

**HTML5 e AngularJS**  
Veja como criar aplicações web  
dinâmicas baseadas em templates

**Django e Python**  
Curso de microblog  
com o Django e extensões

# MATERIAL DESIGN LITE

**Conheça o novo framework  
de componentes do Google**



The image shows a grid of cards, likely from a mobile application. The grid is organized into three columns and four rows. The first column contains a green card with a gold trophy icon, a blue card with a vinyl record icon, and a grey card with a 'leaderboard' icon. The second column contains a light green card with a food icon, a cyan card with a music icon, and a white card with a profile icon. The third column contains a light blue card with a history icon, a light green card with a science and nature icon, and a purple card with a TV and movies icon. Each card has a small arrow pointing to its right.

Food & Drink	General Knowledge	History
Music	Profile	Science and Nature
leaderboard	→	TV & Movies

# MVP

R\$ 1.000.000,00  
INVESTIDOS EM CONTEÚDO  
NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO  
NA SUA CARREIRA...

E MOSTRE AO MERCADO  
QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL  
QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's  
consumido + de **500.000** vezes

POR APENAS  
**R\$ 69,90\*** mensais

\*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR  
MAIS DO MERCADO!



 **DEVMEDIA**

## EXPEDIENTE

### Editor

Diogo Souza ([diogosouzac@gmail.com](mailto:diogosouzac@gmail.com))

### Consultor Técnico

Daniella Costa ([daniella.devmedia@gmail.com](mailto:daniella.devmedia@gmail.com))

### Produção

**Jornalista Responsável** Kaline Dolabella - JP24185

**Capa e Diagramação** Romulo Araujo

### Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse [www.devmedia.com.br/central](http://www.devmedia.com.br/central), ou se preferir entre em contato conosco através do telefone 21 3382-5038.

### Publicidade

[publicidade@devmedia.com.br](mailto:publicidade@devmedia.com.br) – 21 3382-5038

**Anúncios** – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

### Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



**DIOGO SOUZA**

[diogosouzac@gmail.com](mailto:diogosouzac@gmail.com)

Analista de Sistemas Java na Indra Company e já trabalhou em empresas como Instituto Atlântico e Ebix L.A. É instrutor Android, palestrante em eventos sobre Java e o mundo mobile e consultor DevMedia. Conhecimentos e experiências em diversas linguagens e ferramentas de programação e manipulação de dados, bem como metodologias úteis no desenvolvimento de Sistemas diversificados.

# Sumário

### Conteúdo sobre Boas Práticas, Artigo no estilo Solução Completa

#### 04 – Criando aplicação de notícias com AngularJS

[ Júlio Sampaio ]

### Conteúdo sobre Boas Práticas, Conteúdo sobre Novidades

#### 15 – Introdução ao Google Material Design Lite

[ Júlio Sampaio ]

### Artigo no estilo Curso

#### 26 – Aplicação de MicroBlog com Django e Python - Parte 2

[ Júlio Sampaio ]

# Criando aplicação de notícias com AngularJS

Aplicações web flexíveis e customizáveis, baseadas em templates dinâmicos, usando o framework do Google

Com o advento do MVC, muito se ouviu falar nas vantagens que esse novo padrão traria para os projetos e suas respectivas e consequentes produtividade e facilidade. Por muito tempo, entretanto, esse era um conceito aplicado tão-somente ao universo server side, uma vez que o front-end era sempre feito visando muito mais estrutura (HTML) que divisão em camadas, negócio e responsabilidades programáticas. Esse paralelo fez muita gente desacreditar do AngularJS no começo, uma vez que o mesmo vinha exatamente para quebrar esse tabu e injetar mais lógica no cliente, muito mais, evitando que execução desnecessária fosse enviada ao servidor que se encarrega, agora, das funções mais importantes da aplicação, como salvar dados, fazer integrações ou validações mais pesadas que dependam de outros componentes, por exemplo.

Mais que isso, o AngularJS permite estender a HTML convencional e adicionar views dinâmicas de modo a criar um vocabulário próprio. Desenvolvido e mantido pelo Google, o projeto segue à risca o padrão MVC, onde as views são especificadas usando o HTML + AngularJS da própria linguagem de template; os modelos e controladores, por sua vez, são especificados através de objetos e funções do JavaScript. Dentre outras vantagens, ele permite organizar melhor o seu código JavaScript (uma vez que as diretivas permitem abstrair muito do código que faríamos em um projeto normal), criar aplicações responsivas e rápidas, otimizadas para o universo HTTP bem como facilmente integráveis com outros frameworks JavaScript, como o jQuery.

Neste artigo faremos um overview acerca dos seus principais recursos através do desenvolvimento de uma aplicação de cadastro de notícias para um site, desde a configuração do ambiente, a criação do serviço no servidor, até os testes e muito mais.

## Fique por dentro

Este artigo é útil por explorar os principais conceitos do framework AngularJS do Google através da construção de uma aplicação de cadastro de notícias completa, expondo a forma como ele lida com a integração entre HTML e JavaScript, bem como os envios de requisição a Web Services ou outros serviços externos à mesma. Com esse conteúdo, o leitor estará apto não só a criar suas próprias aplicações web, como também criar facilmente fachadas de integração para o universo mobile e front-end.

## Concepção do projeto

O nosso projeto parte do princípio de que o leitor já tem conhecimentos básicos de HTML, CSS e JavaScript, bem como da utilização de ferramentas de edição de código, como Notepad++, Sublime, etc.

Ele basicamente será composto por dois projetos principais: um projeto feito em Java Web, usando a tecnologia Restful da API JAX-RS para fornecer os métodos de comunicação back-end que a nossa aplicação precisará consumir no cliente. Esse tipo de comunicação pode ser feito em qualquer plataforma que o leitor se sentir à vontade, desde que o mesmo tenha conhecimentos sobre Web Services o suficiente para abstrair toda a camada de negócio do sistema; e um segundo projeto que será o principal, no caso o próprio AngularJS.

Para a primeira parte, referente ao back-end, faremos uso das seguintes tecnologias/ferramentas:

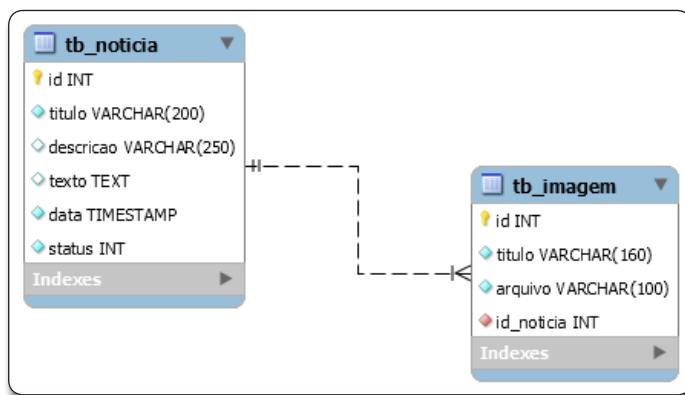
- Java JDK, em sua versão 7 ou superior;
- Eclipse for Java EE Developers (versão para web), em sua versão Mars ou superior;
- Servidor Tomcat (da Apache), na sua versão 7 - em zip (já estamos na versão 8.0, mas para os propósitos do artigo a 7 é mais que suficiente);
- Banco de Dados MySQL e gerenciador gráfico Workbench;
- Notepad++ para a edição dos arquivos do client side.

Para a concepção do banco de dados o leitor também pode optar por qualquer outro banco de sua preferência, lembrando que não mostraremos aqui detalhes de abstração dessa camada. O mesmo vale para a linguagem de programação e plataforma de comunicação dos serviços, que pode ser facilmente substituída por Node.js, por exemplo.

Na **Figura 1** temos a representação do modelo Entidade Relacionamento no MySQL do nosso projeto em questão. Veja que temos apenas duas tabelas para representar todo o salvamento de dados do projeto, a saber:

- **tb\_noticia**: se encarrega de salvar todos os dados inerentes a uma notícia do portal, com campos como título, descrição, texto completo da notícia (que pode inclusive já ser salvo como texto HTML), a data da sua publicação e um status para verificar se a mesma encontra-se disponível ou bloqueada;
- **tb\_imagem**: será associada à tabela **tb\_noticia** para salvar todas as imagens que possam existir na mesma, já que é na base de dados que guardaremos esse tipo de conteúdo. Contém campos de título, descrição e do arquivo propriamente dito, além da coluna de chave estrangeira que estabelece o relacionamento de um-para-muitos com a referida tabela.

Lembrando que o leitor precisa ainda exportar esse modelo para a base de dados física propriamente dita. Para isso, poderá fazer de duas formas: i) através da opção de menu *Database > Forward Engineer*, vai clicando em *Next* até o fim; ii) através da importação do arquivo de script .sql que se encontra disponível no pacote de fontes deste projeto e execução do mesmo no seu banco. No exemplo, criamos a base com o esquema de nome “News\_devmedia”, mas pode dar o nome que melhor desejar.



**Figura 1.** Modelo ER da base de dados do projeto de notícias

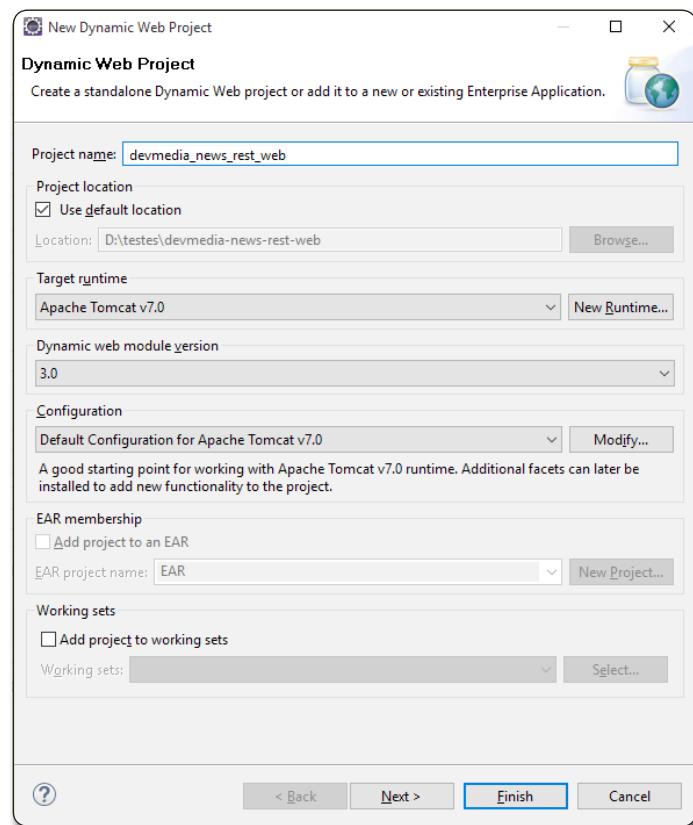
## Mãos à obra

Para início de conversa, precisamos configurar uma estrutura básica que permita ao usuário da aplicação se autenticar e, assim, filtrar quem pode ou não acessar o conteúdo da mesma. Para não aumentar a complexidade da implementação vamos abstrair a parte de acesso a banco e criação de login ou criptografia de dados para essa parte da aplicação. Em vez disso, vamos usar um login fixo, o qual pode ser facilmente modificado pelo leitor ao final deste artigo.

Mas antes disso, precisamos assegurar que o ambiente esteja devidamente configurado. Para os objetivos deste artigo é preciso que o leitor tenha as respectivas ferramentas informadas na seção anterior configuradas, não mostraremos aqui os passos para não perder o foco, mas podemos encontrar facilmente os passos no site oficial de cada uma. Com tudo ok, abra o seu Eclipse, selecione um workspace e vamos criar um projeto web, já que nosso Web Services precisa estar hospedado no formato web em algum servidor (certifique-se de importar o servidor do Tomcat para o Eclipse também, no formato zip – basta descompactar).

Para isso, vá até o menu *File > New > Dynamic Web Project* e, uma vez com o Tomcat já importado no ambiente, dê um nome ao projeto (*devmedia\_news\_rest\_web*), selecionando as opções a seguir, tal como mostra a **Figura 2**:

- **Target Runtime: Apache Tomcat v7.0**;
- **Dynamic web module version: 3.0**;
- **Configuration: Default Configuration for Apache Tomcat v7.0**.



**Figura 2.** Wizard de criação de projeto web no Eclipse

Clique em *Next* duas vezes e na terceira tela marque a checkbox “Generate web.xml deployment descriptor”, finalizando a configuração. Após isso, precisamos converter o nosso projeto para um projeto de tipo Maven (ferramenta de gerenciamento de dependências, bibliotecas e build de projetos) para, assim, não termos de nos preocupar com a manutenção das libs do projeto de forma manual. Portanto, clique com o botão direito sobre o

# Criando aplicação de notícias com AngularJS

projeto e selecione a opção *Configure > Convert to Maven Project* e pronto. Isso será o bastante para que um novo arquivo pom.xml seja adicionado à raiz do projeto, esse é o arquivo que usaremos para inserir nossas dependências.

Dê um duplo click no mesmo e uma interface gráfica será aberta, com várias abas na parte inferior da IDE. Selecione a aba “pom.xml” (a última) e insira o conteúdo apresentado na **Listagem 1**.

**Listagem 1.** Conteúdo do arquivo de configuração pom.xml.

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04     http://maven.apache.org/xsd/maven-4.0.0.xsd">
05   <modelVersion>4.0.0</modelVersion>
06   <groupId>br.com.edu.devmedia.news_rest</groupId>
07   <artifactId>devmedia_news_rest_web</artifactId>
08   <version>0.0.1-SNAPSHOT</version>
09   <packaging>war</packaging>
10  <build>
11    <sourceDirectory>src</sourceDirectory>
12    <plugins>
13      <plugin>
14        <artifactId>maven-compiler-plugin</artifactId>
15        <version>3.3</version>
16        <configuration>
17          <source>1.8</source>
18          <target>1.8</target>
19        </configuration>
20      </plugin>
21      <plugin>
22        <artifactId>maven-war-plugin</artifactId>
23        <version>2.6</version>
24        <configuration>
25          <warSourceDirectory>WebContent</warSourceDirectory>
26          <failOnMissingWebXml>false</failOnMissingWebXml>
27        </configuration>
28      </plugin>
29    </plugins>
30  </build>
31  <dependencies>
32    <dependency>
33      <groupId>org.glassfish.jersey.containers</groupId>
34      <artifactId>jersey-container-servlet</artifactId>
35      <version>2.22.1</version>
36    </dependency>
37    <!-- Required only when you are using JAX-RS Client -->
38    <dependency>
39      <groupId>org.glassfish.jersey.core</groupId>
40      <artifactId>jersey-client</artifactId>
41      <version>2.22.1</version>
42    </dependency>
43    <dependency>
44      <groupId>org.glassfish.jersey.media</groupId>
45      <artifactId>jersey-media-moxy</artifactId>
46    </dependency>
47  </dependencies>
48 </project>
```

Trata-se de um arquivo simples que traz apenas algumas configurações a nível de projeto e build, bem com as três dependências que precisaremos para criar o nosso serviço. Nas linhas 4 e 5 temos os ids de grupo e artefato do projeto, respectivamente,

usados pelo Maven para gerir a forma como nosso projeto será exportado no futuro. Na linha 7 definimos que queremos exportá-lo como um projeto web, em formato war (*web archive*). Na linha 9 definimos em que diretório estarão nossas classes de código fonte, e da linha 10 a 27 configuramos dois plugins necessários ao projeto: o primeiro para definir o nível do compilador para o projeto (1.8) e o segundo para definir a versão de módulo web que estaremos trabalhando (2.6, mas podemos usar qualquer outra mais recente até a 3.1). Por fim, nas linhas 29 a 47 definimos três dependências: as APIs do Jersey para Restful servlet e cliente, e a API do Jersey Media Moxy, que nos auxiliará a enviar e receber objetos no lado JavaScript em formato JSON quando chegarmos na parte do AngularJS.

O Jersey é uma API do Java para criar Web Services de forma fácil e rápida, com poucas configurações e totalmente baseados em *annotations*. É a escolha ideal para o tipo de projeto que estamos criando, pois simplificará tudo no final, além de ser muito semelhante à forma como o AngularJS mapeia as coisas no lado cliente.

Após copiar o conteúdo salve o arquivo e aguarde até que o Maven baixe todas as dependências necessárias (precisaremos de uma conexão com a internet sem proxy para isso). Para conferir se o procedimento todo funcionou, vá até o menu *Properties > Java Build Path > Libraries > Maven Dependencies* e veja todas as libs baixadas.

O próximo passo consiste em configurar o Jersey para que seja reconhecido pelo projeto. Para isso, vá até a pasta *Java Resources > src* e clique com o botão direito, selecione a opção *New > Package* e dê o nome *br.edu.devmedia.entidade*. Clique em *Finish*. Logo após, abra o arquivo *web.xml* que está dentro do diretório *WebContent > WEB-INF* e adicione o conteúdo da **Listagem 2** ao mesmo.

Essas são as configurações do Servlet do Jersey que precisamos mapear para informar ao contexto da aplicação web quem se encarregará de receber as requisições do AngularJS e quem devolverá as respectivas respostas. Trata-se da classe *ServletContainer*. Perceba que dentro dessa configuração estamos passando também um parâmetro de inicialização chamado *jersey.config.server.provider.packages* que recebe o exato nome do pacote que acabamos de criar. Ao fazer isso, estamos dizendo ao Jersey que todas as classes de Web Services estarão contidas dentro deste pacote, por isso não devemos anotar nenhuma outra classe externa ao mesmo como receptora de Web Services.

O segundo parâmetro de inicialização (linha 18) serve para dizer ao mesmo que trabalharemos com JSON e que o mesmo deve fazer a conversão automática de objetos para JSON e vice-versa. No final do arquivo, na linha 24, configuramos o padrão de URLs que usaremos para redirecionar para esse Servlet, com o valor */rest/\**. Isso significa que somente as requisições que tiverem esse padrão na URL serão aceitas por nosso serviço. Pronto, nosso projeto está configurado para receber nossas classes de serviços.

Como nosso serviço só tratará de requisições para login no momento, vamos criar uma nova classe e pacote de entidades para receber os objetos que precisaremos converter do JavaScript para o Java.

---

**Listagem 2.** Conteúdo do arquivo de configurações web.xml.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xmlns="http://java.sun.com/xml/ns/javaee"
04   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
05   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
06   id="WebApp_ID" version="3.0">
07   <display-name>devmedia_news_rest_web</display-name>
08   <welcome-file-list>
09     <welcome-file>index.html</welcome-file>
10   </welcome-file-list>
11   <servlet>
12     <servlet-name>Jersey REST Service</servlet-name>
13     <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
14     <init-param>
15       <param-name>jersey.config.server.provider.packages</param-name>
16       <param-value>br.edu.devmedia.rest</param-value>
17     </init-param>
18     <init-param>
19       <param-name>com.sun.jersey.api.json.POJOMappingFeature
19       </param-name>
20       <param-value>true</param-value>
21     </init-param>
22     <load-on-startup>1</load-on-startup>
23   </servlet>
24   <servlet-mapping>
25     <servlet-name>Jersey REST Service</servlet-name>
26     <url-pattern>/rest/*</url-pattern>
27   </servlet-mapping>
28 </web-app>
```

O primeiro deles será o objeto *Usuario* que conterá as informações de login e senha. Portanto, vá até a pasta *src* e clique com o botão direito selecionando a opção. No campo *Package* informe o valor *br.edu.devmedia.entidade* e no campo *Name* informe *Usuario*. Quando for criado e aberto, adicione o conteúdo da **Listagem 3** ao mesmo. Nessa listagem a única novidade é a anotação *@XmlRootElement* que pertence à API do Jersey e serve para fazer a conversão automática para objeto Java quando esse objeto vier do AngularJS em formato JSON. Além disso, criamos um atributo booleano *logado* para mapear se o usuário está ou não logado no sistema no momento.

O próximo passo é criar a nossa classe de serviço, de fato. Vá até o pacote de terminação *.rest* que criamos, clique com o botão direito e selecione *New > Class*, dê o nome *LoginService* à mesma e clique em *Finish*. Adicione o conteúdo da **Listagem 4**.

Nessa classe já começamos a ver as anotações do Jersey aparecerem com maior intensidade. A primeira delas, na linha 11, é a *@Path* que é responsável por mapear qual URI (URL interna) deverá ser chamada na URL principal para que a requisição seja direcionada para a classe em questão. No método *logarUsuario()* representado na linha 17 estamos mapeando-o com três anotações: *@POST* se encarrega de definir o tipo de método HTTP que esse método Java receberá (o que significa que outros métodos não serão aceitos, como GET ou PUT); *@Produces* define que tipo de conteúdo esse método retornará para o cliente (no caso JSON); e a *@Consumes* define o que ele consumirá, também conteúdo em formato JSON.

---

**Listagem 3.** Código da classe de entidade *Usuario*.

```
package br.edu.devmedia.entidade;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Usuario {

    private String usuario;
    private String senha;

    private boolean logado;

    public String getUsuario() {
        return usuario;
    }

    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }

    public String getSenha() {
        return senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }

    public boolean isLogado() {
        return logado;
    }

    public void setLogado(boolean logado) {
        this.logado = logado;
    }
}
```

---

**Listagem 4.** Código da classe de serviço *LoginService*.

```
01 package br.edu.devmedia.rest;
02
03 import javax.ws.rs.Consumes;
04 import javax.ws.rs.POST;
05 import javax.ws.rs.Path;
06 import javax.ws.rs.Produces;
07 import javax.ws.rs.core.MediaType;
08
09 import br.edu.devmedia.entidade.Usuario;
10
11 @Path("/login")
12 public class LoginService {
13
14     @POST
15     @Produces(MediaType.APPLICATION_JSON)
16     @Consumes(MediaType.APPLICATION_JSON)
17     public Usuario logarUsuario(Usuario usuario) {
18         if ("admin".equals(usuario.getUsuario())) {
19             usuario.setLogado(true);
20         } else {
21             usuario.setLogado(false);
22         }
23
24         return usuario;
25     }
26 }
```

Note que nossa validação focará apenas em validar se o nome de usuário que recebemos por parâmetro será igual a `admin`, e retornar o mesmo objeto de usuário preenchido com tal informação na flag `logado`. Esse é todo o código back-end que precisamos para fazer o login funcionar. Para testar se o mesmo está funcionando, basta adicionar o projeto no servidor Tomcat, iniciar o mesmo e executar a URL em uma ferramenta de testes do tipo, como o SoapUI, por exemplo.

## Criando o cliente da aplicação

Trabalhar com o AngularJS é demasiadamente simples e para evitar aumento de complexidade no projeto vamos importar todos os arquivos de script usando o recurso de CDN (*Content Delivery Network*). Este nos fornece os arquivos em um servidor escalável em tempo real a qualquer momento (na página oficial do framework - seção **Links**). Para isso, a primeira providência é criar a nossa página HTML (sim, com o AngularJS podemos executar tudo em HTML sem a necessidade de extensões server

side, como `.jsp`, `.php`, etc.). Vá até a pasta `WebContent`, clique com o botão direito e selecione *New > HTML File*. Dê um nome (`index.html`) ao arquivo e clique em *Finish*. Adicione o conteúdo da **Listagem 5** ao mesmo.

Vejamos alguns detalhes acerca dessa listagem:

- Logo na primeira linha do código temos um atributo novo sendo exibido na tag `<html>`, trata-se do `ng-app`. Esse atributo, pertencente ao AngularJS, se encarrega de configurar uma aplicação Angular na nossa página. O framework trabalha com essa divisão de responsabilidades via aplicações, onde em um só projeto podemos ter várias dessas. Entretanto, cada uma funciona como um módulo do sistema e, por isso, precisamos nos certificar de mapeá-los corretamente no JavaScript (veremos mais adiante);
- Na linha 6 importamos o arquivo de estilo CSS do Bootstrap, que usaremos para incutir estilo na aplicação, sem ter de criar o nosso do zero;
- Na linha 9 criamos o nosso controller do AngularJS (via atributo `ng-controller`) de nome `loginController`. Este será responsável por

**Listagem 5.** Página index.html para login de usuário.

```
01 <html ng-app="app">
02   <head>
03     <title>Painel Administrativo - Login</title>
04     <meta charset="utf-8">
05
06     <link rel="stylesheet" href="http://getbootstrap.com/dist/css/
07       bootstrap.min.css"/>
08   </head>
09   <body>
10     <div ng-controller="loginController">
11       <div class="container">
12         <div class="row">
13           <div class="col-xs-12">
14             <div class="page-header">
15               <h3>Login do Portal</h3>
16             </div>
17           </div>
18
19         <div class="row">
20           <div class="col-xs-12">
21             <form class="form-horizontal" ng-submit="efetuarLogin()">
22               <div class="form-group">
23                 <label for="inputEmail3" class="col-sm-2 control-label">E-mail
24                   </label>
25                   <div class="col-sm-10">
26                     <input type="text" class="form-control" id="inputEmail3"
27                       placeholder="Informe o seu e-mail"
28                         ng-model="login.usuario" required>
29                   </div>
30               </div>
31               <div class="form-group">
32                 <label for="inputPassword3" class="col-sm-2 control-label">Senha
33                   </label>
34                   <div class="col-sm-10">
35                     <input type="password" class="form-control" id="inputPassword3"
36                       placeholder="Informe a sua senha"
37                         ng-model="login.senha" required>
38                   </div>
39               </div>
40             </form>
41           </div>
42         </div>
43       </div>
44
45     <!-- Modal -->
46     <div id="modal-validacao" class="modal fade" role="dialog">
47       <div class="modal-dialog">
48
49         <!-- Modal content-->
50         <div class="modal-content">
51           <div class="modal-header">
52             <button type="button" class="close" data-dismiss="modal">&times;;
53             </button>
54             <h4 class="modal-title">Alerta!</h4>
55           </div>
56           <div class="modal-body">
57             <p>{{txtModal}}</p>
58           </div>
59           <div class="modal-footer">
60             <button type="button" class="btn btn-default" data-dismiss="modal">
61               Fechar</button>
62           </div>
63         </div>
64       </div>
65
66     <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.5/
67       angular.min.js"></script>
68     <script src="js/app.module.js"></script>
69     <script src="js/loginController.js"></script>
70     <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/
71       jquery.min.js"></script>
72     <script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/
73       bootstrap.min.js"></script>
74   </body>
75 </html>
```

nos prover os métodos e objetos HTTP e de escopo da página, o qual, consequentemente, retornará os objetos dinâmicos e nos permitirá mapear valores e recuperar os mesmos dinamicamente via JavaScript;

- Das linhas 11 a 16 criamos o cabeçalho da página usando as classes CSS do Bootstrap;
- Das linhas 19 a 42 criamos o formulário de autenticação da aplicação. Veja que na linha 21 implementamos nosso form com o atributo *ng-submit* que se encarrega de informar qual função JavaScript do AngularJS será responsável por lidar com o envio do formulário. No AngularJS usamos o conceito de modelos para mapear os campos e valores referentes a uma entidade via JavaScript. Para isso, precisamos mapear cada campo de formulário (input, select, etc.) com a propriedade *ng-model*, passando o objeto e o atributo interno, como fariam em uma aplicação OO convencional (linhas 25 e 31);
- Das linhas 46 a 64 criamos o HTML de uma modal usando o estilo do Bootstrap, bem como suas classes. Veja que dessa vez usamos alguns atributos novos, como o *role* ou *data-dismiss*, mas estes pertencem ao framework do Bootstrap e não ao AngularJS. A grande novidade dessa parte está no atributo do AngularJS da linha 53, o *txtModal*. Ele está entre chaves duplas `{}` em detrimento da exigência do framework em mapear variáveis dessa forma. Assim, quando precisarmos enviar qualquer mensagem de notificação para a página basta acessar o controller passando tal valor;
- Finalmente, nas linhas 66 a 70 efetuamos os imports dos arquivos de script dos frameworks do AngularJS (na sua versão 1.4.5, a mais recente até o momento da publicação), jQuery (versão 1.11.3), Bootstrap (versão 3.3.5), além dos scripts de aplicação e controller que criaremos a seguir.

Agora que criamos o HTML inicial, estamos aptos a entender como o AngularJS trabalha a nível de JavaScript para validar o formulário e enviar a requisição para o nosso Web Service no lado do servidor. O primeiro passo é criar o novo arquivo `app.module.js` dentro da pasta `WebContent/js` (se ela não existir, criar). Insira o código a seguir no mesmo:

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

Esse código é bem conhecido, pois trata-se de uma implementação simplista da criação de um novo módulo de aplicação no AngularJS. A variável deve ser criada sempre fora de qualquer bloco de inicialização para permitir que sua visualização seja feita de forma global. O segundo parâmetro do módulo é um vetor com os dados de inicialização (opcionais).

Após isso, crie também um segundo arquivo de nome `loginController.js` dentro do mesmo diretório e adicione o conteúdo da **Listagem 6** ao mesmo.

Na primeira linha temos a chamada à função `controller()` do AngularJS, que basicamente mapeia um novo controller ao já existente módulo `app` (criado na listagem anterior), passando o

nome como primeiro argumento, e uma função anônima como segundo parâmetro. Esta função recebe dois argumentos também: o `$scope` que se refere ao objeto de escopo do módulo, isto é, tudo que estiver dentro da div que mapeamos como o controller estará também disponível neste objeto via JavaScript; e o `$http`, que é um objeto implícito para efetuar operações via HTTP (GET, POST, etc.).

#### **Listagem 6.** Conteúdo JavaScript do controller de login.

```
01 app.controller('loginController', function($scope, $http) {  
02  
03   $scope.login = {  
04     usuario : "",  
05     senha : ""  
06   };  
07  
08   $scope.efetuarLogin = function() {  
09     if ($scope.login.usuario == "" || $scope.login.senha == "") {  
10       alert("Informe usuário e senha!");  
11       return;  
12     }  
13  
14     $http.post('/devmedia_news_rest_web/rest/login', $scope.login).success  
15       (function(data) {  
16         console.log(data);  
17  
18         if (data.logado) {  
19           window.location = "painel-inicial.html"  
20         } else {  
21           $('#modal-validacao').modal("show");  
22           $scope.txtModal = "Usuário/Senha inválidos!";  
23         }  
24       });  
25   };
```

Dentro do objeto de escopo temos também o *model* que criamos (`login`) na HTML. Dentro dele precisamos definir quais serão os atributos (`usuario` e `senha`), resetando-os. Após isso, na linha 8, mapeamos no mesmo escopo a função que efetuará o login de fato. Nela, fazemos primeiro uma validação simples para ver se os campos foram preenchidos, exibindo uma mensagem de validação (via alert) caso contrário. Aparentemente esse código parece desnecessário, já que definimos os campos na tela como *required* que, nos browsers mais recentes, obrigam o preenchimento dos mesmos. Mas lembre-se que o usuário pode estar acessando sua aplicação de um browser antigo, sem suporte ao HTML5, além de poder perfeitamente inspecionar os mesmos campos e remover tal atributo, fazendo com que o campo não seja mais obrigatório. Portanto, sempre garanta as validações em mais de uma via, para permitir que a aplicação esteja o mais segura possível.

Em seguida, na linha 14 chamamos a função `post()` no objeto `$http` que é bem semelhante à que temos no jQuery. O primeiro argumento se refere à URL de acesso ao serviço que está hospedado na aplicação web, via Jersey. Para esse tipo de implementação, considerando-se que já estamos dentro do mesmo escopo de aplicação, não precisamos referenciar a URL completa (como em `http://localhost:8080/devmedia_news_rest_web/rest/login`), apenas a URI

# Criando aplicação de notícias com AngularJS

interna relativa ao projeto. Como segundo parâmetro, enviamos o objeto de login que está no escopo do AngularJS, já preenchido automaticamente pelo próprio framework. Esse *binding* de informações é padrão no AngularJS, portanto, não precisamos nos preocupar em como ele o realiza.

Logo depois, chamamos a função *success()* que se encarrega de executar o código de sucesso do retorno dessa requisição. Nessa mesma função, recebemos um objeto *data* que corresponde ao *response* do *request*. Nele teremos os dados de retorno do Web Service que, como vimos, estará no formato JSON. O JavaScript já tem suporte nativa a esse tipo de formato de dados, logo, não teremos problema quanto ao acesso e conversão dos nossos. Na linha 15 logamos as informações no Console do navegador, apenas para averiguar se a resposta retornou com sucesso. Na linha 17 testamos se a informação do objeto retornou que o usuário está logado e navegamos para a próxima página de boas-vindas, caso positivo. Caso contrário, mapeamos o objeto de modal (linha 20) e setamos o valor da variável do AngularJS *txtModal* para uma mensagem de usuário e senha inválidos. Pronto, esse é todo o código fonte que precisamos no lado JavaScript para fazer o exemplo funcionar.

Agora só precisamos criar a página de boas-vindas para ter para onde navegar quando o exemplo finalizar.



Figura 3. Tela de login da nossa aplicação

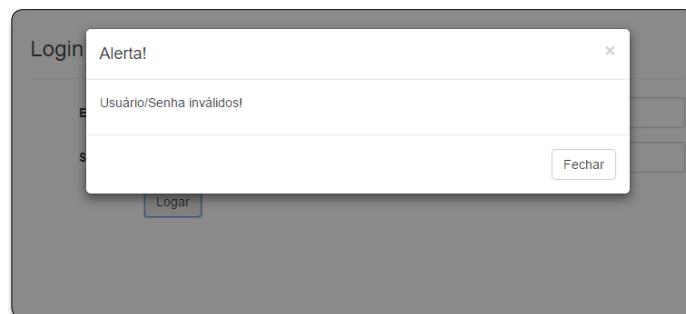


Figura 4. Tela de boas-vindas (logado com sucesso)

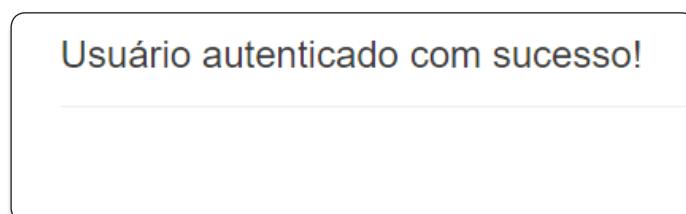


Figura 5. Modal com mensagem de notificação

Para isso, vá até o diretório *WebContent/js* novamente e crie uma nova página HTML de nome *painel-inicial.html* e adicione o conteúdo da **Listagem 7** à mesma. Nela não temos muitas novidades, exceto pelo import do Bootstrap e de um título provisório de cabeçalho.

Para testar tudo, suba a aplicação web no servidor, inicie-o e execute a seguinte URL no browser: [http://localhost:8080/devmedia\\_news\\_rest\\_web/](http://localhost:8080/devmedia_news_rest_web/). O resultado pode ser conferido na **Figura 3**.

Também é aconselhado que o leitor habilite a ferramenta do desenvolvedor no browser para analisar se tudo foi carregado corretamente a nível de JavaScript. Se nenhum erro aparecer nossas configurações estão ok. Lembre-se que fixamos a validação do usuário para o valor *admin*, logo, para testar precisamos informar esse usuário seguido de uma senha qualquer.

Preencha os dados do formulário e clique em Logar para ver uma tela igual à da **Figura 4** (caso de login com sucesso).

Caso a autenticação falhe, por exemplo, informar um usuário inválido, teremos o mesmo resultado demonstrado na **Figura 5**.

## Cadastrando novas notícias

Agora que nosso login está funcional podemos focar na implementação do cadastro de novas notícias na base. Vamos começar então preparando o terreno no lado do servidor, especificamente criando a nova entidade *Noticia* que conterá os atributos e métodos de acesso às propriedades do banco finais. Para isso, crie uma nova classe de nome *Noticia*, no pacote de entidade, e acrescente o conteúdo da **Listagem 8** à mesma.

A classe não traz nenhuma novidade e todos os atributos foram mapeados como sendo do tipo String. Mesmo a data estando no formato Timestamp no banco de dados, faremos a conversão na hora de salvar na base. Veja também que usamos mais uma vez a anotação *@XmlRootElement* para fazer a conversão do valor na hora de enviar via JSON.

## Listagem 7. Conteúdo da página HTML de boas-vindas.

```
<html ng-app="app">
<head>
<title>Home</title>
<meta charset="utf-8">
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.css">
</head>
<body>
<div class="container">
<div class="row">
<div class="col-xs-12">
<div class="page-header">
<h3>Usuário autenticado com sucesso!</h3>
</div>
</div>
</div>
</body>
</html>
```

---

**Listagem 8.** Nova entidade Noticia.

```
package br.edu.devmedia.entidade;

import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlRootElement;
public class Noticia {

    private int id;
    private String titulo;
    private String descricao;
    private String data;
    private String texto;

    // Get's e set's omitidos
}
```

No mesmo pacote, crie um novo enum para guardar os status da notícia, conforme desenhamos no banco também. Acrescente o seguinte conteúdo ao mesmo:

```
public enum Status {
    ATIVA, INATIVA
}
```

Esses valores serão úteis quando precisarmos salvar uma nova notícia na base. Para finalizar o lado servidor crie uma nova classe de serviço, no pacote *br.edu.devmedia.rest*, de nome *NoticiaService* e acrescente o conteúdo demonstrado na **Listagem 9**.

Na linha 20 da listagem temos as configurações de caminho (@Path) que fizemos no mapeamento anterior. Dessa vez, especificamente no método *logarUsuario()*, também estamos usando a mesma anotação, uma vez que teremos mais de uma método na classe de serviço. Quando isso acontece, precisamos acessar o caminho completo acessando a URL dessa forma: *localhost:8080/aplicação/noticia/new*. O restante do procedimento de receber o parâmetro é igual ao que já fizemos.

Em seguida, a partir da linha 27, começamos as configurações de acesso ao banco de dados MySQL. Aqui, usaremos o JDBC do Java, portanto, duas configurações iniciais se fazem necessárias:  
1. Acrescentar a dependências do conector do MySQL para o Java no pom.xml, como mostra a **Listagem 10**;  
2. Criar uma classe de conexão com o banco que criamos antes, retornando objetos do tipo java.sql.Connection, como mostra a **Listagem 11**.

Nessa classe, basicamente passamos a URL de conexão (que termina com o nome do nosso banco criado), bem como o usuário e senha do MySQL (certifique-se de substituir por suas credenciais).

Voltando à **Listagem 9**, na linha 29 criarmos um objeto de formatação de datas, que se encarregará de converter a data recebida em formato String para java.sql.Date (reconhecível pelo banco). O restante do código apenas insere uma nova linha na tabela TB\_NOTICIA, fazendo uso do enum de Status que criamos para setar a notícia com ativa. Note que o método não

---

**Listagem 9.** Classe de serviço para as notícias.

```
01 package br.edu.devmedia.rest;
02
03 import java.sql.Connection;
04 import java.sql.Date;
05 import java.sql.PreparedStatement;
06 import java.sql.SQLException;
07 import java.text.DateFormat;
08 import java.text.ParseException;
09 import java.text.SimpleDateFormat;
10
11 import javax.ws.rs.Consumes;
12 import javax.ws.rs.POST;
13 import javax.ws.rs.Path;
14 import javax.ws.rs.core.MediaType;
15
16 import br.edu.devmedia.config.DatabaseConfig;
17 import br.edu.devmedia.entidade.Noticia;
18 import br.edu.devmedia.entidade.Status;
19
20 @Path("noticia")
21 public class NoticiaService {
22
23     @POST
24     @Consumes(MediaType.APPLICATION_JSON)
25     @Path("/new")
26     public void logarUsuario(Noticia noticia) {
27         Connection con = DatabaseConfig.getConnection();
28
29         DateFormat format = new SimpleDateFormat("dd/MM/yyyy");
30
31         try {
32             PreparedStatement stm = con.prepareStatement(
33                 "INSERT INTO TB_NOTICIA(TITULO, DESCRICAO, TEXTO, DATA, STATUS)
34                 VALUES(?, ?, ?, ?, ?)");
35             stm.setString(1, noticia.getTitulo());
36             stm.setString(2, noticia.getDescricao());
37             stm.setString(3, noticia.getTexto());
38
39             Date data = new Date(format.parse(noticia.getData()).getTime());
40             stm.setDate(4, data);
41             stm.setInt(5, Status.ATIVA.ordinal());
42
43             stm.execute();
44         } catch (SQLException | ParseException e) {
45             e.printStackTrace();
46         }
47     }
48 }
```

---

**Listagem 10.** Código XML de adição da lib do MySQL Conector.

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>5.1.37</version>
</dependency>
```

retorna nada, logo, teremos de lidar com a regra de sucesso/erro no AngularJS.

No lado cliente, começemos pela edição do arquivo de painel-inicial.html. Edite-o com as informações da **Listagem 12**.

Exceto pelas novas configurações do AngularJS (app + controller) e organização da estrutura HTML via classes e divs do Bootstrap, a listagem não traz muitas novidades.

# Criando aplicação de notícias com AngularJS

## Listagem 11. Classe de conexão com o banco.

```
package br.edu.devmedia.config;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConfig {

    public static Connection getConnection() {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            return DriverManager.getConnection("jdbc:mysql://localhost:3306/
news_devmedia", "root", "root");
        } catch (SQLException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Vejamos algumas observações:

- Na linha 2 declaramos a mesma aplicação AngularJS;
- Na linha 22 temos a definição do novo controller painelInicialController, que se encarregará de conter todo o escopo da nova página;
- Na linha 27 definimos o botão de adição de novas notícias com sua respectiva função AngularJS abreCadastroNoticia(). Essa função apenas exibe a div com o formulário de cadastro;
- Na linha 33 criamos o novo formulário com a diretiva ng-show, que serve para definir qual função JavaScript exibe ou esconde o mesmo;
- Na linha 34 definimos a nova função de submit do formulário;
- Dentro do formulário definimos os campos com os respectivos ng-model configurados para o modelo *notícia*;
- Na linha 55 criamos o campo de data que recebe um novo atributo: o ui-mask. Trata-se de um módulo que adicionaremos

## Listagem 12. Novo conteúdo da página painel-inicial.html.

```
01 <!DOCTYPE html>
02 <html ng-app="app">
03   <head>
04     <title>Painel Administrativo</title>
05     <meta charset="utf-8">
06     <link rel="stylesheet" href="http://getbootstrap.com/dist/css/
bootstrap.min.css" />
07     <style>
08       .mbottom {
09         margin-bottom: 10px;
10     }
11   </style>
12 </head>
13 <body>
14   <nav class="navbar navbar-default">
15     <div class="container-fluid">
16       <div class="navbar-header">
17         <a class="navbar-brand" href="#">Dashboard</a>
18       </div>
19     </div>
20   </nav>
21
22   <div ng-controller="painelInicialController">
23     <div class="container">
24       <div class="row">
25         <div class="col-xs-12">
26           <div class="well well-sm">
27             <button type="button" class="btn btn-primary"
ng-click="abreCadastroNoticia()">Nova Notícia</button>
28           </div>
29         </div>
30       </div>
31     </div>
32
33     <div class="container" ng-show="showCadastro">
34       <form ng-submit="cadastrarNovaNoticia()">
35
36       <div class="row mbottom">
37         <div class="col-xs-3 text-right">
38           Título:
39         </div>
40         <div class="col-xs-9">
41           <input type="text" class="form-control"
ng-model="noticia.titulo" required>
42         </div>
43       </div>
44
45       <div class="row mbottom">
46         <div class="col-xs-3 text-right">Descrição:</div>
47         <div class="col-xs-9">
48           <input type="text" class="form-control" ng-model="noticia.descricao">
49         </div>
50       </div>
51
52       <div class="row mbottom">
53         <div class="col-xs-3 text-right">Data:</div>
54         <div class="col-xs-9">
55           <input class="form-control" ng-model="noticia.data"
ui-mask="99/99/9999" model-view-value="true">
56         </div>
57       </div>
58
59       <div class="row mbottom">
60         <div class="col-xs-3 text-right">Texto:</div>
61         <div class="col-xs-9">
62           <textarea class="form-control" ng-model="noticia.texto" rows="5">
63         </div>
64       </div>
65
66       <div class="row mbottom">
67         <div class="col-xs-9 col-xs-offset-3">
68           <button class="btn btn-danger" type="submit">Inserir</button>
69         </div>
70       </div>
71
72     </form>
73   </div>
74 </div>
75
76   <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.5/
angular.min.js"></script>
77   <script src="https://cdnjs.cloudflare.com/ajax/libs/angular-ui-utils/0.1.1/
angular-ui-utils.min.js"></script>
78   <script src="js/painelInicialController.js"></script>
79 </body>
80 </html>
```

do AngularJS para mascarar campos via JavaScript de forma simplificada através de padrões;

- Finalmente, nas linhas 76 a 78 importamos via CDN os arquivos de JavaScript do AngularJS, do módulo UI-Utils do AngularJS e do painelController.js, que criaremos a seguir.

Em seguida, crie um novo arquivo JavaScript de nome painelController.js na pasta js, e adicione o conteúdo da **Listagem 13**.

**Listagem 13.** Conteúdo do arquivo painelController.js.

```
01 var app = angular.module('app', ['ui.mask']);
02
03 app.controller('painelInicialController', function($scope, $http) {
04   $scope.showCadastro = false;
05   $scope.noticia = montarObjNoticia();
06
07   $scope.abreCadastroNoticia = function() {
08     $scope.showCadastro = true;
09   }
10
11   $scope.cadastrarNovaNoticia = function() {
12     var string = $scope.noticia.data;
13     var month = string.substring(0,2);
14     var day = string.substring(2,4);
15     var year = string.substring(4,8);
16     var data_final = month + '/' + day + '/' + year;
17
18     $scope.noticia.data = data_final;
19
20     $http.post('/devmedia_news_rest_web/rest/noticia/new', $scope.noticia)
21       .success(function(data) {
22         alert("Cadastro efetuado com sucesso!");
23         $scope.showCadastro = false;
24         $scope.noticia = montarObjNoticia();
25       }).error(function() {
26         alert("Falha ao cadastrar notícia!");
27     });
28   };
29 });
30
31 function montarObjNoticia() {
32   return {
33     id : -1,
34     titulo :"",
35     descricao :"",
36     texto :"",
37     data : ""
38   };
39 }
```

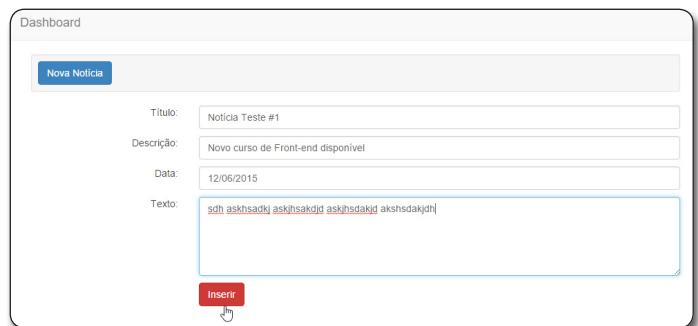
Veja que na primeira linha já temos a primeira mudança significativa no uso da função module() do AngularJS. Aqui sobreescrivemos o objeto global *app* para receber agora o módulo e carregar também a dependências da biblioteca *ui.mask*. Dessa forma, garantimos que o código a seguir a reconhecerá. Vejamos algumas considerações:

- Na linha 3 criamos o controller propriamente dito, com os mesmos objetos de antes;
- Na linha 4 escondemos a div de cadastro, via JavaScript, para exibi-la quando do click no botão de Adicionar Nova Notícia;
- Na linha 5 instanciamos um novo objeto de notícia, através da função auxiliar montarObjNoticia() da linha 31, com valores

padrão vazios;

- Na linha 11 criamos a função de cadastro da nova notícia que:
  - Recebe a data do formulário e a quebra em dia, mês e ano, para assim enviar o valor formatado, já que o AngularJS retorna só o texto;
  - Efetue a requisição post ao novo serviço, passando o objeto de notícia como argumento;
  - Manipula as funções de sucesso e erro, exibindo mensagens específicas para cada uma e limpando o objeto de notícia para um novo cadastro.

Agora basta reiniciar o servidor Tomcat e retestar a aplicação. O resultado pode ser conferido nas **Figuras 6 e 7**.



**Figura 6.** Tela de cadastro com dados de exemplo

A página em localhost:8080 diz:

Cadastro efetuado com sucesso!

Impedir que esta página crie caixas de diálogo adicionais.

OK

**Figura 7.** Mensagem de cadastro com sucesso

## Listagem de notícias

Para efetuar a listagem, vamos fazer o mesmo procedimento no Java, porém agora retornando JSON, em vez de consumindo. Para isso, crie um novo método no nosso NoticiaService, como mostra a **Listagem 14**.

Este pode ser um GET, em vista de não precisarmos enviar nenhum parâmetro e o mesmo ser mais rápido. A lógica dele se encarrega agora de explicitamente criar uma lista Java e retorná-la preenchida com os valores de todas as notícias cadastradas na base. Agora só precisamos atualizar a nossa HTML e JS para receber as mudanças, incluindo agora um novo componente de tabela. Veja na **Listagem 15** as alterações necessárias ao HTML.

Veja que agora estamos usando uma tag nova do AngularJS, a *ng-repeat*, que serve para iterar sobre um array/lista de valores mandados juntos do escopo. Funciona como uma estrutura *forEach* de linguagens mais famosas, como o Java ou C#. Na **Listagem 16** encontramos as alterações necessárias ao arquivo

# Criando aplicação de notícias com AngularJS

painelController.js para fazer o exemplo funcionar. Acrescente as mesmas dentro da declaração da função de controller().

Veja que agora usamos a função get(), ao invés de post(), com o mesmo significado. O resto é só associação e chamadas de métodos. Reinicie o servidor e veja o resultado final, semelhante à Figura 8.

**Listagem 14.** Novo método de listagem de notícias.

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
@Path("/listar")  
public List<Noticia> listarNoticias() {  
    List<Noticia> noticias = new ArrayList<Noticia>();  
  
    Connection con = DatabaseConfig.getConnection();  
  
    try {  
        PreparedStatement stm = con.prepareStatement("SELECT * FROM TB_NOTICIA");  
        ResultSet rs = stm.executeQuery();  
        while (rs.next()) {  
            Noticia noticia = new Noticia();  
            noticia.setId(rs.getInt("id"));  
            noticia.setTitulo(rs.getString("titulo"));  
            noticia.setDescricao(rs.getString("descricao"));  
            noticia.setTexto(rs.getString("texto"));  
            noticia.setData(rs.getDate("data").toString());  
  
            noticias.add(noticia);  
        }  
        con.close();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
    return noticias;  
}
```

**Listagem 15.** Alterações na painel-inicial.html.

```
<div class="container">  
    <div class="row">  
        <div class="col-xs-12">  
  
            <table class="table table-bordered table-striped table-hover">  
                <thead>  
                    <tr>  
                        <th width="90">Data</th>  
                        <th>Título</th>  
                        <th>Descrição</th>  
                    </tr>  
                </thead>  
  
                <tbody>  
                    <tr ng-repeat="noticia in allNoticias">  
                        <td>{{ noticia.data }}</td>  
                        <td>{{ noticia.titulo }}</td>  
                        <td>{{ noticia.descricao }}</td>  
                    </tr>  
  
                </tbody>  
            </table>  
        </div>  
    </div>  
</div>
```

**Listagem 16.** Alterações no painelController.js.

```
$scope.allNoticias = {};  
  
$scope.listarNoticias = function(){  
    $http.get('/devmedia_news_rest_web/rest/noticia/listar')  
        .success(function(data){  
            $scope.allNoticias = data;  
        })  
        .error(function(){  
            alert("Falha em obter as notícias");  
        });  
};  
  
$scope.listarNoticias();
```

Dashboard		
Nova Notícia		
Data	Título	Descrição
2000-01-01	dflkd	llkj
2015-06-12	Notícia Teste #1	Novo curso de Front-end disponível

**Figura 8.** Resultado da listagem de notícias

Essas foram somente algumas configurações e implementações possíveis junto com o AngularJS. Podemos ainda integrar tudo a outros frameworks, como vimos com o jQuery e o Bootstrap, ou incluir suas próprias bibliotecas junto. Essa é uma das grandes vantagens de usar o AngularJS, sua flexibilidade e integração.

## Autor



**Júlio Sampaio**

É analista de sistema e entusiasta da área de Tecnologia da Informação. Atualmente é consultor na empresa Visagio, trabalhando em projetos de desenvolvimento de sistemas estratégicos, é também instrutor JAVA. Possui conhecimentos e experiência em áreas como Engenharia de Software e Gerenciamento de Projetos, tem também interesse por tecnologias relacionadas ao front-end web.



## Links:

**Página oficial do AngularJS.**

<https://angularjs.org/>

**Página oficial do Bootstrap.**

<http://getbootstrap.com/>

# Introdução ao Google Material Design Lite

## Crie aplicações web responsivas usando o mais novo framework de componentes do Google

**I**magine o seguinte cenário: sempre que um novo sistema precisa ser criado, a equipe se divide em alguns passos, como definição da arquitetura inicial (*client e server side*), seleção do banco de dados, servidores (rede, arquivos, banco, ftp, etc.), dentre outros. Mas talvez um dos passos que sempre geram uma certa dor de cabeça seja a definição do layout/design do sistema. Dependendo da empresa/projeto/equipe que se esteja trabalhando, ele poderá se dar de várias formas:

- Ou o designer (que não necessariamente precisa ter conhecimentos sobre jQuery, JavaScript ou frameworks de componentes, como o Bootstrap, por exemplo) precisa fazer todo o trabalho criando telas estáticas para aprovação inicial do cliente, bem como HTML/CSS do mesmo para disponibilizar para os desenvolvedores;
- Ou os desenvolvedores entram nessa tarefa e trabalham em conjunto com o designer para “casar” as definições;
- Ou a equipe sequer faz uso de designer (exceto para imagens, o básico) e trata de ela mesma usar alguma biblioteca pronta para tal finalidade.

Esse é um processo bem flexível e, na verdade, é bom que seja. Assim, garante-se que a união de várias opiniões e experiências defina o que for melhor para o projeto final. Entretanto, suponha que você está criando seu próprio projeto, ou que sua equipe é muito pequena e não terão recursos para contratar um designer. Qual o primeiro passo que você tomaria para a definição de um design que atenda às cada vez mais exigentes expectativas dos seus usuários (design responsivo, componentes vivos, design bonito, fácil de usar, etc.)? Alguns provavelmente pensarão em copiar o design de alguma página estrangeira, ou quem sabe contratar um freelance para fazer apenas o design. Dentre as várias opções em mãos, a mais apropriada para os dias de hoje é, sem dúvida, usar alguma biblioteca de componentes pronta.

Um framework desse tipo já traz tudo que precisamos:

### Fique por dentro

Este artigo é útil por explorar os principais conceitos, na prática, acerca do novíssimo framework baseado em componentes para a web do Google, o Material Design Lite (ou MDL). Baseado totalmente em HTML, JavaScript e CSS, o framework promete trazer uma série de recursos que antes só existiam para o mundo mobile (Android). Aqui veremos um overview completo: desde a configuração, principais recursos, customização, montagem de layouts e templates, bem como a criação de uma página web estilo Pinterest (baseada em blocos). Ao final, você estará apto a criar seus próprios designs com o framework, bem como explorar recursos dos demais que ele usa como base, como o Polymer, por exemplo.

vários componentes prontos que vão desde botões, barras de navegação, menus, formulários, listas, tabelas, até páginas e templates inteiros como a página de contato de um site, ou até mesmo um carrinho de compras.

Variáveis como linha de aprendizado, simplicidade, tamanho dos arquivos, quantidade de recursos disponíveis, bem como navegadores suportados e, sobretudo, preço, estão constantes nesse passo do ciclo de vida do layout. Partindo de um ponto de vista mais realista, poderíamos usar o Bootstrap. Ele é flexível, atende a todas as exigências que comentamos, é gratuito, tem um suporte enorme e excelente da comunidade, é open source e tem como dono nada mais nada menos que o Twitter. Sim, o Bootstrap é uma ótima opção. Entretanto, recentemente, tivemos o lançamento de um novo framework de componentes que promete inovar o mercado web, principalmente por se tratar de um padrão adotado por mais uma gigante do Vale do Silício, o Google. Trata-se do **Google Material Design**, que usamos para definir boas práticas de design em layouts para aplicativos móveis no Android. O sucesso com essa nova plataforma foi tão grande que o Google a redesenhou para se adaptar a todo tipo de dispositivo existente, desde smartphones, tablets, TVs, relógios e óculos inteligentes, e agora, inclusive, para a web.

A especificação original e final foi lançada em meados de 2014, com o objetivo de fornecer todas as instruções para um design bom e bonito para todos os dispositivos, independente do fabricante. Para as versões que fazem uso de simples HTML, CSS e JavaScript, damos o nome de **Material Design Lite** (MDL), uma terminologia mais apropriada considerando-se os ambientes onde irão executar. Dentre as inúmeras vantagens dele, destaca-se a sua leveza (lite) com um tamanho máximo de 27KB (em modo gzip) em código fonte, além das suas poucas dependências externas. Na seção **Links** encontramos a URL para a página oficial do projeto.

Neste artigo trataremos de expor os principais recursos desse novo framework, com alguns exemplos práticos próximos da realidade de aplicações web (que envolvem um aparato de codificação, não apenas design).

## Detalhes da arquitetura

Se dermos uma olhada mais a fundo do framework veremos que, na verdade, ele é uma implementação complementar do projeto *Paper elements*, um subprojeto do framework Polymer que já faz uso dos conceitos de Material Design para construir interfaces web ricas, com elementos interativos, transições, efeitos, etc. Isso significa que podemos integrar o MDL com o Polymer facilmente e tirar proveito de seus componentes para criar designs de forma rápida e responsiva.

A maioria dos componentes que usamos em outras plataformas está disponível no MDL também, desde tooltips, campos de formulários diversos, spinners, até uma grid responsiva característica desse tipo de layout.

Além de ser suportado por todos os browsers mais recentes (Chrome, Firefox, Opera, Edge, etc.), os fontes do MDL são escritos em Sass usando BEM (contração para *Block, Element, Modifier*), trata-se de uma metodologia que visa aumentar a velocidade e produtividade no desenvolvimento de uma forma geral. As suas siglas significam:

- **Block:** um componente lógico e funcional independente, o equivalente a um componente nos famosos Web Components. Um *block* (ou bloco) encapsula comportamento (via JavaScript), templates, estilos (CSS) e outras tecnologias implementadoras. O conceito de independência é usado para facilitar o reuso de estruturas, assim como facilitar o desenvolvimento do projeto e o processo de suporte ao mesmo. Na **Figura 1** vemos um modelo simples de como tais blocos se ajustam uns em relação aos outros; por exemplo: um bloco de cabeçalho (*head*) pode ter uma logo incluída, um formulário de busca e um bloco interno de autorização (login do usuário);

- **Element:** é uma parte constituinte de um bloco que não pode ser usada de fora do mesmo. Por exemplo, um item de menu não

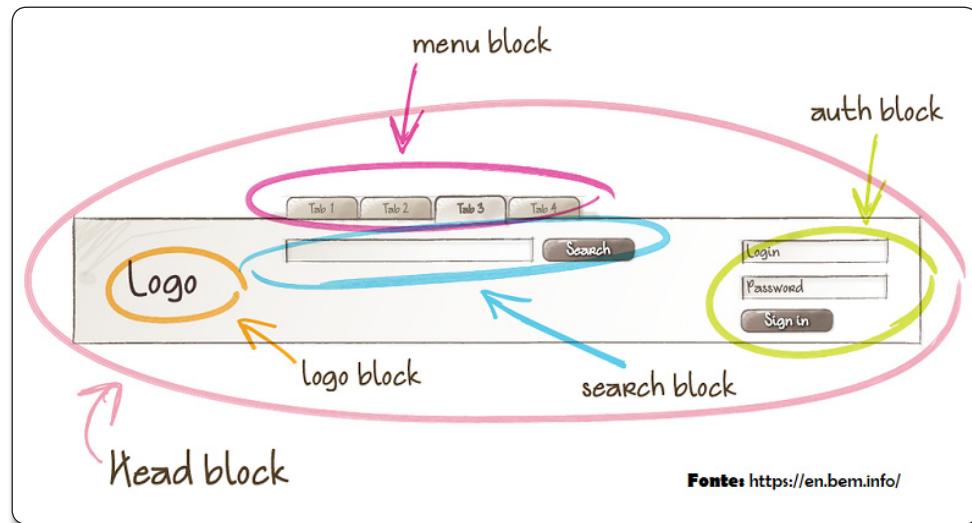


Figura 1. Exemplo de estrutura em blocos no BEM

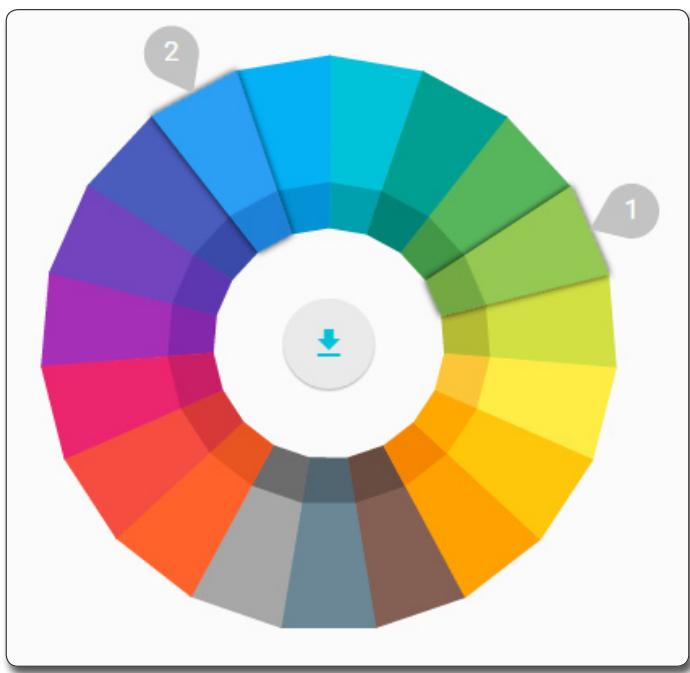
pode ser usado de fora do contexto de um bloco de menu, portanto ele é um *element* (ou elemento).

- **Modifier:** é uma entidade BEM que define a aparência e comportamento de um bloco ou elemento. Eles são similares aos atributos HTML, em essência. Um mesmo bloco tem aparência diferente em relação a outro quando usa um *modifier* (ou modificador).

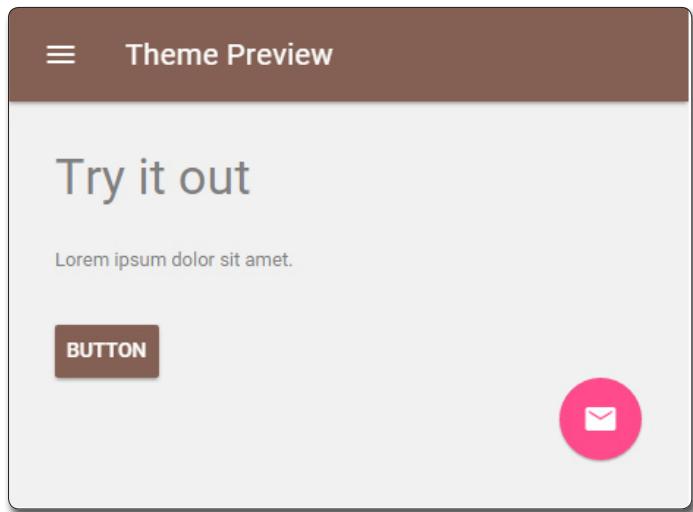
## Mãos à obra

Como vimos, não há muito a se discutir acerca da arquitetura interna do framework. Tudo foi feito para simplificar nosso trabalho. A forma mais fácil de trabalhar com o MDL é referenciar os arquivos de dependências via CDN (*Content Delivery Network*), mas também podemos efetuar o download dos arquivos fisicamente via projeto no GitHub (vide seção **Links**) ou fazer o import via npm (gerenciador de dependências do Node.js) ou Bower. Há também ainda a opção de construir o próprio template usando a ferramenta do *Theme Customizer*, que pré-constrói o CSS e o disponibiliza compactado em um zip. Vejamos como seguir cada uma das opções.

Comecemos então pelo *Theme Customizer*, que não necessita de nenhuma dependência a nível de instalador. Acesse a sua URL via seção **Links** e, na página que abrir, veremos uma paleta com algumas cores, dois números ao redor delas e um botão de download no meio, tal como vemos na **Figura 2**. Os temas gerados são representados por duas cores base: a de número 1 representa a cor primária do template, aquela que aparecerá na maior parte dos componentes; já a de número 2 representa a cor secundária que aparecerá em itens como botões suspensos, tooltips, etc. Ao lado da paleta encontramos uma página simples de exemplo que muda conforme selecionamos cores diferentes. Além disso, a paleta também está preparada para não permitir cores muitos contrastantes de serem selecionadas, dessa forma você sempre terá templates combinando. Faça um teste: após selecionar as duas cores veremos o tema de preview mudar, tal como temos na **Figura 3**.



**Figura 2.** Paleta de cores do Theme Customizer



**Figura 3.** Exemplo de tela de preview com tema mudado pela paleta

Quando finalizar a customização, basta clicar no botão central de download da paleta e um arquivo de nome `material.min.css` será gerado com as devidas configurações. Ou, se preferir, poderá importar o arquivo direto do CDN do Google, uma vez que eles guardam uma referência para cada cruzamento de cores possível que forem selecionadas. Para o nosso exemplo, o código seria o seguinte:

```
<link rel="stylesheet" href="https://storage.googleapis.com/code.getmdl.io/1.0.5/
material.brown-pink.min.css"/>
```

Agora que aprendemos a usar a Paleta, vamos fazer o mesmo import a partir de um ambiente de build. Para isso, precisaremos ter o

Node.js instalado na máquina, pois é a partir dele e do gerenciador de dependências (o npm), que poderemos efetuar os downloads necessários. Não mostraremos os passos para isso aqui para não perder o foco, mas na página oficial do Node.js encontramos o link para download, bem como instruções para instalá-lo.

Crie uma pasta em um caminho de sua preferência para salvarmos os arquivos do projeto. Abra o prompt cmd (ou o equivalente no sistema operacional que esteja usando) e navegue até esse diretório:

```
cd D:\tests\material-design-devmedia
```

Em seguida, execute o seguinte comando via npm:

```
npm install material-design-lite --save
```

Obteremos como resposta a versão do MDL instalado (1.0.5) e uma nova pasta será criada: `node_modules`, na qual teremos o subdiretório “`material-design-lite`” criado também. Nele encontraremos vários arquivos, dentro os quais os únicos que nos interessam são os que estão contidos na subpasta `dist` (de distribuição). Além disso, encontraremos as versões minificadas (usaremos as que tem `min` no fim do nome porque o conteúdo está reduzido e, portanto, mais performático para as páginas web) e normais. Caso tenha o Bower instalado (dá para instalar via npm também), basta executar o seguinte comando:

```
bower install material-design-lite --save
```

O mesmo procedimento será feito, com exceção do nome da pasta (`bower_components`) e das subpastas geradas. Para simplificar o uso de tais arquivos, façamos o seguinte: crie duas novas pastas na raiz do projeto, `css` e `js`, para guardar os arquivos CSS e JavaScript, respectivamente. Em seguida, copie os arquivos `material.min.css` e `material.min.js` para as mesmas, já que serão os únicos que precisamos para trabalhar com o MDL.

Após isso, crie também um novo arquivo `index.html` e adicione o conteúdo da **Listagem 1** ao mesmo. Analisaremos em seguida.

No início da listagem, perceba que estamos usando a tag `<meta>` em função de se tratar de uma página em português, logo essa configuração é necessária para não termos problema de *encoding* na mesma. Na linha 4 importamos o arquivo de CSS do MDL, e em seguida (linha 6), importamos a fonte do *Material Icons*, esta que é importante para que possamos exibir os ícones sem a necessidade de imagens para isso. Essa é mais uma das vantagens do MDL, não precisamos gerenciar aquele monte de ícones, ou criar uma imagem só e sair mapeando posições para exibi-los. Os ícones no Material Design são na verdade letras de uma fonte criada especialmente para isso. Por exemplo, na linha 17 criamos um botão de *Add*, aqueles usados para criar uma nova tarefa ou adicionar algo novo num formulário mobile. O termo “add” é o que confira o símbolo de + para a fonte em questão, atrelado às respectivas classes CSS, claro.

# Introdução ao Google Material Design Lite

**Listagem 1.** Conteúdo do arquivo index.html: Alo Mundo MDL.

```
01 <html>
02 <head>
03   <meta charset="utf-8">
04   <link rel="stylesheet" href="css/material.min.css">
05   <!-- Material Design icon font -->
06   <link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons">
07 <title>Alô Mundo MDL!</title>
08 </head>
09 <body style='padding: 40px'>
10   <!-- Accent-colored raised button with ripple -->
11   <button class="mdl-button mdl-js-button mdl-button--raised mdl-js-ripple-effect mdl-button--accent">
12     Um Botão
13   </button>
14   <br><br>
15   <!-- Colored FAB button -->
16   <button class="mdl-button mdl-js-button mdl-button--fab mdl-button--colored">
17     <i class="material-icons">add</i>
18   </button>
19 </body>
20 <script src="js/material.min.js"></script>
21 </html>
```

Além disso, também criamos um botão no início da implementação para mostrar como essa estrutura se parece no framework. Para testar, salve o arquivo e abra-o no navegador, tal como vemos na **Figura 4**.

Ao clicar no primeiro botão, note que um efeito de clicado será aplicado ao mesmo. Isso é possível através da classe CSS “mdl-js-ripple-effect” que configura essa característica aos elementos. Já a classe “mdl-button--raised”, por exemplo, se encarrega de dar a cor de fundo ao botão, se removê-la verá apenas o texto. O mesmo vale para a “mdl-button--colored” no segundo botão que, por sua vez, removerá a cor, porém manterá a cor padrão: que é o branco prateado. Faça os testes e veja por si.

O MDL também faz uso de uma fonte específica para os designs, trata-se da Roboto, é ela que confere as características mais próximas dos apps Android. Para adicionar no nosso exemplo, inclua a seguinte linha de código dentro da tag head:

```
<link href='http://fonts.googleapis.com/css?family=Roboto:400,300,300italic,500,400italic,700,700italic' rel='stylesheet' type='text/css'>
```

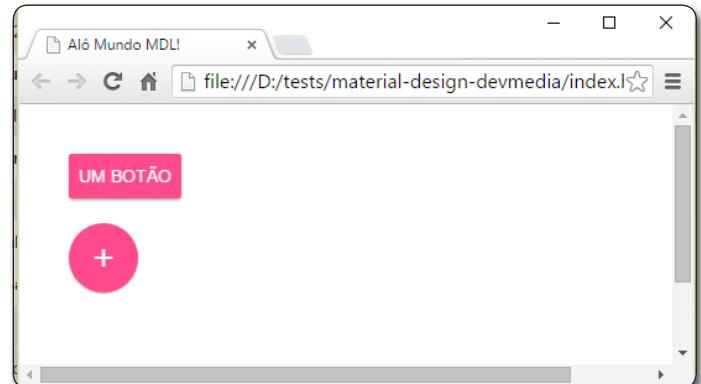
Agora vá até o browser e recarregue a página. O resultado será igual ao da **Figura 5**.

Caso deseje usar o tema que customizamos anteriormente, basta substituir o arquivo de CSS e configurar o botão para o que temos a seguir:

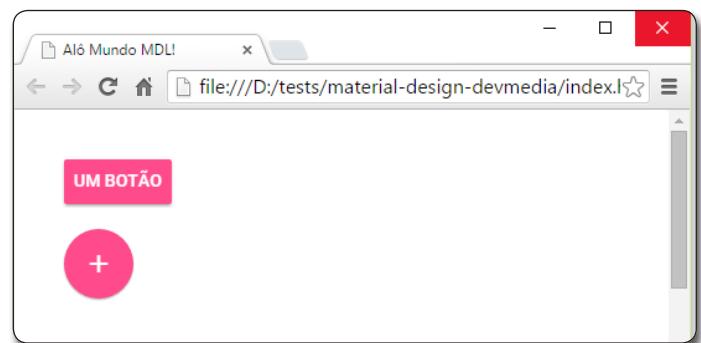
```
<button class="mdl-button mdl-button--colored mdl-button--raised mdl-js-button mdl-js-ripple-effect">Um Botão</button>
```

## A Grid do MDL

O objeto de Layout no MDL é um dos mais importantes pois é nele que definimos a navegação principal, o rodapé, assim como a Grid principal. Esse recurso faz uso do Flexbox, uma *feature* do



**Figura 4.** Alô Mundo com MDL



**Figura 5.** Representação da tela com fonte Roboto

CSS3 que permite dispor os componentes de uma tela de forma rearranjada pensando nos diferentes tipos de tela em que a mesma será exibida. Isso é uma mão na roda, sobretudo por não precisarmos mais fazer uso extensivo de *float* para fazer os elementos flutuarem uns ao lado dos outros, além de permitir que os mesmos se alinhem livremente no escopo em que estiverem inseridos.

No fim das contas, o componente de Grid é nada mais que colunas, mais especificamente 12 colunas para o modo “desktop”, 8 colunas para o modo “tablet” ( $\leq 800\text{px}$ ) e 4 colunas para o modo “smartphone” ( $\leq 500\text{px}$ ).

Para criá-lo, começamos com uma div vazia, e duas classes CSS apropriadas:

```
<div class="content-grid mdl-grid">
  <!-- grid aqui -->
</div>
```

A classe “mdl-grid” funciona como uma row (linha), como temos no Bootstrap, por exemplo. Entretanto, ela nos deixa livre de tamanho padrão, o que nos deixa mais flexíveis para especificar não o tamanho, mas sim o tamanho máximo dessa estrutura. Portanto, crie uma tag style na head da nossa página e adicione o seguinte conteúdo:

```
.content-grid {
  max-width: 950px;
}
```

Isso ainda não exibirá nada de diferente na página, por que ainda precisamos criar as colunas de conteúdo. Para isso, basta adicionar novas divs de classe “mdl-cell” com algum conteúdo dentro. Modifique o conteúdo da div principal conforme a **Listagem 2**.

Perceba que incluímos além das divs de coluna, os outros dois componentes de botão para que vejamos como o MDL associa quaisquer estruturas do framework como passíveis de serem alinhadas em colunas. O resultado pode ser visto na **Figura 6**.

**Listagem 2.** Conteúdo da Grid no MDL.

```
<div class="content-grid mdl-grid">
<div class="mdl-cell">
  Conteúdo 1... Conteúdo 1... Conteúdo 1... Conteúdo 1... Conteúdo 1...
</div>
<div class="mdl-cell">
  Conteúdo 2... Conteúdo 2... Conteúdo 2... Conteúdo 2... Conteúdo 2...
</div>
<div class="mdl-cell">
  Conteúdo 3... Conteúdo 3... Conteúdo 3... Conteúdo 3... Conteúdo 3...
</div>
<button class="mdl-button mdl-button--colored mdl-button--raised mdl-js-button mdl-js-ripple-effect">Um Botão</button>
<button class="mdl-button mdl-js-button mdl-button--fab mdl-button--colored">
  <i class="material-icons">add</i>
</button>
</div>
```



**Figura 6.** Exemplo de grid básica de três colunas



**Figura 7.** Exemplo de grid básica de três colunas com tamanhos predefinidos

Se desejar, também pode configurar suas grids com base no tamanho de cada coluna. Para isso, basta adicionar na mesma div uma segunda classe CSS com o formato *mdl-cell-{numero}-col*, onde *numero* deve ser um valor entre 1 e 12. Veja na **Listagem 3** o código que precisamos para isso e na **Figura 7** o resultado do mesmo. Veja que colocamos valores distintos em cada uma para que víssemos o resultado final, mas mesmo assim tudo junto deve somar 12.

Também é possível configurar tais propriedades para os três ambientes base da Grid: desktop, tablet e phone. Basta lembrar das dimensões máximas de colunas que cada um leva e aplicar a classe CSS correspondente. Veja na **Listagem 4** como ficaria nosso exemplo para tablet. O resultado pode ser visualizado na **Figura 8**.

Para testar no navegador, basta reduzir a dimensão da tela até atingir o tamanho padrão de tablet. O mesmo pode ser feito para dispositivos celulares, bastando adicionar a classe “mdl-cell-{numero}-col-phone” à div, onde *numero* é a quantidade de colunas num total de quatro.

O MDL também disponibiliza uma série de classes CSS que podemos usar para funções utilitárias, como esconder a div quando ela estiver sendo exibido em um determinado tamanho de tela, ou algumas para alinhar elementos de forma diferente, a saber:

- *mdl-cell-hide-desktop*: esconde a coluna no modo de visualização desktop (> 840px);
- *mdl-cell-hide-tablet*: esconde no modo tablet (> 480px e <=840px);
- *mdl-cell-hide-phone*: esconde no modo telefone (<480px);
- *mdl-cell-stretch*: estende a coluna para preencher todo o elemento pai, neste caso, o *mdl-grid*;
- *mdl-cell-top*: alinha a coluna no topo do elemento pai;
- *mdl-cell-bottom*: alinha a coluna na base do elemento pai.

### A barra/menu de navegação

Assim como a maioria das aplicações web de hoje em dia, criar um cabeçalho de navegação, comum a todas as demais páginas, é essencial. Para isso, vamos precisar aprender algumas outras classes CSS do framework que veremos mais à frente. Vamos criar também uma nova página HTML para não misturar com a anterior, assim, ao terminar o cabeçalho podemos incluir novos elementos no corpo da mesma.

Portanto, crie uma nova página chamada *header.html* e acrescente o conteúdo da **Listagem 5** à mesma. Para este exemplo, customizamos um novo arquivo de CSS na paleta do MDL para se adequar melhor às cores da DevMedia.

# Introdução ao Google Material Design Lite



Figura 8. Exemplo de grid básica de três colunas com tamanhos predefinidos para tablet

**Listagem 3.** Código para divs com tamanhos fixos na Grid.

```
<div class="content-grid mdl-grid">
<div class="mdl-cell mdl-cell--6-col">
  Conteúdo 1... Conteúdo 1... Conteúdo 1... Conteúdo 1...
</div>
<div class="mdl-cell mdl-cell--4-col">
  Conteúdo 2... Conteúdo 2... Conteúdo 2... Conteúdo 2...
</div>
<div class="mdl-cell mdl-cell--2-col">
  Conteúdo 3... Conteúdo 3... Conteúdo 3... Conteúdo 3...
</div>
<button class="mdl-button mdl-button--colored mdl-button--raised mdl-js-button mdl-js-ripple-effect">Um Botão</button>
<button class="mdl-button mdl-js-button mdl-button--fab mdl-button--color-red">
  <i class="material-icons">add</i>
</button>
</div>
```

**Listagem 4.** Exemplo de Grid para tablet.

```
<div class="mdl-cell mdl-cell--6-col mdl-cell--1-col-tablet">
  Conteúdo 1... Conteúdo 1... Conteúdo 1... Conteúdo 1...
</div>
<div class="mdl-cell mdl-cell--4-col mdl-cell--2-col-tablet">
  Conteúdo 2... Conteúdo 2... Conteúdo 2... Conteúdo 2...
</div>
<div class="mdl-cell mdl-cell--2-col mdl-cell--5-col-tablet">
  Conteúdo 3... Conteúdo 3... Conteúdo 3... Conteúdo 3...
</div>
```

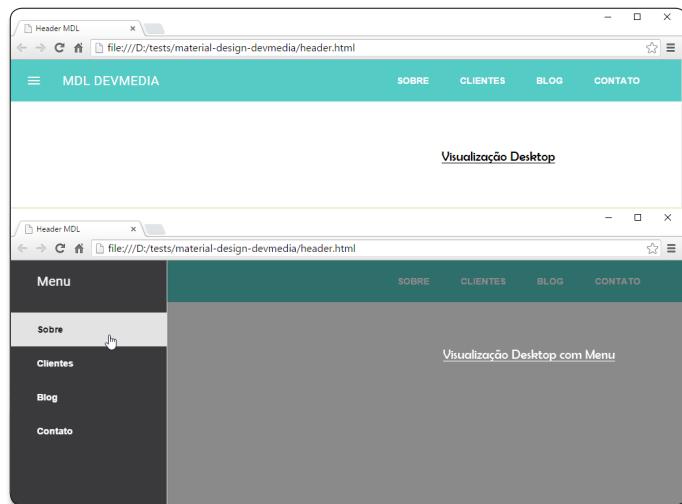
Perceba que temos dois elementos principais nesta listagem: o primeiro é a barra de navegação, representada pelo componente `<header>` que ficará na barra fixa e responsiva do topo da página com as opções de menu representadas por um elemento `<nav>` de classe “`mdl-navigation`” e tendo cada item de menu como um link de classe “`mdl-navigation__link`”; o segundo elemento é o side panel que flutuará na esquerda da página e será aberto automaticamente pelo MDL quando um click for feito no botão do cabeçalho. A classe “`mdl-layout__drawer`” confere essa característica uma vez que esse é o nome comumente dado a esse tipo de estrutura: *Navigation Drawer*. Por fim, temos também a div de classe “`mdl-layout__content`” que conterá o conteúdo do corpo da página.

**Listagem 5.** Página de cabeçalho para navegação e menus.

```
<html>
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="css/material.min.css">
  <!-- Material Design icon font -->
  <link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons">
  <link href="http://fonts.googleapis.com/css?family=Roboto:400,300,300italic,500,400italic,700,700italic' rel='stylesheet' type='text/css'>
  <title>Header MDL</title>
</head>
<body>
  <div class="mdl-layout mdl-js-layout mdl-layout--fixed-header">
    <header class="custom-header mdl-layout__header
      mdl-layout__header--waterfall">
      <div class="mdl-layout__header-row">
        <span class="mdl-layout-title">MDL DevMedia</span>
        <div class="mdl-layout-spacer"></div>
        <nav class="mdl-navigation mdl-layout--large-screen-only">
          <a class="mdl-navigation__link" href="#">Sobre</a>
          <a class="mdl-navigation__link" href="#">Clientes</a>
          <a class="mdl-navigation__link" href="#">Blog</a>
          <a class="mdl-navigation__link" href="#">Contato</a>
        </nav>
      </div>
    </header>
    <div class="mdl-layout__drawer">
      <span class="mdl-layout-title">Menu</span>
      <nav class="mdl-navigation">
        <a class="mdl-navigation__link" href="#">Sobre</a>
        <a class="mdl-navigation__link" href="#">Clientes</a>
        <a class="mdl-navigation__link" href="#">Blog</a>
        <a class="mdl-navigation__link" href="#">Contato</a>
      </nav>
    </div>
    <main class="mdl-layout__content">
      <div class="page-content">
        <!-- Conteúdo aqui -->
      </div>
    </main>
  </div>
</body>
<script src="js/material.min.js"></script>
</html>
```

**Listagem 6.** Arquivo com o CSS no modelo da DevMedia.

```
html>body {  
    font-family: 'Roboto', 'Helvetica', 'Arial', sans-serif !important  
}  
.mdl-layout__drawer-button {  
    height: 38px;  
    margin: 20px 12px;  
}  
@media screen and (max-width: 1024px) {  
    .mdl-layout__header .mdl-layout__drawer-button {  
        margin: 15px;  
    }  
}  
header.custom-header {  
    background: #49c5bf;  
}  
.mdl-layout__header-row span.mdl-layout-title {  
    color: white;  
    text-transform: uppercase;  
}  
.custom-header .material-icons {  
    color: #fff;  
}  
.custom-header a.mdl-navigation__link {  
    color: white;  
    font-weight: 700;  
    font-size: 14px;  
    text-transform: uppercase;  
}  
.custom-header a.mdl-navigation__link:hover {  
    color: #302F31;  
}  
.mdl-layout__drawer span.mdl-layout-title {  
    background: #302F31;  
    color: white;  
}  
.mdl-layout__drawer a.mdl-navigation__link {  
    color: #eee;  
    font-weight: 700;  
    font-size: 14px;  
}  
.mdl-layout__drawer a.mdl-navigation__link:hover {  
    color: #000;  
}  
.mdl-layout__drawer .mdl-navigation .mdl-navigation__link {  
    color: white;  
}  
.mdl-layout__drawer {  
    background: #302F31;  
}
```



**Figura 9.** Resultado final do cabeçalho de navegação

Antes de testar, precisamos também sobreescriver algumas propriedades CSS, uma vez que o template nu e cru não fica muito elegante visualmente. Então, para enaltecer o poder do framework, façamos as modificações para ficarem o mais próximo possível do site da DevMedia, com as cores que ele usa lá. Crie um novo arquivo `header.css` na pasta `css` e adicione o conteúdo da **Listagem 6** ao mesmo. A maioria dos atributos sobreescritos existem para mudar as cores para o novo formato, e podemos ficar à vontade para inserir os próprios. O resultado pode ser visto na **Figura 9**.

No MDL também existe um outro tipo de menu que podemos usar para ações mais rápidas e específicas de certas páginas, como os menus de contexto. No universo móvel eles são conhecidos por aparecerem após o click num botão redondo com três pontos verticais ou quando pressionamos por algum tempo um certo componente. No Android, eles aparecem nas ActionBars, no topo esquerdo, como menus convencionais.

Para implementar na nossa página, vamos desenhar os mesmos na base, para não ocupar o espaço superior que já pertence ao cabeçalho. Para isso, adicione as linhas de código presentes na **Listagem 7** após a tag `<main>` e antes de fechar o corpo.

Note que as primeiras divs e suas respectivas classes servem apenas para criar uma barra fixa na base da página, tal como fizemos com a superior. Dentro da div de classe "wrapper" é onde a mágica acontece: basta criar um componente `<button>` de classe "mdl-button--icon" e adicionar o ícone em seu interior, no caso para o de três pontos verticais, usamos a palavra "more\_vert" (caso deseje o mesmo na horizontal, substitua por "more\_horiz"). Esse mesmo botão também precisa de um id para que possamos associá-lo depois ao atributo `for` da lista que criamos logo a seguir. É assim que o MDL sabe a quem pertence aquele menu. O restante da construção do menu não traz nenhuma novidade.

**Listagem 7.** HTML referente ao menu de contexto.

```
<div class="snippet-demo">  
<div class="snippet-demo-container demo-menu demo-menu__top-right">  
    <div class="container mdl-shadow--2dp">  
        <div class="bar">  
            <div class="wrapper">  
                <!-- Right aligned menu on top of button -->  
                <button id="opcoes_extra" class="mdl-button mdl-js-button mdl-button--icon">  
                    <i class="material-icons">more_vert</i>  
                </button>  
  
                <ul class="mdl-menu mdl-menu--top-right mdl-js-menu mdl-js-ripple-effect" for="opcoes_extra">  
                    <li class="mdl-menu__item">Últimas mensagens</li>  
                    <li class="mdl-menu__item">Filtrar</li>  
                    <li disabled class="mdl-menu__item">Remover Conta</li>  
                    <li class="mdl-menu__item">Sair</li>  
                </ul>  
            </div>  
        </div>  
    </div>  
    </div>  
</div>
```

# Introdução ao Google Material Design Lite

Antes de testar, adicione também as linhas de CSS da **Listagem 8** ao final do arquivo header.css para que o estilo do menu se adeque ao que estamos criando. O resultado pode ser visto na **Figura 10**.



**Figura 10.** Página de template com menu de contexto inserido (modo telefone)

Veja que um dos elementos aparece desabilitado, isso porque o marcamos com o atributo HTML *disabled*, o que automaticamente confere essa característica.

E para finalizar o nosso template, vamos adicionar a famosa caixa de pesquisas no topo do template. Para isso, basta incluir o código da **Listagem 9** logo após a primeira <nav> do cabeçalho.

Isso será o bastante para criar o ícone, exibi-lo de forma responsiva, isto é, quando estiver em modo telefone não esconderá a mesma, e quando clicarmos nela, uma caixa de texto se abre com um efeito, afastando os demais componentes para o lado (**Figura 11**).

Uma vez com o template pronto (cabeçalho + rodapé), agora precisamos inserir algo no corpo da nossa página e ver como ela se comporta. Vamos trabalhar então com dois novos componentes que ainda não citamos aqui: formulários e tabelas. Por padrão, os campos de formulário no MDL são alinhados um ao lado do outro. Por essa razão é interessante que o leitor tenha sempre em mente que o framework não vai fazer 100% do trabalho, o que significa que muitas vezes teremos de pôr a mão na massa e sobreescriver/criar alguns CSSs. Vamos começar pelo HTML então, adicione o conteúdo da **Listagem 10** dentro da div de “page-content”.

**Listagem 8.** CSS para adaptar rodapé de contexto.

```
/* Classes dos menus de seção*/
.demo-menu.demo-menu__top-right .container {
  position: relative;
  width: 100%;
}

.demo-menu.demo-menu__top-right .bar {
  box-sizing: border-box;
  position: relative;
  background: #49c5bf;
  color: white;
  height: 64px;
  width: 100%;
  padding: 16px;
}

.demo-menu.demo-menu__top-right .wrapper {
  box-sizing: border-box;
  position: absolute;
  right: 16px;
}
```

**Listagem 9.** Código para incluir caixa de pesquisas ao lado do menu.

```
<div class="mdl-textfield mdl-js-textfield mdl-textfield--expandable mdl-textfield--floating-label mdl-textfield--align-right">
<label class="mdl-button mdl-js-button mdl-button--icon" for="waterfall-exp">
<i class="material-icons">search</i>
</label>
<div class="mdl-textfield__expandable-holder">
<input class="mdl-textfield__input" type="text" name="sample" id="waterfall-exp"/>
</div>
</div>
```

Para criar componentes de formulário basta usar as tags usuais do HTML para isso, adicionando as respectivas classes CSS do MDL. Perceba que para cada campo, criamos também uma label que agirá, na verdade, como um placeholder do mesmo, com o atributo *for* setado para o *id* de cada campo.

Em seguida, criamos um campo comum (como já vimos) e, logo após, uma tabela para exibir os resultados. A mesma trabalha com o escopo de th/tr para cabeçalho e linhas, respectivamente. Note que precisamos estipular se cada coluna é de tipo numérico ou não, isso porque tal definição influencia no alinhamento dos valores dentro da tabela. Isso é possível através da classe “mdl-data-table\_\_cell--non-numeric”, caso nada seja especificado, a coluna é tida como numérica automaticamente. Veja que também definimos um estilo fixo de largura (*width*) com valor de 100% na tabela para que a mesma ocupe todo o espaço disponível na largura da página pai, caso contrário teríamos ela bem pequena, ocupando apenas o espaço necessário.

Também precisamos incluir algumas regras CSS apenas para alinhar os campos no centro da página. Para isso, inclua as linhas da **Listagem 11** ao final do arquivo header.css. Veja o resultado na **Figura 12**.

**Listagem 10.** Log de saída para migração do banco de dados.

```
<form action="#">
<div class="mdl-textfield mdl-js-textfield">
<input class="mdl-textfield__input" type="text" id="nome"/>
<label class="mdl-textfield__label" for="nome">Nome...</label>
</div>
<div class="mdl-textfield mdl-js-textfield">
<input class="mdl-textfield__input" type="text" id="fone"/>
<label class="mdl-textfield__label" for="fone">Telefone...</label>
</div>
<div class="mdl-textfield mdl-js-textfield">
<input class="mdl-textfield__input" type="text" id="endereco"/>
<label class="mdl-textfield__label" for="endereco">Endereço...</label>
</div>
</form>
<button class="mdl-button mdl-js-button mdl-button--raised mdl-js-ripple-effect">Enviar</button>
<br><br>
<table class="mdl-data-table mdl-js-data-table mdl-shadow--2dp"
style='width: 100%'>
<thead>
<tr>
<th class="mdl-data-table__cell--non-numeric">Nome</th>
<th>Telefone</th>
<th class="mdl-data-table__cell--non-numeric">Endereço</th>
</tr>
</thead>
<tbody>
<tr>
<td class="mdl-data-table__cell--non-numeric">Ana Maria</td>
<td>89898989</td>
<td class="mdl-data-table__cell--non-numeric">R. Teste</td>
</tr>
<tr>
<td class="mdl-data-table__cell--non-numeric">Ana Maria</td>
<td>89898989</td>
<td class="mdl-data-table__cell--non-numeric">R. Teste</td>
</tr>
<tr>
<td class="mdl-data-table__cell--non-numeric">Ana Maria</td>
<td>89898989</td>
<td class="mdl-data-table__cell--non-numeric">R. Teste</td>
</tr>
</tbody>
</table>
```

## Aplicação exemplo: Simulador Pinterest

Agora que já temos a maioria dos componentes mais importantes assimilados, bem como a forma como o framework lida com eles, vamos construir uma aplicação rápida de exemplo usando os modelos do MDL. Trata-se de um simulador do Pinterest (rede social famosa por seus posts em formato de blocos – *tiles*) que exibirá alguns posts de forma responsiva e fará uso de outro componente famoso do Material Design Lite, os *Cards*. Para o exemplo faremos uso também da biblioteca JavaScript do Masonry, uma API para criação de layouts em formato de grid responsivamente (na seção **Links** encontra-se a URL do site). Para não perder tempo com download, usaremos a opção via CDN. Esse exemplo também terá como dependência a biblioteca do jQuery, em sua última versão. Portanto, crie um novo arquivo HTML de nome *pinterest.html* e adicione o conteúdo da **Listagem 12** ao mesmo.

**Listagem 11.** CSS para alinhar campos no centro da página.

```
.mdl-layout__content {
  margin: 0 auto;
}

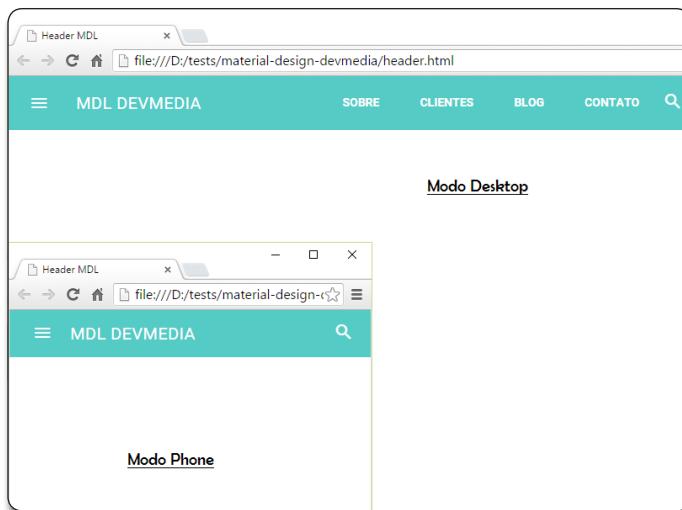
.page-content {
  padding: 20px;
}
```

**Listagem 12.** Conteúdo da página de simulação do Pinterest.

```
<html>
<head>
<meta charset="utf-8">
<link rel="stylesheet" href="css/material.min.css">
<link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons">
<link href="http://fonts.googleapis.com/css?family=Roboto:400,300,300italic,500,400italic,700,700italic' rel='stylesheet' type='text/css'>
<title>Alô Mundo Pinterest!</title>
<style>

</style>
</head>
<body>
<div class="content-grid mdl-grid">

</div>
</body>
<script src="https://code.jquery.com/jquery-2.1.4.min.js"></script>
<script src="js/material.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/masonry/3.3.2/masonry.pkgd.min.js"></script>
</html>
```



**Figura 11.** Inclusão da caixa de pesquisas

A listagem apenas faz os imports que comentamos e cria a estrutura básica inicial. Como adendo, o leitor pode verificar o atalho F12 no Chrome e verificar se alguma mensagem de erro aparece, caso contrário está tudo ok até então. Agora precisamos criar o corpo da nossa página, para tanto inclua o conteúdo da **Listagem 13** dentro da tag `<body>` da página.

Essa é uma listagem relativamente grande, mas ela faz basicamente a mesma coisa em cada li que criamos como item de post. Primeiro criamos uma div fix que fará o papel do cabeçalho,

# Introdução ao Google Material Design Lite

essa parte já conhecemos. Depois, criamos o conteúdo do layout, que trará uma lista (ul) com cada item representado pela classe "grid-item". Perceba que em cada div de card estamos mapeando a imagem como plano de fundo da mesma, o que nos fará precisar baixar as imagens que estão disponíveis no arquivo de fontes desse artigo, no topo da página. Copie as respectivas imagens para sua pasta img e verifique se as referências estão corretas. Cada card tem um título (h2), um texto de descrição, um botão para levar até a respectiva página, e um

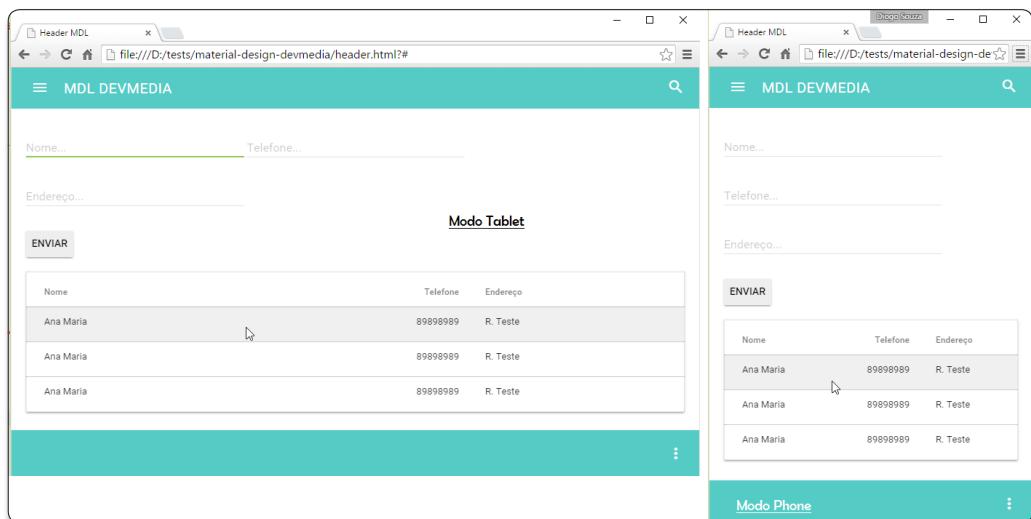


Figura 12. Exemplo de corpo com formulário e tabela

## Listagem 13. Corpo da página de Pinterest.

```
<div class="mdl-layout mdl-layout--fixed-header">
  <header class="mdl-layout__header mdl-layout__header--seamed">
    <div class="mdl-layout__header-row">
      <div class="mdl-layout-title">Material Design Lite Pinterest</div>
    </div>
  </header>

  <main class="mdl-layout__content">
    <div class="mdl-grid">
      <ul class="mdl-cell mdl-cell--11-col grid col-centered">

        <!-- Card #1 -->
        <li class="mdl-card mdl-shadow--2dp grid-item">
          <div class="mdl-card__title">
            style="background: url('img/curso_android.png') center / cover;">
            <h2 class="mdl-card__title-text">Curso de Android</h2>
          </div>

          <div class="mdl-card__supporting-text">
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris sagittis
            pellentesque lacus eleifend lacinia...
          </div>

          <div class="mdl-card__actions mdl-card--border">
            <a class="mdl-button mdl-button--colored mdl-js-
              button mdl-js-ripple-effect">
              Saber Mais...
            </a>
          </div>

          <div class="mdl-card__menu">
            <button class="mdl-button mdl-button--icon mdl-js-button mdl-js-
              ripple-effect">
              <i class="material-icons">share</i>
            </button>
          </div>
        </li>

        <!-- Card #2 -->
        <li class="mdl-card mdl-shadow--2dp grid-item">
          <div class="mdl-card__title" style="background: url('img/curso_java.png')
            center / cover;">
            <h2 class="mdl-card__title-text">Curso de Java</h2>
          </div>

          <div class="mdl-card__supporting-text">
            Lorem ipsum dolor sit amet, consectetur adipiscing elit.
          </div>
        </li>
      </ul>
    </div>
  </main>
</div>
```

botão para compartilhar com redes sociais (share, via *Material Icons*). Após isso, basta repetir a mesma estrutura em cada uma das li's modificando apenas o conteúdo de cada uma delas.

Para fazer o exemplo funcionar ainda precisamos adicionar algumas regras CSS à tag <style> no head da página. Acrescente o código da **Listagem 14** à mesma.

As propriedades são simples e auto inteligíveis. Para finalizar, acrescente também o conteúdo da **Listagem 15** ao fim da sua página, logo abaixo das tags <script>. Essa função servirá para linkar as divs do MDL à biblioteca do Masonry.

Agora pronto, basta rodar a página no browser e veremos algo parecido com a **Figura 13**.

**Listagem 14.** Código CSS para estilo da página.

```
body{
    background-color: #FCFAF1;
    height: 100%;
    margin: 0;
    background-repeat: no-repeat;
    background-attachment: fixed;
}
.mdl-card {
    margin: 10px 10px;
}

.mdl-card__title {
    color: #fff;
    min-height: 176px;
}

.mdl-card__menu {
    color: #fff;
}

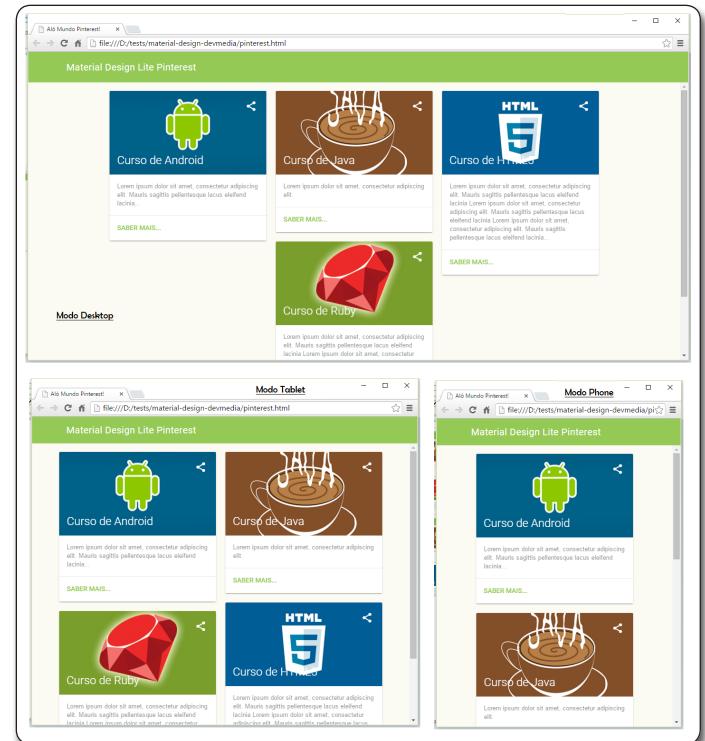
.col-centered {
    display: inline-block;
    float: none;
    margin: 0 auto;
    padding: 0;
    vertical-align: top;
    img{
        text-align: center;
    }
}

.mdl-layout__header {
    color: white;
}
```

**Listagem 15.** Código JavaScript para incluir Masonry.

```
<script>
$(document).ready(function(){
    // cache the window object
    $window = $(window);

    $('.grid').masonry({
        // options
        itemSelector: 'grid-item',
        isFitWidth: true
    });
})</script>
```



**Figura 13.** Página simulador do Pinterest pronta

Essas foram somente algumas das possibilidades de uso desse framework. Como vimos, é muito fácil usar seus componentes, bem como cruzá-los uns com os outros ou integrá-los a bibliotecas externas. Ao longo do artigo fizemos uso do jQuery, do Bootstrap, do Masonry.js, e muitos outros podem ser adicionados.

É importante salientar também que o MDL não é uma solução completa, o que significa que para os propósitos do projeto ele possa não se adaptar melhor em relação a outros, visto que o projeto ainda é bastante recente e alguns componentes estão faltando, principalmente os relacionados a formulários. Até lá continue testando e analisando os mesmos. Bons estudos!

## Autor



### Júlio Sampaio

É analista de sistema e entusiasta da área de Tecnologia da Informação. Atualmente é consultor na empresa Visagio, trabalhando em projetos de desenvolvimento de sistemas estratégicos, é também instrutor JAVA. Possui conhecimentos e experiência em áreas como Engenharia de Software e Gerenciamento de Projetos, tem também interesse por tecnologias relacionadas ao front-end web.



## Links:

### Página oficial do MDL.

<http://getmdl.io/>

### Página do Theme Customizer.

<http://www.getmdl.io/customize/index.html>

# Aplicação de MicroBlog com Django e Python - Parte 2

Aprenda a criar aplicações web fáceis e rápidas com o framework Django e suas extensões

ESTE ARTIGO FAZ PARTE DE UM CURSO

**N**a primeira parte do curso vimos todos os passos para configurar o ambiente, criar o projeto/ aplicação e entender o básico desse poderoso framework para criar formulários, nosso mecanismo de templates, bem como entender as expressões e scripts que o Django faz uso para gerenciar a lógica dos códigos. Nessa segunda parte trataremos de finalizar a implementação com a criação do nosso modelo de entidades, que, por sua vez, será convertido em tabelas no banco de dados SQLite. Veremos quais são as melhores estratégias para gerenciar nossas bases de dados, e que ferramentas podem auxiliar nisso. Também faremos a configuração das nossas views que, em contato direto com os templates, serão responsáveis por exibir a estrutura do site, bem como o estilo (via Bootstrap).

## Atualizando o ambiente

Antes de prosseguir com a nossa implementação, é muito importante que o leitor saiba identificar quando novas versões do Django ou de qualquer outro framework foram lançadas no Python. Geralmente, esse tipo de recurso é facilmente visualizado através da flag `-v` na linha de comando, entretanto isso só funcionará para o core do Python em si, que imprime sua versão, bem como inúmeras outras linhas de código quando o seguinte comando é executado no prompt cmd:

```
python -v
```

## Fique por dentro

Este artigo é útil por explorar recursos importantes do Django Framework, bem como da linguagem de programação Python, tais como conversão de URLs, formatação de valores, manipulação de datas, comunicação com bancos de dados (SQLite 3), navegação, templating, dentre outros. Ao final deste você estará apto a fazer amplo uso do Django para criar quaisquer tipos de aplicações web, com camadas de visão, aplicação e persistência de dados bem definidas e funcionais, além de saber como integrar sua aplicação com frameworks web importantes, como o jQuery e o Bootstrap.

Veja na **Listagem 1** uma parte do log gerado pelo referido comando. Veja que uma simples verificação de versão já faz com que o Python carregue todas as libs principais do seu núcleo, que vão desde threads, I/O, compactação de arquivos (zip), dentre outros.

Para checar a versão do Django instalada, uma vez que ele não tem um binário próprio, precisamos fazer uso da ferramenta utilitária pip que vimos no primeiro artigo. Ela é responsável por gerenciar os pacotes do Python, inclusive suas versões. Execute o seguinte comando:

```
pip freeze
```

Isso será o suficiente para acessar o módulo do Django e retornar com sua versão instalada. Veremos impresso o valor `Django==1.8.5`.

Para atualizar o Django, considerando que a versão 1.9 foi lançada entre a primeira e segunda partes deste artigo, teremos de desinstalá-lo primeiro e instalá-lo novamente em seguida.

**Listagem 1.** Log do comando de versão do Python.

```
import _frozen_importlib # frozen
import imp # builtin
import sys # builtin
import '_warnings' # <class'_frozen_importlib.BuiltinImporter'>
import '_thread' # <class'_frozen_importlib.BuiltinImporter'>
import '_weakref' # <class'_frozen_importlib.BuiltinImporter'>
import '_frozen_importlib_external' # <class'_frozen_importlib.FrozenImporter'>
import '_io' # <class'_frozen_importlib.BuiltinImporter'>
import 'marshal' # <class'_frozen_importlib.BuiltinImporter'>
import 'nt' # <class'_frozen_importlib.BuiltinImporter'>
import '_thread' # previously loaded ('_thread')
import '_thread' # <class'_frozen_importlib.BuiltinImporter'>
import '_weakref' # previously loaded ('_weakref')
import '_weakref' # <class'_frozen_importlib.BuiltinImporter'>
import 'winreg' # <class'_frozen_importlib.BuiltinImporter'>
# installing zipimport hook
import 'zipimport' # <class'_frozen_importlib.BuiltinImporter'>
# installed zipimport hook
...
...
```

Não existe um comando *update* para este módulo em específico. Porém, podemos efetuar os dois passos usando um só comando, o de *upgrade*. Para tanto, execute o seguinte comando no cmd:

```
pip install django --upgrade==1.9
```

Para toda instalação/atualização é necessário informar sempre a versão. Para saber disso, é aconselhado visitar com frequência a página do framework que exibe sempre a última versão. Pronto, com o Django atualizado basta subir o servidor novamente e executar o projeto como antes.

## SQLite: criando os modelos do Blog

O conceito de modelo no universo Django é muito semelhante ao que temos em frameworks de ORM de persistência objeto-relacional, como o Hibernate (para Java) ou o NHibernate (para C#). Criaremos suas classes e atributos na linguagem de programação e mapeia os mesmos com anotações que serão convertidas em comandos SQL para geração das estruturas de tabelas e colunas correspondentes no banco de dados.

No Django, esse trabalho é deveras facilitado pela associação automática feita dos modelos para com as tabelas na nossa base de dados SQLite. Isso por que todo modelo estará intrinsecamente associado a uma instância de tabela no banco e vice-versa.

Na primeira parte do artigo, vimos como mapear as configurações de criação do nosso banco no arquivo *settings.py* (na raiz do diretório do projeto), usando a base SQLite por já estar disponível por padrão junto à instalação do Python, ser fácil e flexível de usar, além de adotar o SQL como as demais bases relacionais de mercado, logo, eventuais alterações para outras bases não surtirão em mudanças tão pesadas. Tais configurações se resumiram a definir a *engine* de geração (cujo valor dado foi “*django.db.backends.sqlite3*”) e o nome da base de dados (*db.sqlite3*). Este segundo parâmetro, em especial, precisa vir acompanhado do caminho absoluto onde o mesmo banco será criado (já ele pode

estar em outra máquina, sendo acessado de forma remota), por isso tivemos de efetuar um *join()* entre a constante *BASE\_DIR* que criamos no início do arquivo e o nome do arquivo propriamente dito. No fim das contas, encontraremos esse arquivo criado no diretório *django\workspace\db.sqlite3*.

A extensão *sqlite3* se refere à versão mais recente do SQLite, mas geralmente esse tipo de banco também pode vir na extensão *.db* ou com nenhuma.

Durante o processo de codificação, precisaremos verificar a consistência tanto física do banco (tabelas, colunas, índices, chaves – primária/estrangeira, etc.), quanto lógica (dados nas tabelas). Portanto, existem duas formas básicas de fazer isso:

1. A primeira é através da própria linha de comando, através da geração de um arquivo txt contendo toda a estrutura de tabelas da base, porém em formato de modelos do Django. Vejamos, execute o seguinte comando no cmd de dentro do diretório workspace:

```
python manage.py inspectdb > meuDB.txt
```

Isso gerará um arquivo de nome *meyDB.txt* no mesmo diretório onde foi executado o comando, com o conteúdo semelhante ao da **Listagem 2**. Nele verificaremos os mesmos imports, declarações de classes, construtores, parâmetros que temos nos modelos convencionais do Django. Veja que cada campo recebe um nome e um tipo de dado (este último veremos com mais detalhes adiante), além da classe *Meta* que declara os metadados daquela tabela, tal como o nome físico da tabela ou se algumas (e quais) de suas colunas precisam ser criadas de forma agrupada.

2. A segunda forma é fazendo uso de alguma ferramenta gráfica de análise DDL/DML de mercado. A vantagem desta abordagem é que podemos ver os dados das tabelas, algo não possível através do item anterior. Além disso, a interface é mais limpa e fácil de manusear, permitindo, inclusive, que façamos alterações na tabela diretamente e vejamos os efeitos surtidos na aplicação quando estiver configurada para consumir tais dados. Para tanto, faremos uso da ferramenta *DB Browser for SQLite*, que é gratuita, simples, além de estar disponível em português para usuários Windows e Mac OS. Para instalar acesse o link disponível na seção **Links** e siga os passos padrão até o fim.

Uma vez com o *SQLiteBrowser* (vamos chamar assim para simplificar) instalado, basta clicar no botão *Abrir banco de dados*, navegar até o diretório onde está o nosso arquivo *db.sqlite3* e clicar em *Abrir*. Após isso, veremos todas as tabelas da sua base criadas até agora, tal como temos na **Figura 1**. Para visualizar os dados das mesmas, vá até a tab *Navegar dados* e selecione a tabela desejada (**Figura 2**).

### Criando modelos: Blog e Categoria

Uma vez entendidos os conceitos relacionados ao SQLite, bem como a melhor maneira de gerenciar os bancos, vejamos agora como criar nossos modelos com as duas primeiras classes de entidade: *Blog* e *Categoria*. Para não se estender muito, focaremos principalmente nestas duas, para exibir na página uma listagem

# Aplicação de MicroBlog com Django e Python - Parte 2

de ambas, assim como mapear suas tabelas de forma relacionada: um blog tem várias categorias, mas cada categoria está relacionada apenas a um blog (1:n).

Portanto, abra o arquivo `models.py` no diretório `django_app` e modifique seu conteúdo para o exibido na **Listagem 3**.

A primeira mudança importante no arquivo está na linha 2 onde importamos um novo módulo do pacote `django.db.models`, o `permalink`, para lidar com links permanentes. Basicamente, este é um utilitário disponibilizado pelo Django em forma de anotações (`@permalink`) para gerar estruturas de links definitivos

para as URLs dos nossos posts. Vejamos alguns detalhes:

- Na linha 6 temos a criação da nossa primeira classe, a de `Blog`. Nela recebemos o modelo do pacote `models` que importamos do Django, para nos auxiliar em alguns passos básicos.

- Das linhas 7 a 10 temos a declaração dos atributos dessa classe, bem como a definição dos tipos de dados de cada um. O Django dispensa tipificação em seus atributos, logo só precisamos definir os respectivos nomes. Para os tipos, faremos uso do pacote `models` também, que traz uma gama considerável de tipos de dados

para mapear os diferentes tipos disponíveis nas bases de dados. Para criar uma coluna de tipo `VARCHAR` ou `TEXT` no banco, o tipo de dados do Django usado deve ser o `CharField` (caso deseje quantificar o tamanho mínimo/máximo de caracteres, sua unicidade, obrigatoriedade, dentre outros) ou `TextField` (caso o campo precise salvar uma quantidade muito grande de caracteres). Para datas usamos o `DataField` (linha 10). O tipo `SlugField` funciona como uma espécie de identificador do post, uma vez que precisamos criar um URL para cada um, salvar no formato texto e esta não pode se repetir. O mesmo acontece com o título, e essa é a razão pela qual criamos ambos os campos com o atributo `unique` setado com o valor `True`.

- Na linha 11 criamos um novo campo de chave estrangeira para relacionar a tabela de blog com a de categoria. Para isso, fazemos uso do tipo `ForeignKey` passando como parâmetro o nome da tabela com a qual ela deve se relacionar.
- Na linha 13 declararmos uma função `__unicode__()` que se encarrega de exibir o valor do objeto quando seu método `toString` for chamado. Basicamente ele retorna o texto do título que é o que vamos exibir no final.
- Na linha 17 declararmos o método `get_url_absoluta()` que se encarrega de formatar a URL do nosso post antes que ela seja salva na base. Veja que anotamos este método com a annotation `@permalink` que descrevemos antes para gerar uma URL automática com base no texto do título. Por exemplo, considerando um título de texto “Alô Mundo Django” (mesmo valor que seria retornado pelo método `__unicode__()`), o valor final retornado pelo método `get_url_absoluta()` seria: `/django_app/view/alô-mundo-django.html`. Esse valor precisa ter a uma configuração específica no arquivo `urls.py` que faremos mais à frente.
- A classe `Categoria`, na linha 20, segue o mesmo processo da classe anterior, contendo apenas os atributos `título` e `url`.

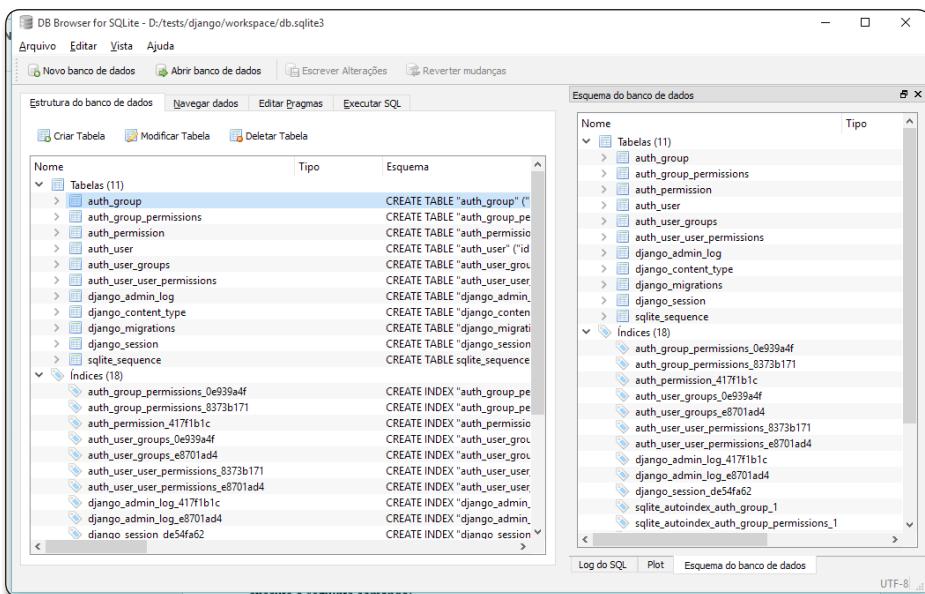


Figura 1. Visualizando tabelas da base via SQLiteBrowser

name	seq
django_migrations	10
django_content_type	8
auth_permission	24
auth_user	1

Figura 2. Visualizando dados da tabela sqlite\_sequence via SQLiteBrowser

**Listagem 2.** Conteúdo do arquivo meuDB.txt.

```
from __future__ import unicode_literals

from django.db import models

class AuthGroup(models.Model):
    id = models.IntegerField(primary_key=True) # AutoField?
    name = models.CharField(unique=True, max_length=80)

    class Meta:
        managed = False
        db_table = 'auth_group'

class AuthGroupPermissions(models.Model):
    id = models.IntegerField(primary_key=True) # AutoField?
    group = models.ForeignKey(AuthGroup)
    permission = models.ForeignKey('AuthPermission')

    class Meta:
        managed = False
        db_table = 'auth_group_permissions'
        unique_together = (('group_id', 'permission_id'),)

class AuthPermission(models.Model):
    id = models.IntegerField(primary_key=True) # AutoField?
    content_type = models.ForeignKey('DjangoContentType')
    codename = models.CharField(max_length=100)
    name = models.CharField(max_length=255)

    class Meta:
        managed = False
        db_table = 'auth_permission'
        unique_together = (('content_type_id', 'codename'),)
...
```

**Listagem 3.** Novo conteúdo da models.py.

```
01 from django.db import models
02 from django.db.models import permalink
03
04 # Create your models here.
05
06 class Blog(models.Model):
07     titulo = models.CharField(max_length=100, unique=True)
08     url = models.SlugField(max_length=100, unique=True)
09     corpo = models.TextField()
10     data = models.DateField(db_index=True, auto_now_add=True)
11     categoria = models.ForeignKey('django_app.Categoria')
12
13     def __unicode__(self):
14         return '%s' % self.titulo
15
16     @permalink
17     def get_url_absoluta(self):
18         return ('ver_post_blog', None, {'slug': self.url })
19
20 class Categoria(models.Model):
21     titulo = models.CharField(max_length=100, db_index=True)
22     url = models.SlugField(max_length=100, db_index=True)
23
24     def __unicode__(self):
25         return '%s' % self.titulo
26
27     @permalink
28     def get_url_absoluta(self):
29         return ('ver_categoria_blog', None, {'slug': self.url })
```

Por exemplo, como nossa aplicação Django se chama “`django_app`”, cada tabela é criada com este prefixo adicionado ao nome da classe, logo, nossa classe de `Blog` se chamará `django_app_blog` na base de dados. Além disso, cada tabela recebe automaticamente uma coluna de chave primária de nome “`id`”, tendo seu valor também incrementado de forma automática, já que a mesma é criada como `AUTOINCREMENT` pelo Django.

Este exemplo está preparado para lidar apenas com um relacionamento 1:n entre categorias e blog. Caso deseje mais de uma categoria por post de blog, será necessário fazer uso do tipo `ManyToMany`. É aconselhado que o leitor de aprofunde sobre a referida classe na documentação do framework, uma vez que mudanças no projeto serão requeridas.

### Configurando acesso de admin

Como a visão principal do blog até o momento será a de usuário comum, precisamos nos certificar de que o usuário `admin`, uma vez logado no sistema, consiga acessar as mesmas propriedades. Para isso, altere o conteúdo do arquivo `admin.py` na pasta da aplicação para o contido na **Listagem 4**.

**Listagem 4.** Conteúdo do arquivo de administração admin.py.

```
01 from django.contrib import admin
02 from django_app.models import Blog, Categoria
03
04 class BlogAdmin(admin.ModelAdmin):
05     exclude = ['data']
06     prepopulated_fields = {'slug': ('titulo',)}
07
08 class CategoriaAdmin(admin.ModelAdmin):
09     prepopulated_fields = {'slug': ('titulo',)}
10
11 admin.site.register(Blog)
12 admin.site.register(Categoria)
```

Como o arquivo de modelos está disponível na forma de um módulo para o Django, precisamos importar suas classes de `Blog` e `Categoria` neste arquivo (linha 2). O segredo dessa implementação consiste em criar uma classe de sufixo `-Admin` para cada uma já criada como modelo. Os atributos `exclude` e `prepopulated_fields` servem para excluir campos do modelo e pré-popular alguns deles quando da inicialização da página, respectivamente. No final, só precisamos registrar cada um dos modelos no objeto global de `admin`, via método `register()`.

### Criando as views do Blog

Nosso blog precisará essencialmente de três ações básicas:

- Exibir todas as categorias e últimos posts;
- Exibir os posts de uma categoria específica;
- Exibir um post.

Para essa alteração teremos de modificar o arquivo `views.py` que, até o momento, consta apenas com o retorno da página de `index`. Para isso, altere o seu conteúdo para o demonstrado na **Listagem 5**.

# Aplicação de MicroBlog com Django e Python - Parte 2

## Listagem 5. Novo conteúdo do arquivo views.py.

```
01 from django.shortcuts import render, render_to_response, get_object_or_404
02 from django_app.models import Blog, Categoria
03 from django.http import HttpResponseRedirect
04
05 def index(request):
06     return render_to_response('django\\home.html', {
07         'categorias': Categoria.objects.all(),
08         'posts': Blog.objects.all()[:5]
09     })
10
11 def ver_post(request, slug):
12     return render_to_response('django\\ver_post.html', {
13         'post': get_object_or_404(Blog, url=slug)
14     })
15
16 def ver_categoria(request, slug):
17     categoria = get_object_or_404(Categoria, url=slug)
18     return render_to_response('django\\ver_categoria.html', {
19         'categoria': categoria,
20         'posts': Blog.objects.filter(categoria=categoria)[:5]
21     })
```

Vejamos algumas considerações:

- Na linha 1 importamos as bibliotecas de renderização das respostas HTTP que o Django processará. Além da render, que já estava inclusa, importamos também as de `render_to_response` (para renderizar diretamente no objeto de `HttpResponse`) e `get_object_or_404` (que busca um objeto de modelo com base nos parâmetros de filtragem passados como argumento, retornando um objeto com o erro HTTP 404 caso nada seja encontrado);
- Na linha 2 importamos as já conhecidas classes de `Blog` e `Categoria`;
- Na linha 5 mudamos a nossa função de `index` padrão que antes redirecionava para a página `index.html` na página de templates. Agora, enviamos para o response a URL “`django\\home.html`” (a barra dupla se deve ao fato do mapeamento que faremos de URL a seguir com uma expressão regular que requer tal configuração), a qual se encarregará de exibir a página de boas-vindas do blog que ainda será criada. Na declaração enviamos um parâmetro com os dois objetos de lista: de categorias (via método `all()` do atributo `objects`, disponível em todos os objetos de modelo) e de posts (também via método `all()`, porém passando o valor `:5` para o vetor, o que fará com que apenas os cinco últimos posts sejam retornados);
- Na linha 11 definimos a função `ver_post()` que receberá o parâmetro `slug`, referente ao identificador da URL do post a ser exibido individualmente. Ela faz uso do método `get_object_or_404()` para recuperar o objeto filtrando pelo atributo `url`, recebido como parâmetro.
- Na linha 16 criamos a função `ver_categoria()` que também recebe o mesmo `slug`, porém para filtrar agora pela `Categoria`. Veja que por termos vários posts dentro de uma categoria, o filtro precisará trazer essa lista também inclusa. Para isso, fazemos uso do método `filter()` do atributo `objects`, disponível em todos os objetos de modelo.

Após isso, precisamos definir também as configurações de URL para permitir e direcionar o fluxo de navegação das páginas referentes a cada post/categoria. Para isso, abra o arquivo `urls.py` e altere seu conteúdo para o que temos na **Listagem 6**.

## Listagem 6. Novo conteúdo do arquivo urls.py.

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^django_app/', include('django_app.urls')),
    url(r'^django_app/view/(?P<slug>[^.]+).html', 'django_app.views.ver_post',
        name='ver_post_blog'),
    url(r'^django_app/categoria/(?P<slug>[^.]+).html', 'django_app.views
        .ver_categoria', name='ver_categoria_blog'),
]
```

Agora as coisas começam a se conectar na aplicação. Além das URL's referentes ao domínio de admin e à raiz do projeto (`django_app/`), agora temos mapeadas as de posts e categorias. As duas últimas url's configuradas passam parâmetros customizados à view para lidar com o conteúdo do texto que será formado para nossas URLs de posts. O valor `(?P<slug>[^.]+)` é uma regex (expressão regular) que mapeia o valor final da URL com extensão final `.html`, além de verificar se o `slug` está correto. O segundo parâmetro, como de praxe, recebe a função Django que deve ser chamada para gerenciar tal requisição na `views.py` (que recebem os mesmos nomes das duas últimas funções criadas na listagem anterior). O último parâmetro, por sua vez, recebe o nome que demos ao primeiro parâmetro da função `get_url_absoluta()`, quando criamos nossos modelos na `models.py`. É muito importante que o leitor se atente a essas configurações para não ter problemas quando for executar a aplicação final.

## Definindo o template

O nosso próximo passo é configurar as páginas de template que usaremos para exibir os dados trafegados das views. Portanto, vá até o diretório `\django\lib\site-packages\django\contrib\admin\templates\django` e crie uma nova página de template de nome `base.html`, e insira o conteúdo da **Listagem 7** à mesma.

Ela basicamente cria uma estrutura vazia com um bloco para o título da página (`<title>`) a ser customizado em cada um individualmente. Da mesma forma que o título de cada página (`<h1>`) e o conteúdo. Como já vimos a estrutura de scripts do Django, não vamos nos adentrar aos detalhes de `include` e `export` que o mesmo disponibiliza. Essa página também precisará ser estilizada com seu CSS, aqui iremos usar o Bootstrap por padrão.

A próxima página é a própria `home.html`. No mesmo diretório, crie-a e adicione o conteúdo da **Listagem 8**.

Nessa página usamos alguns componentes novos, a saber:

- Na linha 1 estendemos nossa página do template `base.html`. Essa é uma outra forma de importar um template em uma página, além

**Listagem 7.** Página de template base.html.

```
<html>
<head>
<title>{% block head_title %}Blog Django DevMedia{% endblock %}</title>
<!-- FAVICON HTMLS-->
<link rel="icon" href="http://www.devmedia.com.br/favicon.png"/>
</head>
<body>
<h1>{% block title %}Bem vindo ao meu block{% endblock %}</h1>
{% block content %}

{% endblock %}
</body>
</html>
```

**Listagem 8.** Página inicial do blog: home.html.

```
01 {% extends 'django/base.html' %}
02 {% block title %}Bem vindo ao Blog Django DevMedia{% endblock %}
03
04 {% block content %}
05   <h2>Categorias</h2>
06   {% if categorias %}
07     <ul>
08       {% for categoria in categorias %}
09         <li><a href="{{ categoria.get_url_absoluta }}>{{ categoria.titulo }}</a></li>
10     {% endfor %}
11     </ul>
12   {% else %}
13     <p>Não há nenhuma categoria adicionada.</p>
14   {% endif %}
15
16   <h2>Posts</h2>
17   {% if posts %}
18     <ul>
19       {% for post in posts %}
20         <li><a href="{{ post.get_url_absoluta }}>{{ post.titulo }}</a></li>
21       {% endfor %}
22     </ul>
23   {% else %}
24     <p>Não há nenhum post adicionado.</p>
25   {% endif %}
26
27 {% endblock %}
```

do *include* que vimos. A expressão `{% extends %}` estabelece uma relação de herança entre a página atual e sua página mãe (`base.html`). A maior diferença em relação a de *include* está no fato de que quando importamos via *exclude* a página inteira se torna um template, e aí só temos de preencher cada uma de suas subseções informando o nome de cada bloco (*block*).

- Na linha 2 sobrescrevemos nosso primeiro bloco: o de título da página. Para fazer isso, basta envolver entre chaves o valor do bloco de mesmo nome que foi declarado no template. O conteúdo HTML deve ser posicionado entre os scripts e os mesmos devem ser fechados sempre com um *endblock*.
- Na linha 4 sobrescrevemos nosso conteúdo (*content*) com a lista de categorias e posts carregados direto da base de dados. Primeiro testamos se suas respectivas listas estão preenchidas no documento (linhas 6 e 17) para, em seguida, iterar sobre as mesmas (linhas 8 e 19). Caso existam categorias/posts a serem exibidos, criamos uma lista `<ul>` com links preenchidos para direcionar para as

```
(django) D:\tests\django\workspace>python manage.py migrate
System check identified some issues:

WARNINGS:
?: (1.8.W001) The standalone TEMPLATE_* settings were deprecated in Django
1.8 and the TEMPLATES dictionary takes precedence. You must put the value
s of the following settings into your default TEMPLATES dict: TEMPLATE_DIR
S.

Operations to perform:
  Synchronize unmigrated apps: messages, staticfiles
  Apply all migrations: admin, contenttypes, auth, sessions
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
    Installing custom SQL...
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying sessions.0001_initial... OK
```

**Figura 3.** Resultado da execução do comando `migrate` no console

URL's retornadas pelo método `get_url_absoluta()` de cada objeto. Caso contrário, exibimos uma mensagem de conteúdo vazio sendo retornado (linhas 12 e 23).

Para testar a nossa aplicação, precisamos executá-la no servidor. Entretanto, antes temos de nos certificar que a base de dados vai estar devidamente criada para tal. Portanto, precisamos “migrar” as alterações que fizemos, principalmente os modelos de entidades, para uma nova versão do projeto. Esse é um recurso que o Django disponibiliza para te permitir evoluir sua aplicação, tanto em configuração quanto em integração aos diferentes componentes com os quais ela possa estar se comunicando, o que também inclui o banco de dados.

Execute então o seguinte comando na sua linha de comandos:

```
python manage.py migrate
```

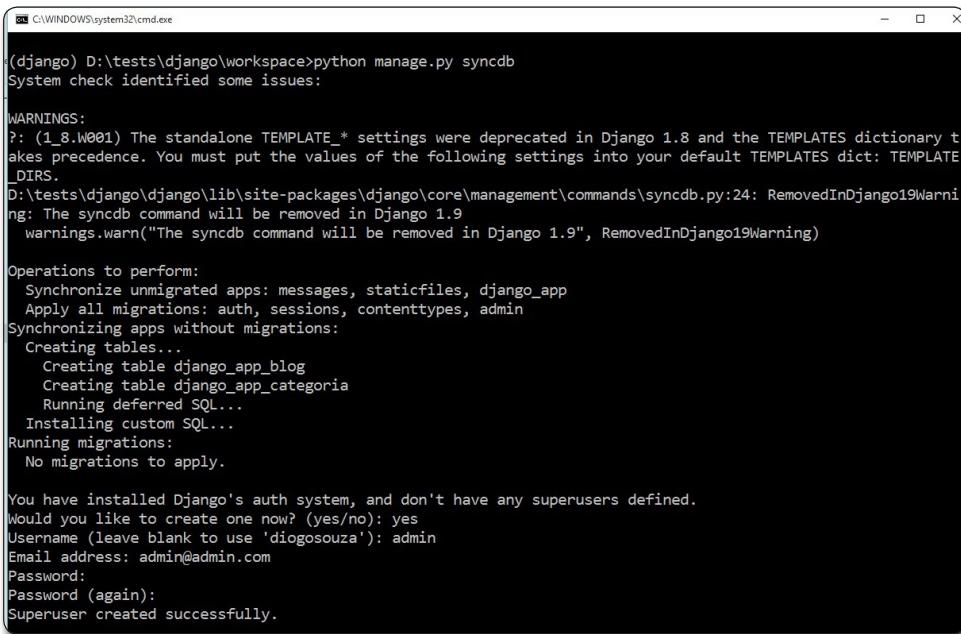
Este comando exibirá uma lista de logs no console semelhante à exibida na **Figura 3**.

Para verificar se ele funcionou, abra o banco novamente no SQLiteBrowser e veja se as novas tabelas foram criadas. Caso não tenham, isso acontece porque o Django cria uma pasta de segurança de nome *migrations* para salvar o histórico de migrações que já foi feito no projeto. Remova a mesma de dentro da pasta raiz da aplicação e execute o comando em seguida:

```
python manage.py syncdb
```

Este comando será responsável por buscar todas as últimas alterações nos arquivos de modelos (*models.py*) e transcrevê-las para a base de dados, forçando o processo todo. O resultado do log pode ser visto na **Figura 4** e a visualização das tabelas recém-criadas na **Figura 5**. Veja como os tipos de dados são criados no SQLite em alusão aos que configuramos nos modelos usando os

# Aplicação de MicroBlog com Django e Python - Parte 2



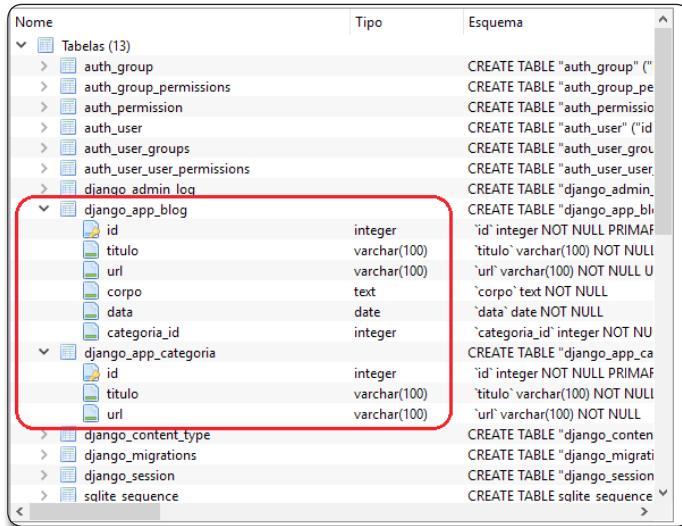
```
(django) D:\tests\django\workspace>python manage.py syncdb
System check identified some issues:

WARNINGS:
?: (1_8.W001) The standalone TEMPLATE_* settings were deprecated in Django 1.8 and the TEMPLATES dictionary takes precedence. You must put the values of the following settings into your default TEMPLATES dict: TEMPLATE_DIRS.
D:\tests\django\lib\site-packages\django\core\management\commands\syncdb.py:24: RemovedInDjango19Warning: The syncdb command will be removed in Django 1.9
    warnings.warn("The syncdb command will be removed in Django 1.9", RemovedInDjango19Warning)

Operations to perform:
  Synchronize unmigrated apps: messages, staticfiles, django_app
  Apply all migrations: auth, sessions, contenttypes, admin
Synchronizing apps without migrations:
  Creating tables...
    Creating table django_app_blog
    Creating table django_app_categoria
    Running deferred SQL...
  Installing custom SQL...
Running migrations:
  No migrations to apply.

You have installed Django's auth system, and don't have any superusers defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'diogosouza'): admin
Email address: admin@admin.com
Password:
Password (again):
Superuser created successfully.
```

Figura 4. Log de execução do comando de syncdb



Nome	Tipo	Esquema
auth_group		CREATE TABLE "auth_group" ("
auth_group_permissions		CREATE TABLE "auth_group_pe
auth_permission		CREATE TABLE "auth_permissio
auth_user		CREATE TABLE "auth_user" ("id
auth_user_groups		CREATE TABLE "auth_user_grou
auth_user_user_permissions		CREATE TABLE "auth_user_userr
django_admin_log		CREATE TABLE "django_admin_
django_app_blog	id	CREATE TABLE "django_app_bla
	titulo	'id' integer NOT NULL PRIMAF
	url	'titulo' varchar(100) NOT NUL
	corpo	'url' varchar(100) NOT NULL U
django_app_categoria	data	'corpo' text NOT NULL
	categoria_id	'data' date NOT NULL
	id	'categoria_id' integer NOT NU
	titulo	CREATE TABLE "django_app_ca
	url	'id' integer NOT NULL PRIMAF
		'titulo' varchar(100) NOT NUL
django_content_type		'url' varchar(100) NOT NULL
		CREATE TABLE "django_conten
		CREATE TABLE "django_migrati
		CREATE TABLE "django_session
salite sequence		CREATE TABLE "sqlite_sequenc

Figura 5. Visualizando tabelas criadas no SQLiteBrowser



Bem vindo ao Blog Django DevMedia

## Categorias

Não há nenhuma categoria adicionada.

## Posts

Não há nenhum post adicionado.

Figura 6. Página home.html exibindo lista de posts/categorias vazia

tipos do Django (encontraremos a relação de tipos aceitos pelo SQLite na URL disponível na seção **Links**).

Agora é só iniciar o servidor novamente e todas as alterações serão consideradas no build. Ao executar a URL [http://localhost:8000/django\\_app/](http://localhost:8000/django_app/) no navegador veremos um resultado semelhante ao da **Figura 6**.

Agora precisamos criar a estrutura de páginas HTML para conter a lógica de exibição dos posts e categorias individualmente. Para isso, dentro do mesmo diretório, crie um novo arquivo de nome `ver_categoria.html` e adicione o conteúdo da **Listagem 9** ao mesmo.

Na linha 1 estendemos novamente do template base.html e na linha 2 sobrecrevemos o título da página (cabeçalho HTML) com um texto concatenado com o título recebido da views.py. O código da linha 3 é responsável por sobrecrever o título do corpo da página.

Listagem 9. Conteúdo HTML da página ver\_categoria.html.

```
01 {% extends 'django\base.html' %} 
02 {% block head_title %}Vendo categoria {{ categoria.titulo }}{% endblock %} 
03 {% block title %}{{ categoria.titulo }}{% endblock %} 
04 
05 {% block content %} 
06   {% if posts %} 
07     <ul> 
08       {% for post in posts %} 
09         <li><a href="{{ post.get_url_absoluta }}>{{ post.titulo }}</a></li> 
10       {% endfor %} 
11     </ul> 
12   {% else %} 
13     <p>Não existem posts para essa categoria.</p> 
14   {% endif %} 
15   <p> 
16     <a href='/django_app'>- Home</a> 
17   </p> 
18 {% endblock %}
```

Em seguida (linhas 5 a 18), sobrecrevemos o conteúdo do corpo, verificando se a variável posts foi setada e, caso positivo, iteramos sobre a lista de posts criando links para cada um deles. Caso contrário, exibimos a mensagem de que não há posts disponíveis. É interessante salientar que quando o usuário chegar neste ponto, nosso método de filtro dos posts por categoria já terá sido chamado, trazendo apenas a lista de posts correspondente. No final, na linha 16, exibimos um link para retornar à página inicial da aplicação.

Crie agora um novo arquivo de nome `ver_post.html` e adicione o conteúdo exibido na **Listagem 10**.

**Listagem 10.** Conteúdo HTML da página ver\_post.html.

```

01 {% extends 'django\base.html' %}
02 {% block head_title %}{% post.titulo %}{% endblock %}
03 {% block title %}{% post.titulo %}{% endblock %}
04
05 {% block content %}
06   <h3>Categoria: <a href='{{ post.categoria.get_url_absoluta }}'>
07     {{ post.categoria.titulo }}</a></h3>
08   <p>
09     <a href='/django_app'><- Home</a>
10   </p>
11 {% endblock %}

```

Essa página traz praticamente as mesmas configurações iniciais da anterior, mudando apenas o seu conteúdo que exibe um link para a categoria relacionada àquele post em específico, bem como o texto do corpo do mesmo (linha 7). No final, o mesmo link de volta ao Home é adicionado.

Uma vez com a estrutura de template devidamente funcional, podemos focar em enxertar dados na base para serem consumidos pela aplicação. Para isso, acesse o SQLiteBrowser e, na tab *Executar SQL*, execute o código SQL representado pela **Listagem 11**. Após executar, não esqueça de clicar em *Escrever Alterações* para confirmar o *commit* dos dados na base, efetivamente.

O leitor pode ficar à vontade para modificar os dados da forma que desejar. Isso já será o bastante para testar a aplicação com dados. Reinicie o servidor e acesse-a novamente. O resultado está representado na **Figura 7**.

Se desejar inserir também a data na exibição de cada post, basta inserir o seguinte código na ver\_post.html:

```
<h4>Data: {{ post.data|date:"d/m/Y" }}</h4>
```

Como o campo *data* foi definido com tipo *DateField*, basta modificar seu padrão (*pattern*) de formatação para o formato de data brasileiro (uma vez que elas são salvas em formato americano, por default). Observe também a forma como as URLs são apresentadas na barra de navegação, exatamente com os nomes que definimos. É interessante que o usuário selecione um padrão para todo o blog (o mais comum é o que usamos, separando as palavras por hífen).

**Listagem 11.** Código SQL de carga de dados.

```

INSERT INTO django_app_categoria(titulo, url) VALUES('Iniciante','iniciante');
INSERT INTO django_app_categoria(titulo, url) VALUES('Java','java');
INSERT INTO django_app_categoria(titulo, url) VALUES('Engenharia de Software','engenharia-software');
INSERT INTO django_app_categoria(titulo, url) VALUES('Front-End','front-end');
INSERT INTO django_app_categoria(titulo, url) VALUES('PHP','php');

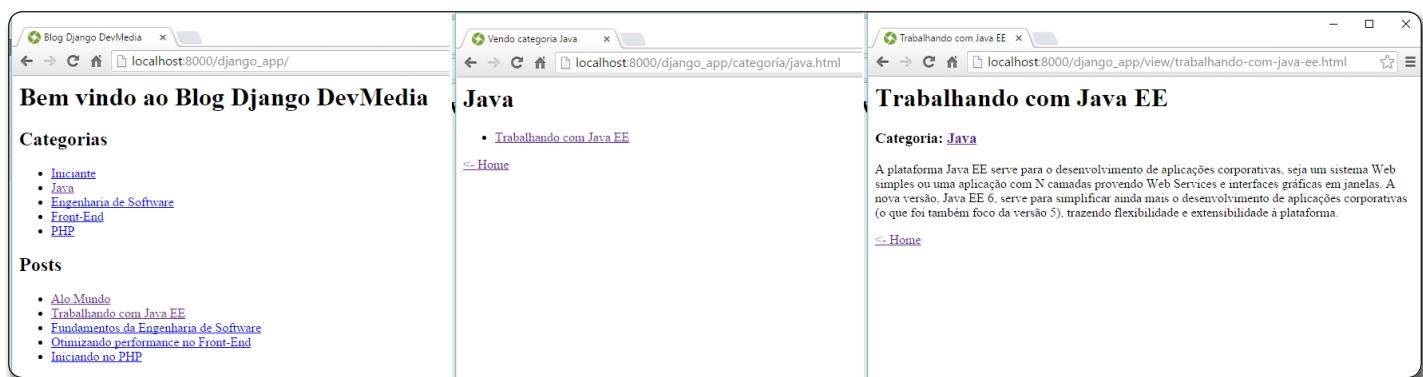
INSERT INTO django_app_blog(titulo, url, data, corpo, categoria_id) VALUES('Alo Mundo','alo-mundo','2015-09-12,'Testando Alo Mundo Iniciante', 1);
INSERT INTO django_app_blog(titulo, url, data, corpo, categoria_id) VALUES('Trabalhando com Java EE','trabalhando-com-java-ee','2015-09-13,'A plataforma Java EE serve para o desenvolvimento de aplicações corporativas, seja um sistema Web simples ou uma aplicação com N camadas provendo Web Services e interfaces gráficas em janelas. A nova versão, Java EE 6, serve para simplificar ainda mais o desenvolvimento de aplicações corporativas (o que foi também foco da versão 5), trazendo flexibilidade e extensibilidade à plataforma', 2);
INSERT INTO django_app_blog(titulo, url, data, corpo, categoria_id) VALUES('Fundamentos da Engenharia de Software','fundamentos-engenharia-software','2015-09-14,'Na maioria das instituições brasileiras de ensino superior, o conjunto de conhecimentos e técnicas conhecido como Engenharia de Software é ensinado em uma ou duas disciplinas dos cursos que têm os nomes de Ciência da Computação, Informática ou Sistemas de Informação. Raramente, em mais disciplinas, muitas vezes opcionais, e muitas vezes oferecidas apenas em nível de pós-graduação. Algumas instituições oferecem cursos de pós-graduação em Engenharia de Software, geralmente no nível de especialização', 3);
INSERT INTO django_app_blog(titulo, url, data, corpo, categoria_id) VALUES('Optimizando performance no Front-End','optimizando-performance-front-end','2015-09-14,'O desenvolvimento de aplicações web que, até pouco tempo, focava em medições de performance apenas na arquitetura servidor e excluía toda a parte gráfica (principalmente por se tratar de GUIs desktop e estas serem deveras performativas), se vê hoje forçado a entender como funciona a web, seus diversos protocolos, navegadores e, mais importante, as linguagens de programação que fazem tudo funcionar', 4);
INSERT INTO django_app_blog(titulo, url, data, corpo, categoria_id) VALUES('Iniciando no PHP','iniciando-no-php','2015-09-15,'linguagem PHP é muito utilizada para o desenvolvimento de WEB e Sites e Intranet. Hoje como cases é possível citar grandes portais da internet, como o Facebook e o Twitter', 5);

```

nes), evitando espaços em branco, uma vez que, se usados, serão transformados em caracteres especiais e deixarão a URL não muito apresentável. Lembre-se: as URLs são muito importantes para os motores de busca, pois definem muito do que aquela página vai dizer. É por isso que ferramentas como Blogger ou WordPress fazem uso extensivo desse tipo de recurso.

## Estilizando o blog

Nossa blog encontra-se completamente funcional, todavia, ainda precisamos estilizar o mesmo com os recursos de design fornecidos pelo Bootstrap. Para isso, faremos uso de um template



**Figura 7.** Visualizando blog final com dados

# Aplicação de MicroBlog com Django e Python - Parte 2

gratuito disponibilizado pelo site Start Bootstrap, cujo link para download encontra-se na seção **Links**. O mesmo já vem preparado para lidar especificamente com blogs e sua estrutura HTML já nos facilita o acoplamento ao nosso blog em si. Portanto, efetue o download e copie os arquivos contidos nas pastas `css.js` e `fonts` para a nossa pasta `static` do `django_project`.

Dentro do arquivo de download encontramos o template inserido na página `index.html`. Abra-a no navegador e veja como o perfil do blog se parece.

A primeira mudança que teremos de fazer é no nosso template `base.html`, uma vez que é ele o responsável por definir o cabeçalho, rodapé e corpo da página, deixando os dados para serem enxertados pelas demais views. Altere o seu conteúdo para o demonstrado na **Listagem 12**.

Essa implementação é um pouco maior, por que traz toda a nova estrutura baseado no Bootstrap, o qual, por sua vez, trabalha muito com divs aninhadas e classes CSS para definir seus estilos. Vejamos os principais detalhes:

- Na linha 2 importamos os arquivos estáticos mais uma vez, isso porque dessa vez carregaremos os arquivos de CSS, JS e fontes diretamente da pasta `/static`;
- Nas linhas 4 a 14 criamos o cabeçalho HTML da aplicação, o qual conterá com as tags `meta` de compatibilidade com o Internet Explorer (linha 6) e de responsividade (linha 7). Além disso, mantivemos a tag `title` da antiga página com um bloco do Django para ser substituído nas demais. Nas linhas 12 e 13 importamos os arquivos de CSS do Bootstrap e do blog-home referente ao template baixado;

**Listagem 12.** Conteúdo do template `base.html` estilizado com o Bootstrap.

```
01 <!DOCTYPE html>
02 {% load staticfiles %} <!-- carrega os arquivos estáticos -->
03 <html>
04   <head>
05     <meta charset="utf-8">
06     <meta http-equiv="X-UA-Compatible" content="IE=edge">
07     <meta name="viewport" content="width=device-width, initial-scale=1">
08     <title>{% block head_title %}Blog Django DevMedia{% endblock %}</title>
09
10    <link rel="icon" href="http://www.devmedia.com.br/favicon.png"/>
11
12    <link href='{% static "css/bootstrap.min.css" %}' rel="stylesheet"/>
13    <link href='{% static "css/blog-home.css" %}' rel="stylesheet"/>
14 </head>
15 <body>
16   <nav class="navbar navbar-inverse navbar-fixed-top" role="navigation">
17     <div class="container">
18       <div class="navbar-header">
19         <button type="button" class="navbar-toggle" data-toggle="collapse"
20             data-target="#bs-example-navbar-collapse-1">
21           <span class="sr-only">Toggle navigation</span>
22           <span class="icon-bar"></span>
23           <span class="icon-bar"></span>
24           <span class="icon-bar"></span>
25         </button>
26         <a class="navbar-brand" href="#">Blog Django</a>
27       </div>
28       <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
29         <ul class="nav navbar-nav">
30           <li><a href="#">Sobre</a></li>
31           <li><a href="#">Quem Somos</a></li>
32           <li><a href="#">Contato</a></li>
33         </ul>
34       </div>
35     </div>
36   </nav>
37   <div class="container">
38     <div class="row">
39       <div class="col-md-8">
40         <h1 class="page-header">
41           {% block title %}Bem vindo ao meu block{% endblock %}
42         </h1>
43         {% block content %}{% endblock %}
44         <ul class="pager">
45           <li class="previous"><a href="#">&larr; Anteriores</a></li>
46           <li class="next"><a href="#">Novas &rarr;</a></li>
47         </ul>
48       </div>
49       <div class="col-md-4">
50         <h4>Categorias do Blog</h4>
51         <div class="row">
52           {% block categorias-content %}
53             {% if categorias %}
54               <div class="col-lg-12">
55                 <ul class="list-unstyled">
56                   {% for categoria in categorias %}
57                     <li>
58                       <a href="{{ categoria.get_url_absoluta }}>{{ categoria.titulo }}</a>
59                     </li>
60                   {% endfor %}
61                 </ul>
62               </div>
63             {% else %}
64               <p>Não há nenhuma categoria adicionada.</p>
65             {% endif %}
66           {% endblock %}
67         </div>
68       </div>
69       <div class="well">
70         <h4>Sua Propaganda Aqui..</h4>
71         <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Inventore,
72           perspiciatis adipisci accusamus laudantium odit aliquam repellat
73           tempore quos aspernatur vero.</p>
74       </div>
75     </div>
76     <hr>
77     <footer>
78       <div class="row">
79         <div class="col-lg-12">
80           <p>Copyright &copy; Todos os direitos reservados 2015</p>
81         </div>
82       </footer>
83     </div>
84     <script src='{% static "js/jquery.js" %}'></script>
85     <script src='{% static "js/bootstrap.min.js" %}'></script>
86   </body>
87 </html>
```

- Nas linhas 16 a 35 criamos a barra de navegação da página, usando o componente navbar do próprio Bootstrap, que cria um menu fixo e responsivo no topo da página com as opções definidas através de uma lista HTML aninhada com links. Estes menus não apontam para nenhuma página funcional, mas o leitor já tem insumos o suficiente para fazer isso no futuro;
- Nas linhas 38 a 47 criamos o cabeçalho e corpo da página, reutilizando o bloco *content* que já havíamos criado antes e que será enxertado pela página de home.html. Em seguida, criamos também os ícones de navegação de artigos, que podemos usar para limitar a quantidade de artigos exibidos por página, paginando os dados diretamente do banco;
- Nas linhas 48 a 74 reusamos a div que lista as categorias do blog. Para isso, criamos um bloco *categorias-content* e já o preenchemos com a iteração dos elementos. O leitor pode estar se perguntando o porquê de não termos adicionado este bloco em outra página, o motivo seria que dessa forma teríamos que iterar repetidamente as categorias em todas as páginas que quiséssemos exibi-las, isto é, como a div de categorias aparecerá em todas as páginas (diferente da div de últimos posts que só aparecerá na home.html) teríamos de mandar a lista de categorias e iterar sobre ela em todas as mesmas páginas. Veja que só recordamos a lógica da home.html e a integrarmos à lista *ul* do template Bootstrap;
- Na linha 76 criamos um *footer* básico apenas com texto, e nas linhas 84 e 85 importamos os arquivos de JavaScript estáticos do jQuery e Bootstrap.

O próximo passo é modificar a página home.html para lidar com as novas regras. Portanto, altere seu conteúdo para o que temos na **Listagem 13**.

**Listagem 13.** Conteúdo da página home.html estilizado com o Bootstrap.

```

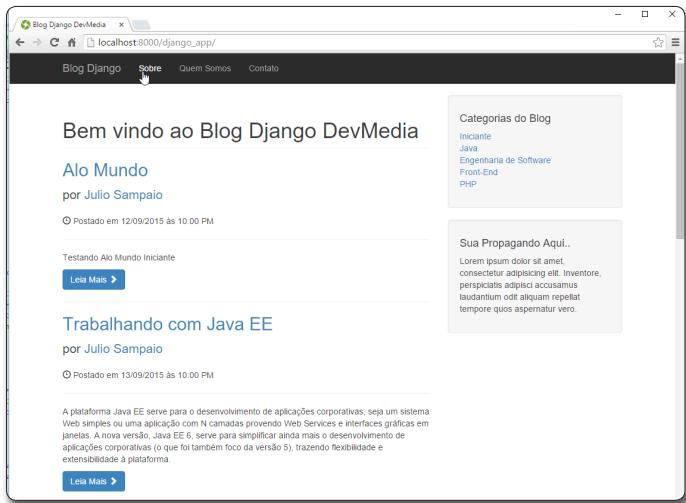
01 {% extends 'django\base.html' %}
02 {% block title %}Bem vindo ao Blog Django DevMedia{% endblock %}
03
04 {% block content %}
05   {% if posts %}
06     {% for post in posts %}
07       <h2>
08         <a href="{{ post.get_url_absoluta }}>{{ post.titulo }}</a>
09       </h2>
10       <p class="lead">
11         por <a href="#">Julio Sampaio</a>
12       </p>
13       <p><span class="glyphicon glyphicon-time"></span>
14         Postado em {{ post.data|date:"d/m/Y" }} às 10:00 PM</p>
15       <hr>
16       <!-- 
17       <hr -->
18       <p>{{ post.corpo }}</p>
19       <a class="btn btn-primary" href="{{ post.get_url_absoluta }}>Leia Mais
20         <span class="glyphicon glyphicon-chevron-right"></span></a>
21       <hr>
22     {% endfor %}
23     {% else %}
24       <p>Não há nenhum post adicionado.</p>
25     {% endif %}
26 {% endblock %}

```

Veja que mantivemos o bloco *content* como antes, porém removemos a iteração sobre os itens de categoria da página. Basicamente o que a listagem faz é concatenar os valores de cada post no loop com a estrutura HTML do template. Na linha 10 criamos um parágrafo para salvar o nome do autor, caso desejemos implementar um gerenciamento de usuários padrão no blog. Na linha 13 postamos a data que terá um ícone associado (aqui podemos implementar outros padrões de formatação de datas), junto com a hora (que não foi implementada mas pode ser usada junto com um tipo apropriado no Django).

Na linha 15 temos uma imagem comentada. Podemos usá-la para exibir uma imagem padrão do post, salvando-a no banco de dados e recuperando-a na referida tag *img*. Finalmente, o corpo do post na linha 17 e um link para navegar até ele na linha seguinte.

Como as alterações foram somente em HTML, basta reiniciar o servidor e recarregar a página no browser. O resultado pode ser conferido na **Figura 8**.



**Figura 8.** Resultado do blog estilizado com o Bootstrap

Se tentarmos clicar em um dos posts ou categorias navegará até as páginas anteriores, que ainda não estarão com o estilo do Bootstrap. O primeiro passo é retornar a lista de categorias para ambas as páginas no views.py, isso porque o template exige esse dado para todas as páginas. Portanto, insira a seguinte linha de código no método return dos métodos *ver\_post()* e *ver\_categoria* em views.py:

```
'categorias': Categoria.objects.all(),
```

Em seguida, siga as instruções das **Listagens 14** e **15** para modificar os arquivos de *ver\_post.html* e *ver\_categoria.html*, respectivamente.

As alterações aqui consistem basicamente em incluir a div de classe *well* que imprime um estilo igual ao do box de categorias da página inicial. Fizemos isso para exibir as informações da categoria do post e da data de postagem. No final (linhas 11 a 13)

## Aplicação de MicroBlog com Django e Python - Parte 2

The screenshot shows a web browser window with the title 'Trabalhando com Java EE'. The URL in the address bar is 'localhost:8000/django\_app/view/trabalhando-com-java-ee.html'. The page has a dark header with navigation links: 'Blog Django', 'Sobre', 'Quem Somos', and 'Contato'. The main content area features a large heading 'Trabalhando com Java EE'. Below it, a box contains the text 'Categoria: Java' and 'Postado em: 13/09/2015'. The main body of the post discusses Java EE's role in corporate application development. To the right, there are two sidebar boxes: 'Categorias do Blog' listing 'Iniciante', 'Java', 'Engenharia de Software', 'Front-End', and 'PHP'; and 'Sua Propaganda Aqui..' with placeholder text.

Figura 9. Página de post visualizada sob novo estilo.

The screenshot shows a web browser window with the title 'Vendo categoria Java'. The URL in the address bar is 'localhost:8000/django\_app/categoria/java.html'. The page has a dark header with navigation links: 'Blog Django', 'Sobre', 'Quem Somos', and 'Contato'. The main content area features a large heading 'Java'. Below it, a box contains a single item: '• Trabalhando com Java EE'. The main body of the post discusses Java EE's role in corporate application development. To the right, there are two sidebar boxes: 'Categorias do Blog' listing 'Iniciante', 'Java', 'Engenharia de Software', 'Front-End', and 'PHP'; and 'Sua Propaganda Aqui..' with placeholder text. At the bottom, there is a copyright notice: 'Copyright © Todos os direitos reservados 2015'.

Figura 10. Página de categoria visualizada sob novo estilo

inserimos o link para home entre as tags *hr* que imprimem uma linha horizontal divisória na página. O resultado pode ser visualizado na **Figura 9**.

**Listagem 14.** Alterações no arquivo ver\_post.html.

```
01 {% extends 'django\base.html' %}  
02 {% block head_title %}{{ post.titulo }}{% endblock %}  
03 {% block title %}{{ post.titulo }}{% endblock %}  
04  
05 {% block content %}  
06   <div class="well">  
07     <h4>Categoria: <a href="{{ post.categoria.get_url_absoluta }}>  
08       {{ post.categoria.titulo }}</a></h4>  
09   <h4>Postado em: {{ post.data|date:"d/m/Y" }}</h4>  
10  </div>  
11  {{ post.corpo }}  
12  <hr>  
13  <p><a href='/django_app'>- Voltar para a Home</a></p>  
14 {% endblock %}
```

**Listagem 15.** Alterações no arquivo ver\_categoria.html.

```
01 {% extends 'django\base.html' %}  
02 {% block head_title %}Vendo categoria {{ categoria.titulo }}{% endblock %}  
03 {% block title %}{{ categoria.titulo }}{% endblock %}  
04  
05 {% block content %}  
06   {% if posts %}  
07     <div class="well">  
08       <ul>  
09         {% for post in posts %}  
10           <li><a href="{{ post.get_url_absoluta }}>{{ post.titulo }}</a></li>  
11         {% endfor %}  
12       </ul>  
13     </div>  
14   {% else %}  
15     <p>Não existem posts para essa categoria.</p>  
16   {% endif %}  
17   <hr>  
18   <p><a href='/django_app'>- Voltar para a Home</a></p>  
19   <hr>  
20 {% endblock %}
```

No arquivo de categorias, inserimos a mesma div *well* para exibir a lista de posts dentro da mesma, assim como o link de home entre tags *hr*. O resultado pode ser conferido na **Figura 10**.

O leitor pode conferir todo o código fonte do projeto disponível no link de fontes do topo do artigo.

Uma das maiores vantagens de se trabalhar com o Django (e o Python) é a simplicidade e os poucos passos que precisamos para começar a fazer qualquer coisa funcionar, principalmente aquelas que em outras linguagens exigiriam enormes esforços de configuração e integração com frameworks e extensões diversos. O Django já nos fornece uma lista de módulos prontos para auxiliar em tarefas como geração de URLs, conversão de tipos, formatação de dados, navegação entre páginas, isso sem falar de um de seus maiores diferenciais: gerar poderosos templates de forma flexível. Bons estudos!

## Autor



### Júlio Sampaio

É analista de sistema e entusiasta da área de Tecnologia da Informação. Atualmente é consultor na empresa Visagio, trabalhando em projetos de desenvolvimento de sistemas estratégicos, é também instrutor JAVA. Possui conhecimentos e experiência em áreas como Engenharia de Software e Gerenciamento de Projetos, tem também interesse por tecnologias relacionadas ao front-end web.



## Links:

### Página de download do SQLiteBrowser.

<http://sqlitebrowser.org/>

### Relação de tipos de dados para campos de modelos no Django.

<https://docs.djangoproject.com/en/1.9/ref/models/fields/#model-field-types>

### Lista de tipos de dados do SQLite 3.

<https://www.sqlite.org/datatype3.html>

### Página de download do template de blog Bootstrap.

<http://startbootstrap.com/template-overviews/blog-home/>

# Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

**Porta 80** é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia. Somos uma equipe composta de gente que entende e gosta do que faz,  
**assim como você.**



## Estrutura

100% NACIONAL.  
Servidores de primeira linha, links de alta capacidade.

## Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

## Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

## 1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.  
**Conheça!**



# Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |  
Aplicações | Streaming | Email corporativo

[porta80.com.br](http://porta80.com.br) | [comercial@porta80.com.br](mailto:comercial@porta80.com.br) | [twitter.com/porta80](http://twitter.com/porta80)

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486