo}}</h2>
      `
})
export class FilmesComponent {
    titulo: string;

    constructor() {
        this.titulo = 'Lista de Filmes';
    }
}
```

- Nesse exemplo, criamos um `titulo` do tipo `string`. Os tipos básicos de TypeScript são:
  - `boolean`
  - `number`
  - `string`

- Podemos criar vetores de objetos de duas formas:

```
let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];
```

- Uma tupla é uma espécie de vetor em que os tipos são diferentes:

```
let x: [string, number] = [ 'Olá mundo!', 10];
```

- Também é possível criarmos uma enumeração (iniciada em zero ou com valores atribuídos manualmente):

```
enum Cores {Vermelho, Verde Azul};
let c: Cores = Cores.Verde;   // equivale a 1

enum Cores {Vermelho = 1, Verde = 2, Azul = 4};
let c: Cores = Cores.Verde;   // equivale a 2
```

- Em algumas situações, o valor da variável pode ser desconhecido previamente e precisamos usar um tipo genérico. A solução é o tipo `any`.

```
let desconhecido: any;
desconhecido = 4;
desconhecido = "uma string";
desconhecido = false;
```

- Quando uma função não retorna valor, o tipo a ser usado é `void`.

```
function alerta(msg): void {
  alert(msg);
}
```

- Existem também duas formas de fazermos coerção de tipos:

```
let coisa: any = "esta é uma string";
let comprimento: number = (<string>coisa).length;

let coisa: any = "esta é uma string";
let comprimento: number = (coisa as string).length;
```

8. DIRETIVAS

- Uma diretiva é usada para controlar algum aspecto da apresentação por meio da manipulação do DOM.

- Por exemplo, a diretiva NgFor é usada para criar elementos para cada item de uma coleção. No componente do filme, podemos usar essa diretiva para criar a lista dos filmes.

```
import { Component } from '@angular/core';

@Component({
    selector: 'filmes',
    template: `
      <h2>{{titulo}}</h2>
      <ul>
        <li *ngFor="let filme of filmes">{{filme}}</li>
      </ul>
      `
})
export class FilmesComponent {
    titulo: string;
    filmes: string[];

    constructor() {
        this.titulo = 'Lista de Filmes';
        this.filmes = ['Titanic', 'Jurassic Park', 'Avatar',
                       'Guerra nas Estrelas'];
    }
}
```

- Criamos, aqui, um vetor de filmes (`string[]`) contendo 4 filmes. Obviamente, o correto seria que essa lista viesse do banco de dados. Por ora, será inserida manualmente no código.

- O que interessa é a diretiva NgFor. A sintaxe exige o uso do * antes dela e há várias alternativas para sua inserção em um elemento. Novamente, por ora, essa será suficiente.

- A diretiva cria uma repetição do elemento `li` para cada item do vetor `filmes`, que será armazenado em uma variável de bloco `filme`. Essa variável será impressa, por meio da interpolação, como conteúdo do `li`.

- Existem várias outras diretivas e podemos criar as nossas próprias. Antes de tratarmos delas, veremos outros elementos do Angular.

9. SERVIÇOS

- O problema do exemplo do filme, é que os nomes dos filmes estão inseridos manualmente no componente. O correto seria ter esses filmes recuperados de algum outro lugar como, por exemplo, de um banco de dados.

- A classe que cuida de todos os processos não relacionados à visão é a classe de serviços (`Service`).

- Criamos um serviço, mas, por ora, com os dados inseridos manualmente também (só que fora da apresentação). O serviço ficará no arquivo `filmes.service.ts`.

```
export class FilmesService {
    getFilmes(): string[] {
        return ['Titanic', 'Jurassic Park', 'Avatar', 'Guerra nas Estrelas'];
    }
}
```

- O serviço não apresenta nada complicado. É uma classe normal (exportada), que possui um único método. Esse método deveria consultar o banco de dados e retornar a lista de filmes armazenados. Até vermos como fazer isso, deixaremos a inserção manual da lista.

- O componente `filmes.component.ts` precisa ser atualizado.

```
import { Component } from '@angular/core';
import { FilmesService } from './filmes.service';

@Component({
    selector: 'filmes',
    template: `
      <h2>{{titulo}}</h2>
      <ul>
        <li *ngFor="let filme of filmes">{{filme}}</li>
      </ul>
      `
})
export class FilmesComponent {
    titulo: string;
    filmes: string[];

    constructor(filmesService: FilmesService) {
        this.titulo = 'Lista de Filmes';
        this.filmes = filmesService.getFilmes();
    }
}
```

- A primeira ação a executar aqui é a importação da classe FilmesService, para que possa ser usada no componente.

- Observe que não há uma importação específica da classe FilmesService no código. No entanto, um objeto dessa classe é recebido (automaticamente) no construtor do componente. Esse tipo de importação é chamado de Injeção de Dependências (*dependency injection*).

- A injeção de dependências é uma forma de informar a uma instância de uma classe quais são as dependências que ela possui. Normalmente, essas dependências serão do tipo `service` como os que estamos criando. Essas dependências estarão disponíveis ao construtor como mostrado no código.

```
constructor(filmesService: FilmesService) { ... }
```

- Para que isso funcione, no entanto, devemos incluir a classe FilmesService na lista de serviços do módulo por meio do atributo `providers`.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';
import { FilmesComponent } from './filmes.component';
import { FilmesService } from './filmes.service';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, FilmesComponent ],
  providers:    [ FilmesService ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

- O `providers` é um vetor de dependências que o módulo possui. Basicamente, a lista de classes que serão necessárias para os componentes serem executados. Como essas classes são *injetadas* automaticamente nos componentes, então esse padrão é chamado de injeção de dependências.

- Portanto, a criação do objeto da classe `FilmesService` é feita automaticamente pela aplicação.

10. LIGAÇÃO DE DADOS

- Até agora, vimos a apresentação de dados por meio de uma interpolação. A interpolação coloca, no *template*, uma expressão que pode usar uma propriedade do componente ou um método.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: '<p>{{a+"+"+b}} = {{soma()}}</p>'
})

export class AppComponent {

  a:number = 1;
  b:number = 2;
  soma():number {
    return this.a+this.b;
  }

}
```

- Nesse exemplo, há a interpolação de uma expressão: `a + " + " + b` e de uma função `soma()`.

- Uma interpolação é um exemplo de uma ligação unidirecional (*one-way data binding*), isto é, o valor da expressão fica ligado ao valor da propriedade ou método.

ATRIBUTOS

- A ligação de dados também pode ser feita em atributos dos elementos da página. Por exemplo, podemos criar uma variável chamada `imagem` na classe:

  ```
  imagem: string = "imagens/avatar.jpg";
  ```

  E fazermos a inserção dela no *template* de duas formas. A primeira é usando uma expressão de interpolação:

  ```
  <img src="{{imagem}}" />
  ```

  E a segunda forma é dizermos que o conteúdo do atributo é, por definição, uma expressão de interpolação, ao colocarmos o nome do atributo entre colchetes.

  ```
  <img [src]="imagem" />
  ```

  Essa segunda forma tende a ser a mais usada.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: '<img [src]="imagem" />'
})

export class AppComponent {
  imagem:string = "imagens/avatar.jpg";
}
```

- A vantagem dessa notação de colchetes é que ela permite operações mais sofisticadas. Por exemplo, o atributo `style` da HTML é usado para fazermos formatações dos elementos. Por meio desse atributo, temos acesso a todas as propriedades da CSS. Ao invés de fazermos uma definição única do atributo `style`, podemos fazer isso por propriedade.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <div [style.width]="largura"
         [style.height]="altura"
         [style.backgroundColor]="cor">
    </div>
    `
})

export class AppComponent {
  largura: string = '200px';
  altura: string = '200px';
  cor: string = 'crimson';
}
```

- Podemos fazer algo semelhante com os atributos de valores lógicos. Por exemplo, podemos desabilitar um campo a partir de uma variável.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    Nome: <input type="text" [disabled]="desabilitado" />
    `
})

export class AppComponent {
  desabilitado: boolean = true;
}
```

- Finalmente, podemos fazer uma operação semelhante a essa para incluirmos ou excluirmos classes em um elemento.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    Nome: <input type="text" [class.erro]="deuErro" />
    `
})

export class AppComponent {
  deuErro: boolean = true;
}
```

- Para ver esse exemplo funcionando, seria interessante acrescentar alguma configuração da classe `erro` no arquivo `styles.css`.

11. EVENTOS

- A criação de eventos é tão simples quanto a definição das propriedades dos elementos. Basta associar o evento desejado a uma função. Para isso, colocamos o nome do evento entre parênteses.

```
import { Component } from '@angular/core';
import {}

@Component({
  selector: 'my-app',
  template: `
    <button (click)="clique()">Clique aqui.</button>
    <p><span id="res">{{n}}</span> cliques.</p>
    `
})

export class AppComponent {

  n: number = 0;
  clique(): void {
    this.n++;
  }

}
```

- Nesse exemplo, o evento `click` está associado ao método `clique()`. Além disso, o valor de `n`, que indica o número de cliques realizados, está em uma expressão de interpolação para apresentar o valor ao usuário.

- Qualquer evento pode ser usado dessa forma. Os eventos `mouseover` e `mouseout` são mostrados no próximo exemplo.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <div id="box"
         (mouseover)="ativa()"
         (mouseout)="desativa()"
         [class.sobre]="ativado">
    </div>
    `
})

export class AppComponent {
  ativado: boolean = false;
  ativa(): void { this.ativado = true; }
  desativa(): void { this.ativado = false; }
}
```

- Além dessas mudanças no código, foram feitas as seguintes definições no arquivo `styles.css`.

```
#box {
  width: 250px;
  height: 150px;
  border: thin solid black;
}
.sobre {
  background-color: crimson;
}
```

- É possível ter acesso aos dados do evento (`$event`), desde que ele seja informado como parâmetro da função.

```
import { Component } from '@angular/core';
import {}

@Component({
  selector: 'my-app',
  template: `
    <button (click)="clique($event)">Clique aqui.</button>
    <p><span id="res">{{n}}</span> cliques.</p>
    `
})

export class AppComponent {

  n: number = 0;
  clique(evento): void {
    this.n++;
    console.log(evento);
  }

}
```

12. ESTRUTURA DO PROJETO

- Da mesma forma que uma aplicação Angular, uma aplicação Ionic possui um arquivo `packages.json` que define todas as dependências, que serão armazenadas na pasta `node_modules`.
  - O pacote `ionic-angular` contém os componentes Ionic, usados na criação de aplicações híbridas.
  - O pacote `ionicons` contém os ícones clássicos das aplicações de smartphones, de acordo com cada plataforma.
  - O pacote `ionic-native` contém os objetos que permitem o acesso aos recursos nativos dos dispositivos (câmera, agenda, etc.).

- Um projeto em Ionic armazena todo o código compilado da aplicação no diretório `www` (o código original é armazenado no diretório `src`).

- O arquivo que serve como ponto de entrada para uma aplicação Ionic é o `www/index.html`. Esse arquivo importa os *scripts* e as folhas de estilos necessários.

- A aplicação será apresentada no local em que for inserido o elemento (ou diretiva) `<ion-app>`.

  ```
  <ion-app></ion-app>
  ```

- O arquivo index.html também possui um vínculo para os seguintes *scripts*:
  - `www/cordova.js` É esse arquivo que contém o Cordova, para serve como *middleware* para a sua aplicação. No entanto, o Cordova é realmente importado por meio de injeção de dependências, que a sua aplicação é construída. A tentativa de acessar o *script* diretamente resultará em um erro 404.
  - `www/build/js/main.js` Esse arquivo combina tudo que a sua aplicação precisa, isto é, o Ionic, o Angular e o código JavaScript que você escreveu.
  - `www/build/js/main.css` Esse arquivo contém todas as regras CSS geradas a partir das regras SASS que você escreveu.

- No diretório `src` armazenamos o código original, da aplicação. Quando executamos o comando `ionic serve`, o código dentro de `app` é transpilado para algo que o navegador entenda, como a versão ES5 da JavaScript.
  - Transpilação é a conversão do código em uma linguagem (ex.: TypeScript) para outra (ex.: ES5).

- Como qualquer aplicação Angular, todos os componentes (páginas) que criarmos em uma aplicação Ionic devem ser declarados dentro do NgModule (`src/app/app.module.ts`).
  - Uma diferença importante aqui é que os componentes Ionic não são inseridos na aplicação por meio do elemento indicado da propriedade `selector` do *decorator*, como em uma aplicação Angular.
  - Assim, eles precisam ser especificados também na propriedade `entryComponents` do NgModule. De uma forma simplificada, o objetivo dessa propriedade é exatamente esse: indicar quais componentes serão carregados por código e não por seletores.
  - Em uma aplicação Ionic, também é importante incluir todos os serviços por meio da propriedade `providers` do NgModule e não nos *decorators* de cada componente.

- A formatação das páginas em aplicações Ionic é feita com SASS – Syntactically Awesome Style Sheets. SASS é um pré-processador CSS que permite que façamos declarações de variáveis, criação de regras hierárquicas e muito mais. É importante lembrar que um arquivo SASS não é usado diretamente, mas compilado para uma versão CSS clássica.

- Enquanto o código do componente principal fica no diretório `src/app`, as páginas da nossa aplicação ficarão na pasta `src/pages`. Cada página possui, geralmente, três arquivos: o componente em TypeScript, o template em um arquivo HTML e as regras de formatação em um arquivo SASS.

ARQUIVO PRINCIPAL

- O código para gerar o conteúdo do elemento `<ion-app>` fica no arquivo `src/app/app.component.ts`:

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
import { StatusBar } from 'ionic-native';
import { TabsPage } from '../pages/tabs/tabs';

@Component({
  template: '<ion-nav [root]="rootPage"></ion-nav>'
})
export class MyApp {

  rootPage = TabsPage;

  constructor(private platform: Platform) {
    platform.ready().then(() => {
      // Okay, so the platform is ready and our plugins are available.
      // Here you can do any higher level native things you might need.
      StatusBar.styleDefault();
    });
  }
}
```

- O elemento `<ion-nav>`, no *template*, cria um controlador de navegação da classe `NavController`. Esse controlador permite a construção de uma pilha de páginas, sendo que a página que estiver no topo da pilha será a página apresentada. Navegar para uma página significa acrescentá-la à pilha. Voltar à página anterior significa retirar uma página da pilha. A página inicial da pilha é definida pela propriedade `root` do elemento `<ion-nav>`.

- Neste exemplo, a página `TabsPage` foi importada do arquivo `src/pages/tabs/tabs.ts`. Em seguida, essa página foi atribuída à variável `rootPage`, que foi passada ao atributo `root`.
  - Note que a forma de importação de conteúdo é diferente da baseada em diretivas usada em Angular e é por isso que elas precisam ser indicadas na propriedade `entryComponents` do NgModule.

- Os métodos encadeados `ready()` e `then()` do objeto `platform` (injetado no construtor) garantem que o código seguinte só será executado caso a plataforma tenha sido carregada.

- Neste exemplo, a barra de Status (sinal, hora, bateria, etc.) é formatada da forma padrão. Essa barra, obviamente, não aparece quando abrimos a apresentação no navegador do computador.

PÁGINAS

- Todas as páginas da aplicação estão armazenadas na pasta `src/pages`. Cada uma é composta por três arquivos: um componente em TypeScript, um *template* em HTML e um arquivo SCSS (um estilo de SASS).
  - Como os *decorators* são do Angular, tanto o template quanto as regras CSS (e não SCSS) podem ficar dentro do *decorator* do componente.

- A página `tabs` é composta pelo arquivo `src/pages/tabs/tabs.ts`

```
import { Component } from '@angular/core';
import { HomePage } from '../home/home';
import { AboutPage } from '../about/about';
import { ContactPage } from '../contact/contact';

@Component({
  templateUrl: 'tabs.html'
})
export class TabsPage {

  // this tells the tabs component which Pages
  // should be each tab's root Page
  tab1Root: any = HomePage;
  tab2Root: any = AboutPage;
  tab3Root: any = ContactPage;

  constructor() {

  }
}
```

e pelo arquivo `src/pages/tabs/tabs.html`:

```
<ion-tabs>
  <ion-tab [root]="tab1Root" tabTitle="Home" tabIcon="home"></ion-tab>
  <ion-tab [root]="tab2Root" tabTitle="About" tabIcon="information-circle">
                                                              </ion-tab>
  <ion-tab [root]="tab3Root" tabTitle="Contact" tabIcon="contacts"></ion-tab>
</ion-tabs>
```

- As abas de navegação são criadas pelo elemento `<ion-tabs>`. Cada `<ion-tab>` dentro do `<ion-tabs>` é um elemento da classe `NavController` (da mesma forma que `<ion-nav>`). Assim, também possuem o atributo `root`, que define a página a ser apresentada, e possuem sua própria pilha de navegação.
  - A página `tabs` não possui um conteúdo próprio, mas incorpora o conteúdo das páginas descendentes.
  - Os ícones disponíveis podem ser vistos em: http://ionicframework.com/docs/v2/ionicons/.

- As três páginas usadas na navegação `home`, `about` e `contact`, são também compostas por três arquivos cada, mas as diferenças quase que se resumem ao seu conteúdo (no arquivo com extensão HTML).

- Os arquivos `src/pages/home/home.ts`, `src/pages/about/about.ts` e `src/pages/contact/contact.ts` contêm as declarações dos componentes. As duas únicas diferenças entre eles são o nome da classe e o *template* (HTML) importado. O arquivo `home.ts` possui o seguinte conteúdo:

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';

@Component({
  selector: 'page-home',
  templateUrl: 'home.html'
})
export class HomePage {

  constructor(public navCtrl: NavController) { }

}
```

- Os seletores usados nos *decorators* estabelecem o vínculo com outros aspectos da aplicação, como a formatação no arquivo SCSS.

- Os arquivos SCSS estão praticamente vazios e devem ser usados para formatação personalizada.

- O arquivo `src/pages/home/home.html` possui o seguinte conteúdo:

```
<ion-header>
  <ion-navbar>
    <ion-title>Home</ion-title>
  </ion-navbar>
</ion-header>

<ion-content padding>
  <h2>Welcome to Ionic 2!</h2>
  <p>
    This starter project comes with simple tabs-based layout for apps
    that are going to primarily use a Tabbed UI.
  </p>
  <p>
    Take a look at the <code>app/</code> directory to add or change tabs,
    update any existing page or create new pages.
  </p>
</ion-content>
```

- O elemento `<ion-header>` é usado para criar um cabeçalho para a página, assim como o elemento `<ion-footer>` é usado para criar um rodapé. Os dois elementos devem estar na raiz da página (isto é, não podem ser descendentes de outros elementos).

- Um elemento `<ion-header>` pode conter uma única barra de navegação (`<ion-navbar>`) e uma ou mais barras de botões (`<ion-toolbar>`). O elemento `<ion-footer>` só pode conter barras de botões.

- Uma barra de navegação pode conter grupos de botões (`<ion-buttons>`) e um título (`<ion-title>`).

- O conteúdo da página, em HTML ou diretivas Ionic, é encaixado dentro do elemento `<ion-content>`. O atributo `padding` é usado para acrescentar 10px de margem interna em todas as direções. Alternativas são: `padding-vertical`, `padding-horizontal`, `padding-top`, `padding-right`, `padding-bottom` e `padding-left`.

- Existem inúmeras diretivas (elementos) Ionic que podem ser usados nas páginas como, por exemplo, as listas.
  - Uma lista é declarada pela diretiva `<ion-list>`.
  - Uma lista tem um cabeçalho (`<ion-list-header>`) e um conjunto de itens (`<ion-item>`).
  - O conteúdo básico de um item é apenas texto, mas também podemos incluir ícones nos itens de lista. Isso é feito por meio da diretiva `<ion-icon>`. A especificação do ícone que será incluído é feita por meio do atributo `name`. A lista de nomes de ícones pode ser encontrada em: http://ionicframework.com/docs/vs/ionicicons/.

- Todos esses componentes serão carregados dinamicamente e, portanto, não precisam de seletores em seus *decorators*.

13. SASS

- A formatação das páginas em Ionic deve ser feita com SASS (*Syntactically Awesome Style Sheets*), que deve, mais tarde, ser compilada em um arquivo CSS. Esse pré-processamento nos permite usar recursos bastante interessantes como a declaração de variáveis e a declaração de regras hierárquicas.
  - http://sass-lang.com.

- As regras SASS compiladas ficam armazenadas no arquivo `www/build/main.css`.

- O formato SASS permite, entre outros, as seguintes declarações:
  - Regras hierárquicas
  - Variáveis
  - Funções (*mixins*)

- Cada página em Ionic, já possui um escopo. Esse escopo aparece no arquivo `src/pages/nome_pagina/nome_pagina.scss`.
  - Por exemplo, a página `home` possui o arquivo `src/pages/home/home.scss`, que já contém a regra `page-home`, automaticamente associada ao conteúdo dessa página.

- O seguinte exemplo mostra os recursos de regras hierárquicas e de variáveis.

```
$cor: red;

page-home {
    h2 {
        color: $cor;
    }
    p {
        font-style: italic;
        background-color: $cor;
    }
}
```

- Nesse exemplo, a variável `$cor` recebeu o valor "`red`". A cada vez que essa variável for usada, o valor "`red`" será assumido. Isso facilita as mudanças, sem a necessidade de revisão de todo o código.

- As regras para os elementos `h2` e `p` estão dentro da regra do elemento `page-home` (automaticamente criado), isto é, só valem para os elementos descendentes de `page-home`.

- Se estivéssemos usando CSS puro, o código acima seria declarado da seguinte forma:

```
page-home h2 {
    color: red;
}
page-home p {
    font-style: italic;
    background-color: red;
}
```