guia definitivo do angular

Stewan Pacheco

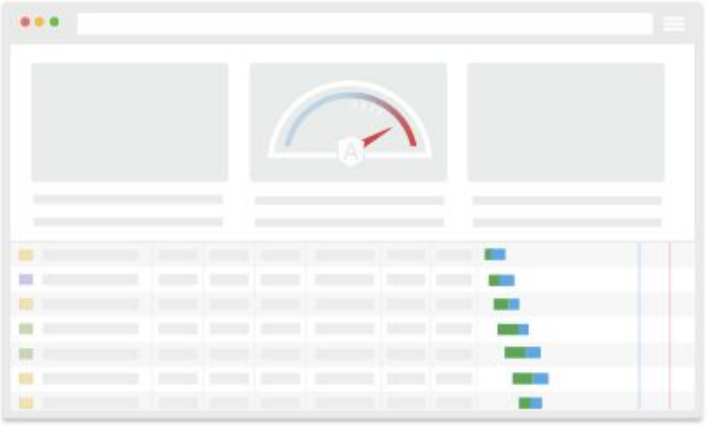
Tópicos

Introdução

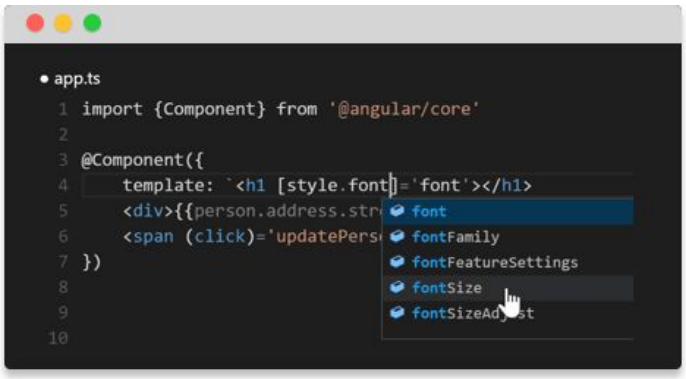
A nova versão do angular nasce para ser apenas um Framework, mobile ou desktop. Você desenvolve apenas de uma maneira, reutilizando código para qualquer destino de implementação. Seja ela web, web móvel, móvel nativa e área de trabalho nativa.

Velocidade & Performance

O novo angular consegue alcançar a velocidade máxima possível na plataforma Web atualmente, e levá-la mais longe, via Web Workers e renderização no lado do servidor.



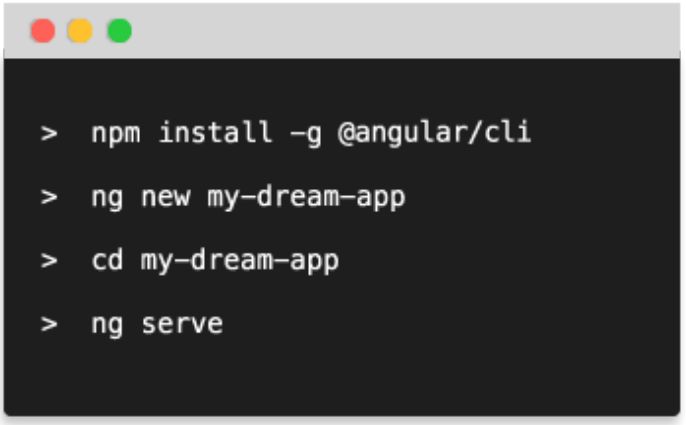
Tooling Incrível



Construa sua aplicação rapidamente de forma simples e declarativa, utilizando uma IDE específica para desenvolvimento Angular e tecnologias web. O Visual Studio Code vem com IntelliSense, Depuração de código, integração com Git e gerenciador de extensões.

Angular CLI

A interface de linha de comando do angular vai desde executar um mini servidor para rodar no browser, a gerar builds finais para produção. Além de ajudar a prototipar rapidamente novas aplicações, criar componentes, serviços e todo tipo de recurso que o angular pode proporcionar.



Porque utilizar o Angular?

Simplesmente por ser amado por **milhões de desenvolvedores** ao redor do planeta.

O Angular está presente desde aplicações simples a aplicações de grandes corporações.

Além de ser largamente utilizado em grandes aplicações do próprio Google, como o **Google Analytics**, **Adwords**, **Adsense**, **Google Trends**, etc.

Fundamentos do Angular

Aplicações Angular são feitas a partir de um conjunto de *web components*. Um web componente é a combinação de estilo CSS + *template HTML* + *classe javascript* que irá dizer ao angular como controlar uma parte da aplicação.

Além de componentes, no Angular possuímos uma série de bibliotecas (Classes) que resolvem cada uma um problema específico.

Visão Geral

Aqui está um exemplo de um componente que exibe uma simples string:

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

O resultado do exemplo acima será um simples: "Hello Angular", dentro de uma *tag* padrão **<h1>** do *html*.

Experimente executar o exemplo acima a partir deste plunker

Quando o site carregar, altere a linha

```
template: `<h1>Hello {{name}}</h1>`
```

para

```
template: `<h1>Olá {{name}}</h1>`
```

Aguarde e veja o resultado.

Nota:
Este exemplo foi escrito usando a linguagem TypeScript, que é um super conjunto (superset) do JavaScript. Angular usa TS para tornar mais fácil e dar maior produtividade a vida do desenvolvedor por meio de ferramentas para desenvolvimento. Apesar de não ser recomendado, você também pode escrever código Angular usando JavaScript puro, este guia explica como.

Analisando o código do exemplo, em linhas gerais estamos

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

importando a classe *Component* a partir do **núcleo** do angular

```
@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})
```

O trecho acima é o decorador de componente do angular. Decoradores de componente sempre vem exatamente uma linha acima da classe a ser trabalhada e eles possuem o propósito de esclarecer ao angular como este componente deve trabalhar:

Selector: irá informar ao angular qual nome deverá utilizar na tag HTML. Neste caso o resultado final seria **<my-app>**

Template: o HTML em si do componente. Neste caso uma tag **<h1>** com uma variável de template **{{name}}**

```
export class AppComponent {
  name = 'Angular';
}
```

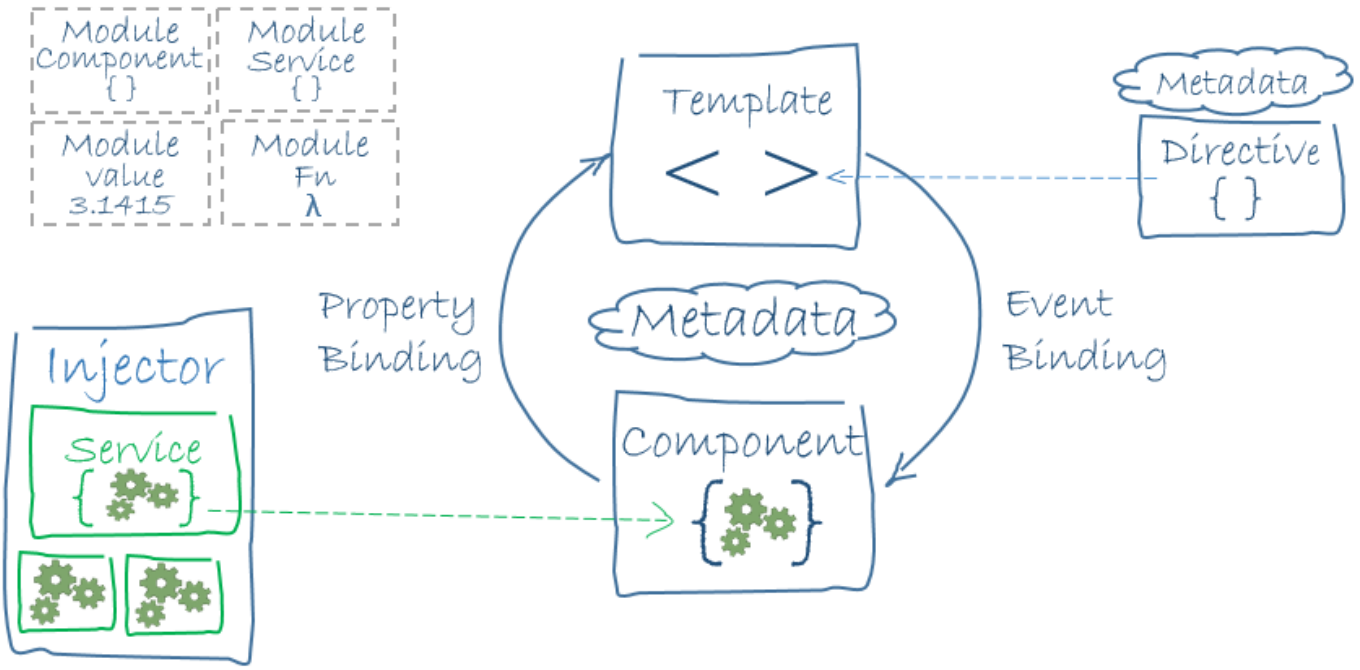
O trecho acima está exportando a classe **AppComponent** afim de que esteja disponível para importação em outros arquivos do projeto. Neste caso, a classe *AppComponent* é importada no arquivo *app.module*, verifique no plunker, lado esquerdo, clique no arquivo *app.module*.

O objetivo do *app.module* é basicamente importar todos os recursos que a aplicação irá utilizar e defini-las em um módulo – onde possamos fazer o bootstrap – ou inicialização da nossa aplicação. Esta inicialização é feita no arquivo src/*main.ts*, por meio do método *platformBrowserDynamic().bootstrapModule(AppModule)*

Voltando ao último trecho de código do exemplo, além da exportação, também é definida uma propriedade no escopo da classe, a propriedade *name* é "ligada" ao template por meio de um recurso denominado **Data Binding,** ou **Property Binding**.

É legal você conferir esta parte da documentação (em inglês), que ensina outro jeito de aprender angular. Neste ebook, vou ser o mais objetivo possível para você começar a desenvolver, sem ter que ver conceitos mais complexos por agora.

Arquitetura



Ok, WTF is this?

No angular tudo é centrado no **Componente**. Conforme você pode analisar mais ao centro do diagrama o componente angular é definido por meio de um ***Metadata***, que nada mais é que aquele objeto {} definido dentro do decorador **@Component**. Este mesmo componente possui um ***Template*** e a comunicação de dados entre a parte lógica do Componente e o *Template* é realizada por meio de *Property Bindings*, ou Ligação por meio de propriedades.

Quebrando estes termos técnicos temos:

Componente: Soma de uma classe **javascript + template HTML + Metadata**, conforme o exemplo abaixo, o metadata seria o que está entre { }

```
@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})
```

O bind de propriedade (Property Binding) seria a definição da propriedade no escopo da classe, já com valor inicial, que é ligada diretamente ao template

```
export class AppComponent {
   name = 'Angular';
}
```

Diretivas

Diretivas no angular são atributos HTML especiais que aceitam uma lógica de programação diretamente no template. Alguns exemplos básicos são:

```
<section
*ngIf="showSection==true">
```

Remove ou adiciona uma nova parte no DOM baseada em uma expressão, neste caso se showSection for true.

```
<li *ngFor="let item of list">
```

Cria um novo template dinâmico em tempo real baseado no elemento <li>, instanciando uma nova "view" para cada item da lista.

```
<div [ngClass]="{'is-active':
pagina == 'home'}">
```

Bind (ligação) de classe no html, a chave do objeto representa o nome da classe a ser aplicada, o valor do objeto representa a condição ou expressão.

```
<input [(ngModel)]="userName">
```

O famoso two-way data-binding, ou ligação de dados em duas vias, além de controle de validação dos formulários.

Consulte mais sobre outras diretivas na sessão **Built-in Directives** deste Cheat Sheet da documentação do Angular.

Serviços

```
@Injectable()
export class MyService() {}
```

Serviços são classes Singleton.

Singleton é um padrão de projeto de software (do inglês Design Pattern). Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto.

Basicamente elas servem para guardar a lógica do negócio. Para fazer uma simples alusão, imagine duas xícaras de café, estas xícaras são dois componentes angular, as duas precisam ter café, o que seria melhor, escrever o código de fazer café repetidamente nas 2 xícaras ou escrever de uma única vez um terceiro elemento chamado bule (provider)?

Com este bule poderíamos com um único código servir as duas xícaras.

O novo angular permanece com a mesma idéia de injeção de dependência da versão anterior, com a diferença de agora estarmos de fato trabalhando com Classes e orientação a objetos.

Para criar um serviço no novo angular, basta criar uma Classe e decorá-la com decorador *@Injectable()*, o angular possui vários decoradores, você pode conferir a lista completa deles no Cheat Sheet do angular. Vamos ver isto melhor na prática mais a frente.

Injeção de dependências

Exemplo do bule de café usado anteriormente na versão angular. Vamos criar um componente xícara que utiliza o provider bule para fazer café

```
// typescript
import { Component } from '@angular/core';
import { BuleService } from '../../providers/bule.service';

@Component({
  selector: 'app-xicara',
  templateUrl: './xicara.component.html'
})
export class XicaraComponent {
  // declara uma propriedade de escopo global na classe
  // usaremos ela mais a frente pra fazer a ligação dos dados com a view
  cafe:any;

  // constructor é o primeiro método executado pelo angular
  // é a inicialização do componente xícara
  // aqui instanciamos a utilização do serviço BuleService
  // na variável bule para poder ser utilizada através do this.bule
  constructor(public bule: BuleService) { }

  // comportamento de fazer cafe
  fazerCafe() {
    this.cafe = this.bule.fazerCafe();
  }
}
```

```
// html do elemento xicara
{{ cafe }}


// html do elemento pai que for usar o componente xicara
<app-xicara></app-xicara> // xicara 1
<app-xicara></app-xicara> // xicara 2
```

Injetar dependência significa que você pode reutilizar código por toda sua aplicação. Na primeira e segunda linha estamos importando duas dependências, a classe **Component** do core do angular – para ser utilizada mais abaixo como decorador em **@Component** – e a classe **BuleService** – que contém toda a lógica relacionada a fazer café – para injetar na classe **XicaraComponent**.

Começando a desenvolver

Para começar a brincar de verdade com angular é necessário, primeiramente, preparar o ambiente.

Introdução ao Node.js

Em linhas gerais o node.js é uma plataforma escrita em javascript feita para rodar código javascript. Em uma simples analogia, o Node.js seria a soma do *PHP+Apache*. é importante frisar que o Node.js roda código javascript apenas no lado do servidor.

Utilizaremos o node aqui não para criar *API's* ou renderizar *HTML* no lado do servidor. Iremos utilizar o node como ferramenta principal para desenvolvimento. Por meio dele vamos instalar a linha de comando do angular e executar nossa aplicação localmente.

Instalando o NPM

NPM vem de *Node Package Manager* ou gerenciador de pacotes do Node. O *NPM* é distribuído juntamente com a plataforma do node e é por meio dele que iremos instalar todas nossas dependências de desenvolvimento e bibliotecas.

Você pode instalar o Node/NPM de duas maneiras:

1 - Maneira tradicional diretamente do site: nodejs.org
2 - Pelo gerenciador de versões para o node, o *NVM - Node Version Manager*.

Instalando Node da maneira tradicional

Neste caso basta baixar o binário do site nodejs.org, executar e seguir os passos. Para testar se tudo está ok, digite ***npm*** no seu terminal de comando.

Dica para usuários windows

Se estiver utilizando o windows, recomendo fortemente que baixe e instale o Git for windows e utilize a linha de comando pelo git em tudo que for executar pelo terminal. Procure no menu iniciar por ***Git bash***. Dependendo da versão do windows (8), talvez seja necessário executar como administrador.

Instalando Node pelo NVM

No caso do *NVM*, você poderá ter múltiplas versões instaladas do Node ao mesmo tempo em seu sistema operacional. Este inclusive é o jeito que eu trabalho. Existem duas versões:

Para sistemas Linux Based ou Mac

https://github.com/creationix/nvm#install-script

Para sistemas Windows

https://github.com/coreybutler/nvm-windows/releases

Talvez seja necessário reiniciar o Windows para linha de comando do *NVM* funcionar

Instalando o Node pelo NVM

Após a instalação do *NVM*, abra a linha de comando do seu sistema e digite

```
nvm install latest
```

Esse comando irá instalar a última versão do node pelo *NVM*. Caso queira instalar uma versão específica, execute

```
nvm install x.x.x
```

Para listar as versões disponíveis do node

```
nvm ls
```

para verificar se está tudo OK, execute **node --version** no terminal de comando. Caso não esteja funcionando, rode o comando **nvm use x.x.x**

Consulte a documentação no github do *NVM* que estiver utilizando para maiores informações.

Instalando o Angular CLI

Uma vez instalado o Node/NPM, precisamos instalar a interface de linha de comando do Angular. Agora sim vamos começar de fato a trabalhar com node, pelo gerenciador de pacotes npm.

```
npm install -g @angular/cli@latest
```

-g *= instalar de maneira global, ou seja, a partir de qualquer diretório você poderá utilizar os comandos do angular (pela linha de comando);*
@angular *= nome da organização ou usuário;*
cli *= nome do repositório (ou projeto);*
@latest *= última versão disponível;*

Comandos da CLI do Angular

Você pode conferir a lista de todos os comandos disponíveis na documentação do Github, mas basicamente o que iremos utilizar são:

ng new

cria uma nova aplicação

exemplo:

```
ng new --help
```

exibe todas as opções para o comando **new**

```
ng new myApp
```

cria toda a estrutura inicial de diretórios e arquivos, além de instalar as dependências necessária para começar uma nova aplicação angular.

ng serve

serve a aplicação criada anteriormente para ser executada em tempo real no browser. Isto significa que tudo que você alterar no código é compilado e exibido no navegador automaticamente sem a necessidade de reload manual, ou f5. O endereço padrão é o localhost:4200

ng generate

Gera novos recursos angular para a aplicação corrente.
Estes são os esqueletos gerados atualmente:

- class
- component
- directive
- enum
- guard
- interface
- module
- pipe
- service

exemplo:

```
ng generate component banner
```

gera um novo componente de nome *banner* no caminho *src/app*, além de fazer referência automática no arquivo *app.module*. Para utilizá-lo em qualquer template da sua app, bastará chamar a tag **<app-banner>**.

Este prefixo 'app' é adicionado automaticamente pelo angular-cli para que o novo elemento não entre em conflito com tags nativas do html, por exemplo o próprio <header>. Este prefixo pode ser facilmente alterado diretamente no arquivo de configuração da CLI (.angular-cli.json), gerado ao criar um novo projeto.

ng build

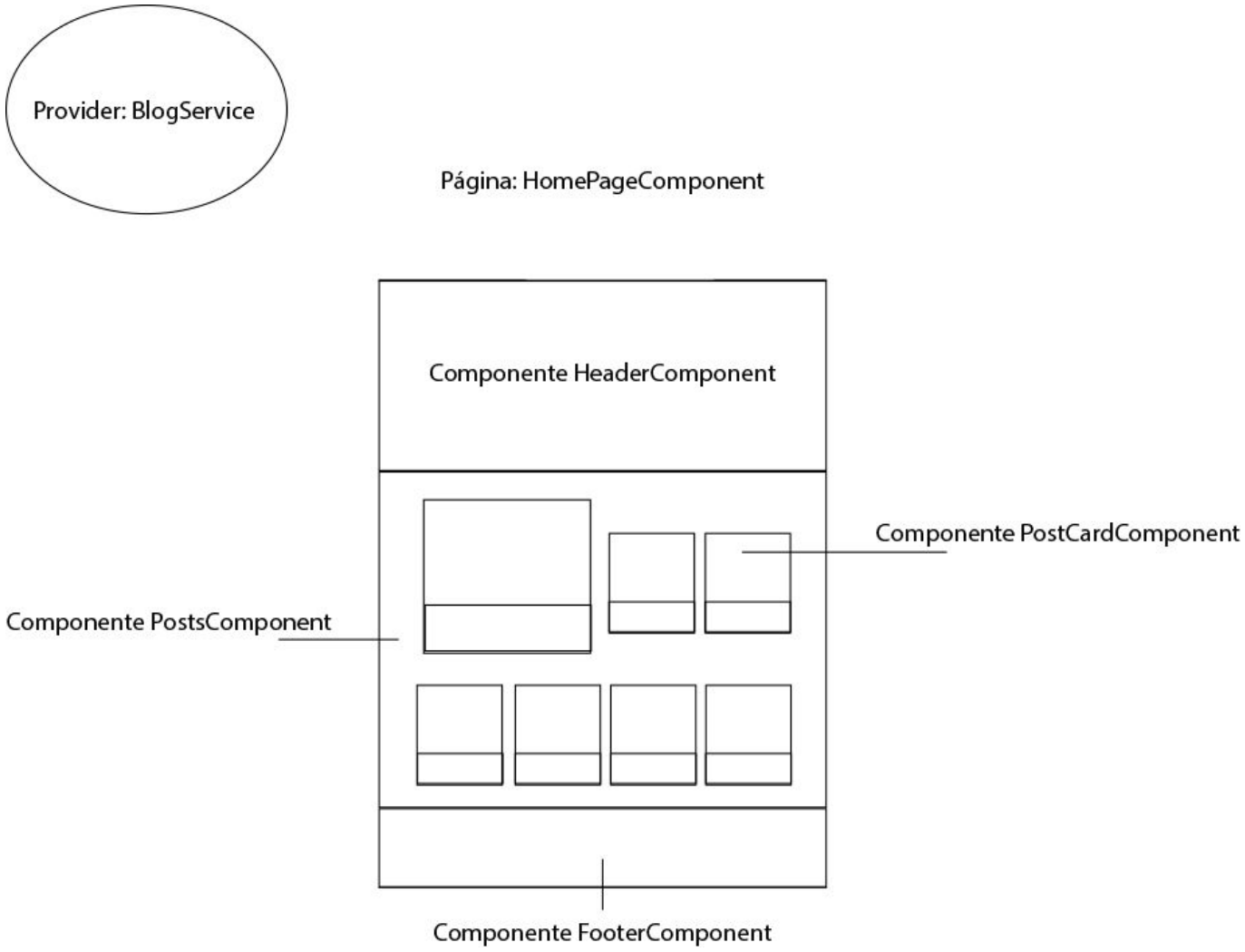
gera um "empacotamento" de todo o projeto, no formato de arquivos HTML + Javascript + CSS para ser publicado em qualquer servidor estático.

Projetando nossa aplicação

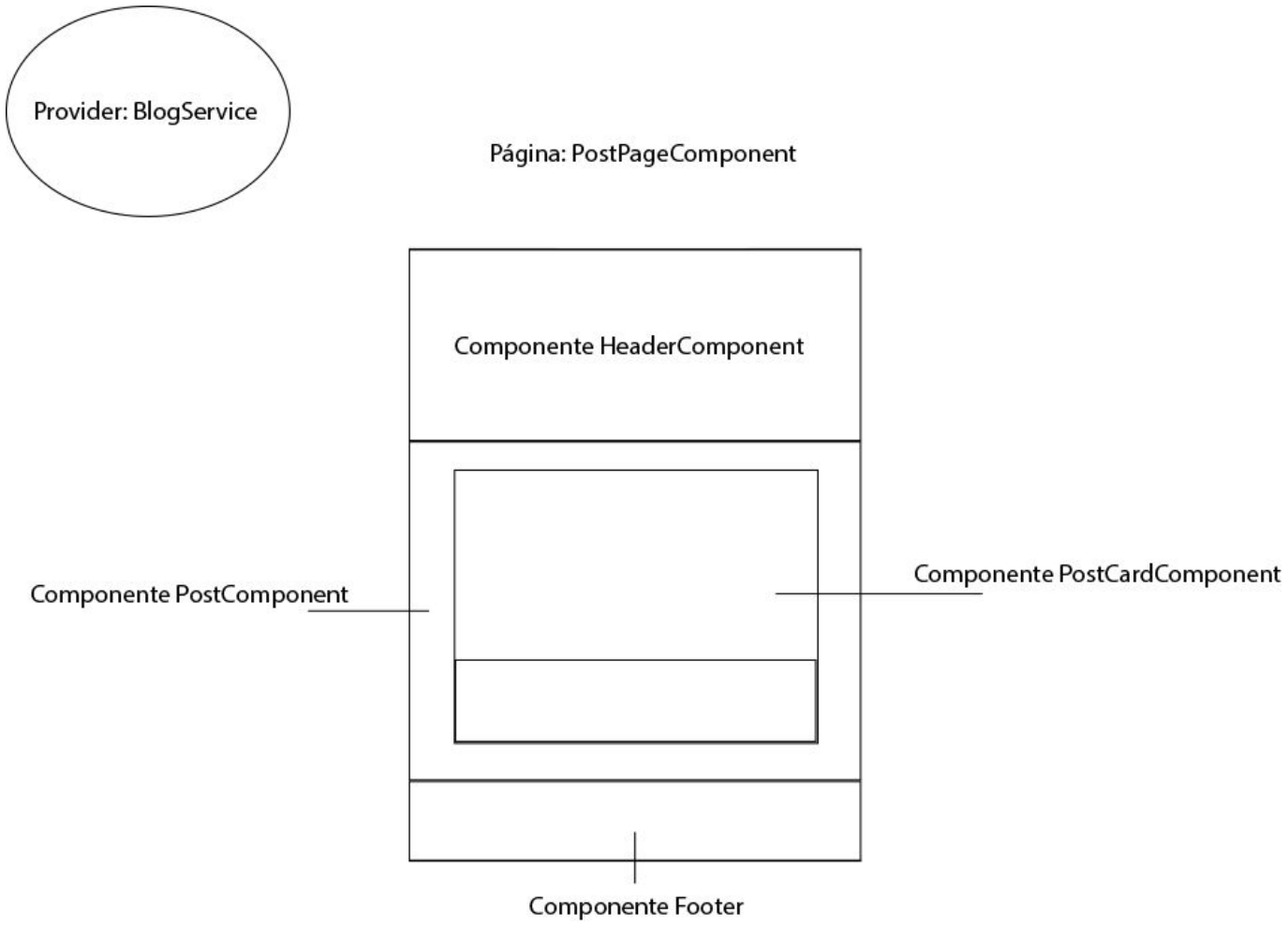
Para realmente entendermos como funcionam estes conceitos na prática, vamos rascunhar um pouco:

Página Home

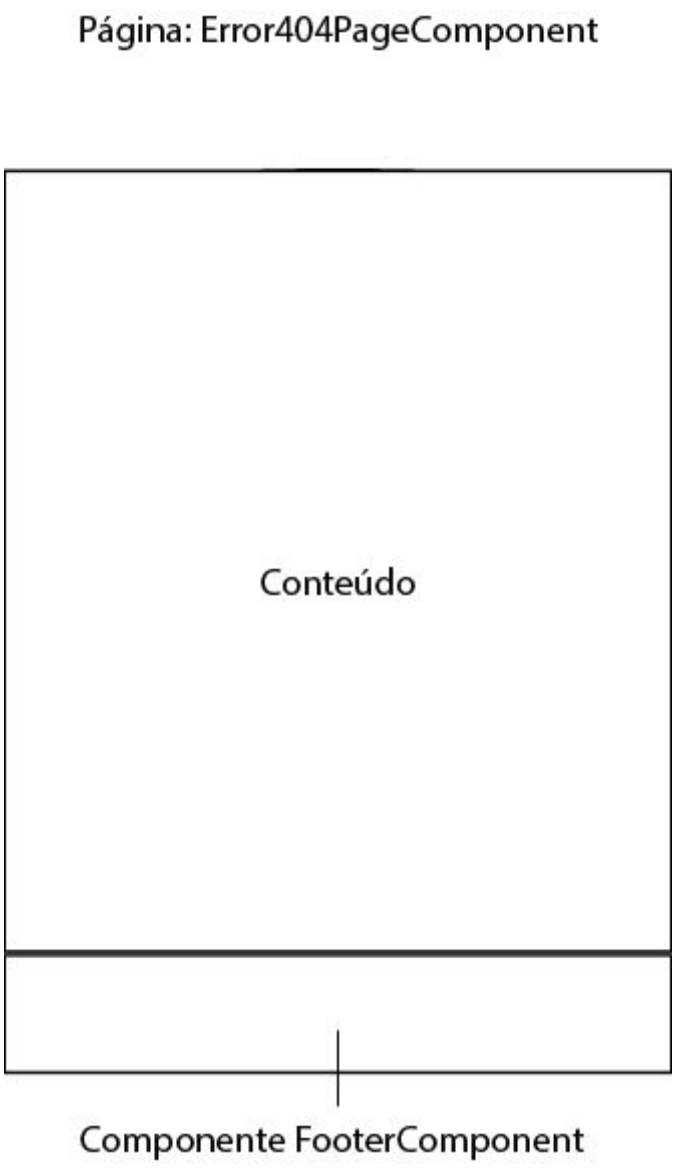
Este seria o rabisco da nossa página home, que não deixa de ser também um componente



Página Post


Provider: BlogService
Página: PostPageComponent
Componente HeaderComponent
Componente PostComponent
Componente PostCardComponent
Componente Footer

Página 404


Página: Error404PageComponent
Conteúdo
Componente FooterComponent

Mãos à obra

A partir de agora vamos ver detalhadamente passo a passo para desenvolver nossa app.

Passo 1 - Iniciando um novo projeto angular

```
ng new blog --style=sass
```

Inicia o projeto especificando o esquema de estilo que vamos utilizar, no caso o SASS.

SASS – para quem não conhece – é basicamente uma linguagem de programação para o CSS. Com ele é possível definir variáveis, criar estruturas de repetição, entre outras coisas.

Vamos utilizar esta tecnologia para poder trabalhar com o bulma, que é um framework css escrito em cima do SASS.

Após a instalação abra o projeto no seu Visual Studio Code

Analisando a estrutura criada

node_modules - todas as dependências de desenvolvimento, esta pasta é criada a partir do comando *npm install*, você precisa ter um arquivo *package.json* com as dependências definidas

src - contém todos os arquivos fontes do projeto.

src/app - Arquivos fontes contendo lógica, template, estilos e testes unitários. Tudo que você gerar pelo comando da CLI *generate*, irá para *src/app*

src/assets - Arquivos como Imagens e fontes

Nota: Não crie folhas de estilos CSS nesta pasta, cada estilo deve compor um componente. Caso precise criar um estilo que atenda sua app de maneira global, utilize o arquivo src/styles.sass

src/environments - Objeto para guardar opções baseados nos ambientes a serem executados. Você pode definir configurações de desenvolvimento no arquivo *environment.ts*, e de produção no environment.prod.ts. O angular-cli irá usar como padrão o arquivo environment para desenvolvimento, mas se na hora da build quiser usar o de produção, basta passar o sinal de produção

```
Ex: ng build --env=prod
```

O mapeamento de ambientes é feito através do arquivo de configurações .angular-cli.json localizado na raiz do projeto

src/index.html - Como estamos trabalhando uma aplicação de página única – ou SPA (Single Page App) – este é o template principal da aplicação. Dificilmente mexemos neste arquivo.

Nota: para trabalhar com várias rotas ou URL's, você deve configurar seu servidor para que todas requisições batam no arquivo index.html

src/main.ts - arquivo principal de inicialização. Aqui são importadas todas as dependências e iniciadas pelo método **platformBrowserDynamic().bootstrapModule(AppModule);**

src/styles.sass - arquivo global de estilos, utilizando o compilador SASS. Tudo que é escrito aqui ficará disponível para todos os componentes da aplicação

src/typings.d.ts - arquivo para definição de tipos do TypeScript

.angular-cli.json - arquivo de configuração da CLI do angular

package.json - arquivo global de dependências do node. Tudo que estiver aqui é baixado para a pasta node_modules ao se executar o comando *npm install*

Qualquer outro arquivo que não estiver citado, refere-se a configuração do TypeScript, testes unitários ou ferramentas de desenvolvimento que não precisamos nos preocupar por agora.

Passo 2 - Adicionando Bulma e Font Awesome

```
npm install --save bulma
```

Irá baixar na pasta **node_modules** este framework css, escrito em **SASS**, para trabalharmos melhor e de maneira rápida o visual da nossa app

--save = salva o bulma como dependência local no arquivo package.json

Para adicionar o Font Awesome copie e cole a tag <link> abaixo logo antes de fechar a tag <head> em *src/index.html*

```
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <link rel="icon" type="image/x-icon" href="favicon.ico">
 <link
href="//maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-awesome.min.css"
rel="stylesheet"
integrity="sha384-wvfXpqpZZVQGK6TAh5PVlGOfQNHSoD2xbE+QkPxCAFlNEevoEH3Sl0sibVc
OQVnN" crossorigin="anonymous">
</head>
```

Passo 3 - Configurando o bulma pra trabalhar no angular

Agora precisamos dizer ao angular para importar o bulma. Vá até o arquivo **src/style.sass** e adicione a seguinte linha no código

```
$primary: #ff3860
@import '~bulma'
```

No código acima estamos substituindo a cor da variável **$primary** e, em seguida, importando o bulma. Assim tudo que estiver escrito no bulma utilizando a variável $primary, será substituído pela cor que acabamos de definir.

Veja a lista completa de variáveis que o bulma utiliza em https://goo.gl/wyMrgb
Veja mais sobre customização do bulma em https://goo.gl/Ab9DYD

Passo 4 - Criando as páginas que irão servir as rotas

Como iremos trabalhar com rotas, primeiro devemos criar as páginas que irão ser chamadas ou utilizadas por estas rotas.

Serão 3 rotas

- Home page (/)
- Post page (/:slug/:id)
- Error 404 page (/*)
  Neste caso qualquer coisa que estiver fora dos dois primeiros casos, será error 404

Rode na sua linha de comando

```
ng g c pages/home-page
ng g c pages/post-page
ng g c pages/error-404-page
```

Nota: O nome das pages podem variar de acordo com a sua necessidade como por ex:

```
ng g c pages/about-page
```

Serão criados um componente para cada página, no caminho src/pages. Você pode utilizar o caminho e dar o nome que quiser para criar seus componentes, porém por uma questão de convenção, estamos criando aqui no caminho 'pages'. A grande questão aqui é

que estas páginas componentes são as que serão chamadas pelo componente de rota do angular, que vamos definir agora em rotas.

Passo 5 - Configurando as Rotas

Para trabalhar com rotas no angular, utilizaremos o pacote **@angular/router** , que já vem no *package.json* ao iniciar a aplicação pelo comando **ng new**

Entretanto é necessário alguns passos extras para fazê-lo funcionar:

Passo 5.1 - criar um arquivo chamado **app.router.ts** em src/app

Passo 5.2 - importar no arquivo de rotas as dependências do router e as páginas que acabamos de criar anteriormente

```
import { Routes, RouterModule } from '@angular/router';
import { HomePageComponent } from './pages/home-page/home-page.component';
import { PostPageComponent } from './pages/post-page/post-page.component';
import { Error404PageComponent } from
'./pages/error-404-page/error-404-page.component';
```

Passo 5.3 - declarar uma constante com a configuração das rotas, definindo o caminho e o componente que será utilizado para este caminho

```
const routes: Routes = [
    // home
    {
        path: '',
        component: HomePageComponent
    },
    // blog post
    {
        path: ':slug/:id',
        component: PostPageComponent,
    },
    // handling 404
    {
        path: '**', component: Error404PageComponent
    }
];
```

Passo 5.4 - exportar uma constante RoutingModule contendo o módulo de rotas configurado

```
export const RoutingModule = RouterModule.forRoot(routes);
```

Passo 5.5 - importar o `RoutingModule` no arquivo **app.module.ts**

Abra o arquivo *src/app/app.module.ts* e importe manualmente o módulo de rotas que acabamos de criar, depois informe ao angular pela array **imports[]**

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

// route module
import { RoutingModule } from './app.router';
import { AppComponent } from './app.component';
import { HomePageComponent } from './pages/home-page/home-page.component';
import { PostPageComponent } from './pages/post-page/post-page.component';
import { Error404PagesComponent } from
'./pages/error-404-pages/error-404-pages.component';

@NgModule({
  declarations: [
    AppComponent,
    HomePageComponent,
    PostPageComponent,
    Error404PagesComponent
  ],
  imports: [
    RoutingModule,
    BrowserModule,
    FormsModule,
    HttpModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Nota: não importa a ordem de importação das dependências

Passo 5.6 - chamar o componente de rota no template principal app.component.html

Abra o arquivo *src/app/app.component.html,* apague todo o conteúdo e adicione o seguinte componente

```
<router-outlet></router-outlet>
```

Com isto, o angular irá gerenciar todas as rotas pelo arquivo **app.router**

Passo 6 - Executando nossa app pra ver se tudo vai bem

Execute em sua linha de comando

```
ng serve
```

Após alguns segundos você poderá abrir seu navegador no endereço http://localhost:4200/

deverá exibir

home-page works!

Passo 7 - Criar o provider blog que servirá nossos componentes

Neste provider vamos utilizar uma biblioteca de terceiros chamada lodash _ para ilustrar a importação de bibliotecas externas no angular e reaproveitar alguns métodos desta incrível lib.

Primeiramente execute o comando abaixo no seu terminal, na raiz do projeto blog

```
npm install --save lodash
```

Agora vamos criar um novo serviço (provider) pelo angular-cli

```
ng g s providers/blog
```

g = generate
s = service
providers = pasta a ser criada pelo angular-cli
blog = serviço a ser criado pelo angular-cli

Importar e referenciar este provider no arquivo **app.module.ts** uma vez que atualmente o angular-cli não faz referência automática para providers, somente componentes.

```
// route module
import { RoutingModule } from './app.router';
// services
import { BlogService } from './providers/blog.service';
```

Agora informe o provider no array providers do @NgModule

```
@NgModule({
  ...
  providers: [BlogService],
  bootstrap: [AppComponent]
})
```

Abra o arquivo *src/app/providers/blog.service.ts*, apague todo o conteúdo e cole o seguinte código. Leia atentamente aos comentários.

```
// importa lodash como _
import * as _ from 'lodash';
// usado para decorar serviços angular
import { Injectable } from '@angular/core';
// usado para navegarmos entre rotas
import { Router } from '@angular/router';
// usado para requisições GET, POST, etc
import { Http } from '@angular/http';

// declara uma constante base url
const BASEURL = window.location.href;

@Injectable() // decora a classe BlogService para ser "injetável" dentro do
angular
// exporta nossa classe para ser "importável" rs, por outros arquivos
export class BlogService {

  // declara uma variável de escopo global na classe com a url do serviço
externo que iremos consumir. experimente copiar e colar esta url no seu
navegador.
```

```
  getUrl: string = 'https://jsonplaceholder.typicode.com/posts';
  constructor(
    public http: Http, // injetando serviço Http do angular
    public router: Router // injetando serviço router do angular
  ) { }

  // pegar a lista de posts
  posts() {
    // retorna uma promessa que quando resolvida irá conter a lista de posts
    return new Promise((resolve, reject) => {
      // utiliza o método get do http que injetamos mais acima passando como
opção a url que irá conter nossos posts. Poderia ser qualquer serviço externo
que devolve uma lista no formato JSON
      this.http.get(this.getUrl).subscribe((data: any) => {
        // declara uma variável local posts
        let posts = JSON.parse(data._body);
        // loop nos posts pra criar umas propriedades extras
        posts.map((post, i) => {
          // título no-formato-de-slug
          posts[i].titleSlug = _.kebabCase(post.title);
          // a rota deste post
          posts[i].router = '/' + posts[i].titleSlug + '/' + posts[i].id;
          // a url deste post
          posts[i].url = BASEURL + posts[i].router;
        });
        resolve(posts); // resolve a lista de posts
      }, (err) => {
        reject(err); // rejeita a promessa com o erro
      });
    })
  }
```

```
  // pegar um post especifico, passando como parametro o id do post
  post(id: any) {
    return new Promise((resolve, reject) => {
      // pegamos a lista de posts
      this.posts().then((posts: any[]) => {
        // filtramos procurando pelo post especifico
        let post = _.find(posts, (p) => {
          return p.id == id;
        });
        // se tiver post resolve, senão rejeita (erro 404)
        return post ? resolve(post) : reject('post not found');
      });
    })
  }
}
```

Para verificar se tudo está ok no momento, basta não haver nenhum erro no navegador (chrome f12 console) ao executar **ng serve**

Passo 8 - Criar o componente header e chamá-lo na home page

Agora vamos colocar algum visual e dar vida a nossa app.

Primeiramente execute o comando

```
ng g c components/header
```

Será criado um novo componente no caminho *src/app/components/header*

Abra o arquivo *src/app/components/header.component.html* apague todo o conteúdo e adicione o html

```
<section class="hero is-danger is-medium">
  <div class="hero-body">
    <div class="container">
      <h1 class="title">
        Angular
      </h1>
      <h2 class="subtitle">
        Bulma Blog
      </h2>
    </div>
  </div>
</section>
```

salve o arquivo, depois abra *src/app/pages/home-page/home-page.component.html*

apague todo o conteúdo e adicione

```
<app-header></app-header>
```

Salve e veja o resultado em seu navegador com **ng serve**

Passo 9 - Criando o componente post card

Agora criaremos nosso componente post-card, execute o comando abaixo

```
ng g c components/post-card
```

Abra o arquivo *src/app/components/post-card.component.ts* para adicionarmos um decorador novo, o decorador de propriedade @Input(). Com este decorador podemos *inputar* (dar entrada) de dados no nosso componente a partir de um componente pai.

importe a classe *Input* do core do angular

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

depois declare uma propriedade decorada na classe, chamada "post" do tipo "any"

```
export class PostCardComponent implements OnInit {
  @Input() post: any;
  constructor() { }
  ngOnInit() { }
}
```

Agora abra o template do componente, apague todo o conteúdo e cole o html abaixo

```
<div class="card">
  <div class="card-image">
    <figure class="image is-4by3">
      <a [routerLink]="post.router">
        <img src="http://bulma.io/images/placeholders/1280x960.png"
alt="Image" />
      </a>
    </figure>
  </div>

  <div class="card-content">
    <div class="media">
      <div class="media-content">
        <h2 class="title is-4"><a [routerLink]="post.router">{{ post.title
}}</a></h2>
      </div>
    </div>

    <div class="content">
      <div [innerHTML]="post.body">
        <!-- aqui vira o conteudo de post.body -->
      </div>
      <br>
      <small>11:09 PM - 1 Jan 2016</small>
    </div>
  </div>
</div>
```

No código acima, além de html, temos 3 caras diferentes

```
1) <a [routerLink]="post.router">
```

O **[routerLink]** é uma diretiva nativa do angular router usada para navegar entre rotas. Neste caso, o post.router está sendo gerado lá no mapeamento dentro do provider blog.service. No final este <a> se parecerá com algo do tipo <a href="/lorem-ipsum/1">

```
2) {{ post.title }}
```

Aqui o angular está dando saída no dado que foi ligado a partir da variável post, que acabamos de definir mais acima com **@Input**. Isso permitirá com que nosso componente receba dado de cima, a partir do atributo post

exemplo:

```
// ts
const objetoDoPost =
      {
            title: 'Lorem ipsum'
      }
// html
<app-post-card [post]="objetoDoPost"></app>
```

```
3) <div [innerHTML]="post.body"></div>
```

[innerHTML] é uma diretiva nativa do angular para que possamos inserir código html dentro de um elemento html. Neste caso o vai receber exatamente o que existir em post.body que pode ser um código html.

Passo 10 - Criando o componente posts e utilizando o provider Blog

Agora a coisa fica mais legal. Vamos criar o componente pai **posts** que irá utilizar em seu template o componente **post-card**, já consumindo os dados a partir do provider **blog.service**

Doidera?
vamos lá

```
ng g c components/posts
```

Trabalhando a lógica

Abra o arquivo *src/app/components/posts.component.ts*

Importe o BlogService

```
import { BlogService } from '../../providers/blog.service';
```

Declare uma propriedade para controlarmos o estado de carregamento

```
export class PostsComponent implements OnInit {
  loading: boolean = true; // já começara com estado 'carregando'
  constructor() { }
  ngOnInit() {
  }
}
```

Declare uma propriedade para guardar a lista de posts que vamos obter pelo provider

```
export class PostsComponent implements OnInit {
  loading: boolean = true;
  posts: any[];
  constructor() { }
  ngOnInit() {
  }
}
```

Injete o serviço do blog no construtor da classe do nosso novo componente

```
export class PostsComponent implements OnInit {
  loading: boolean = true;
  posts: any[];
  constructor(public blog: BlogService) { }
  ngOnInit() {
  }
}
```

Agora vamos escrever um código que irá pegar os dados (posts) por meio do *hook* **ngOnInit(),** que é o método que o angular executa depois do **constructor()**

```
 ngOnInit() {
    // aqui vamos chamar o método posts do provider/serviço blog que criamos
anteriormente
    // que quando for resolvido retornará um array (lista) de posts do tipo
any
    this.blog.posts().then((posts: any[]) => { // acessamos this.blog pois
instanciamos no constructor desta classe a variável blog do tipo BlogService
      setTimeout(() => { // damos um tempo pra exibir o resultado no template
        this.posts = posts; // ligamos o resultado na view (template)
        this.loading = false; // paramos o carregamento
      }, 1 * 1000) // define o tempo de carregamento pra 1 segundo
    })
  }
```

Trabalhando o Template

Copie e cole o HTML abaixo em *src/app/components/posts/posts.component.html*

```
<section class="posts">
  <br /><br />
  <div class="container">
    <div class="heading">
      <h2 class="subtitle">
        <span [hidden]="loading != true">Carregando...</span> Latest
<strong>fake</strong> news
      </h2>
    </div>

    <div class="columns is-desktop is-multiline">
        <app-post-card class="column" *ngFor="let post of posts; let i =
index" [ngClass]="{'is-half':i==0||i%11==0, 'is-one-quarter':i>0&&i%11!=0}"
[post]="post"></app-post-card>
    </div>

  </div>
  <br /><br />
</section>
```

Neste HTML Você encontra quatro caras diferentes

1)

```
<span [hidden]="loading!=true">Carregando...</span>
```

O [hidden] é uma diretiva interna do angular para trabalhar o estado de visão dos elementos. Neste caso, estamos escondendo o elemento sempre que a propriedade loading da classe for diferente de true, ou seja, false. Em outras linhas, se loading for true, mostra o com o texto "Carregando..."

2)

```
*ngFor="let post of posts; let i = index"
```

Aqui estamos *loopando* o componente **post-card** que criamos anteriormente, de acordo com os dados que vieram do provider Blog. O angular irá criar uma variável local **post** para cada repetição, juntamente com uma variável contadora **i**

3)

```
[ngClass]="{'is-half':i==0||i%11==0, 'is-one-quarter':i>0&&i%11!=0}"
```

Outra diretiva nativa do angular, o **[ngClass]** define nomes de classe css dinamicamente, de acordo com a expressão passada. Neste caso, nosso **post-card** terá a classe **is-half** sempre que o contador **i** for igual a **zero** ou se o mod (resto da divisão) de i por 11 for igual a **zero**. Isto dará o efeito de a cada 11 cards, o primeiro card seja maior que os outros. A mesma lógica se aplica para a classe **is-one-quarter**, só muda a expressão.

O **is-half** e o **is-one-quarter** são classes de grid do bulma.
Documentação: http://bulma.io/documentation/grid/columns/

4)

```
[post]="post"
```

Esta é uma diretiva que criamos quando definimos o **@Input** na classe do componente **post-card**. Aqui estamos passando o objeto "post" criado dinamicamente pela diretiva ***ngFor** vista anteriormente

Agora basta anexar o elemento *<app-posts></app-posts>* ao template da página home

Abra o arquivo *src/app/pages/home-page/home-page.component.html* e adicione

```
<app-header></app-header>
<app-posts></app-posts>
```

Salve e veja o resultado no seu navegador pelo comando **ng serve**

Passo 11 - Criando o componente footer e adicionando a home page

Execute o comando

```
ng g c components/footer
```

Copie e cole o template abaixo em src/app/pages/footer.component.html

```
<footer class="footer">
  <div class="container">
    <div class="content has-text-centered">
      <p>
        <strong>Angular Bulma Blog</strong>
        <br />
        por <a href="https://stewan.io">Stewan Pacheco</a> <a class="icon"
href="https://github.com/stewwan">
          <i class="fa fa-github"></i>
        </a>
      </p>
      <p>
        <strong>Curtiu o projeto?</strong><br /><a
href="https://stewan.io/ebooks/guia-definitivo-do-angular">Baixe meu ebook de
angular</a> e veja passo a passo + código fonte
      </p>
    </div>
  </div>
</footer>
```

Chame na home page

Abra o arquivo *src/app/pages/home-page/home-page.component.html*

Adicione o html

```
<app-header></app-header>
<app-posts></app-posts>
<app-footer></app-footer>
```

Adicione um estilo ao footer

Abra o arquivo *src/app/components/footer/footer.component.sass*

adicione o estilo abaixo

```
.footer
    padding-top: 50px
    border-top: 1px solid #eee
    background-color: #fff

a.icon
    margin-top: 5px
    display: inline-block
    line-height: 20px
```

Salvou **ng serve** abraço!, home page **concluída**

Passo 12 - Criando o componente post para página interna

Este componente irá usar o provider *blog.service* para exibir um post baseado no id passado pela rota (url)

Devido as configurações de rota que fizemos no Passo 5.3, uma url como esta

http://localhost:4200/dolorem-dolore-est-ipsam/54

cairia no componente **post-page** criado no Passo 4, e o componente **post-page** usaria o componente **post** que criaremos agora:

Execute o comando no seu terminal

```
ng g c components/post
```

Trabalhando a lógica

Abra o arquivo gerado em src/app/components/post/post.component.ts e substitua pelo código

```
// importando dependencias do angular core
import { Component, OnInit } from '@angular/core';
// importando dependencias de rota
import { Router, ActivatedRoute } from '@angular/router';
// importando o provider do blog
import { BlogService } from '../../providers/blog.service';

// decorando o componente para informar como o angular deve trabalhar
@Component({
  selector: 'app-post',
  templateUrl: './post.component.html',
  styleUrls: ['./post.component.sass']
})
// exportando a classe deste componente para ser importável por outros
arquivos (no caso, o app.module.ts)
export class PostComponent implements OnInit {

// declarando uma variável objeto do tipo any que irá armazenar o post assim
que for resolvido pelo provider
  post: any = {};

  constructor(
    // instanciando a rota ativa
    public activatedRoute: ActivatedRoute,
    // instanciando o roteador
```

```
    public router: Router,
    // instanciando nosso provider de blog
    public blog: BlogService) {
    // chama a rota ativa
    this.activatedRoute
      .params
      // se inscreve para quando for resolvida
      .subscribe(
      // recebe os parametros
      params => {
        // declara uma variável id baseada no parametro id da url
        let id = params['id'];
        // pego o post em questão
        this.blog.post(id)
          // quando resolvido
          .then((post: any) => {
            // liga os dados na view (template)
            this.post = post;
          })
          // tratamento de erro
          .catch((e) => {
            console.error(e);
            // força navegação para página 404
            this.router.navigate(['/404']);
          })
      });
  }
  ngOnInit() { }
}
```

Trabalhando o template

Abra o arquivo *src/components/post/post.component.html* e substitua pelo código HTML abaixo

```
<section class="posts">
  <br /><br />
  <div class="container">
    <div class="heading">
      <h2 class="subtitle">
       <a [routerLink]="'/'">Blog</a> > {{ post.title }}
      </h2>
    </div>
    <br />
    <div class="columns">
        <app-post-card class="column" [post]="post"></app-post-card>
    </div>
  </div> <br /><br />
</section>
```

Agora substitua o HTML abaixo em *src/app/pages/post-page/post-page.component.html*

```
<app-header></app-header>
<app-post></app-post>
<app-footer></app-footer>
```

Adicionando um estilo ao componente post

Abra o arquivo *src/app/components/post/post.component.sass*

```
.heading
    margin: 20px
.container
    max-width: 980px
```

Boa, salve e tente acessar um post pelo navegador usando **ng serve.**

Página interna do post concluída **\o/**

Passo 13 - Customizando a página de Erro 404

Abra o arquivo *src/app/pages/error-404-page/error-404-page.component.html* e substitua pelo código abaixo

```
<section class="hero is-warning is-fullheight">
  <div class="hero-body">
    <div class="container">
      <h1 class="title">
        Page not found
      </h1>
      <h2 class="subtitle">
        <a href="/">Back to home</a>
      </h2>
    </div>
  </div>
</section>
<app-footer></app-footer>
```

Salve e para testar se está ok, abra seu navegador em um endereço qualquer como http://localhost:4200/asdf

Booom!!!

Build para produção

Quando estamos desenvolvendo com angular, mexemos puramente com arquivos **SASS** e **Typescript**. Entretanto, nenhum navegador entende esses tipos de arquivos, por isso é necessário gerar uma build final de tudo isso no formato em que o navegador possa entender.

Rode o comando

```
ng build --prod
```

Todo o projeto será compilado e exportado para a pasta **dist** minificados em arquivos puramente **javascript** e **css**. Com isto basta que você hospede o conteúdo desta pasta em qualquer servidor estático.

Hospedando no Github Pages

O Github além de ser um repositório mundial de código fonte dos mais variados tipos, é também um mega servidor estático gratuito. Podemos hospedar qualquer arquivo HTML, CSS, Javascript e imagens, de forma que estejam disponíveis na web por meio de um endereço. Para ilustrar este cenário, vamos hospedar nosso blog na seguinte url

http://seu_usuario.github.io/nome_repositorio

Se ainda não tem, abra uma conta gratuita no Github. Depois crie um novo repositório com qualquer nome.
Agora altere uma configuração no *angular-cli* para que ao invés de ele gerar a build na pasta **dist**, gere em uma pasta específica, denominada "docs".

Mas porque "docs" Stevão?
Haha pergunte ao Github.

Abra o arquivo *.angular-cli.json*, e na opção "**outDir**" altere de "**dist**" para "**docs**".

Salve o arquivo e execute novamente o comando

```
ng build --prod --bh=/nome_respositorio/
```

Nota: Para usuários windows existe um bug atualmente discutido aqui, onde não é inserido corretamente o **base href** no arquivo index.html pela opção --bh. Um workaround é você abrir manualmente o arquivo em **docs/index.html** e corrigir deixando exatamente assim:

```
<base href="/nome_respositorio/">
```

Agora é preciso commitar as alterações e fazer o *push* para o github.

Se ainda não está familiarizado com o git, vamos pelo começo:

execute o comando abaixo na raiz do projeto para iniciar um repositório git

```
git init
```

adicione o endereço de um repositório remoto

```
git remote add origin https://github.com/seu_usuario/nome_repositorio
```

Você pode testar se tudo foi bem, rodando o comando

```
git remote show origin
```

Ele deverá mostrar o *remote* `origin` registrado.

Grave as alterações e envie para o *remote* que acabamos de criar:

```
git add -A
git commit -am "primeiro commit"
git push origin master
```

Pode acontecer de pedir seu login e senha do Github. Agora precisamos apenas dizer ao Github Pages para "ouvir" a pasta docs. Abra as configurações do repositório (Settings) e mais abaixo na aba **GitHub Pages** selecione a opção **Master branch /docs folder**.

Boooom!!!, aguarde alguns segundos e abra seu navegador no endereço

http://seu_usuario.github.io/nome_do_repositorio

Trabalhando SEO no Angular

Como é de se esperar, sites feitos puramente com javascript não são indexados por mecanismos de busca. Atualmente para resolver este problema existem três possibilidades:

1 - Servir uma versão totalmente html “pré renderizada” especialmente para os bots.

Esta é a opção que utilizo em todos meus sites angular, incluindo meu blog. Porém requer conhecimentos e passos extras, caso queira se aventurar, recomendo começar pelo serviço prerender.io que possui plano inicial *free*.

2 - Construir sua app utilizando **angular universal**

Angular Universal era, até pouco tempo, um projeto a parte da comunidade, que atualmente está mesclado no repositório principal do angular, mas ainda possui muitos problemas. Seria basicamente você construir sua app para funcionar de duas maneiras ao mesmo tempo: no browser e no servidor. Assim, o usuário baixaria a versão client no browser dele e o bot receberia a versão do servidor, já renderizada na forma em que ele precisa ler.

3 - Trabalhar um SEO básico

Uma vez que todos os bots caem no mesmo arquivo, o index.html, podemos então trabalhar um SEO básico da seguinte forma:

Abra o arquivo src/index.html e adicione as seguintes meta tags

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Blog</title>
  <base href="/">
  <meta name="title" content="Angular Bulma Blog">
  <meta name="og:title" property="og:title" content="Angular Bulma Blog">
  <meta name="description" content="Um blog construído com angular e bulma">
  <meta name="og:description" property="og:description" content="Um blog
construído com angular e bulma">
  <meta name="og:image" property="og:image"
content="https://stewwan.github.io/angular-bulma-blog/assets/guia-angular.png
">
```

Pronto, agora ao menos os bots vão ter alguma informação da sua página. Quando o site crescer, recomendo fortemente que utilize a técnica número 1.

Treta do 404 no Github Pages

Ao trabalhar com Angular no Github Pages ou qualquer servidor estático existe um grande problema a se resolver. Se você fez tudo certo até aqui, experimente acessar o endereço

https://seu_usuario.github.io/seu_repositorio/qui-est-esse/2

Deverá aparecer a página 404 do Github.

WTF? Este é um comportamento completamente normal pois quando trabalhamos com rotas em servidores estáticos, caso a rota (url) em questão não exista de fato fisicamente falando, no caminho requisitado, o servidor tende a exibir a página de erro 404.

No caso está sendo exibida a página padrão do Github, mas se você quiser pode exibir sua própria página 404, basta adicionar um arquivo 404.html na raiz da build que o angular gerou, /docs (ou /dist).

E é fazendo isso que vamos hackear essa treta, bastando copiar o arquivo index.html para 404.html gerado em /docs, commitar a alteração e enviar novamente pro Github.

Agora acesse novamente a url do post pra ver se está tudo ok

https://seu_usuario.github.io/seu_repositorio/qui-est-esse/2

Download do código fonte

Código fonte do projeto disponível para download em
https://github.com/stewwan/angular-bulma-blog/archive/master.zip

Repositório no Github
https://github.com/stewwan/angular-bulma-blog/

Créditos

- Angular
- Angular CLI
- Angular CheatSheet
- Bulma
- Lodash
- Font awesome
- Github Pages
- Node
- Node Version Manager Win / Linux

Obrigado por estar aqui

Neste ebook tentei abordar de forma simples e objetiva, os fundamentos básicos para se começar a trabalhar com o novo Angular de maneira organizada, escalada e seguindo as melhores práticas.

Espero que tenha gostado, comente seu feedback na página do ebook, não deixe de compartilhar com seus amigos.

Se curtiu deste trabalho, siga-me também pelas redes sociais para ficar por dentro das novidades.

Github: https://github.com/stewwan
Twitter: https://twitter.com/StewanPacheco
Instagram: https://www.instagram.com/stewan.io
Facebook: https://fb.me/Stewan.io
Youtube: https://goo.gl/ZaCr71

Abs,
Stewan.