

# Desenvolvimento web com Django

Jackson Gomes \ [jgomes@ceulp.edu.br](mailto:jgomes@ceulp.edu.br)

2018

# Sumário

<b>Prefácio</b>	<b>v</b>
Convenções	v
Ambiente de execução do Python e do Django	vi
Servidor web	vii
<b>1 Introdução</b>	<b>1</b>
1.1 Ambiente do projeto e dependências	1
1.2 Hello World, Django!	2
1.3 Hello World subiu às nuvens!	4
1.3.1 Configuração inicial do Heroku	4
1.4 Conclusão	9
<b>2 Aplicativo Notícias</b>	<b>11</b>
2.1 Configuração inicial	11
2.2 Projeto e aplicativo Django	11
2.3 Criando o aplicativo	11
2.4 Criando o banco de dados	12
2.5 Estrutura do projeto	13
2.6 Criação do Model	14
2.7 Interface Administrativa ou Django Admin	16
2.8 Personalizando o idioma e o fuso horário	19
2.9 Habilitando o aplicativo de notícias no Django Admin	19
2.10 Incrementando o modelo de dados	22
2.11 Personalizando o site	23
2.11.1 Criando a HomePageView	24
2.11.2 Templates	24
2.11.3 URLs	25
2.12 Fazendo deploy no Heroku	26
2.13 Django Admin no Heroku	27
2.14 Ciclo do app no Heroku e o banco de dados SQLite	29
2.15 Testes	29
2.16 Conclusão	34
<b>3 Django ORM</b>	<b>38</b>
3.1 Shell do Django	38

3.2	Criando objetos . . . . .	38
3.3	Atualizando objetos . . . . .	39
3.4	Recuperando objetos . . . . .	39
3.4.1	Recuperar todos os objetos . . . . .	39
3.4.2	Recuperar objetos específicos usando filtros . . . . .	40
3.4.3	QuerySets são lazy . . . . .	41
3.5	Recuperar um objeto único . . . . .	41
3.6	Field lookup . . . . .	42
3.7	Excluindo objetos . . . . .	42
<b>4</b>	<b>Melhorando o modelo de dados do Aplicativo Notícias</b>	<b>43</b>
4.1	Relacionamentos . . . . .	43
4.1.1	Relacionamento um-para-um . . . . .	43
4.1.2	Relacionamentos muitos-para-um . . . . .	44
4.1.3	Relacionamentos muitos-para-muitos . . . . .	45
4.2	Configurando o Django Admin . . . . .	46
4.3	Conclusão . . . . .	49
<b>5</b>	<b>Acesso público: views e templates</b>	<b>50</b>
5.1	Views baseadas em funções . . . . .	50
5.2	Configurações de URLs (acessando views) . . . . .	51
5.3	Templates . . . . .	52
5.4	Views e URLs com parâmetros . . . . .	53
5.5	Gerando URLs nos templates . . . . .	55
5.6	Views baseadas em classes . . . . .	56
5.6.1	View . . . . .	56
5.6.2	TemplateView . . . . .	57
5.6.3	DetailView . . . . .	59
5.6.4	ListView . . . . .	60
5.7	Conclusão . . . . .	60
<b>6</b>	<b>Formulários e mais sobre templates</b>	<b>62</b>
6.1	Model MensagemDeContato . . . . .	62
6.2	Django Admin . . . . .	63
6.3	Formulários . . . . .	63
6.4	FormView . . . . .	64
6.5	Template para o formulário . . . . .	65
6.6	Se tem URL de sucesso, tem URL de erro? . . . . .	67
6.7	Conclusão . . . . .	69
<b>A</b>	<b>Configuração do ambiente Python</b>	<b>70</b>
A.1	Windows . . . . .	70
A.2	Linux (Ubuntu) . . . . .	70
A.3	Ambiente com permissões restritas . . . . .	71
A.4	Usando o virtualenv . . . . .	71
A.4.1	Criação de um ambiente do projeto . . . . .	71

<i>SUMÁRIO</i>	iii
A.4.2 Ativação do ambiente . . . . .	71
A.4.3 Desativação do ambiente . . . . .	72
A.4.4 Instalação de pacotes . . . . .	72
A.5 Usando o pipenv . . . . .	73
A.5.1 Ativação do ambiente . . . . .	73
A.5.2 Desativação do ambiente . . . . .	73
A.5.3 Instalação de pacotes . . . . .	73
<b>B Git</b>	<b>75</b>
<b>C Utilizando Heroku CLI</b>	<b>76</b>
C.1 Fazendo login . . . . .	76
<b>Referências</b>	<b>77</b>

# Lista de Tabelas

2.1	Tabela com tipos de campos no model do Django . . . . .	14
2.1	Tabela com tipos de campos no model do Django . . . . .	15

# Lista de Figuras

1	Exemplo de comunicação cliente-servidor . . . . .	viii
1.1	Janela do browser carregando o projeto django . . . . .	3
1.2	Janela do browser carregando o projeto django no ambiente remoto (produção) . .	7
1.3	Tela da visão geral do aplicativo no Heroku . . . . .	8
1.4	Workflow para o desenvolvimento com Django e Heroku . . . . .	10
2.1	Estrutura do projeto Noticias . . . . .	13
2.2	Tela de autenticação do Django Admin . . . . .	17
2.3	Tela inicial da administração do site no Django Admin . . . . .	18
2.4	Tela inicial da administração do site no Django Admin mostrando o aplicativo “APP_NOTICIAS” . . . . .	20
2.5	Tela da lista de notícias . . . . .	21
2.6	Tela do cadastro de notícia . . . . .	21
2.7	Tela da lista de notícias apresentando registros . . . . .	22
2.8	Tela inicial do software Notícias . . . . .	27
2.9	Tela da lista de notícias apresentando registros . . . . .	35
2.10	Workflow para o desenvolvimento com Testes . . . . .	37
4.1	Adicionar notícia com autor e tags . . . . .	47
4.2	Adicionar notícia - popup de adicionar pessoa . . . . .	48
4.3	Adicionar notícia - popup de adicionar tag . . . . .	48
4.4	Adicionar notícia com autor e tags completo . . . . .	49
6.1	Formulário de contato . . . . .	66
6.2	Formulário de contato com as mensagens de erro . . . . .	67
6.3	Página de confirmação da mensagem de contato . . . . .	68
6.4	Formulário de contato com as mensagens de erro . . . . .	68

# Lista de Códigos-fontes

2.1	Código inicial do model Noticia . . . . .	14
2.2	Código inicial para configuração do Django admin . . . . .	19
2.3	Código inicial para as views do software Notícias . . . . .	24
2.4	Código inicial para o template "home" do software Notícias . . . . .	25
2.5	Código inicial para as URLs do aplicativo Notícias . . . . .	25
2.6	URLs do projeto Notícias . . . . .	26

# Prefácio

Este é um livro sobre tecnologias de desenvolvimento de software para a web com foco no **Django**, um *framework* de desenvolvimento web. Um *framework* representa um modelo, uma forma de resolver um problema. Em termos de desenvolvimento de software para a web um framework fornece ferramentas (ie. código) para o desenvolvimento de aplicações. Geralmente o propósito de um framework é agilizar as atividades de desenvolvimento de software, inclusive, fornecendo código pronto (componentes, bibliotecas etc.) para resolver problemas comuns, como uma interface de cadastro.

O objetivo deste livro é fornecer uma ferramenta para o desenvolvimento de habilidades de desenvolvimento web com Django, com a expectativa de que você comece aprendendo o básico (o “hello world”) e conclua com habilidades necessárias para o desenvolvimento de software que conecta com banco de dados ou fornece uma API HTTP REST, por exemplo.

## Convenções

Os trechos de código apresentados no livro seguem o seguinte padrão:

- **comandos:** devem ser executados no prompt; começam com o símbolo `$`
- **códigos-fontes:** trechos de códigos-fontes de arquivos

A seguir, um exemplo de comando:

```
$ mkdir hello-world
```

O exemplo indica que o comando `mkdir`, com a opção `hello-world`, deve ser executado no prompt para criar uma pasta com o nome `hello-world`.

A seguir, um exemplo de código-fonte:

```
1 class Pessoa:
2     pass
```

O exemplo apresenta o código-fonte da classe `Pessoa`. Em algumas situações, trechos de código podem ser omitidos ou serem apresentados de forma incompleta, usando os símbolos `...` e `#`, como no exemplo a seguir:



```
1 class Pessoa:
2     def __init__(self, nome):
3         self.nome = nome
4
5     def salvar(self):
6         # executa validação dos dados
7         ...
8         # salva
9         return ModelManager.save(self)
```

## Ambiente de execução do Python e do Django

Este livro é voltado para a versão **3.x** do Python e versão **2.x** do Django. Seção **A** apresenta um rápido tutorial sobre configuração de um ambiente Python com **pip** e **virtualenv** ou **pipenv**. Essas ferramentas são fundamentais para a configuração do ambiente de desenvolvimento.

**pip** é um gerenciador de pacotes para o Python (PYPA, [s.d.]). Uma vez que você precise de recursos adicionais, pode instalar pacotes. Por exemplo, o comando a seguir demonstra como instalar o Django:

**virtualenv** é uma ferramenta utilizada para gerenciar ambientes de projeto, isolados entre si e do ambiente global do Python (PYPA, [s.d.]). Com isso, cada projeto pode ter seus pacotes e versões diferentes.

**pipenv** reúne as funcionalidades do **pip** e do **virtualenv** e é uma alternativa mais moderna para o desenvolvimento de software Python (PYPA, [s.d.]).

Este livro não leva em consideração o Sistema Operacional do seu ambiente de desenvolvimento, mas é importante que você se acostume a certos detalhes e a certas ferramentas, como o **prompt** ou **prompt de comando**.

Além destas ferramentas também são utilizadas:

- **Git**
- **Heroku**

O **Git** é um gerenciador de repositórios com recursos de versionamento de código (GIT COMMUNITY, [s.d.]). É uma ferramenta essencial para o gerenciamento de código fonte de qualquer software.

O **Heroku** é um serviço de **PaaS** (de *Platform-as-a-Service*) e fornece um ambiente de execução conforme uma plataforma de programação, como o Python, um tecnologia de banco de dados, como MySQL e PostgreSQL e ainda outros recursos, como cache usando Redis (SALESFORCE.COM, [s.d.]).

---

**Calma! Não pira!**

(In)Felizmente você não vai usar todas as tecnologias lendo o conteúdo desse livro. Fica para outra oportunidade.

---

Para utilizar o Heroku você precisa criar uma conta de usuário. Acesse <https://www.heroku.com/> e crie uma conta de usuário.

Depois que tiver criado e validado sua conta de usuário instale o **Heroku CLI**, uma ferramenta de linha de comando (prompt) que fornece uma interface de texto para criar e gerenciar aplicativos Heroku. Detalhes da instalação dessa ferramenta não são tratados aqui, mas comece acessando <https://devcenter.heroku.com/articles/heroku-cli>.

## Servidor web

Um **servidor web** é um programa que fornece um serviço de rede que funciona recebendo e atendendo requisições de clientes. Um **cliente**, por exemplo, é o browser.

Um **cliente** solicita um arquivo ao **servidor web**, que recebe a solicitação, atende a solicitação e retorna uma resposta para o cliente.

Esse modelo é chamado **cliente-servidor** (WIKIPÉDIA, 2018) e, na web, utiliza o protocolo **HTTP** (de *Hypertext Transfer Protocol*), que determina as regras da comunicação:

- como o cliente deve enviar uma solicitação para o servidor
- como o servidor deve interpretar a solicitação
- como o servidor deve enviar uma resposta para o cliente
- como o cliente deve interpretar a resposta do servidor

Para ilustrar esse processo a Figura 1 demonstra a comunicação entre cliente e servidor.

Como a Figura 1 apresenta, quem inicia a comunicação é o cliente. O servidor recebe a solicitação e retorna uma resposta. A resposta pode ser interpretada como sucesso ou erro. No caso da figura, se o servidor encontrar o arquivo, ele retorna um código de resposta do HTTP com o número 200 e o conteúdo HTML do arquivo `index.html`, caso contrário ele retorna um código de resposta HTTP com o número 404, indicando que o arquivo não foi encontrado.

Como o Django é um framework para desenvolvimento de software esse processo será bastante utilizado e ficará bastante evidente durante seu aprendizado.

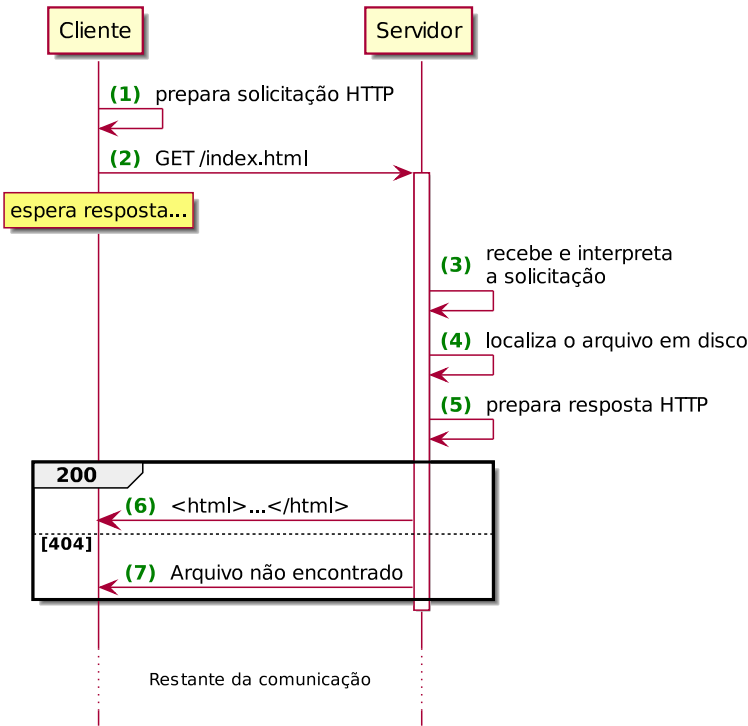


Figura 1: Exemplo de comunicação cliente-servidor

# Capítulo 1

## Introdução

O Django surgiu como uma ferramenta para agilizar o desenvolvimento de software web que, geralmente, tem algumas tarefas comuns (DJANGO SOFTWARE FOUNDATION, [s.d.]). Por exemplo, um software web tem uma interface administrativa, que permite cadastro e gerenciamento de dados, e uma interface pública, que permite consulta dos dados. O Django fornece mecanismos para tornar isso algo bastante rápido de construir.

A estrutura do Django é baseada no padrão de projeto **Model-View-Controller** (MVC). O MVC é um padrão de projeto arquitetural, o que significa que ele determina, inclusive, como os elementos do software comunicam entre si (WIKIPEDIA CONTRIBUTORS, 2018). Desta forma:

- **Model:** representa a camada de dados
- **View:** representa a interface
- **Controller:** representa a lógica que liga os dois elementos anteriores

Falando em **projeto**, esta é uma unidade importante do Django, que organiza o software em **projeto** e seus **aplicativos**. Tanto o projeto como o aplicativo são pacotes Python que podem ser desenvolvidos com o intuito de serem redistribuídos e reutilizados em outros softwares. Isso ficará mais claro nos capítulos seguintes.

### 1.1 Ambiente do projeto e dependências

Uma etapa importante de todo projeto Django é a configuração do ambiente. Antes de prosseguir, garanta que seu ambiente esteja com as ferramentas devidamente configuradas (veja Seção A e depois volte para cá). Além disso, como há mais de uma forma de gerenciar pacotes do projeto, o restante desse livro não vai indicar qual ferramenta utilizar, mas considerar que você já sabe realizar essa tarefa.

O **Django** é distribuído como um pacote do Python. Isso significa que o ambiente do seu projeto precisa ter instalado o pacote `django`.

A seção a seguir demonstra como criar um **hello world** Django.

## 1.2 Hello World, Django!

Crie uma pasta para seu projeto utilizando o programa `mkdir` (ou outra forma). Por exemplo, considere que a pasta do projeto se chame `hello-world-django`. Em seguida, acesse a pasta utilizando o programa `cd`.

```
$ mkdir hello-world-django
$ cd hello-world-django
```

Realize a ativação do ambiente do projeto e instale o pacote `django`.

Com o pacote `django` instalado no ambiente do projeto é hora de criar um **projeto django**. Para isso, utilize o programa `django-admin`. O exemplo a seguir demonstra como criar o projeto `hello_world_django` na pasta local:

```
$ django-admin startproject hello_world_django .
```

O programa `django-admin` está sendo executado utilizando, nesta ordem:

- `startproject`: o comando usado para criar um projeto (há outros)
- `hello_world_django`: o nome do projeto django
- `.`: o local do projeto django (`.` representa a pasta local)

Um **projeto django** possui uma estrutura de arquivos bastante particular, veja:

```
/
├── Pipfile
├── Pipfile.lock
├── hello_world_django/
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

Além dos arquivos do gerenciador de pacotes **pipenv** (`Pipfile` e `Pipfile.lock`) estão:

- `manage.py`: um programa Python utilizado para executar determinadas tarefas no projeto django
- `hello_world_django`: o diretório que contém os arquivos do projeto.

Os arquivos do projeto django (no diretório `hello_world_django`):

- `__init__.py`: indica que o conteúdo da pasta atual pertence a um pacote Python
- `settings.py`: contém configurações do projeto django
- `urls.py`: contém especificações de caminhos, URLs e **rotas** do projeto
- `wsgi.py`: contém a configuração de execução do projeto django em um servidor web

Agora inicie o servidor web local para utilizar o software, executando:

```
$ python manage.py runserver
```

Neste momento você verá uma saída como a seguinte:

```
Performing system checks...

System check identified no issues (0 silenced).

You have 14 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

July 12, 2018 - 00:30:14
Django version 2.0.7, using settings 'hello_world_django.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Essa é a saída do programa Python `manage.py` com o argumento `runserver`.

Neste momento, use o browser e navegue até <http://localhost:8000> ou <http://127.0.0.1:8000>) e você algo como o que a Figura 1.1.

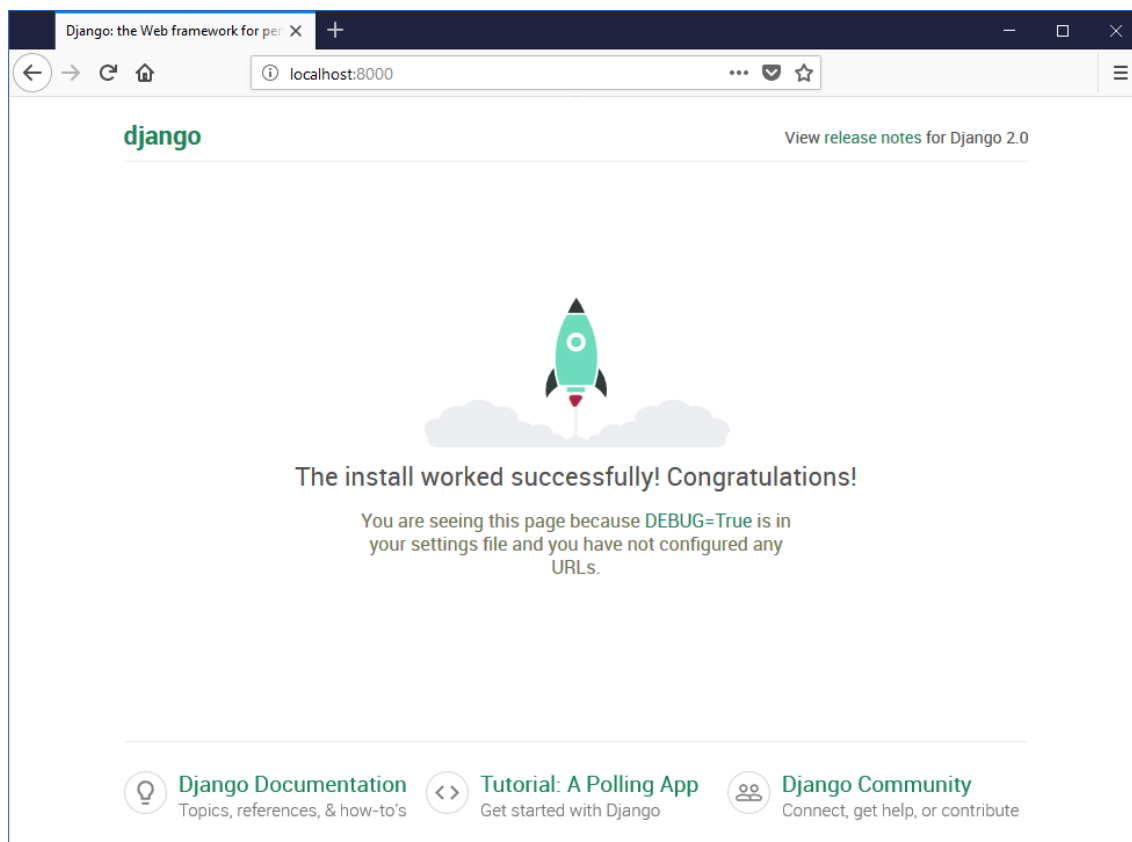


Figura 1.1: Janela do browser carregando o projeto django

A Figura 1.1 apresenta a “página inicial” do projeto Django em execução. Ela é criada automaticamente pelo Django para servir como uma verificação rápida de que tudo está realmente funcionando no seu ambiente de desenvolvimento.

Voltando ao prompt onde você iniciou o servidor web local, perceba que começam a aparecer algumas linhas de **log** à medida que o browser faz solicitações ao servidor web local, por exemplo:

```
[12/Jul/2018 00:32:50] "GET / HTTP/1.1" 200 16348
[12/Jul/2018 00:46:40] "GET / HTTP/1.1" 200 16348
[12/Jul/2018 00:46:40] "GET /static/admin/css/fonts.css HTTP/1.1" 200 423
```

Essas linhas são geradas pelo servidor web local para demonstrar que está ocorrendo alguma atividade, ou seja, está recebendo solicitações de um cliente (o seu browser).

## 1.3 Hello World subiu às nuvens!

Enquanto você está utilizando seu servidor web local somente você consegue acessar seu software web. Por isso, seu software precisa ir para a nuvem. Para fazer isso vamos utilizar o **Heroku**. Antes de continuar, dois conceitos importantes:

- **ambiente de desenvolvimento:** corresponde ao seu computador, contendo os arquivos e recursos que você utiliza para desenvolver o software; o software utiliza o servidor web local e só pode ser acessado por você
- **ambiente de produção:** corresponde ao servidor remoto que você utiliza para disponibilizar seu software para outras pessoas (no caso, o Heroku)

É importante estabelecer uma **regra de ouro**: *só vai para a produção o que está 100% funcionando no ambiente de desenvolvimento*. Utilizar isso como um princípio garante que o software que as pessoas vão utilizar esteja realmente funcionando como deveria. Em outros capítulos você vai aprender a dar essa garantia de uma maneira mais sistemática. Por enquanto, garanta que o servidor web local não apresente erros.

### 1.3.1 Configuração inicial do Heroku

O Heroku precisa que você crie o arquivo **Procfile**, que especifica configurações do ambiente de execução do Python, com o seguinte conteúdo:

```
web: gunicorn hello_world_django.wsgi --log-file -
```

Perceba que “hello\_world\_django” é o nome do projeto Django e pode ser adaptado à sua realidade.

O conteúdo do **Procfile** indica para o Heroku que ele vai utilizar o servidor web **gunicorn** que, diferentemente do servidor web local que você acabou de utilizar, é voltado para o ambiente de produção.

Instale o pacote `gunicorn` no seu ambiente de projeto.

Altere o arquivo `hello_world_django/settings.py`, modificando `ALLOWED_HOSTS` da seguinte forma:

```
ALLOWED_HOSTS = ['*']
```

Essa configuração permite que seu software possa ser acessado a partir de outros computadores (hosts).

Os arquivos são enviados ao Heroku por meio do **Git**, por isso é necessário executar alguns procedimentos, começando pela inicialização do **repositório Git local**. Para isso execute o comando:

```
$ git init
```

Depois configure informações do seu usuário:

```
$ git config user.email "email@servidor.com"
$ git config user.name "Nome do usuário"
```

Substitua `email@servidor.com` pelo e-mail utilizado na sua conta do Heroku.

Em seguida adicione todos os arquivos do diretório atual no próximo **commit**:

```
$ git add .
```

e faça um **commit**:

```
$ git commit -m "Commit inicial para produção"
```

Crie o aplicativo do Heroku usando o comando a seguir:

```
$ heroku create
```

A saída desse comando é importante e é mais ou menos assim:

```
Creating app... done,   lit-sands-61516
https://lit-sands-61516.herokuapp.com/ | https://git.heroku.com/lit-sands-61516.git
```

O nome do aplicativo é criado automaticamente pelo Heroku. Nesse caso é **lit-sands-61516**. A saída também informa a URL do aplicativo (`https://lit-sands-61516.herokuapp.com/`) e a URL do **repositório Git remoto** (`https://git.heroku.com/lit-sands-61516.git`).

Em seguida, conecte seu repositório Git local com o repositório remoto:



```
$ heroku git:remote -a lit-sands-61516
```

Isso faz com que o Git seja configurado para enviar arquivos para o Heroku.

Configure o Heroku para ignorar arquivos estáticos (como arquivos CSS e JavaScript):

```
$ heroku config:set DISABLE_COLLECTSTATIC=1
```

Envie o conteúdo do commit atual para o Heroku:

```
$ git push heroku master
```

O comando atualiza (sincroniza) o repositório local e o repositório remoto. A saída desse programa deve ser algo parecido com isso:

```
Counting objects: 11, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (10/10), done.
Writing objects: 100% (11/11), 3.40 KiB | 1.13 MiB/s, done.
Total 11 (delta 0), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
remote: -----> Installing python-3.6.6
remote: -----> Installing pip
remote: -----> Installing dependencies with Pipenv ...2018.5.18
remote:           Installing dependencies from Pipfile.lock (a8faad)...
remote: -----> Discovering process types
remote:           Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:           Done: 59.3M
remote: -----> Launching...
remote:           Released v5
remote:           https://lit-sands-61516.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/lit-sands-61516.git
 * [new branch]      master -> master
```

Pela saída é possível entender que o Heroku identifica o ambiente de execução do Python, as dependências do aplicativo (nesse caso estão no arquivo `Pipfile.lock`) e o tipo de processo que vai ser criado (`web`, utilizando `gunicorn`). Por fim o Heroku disponibiliza o aplicativo no ambiente de produção (faz o **deploy**).

Para concluir inicialize o aplicativo no Heroku utilizando o comando:

```
$ heroku ps:scale web=1
```

A saída do comando deve ser algo como:

```
Scaling dynos... done, now running web at 1:Free
```

Pronto, agora acesse seu aplicativo no endereço informado pelo Heroku (nesse caso <https://lit-sands-61516.herokuapp.com/>). Perceba que o resultado é o mesmo da execução do seu aplicativo no ambiente local, como ilustra a Figura 1.2.

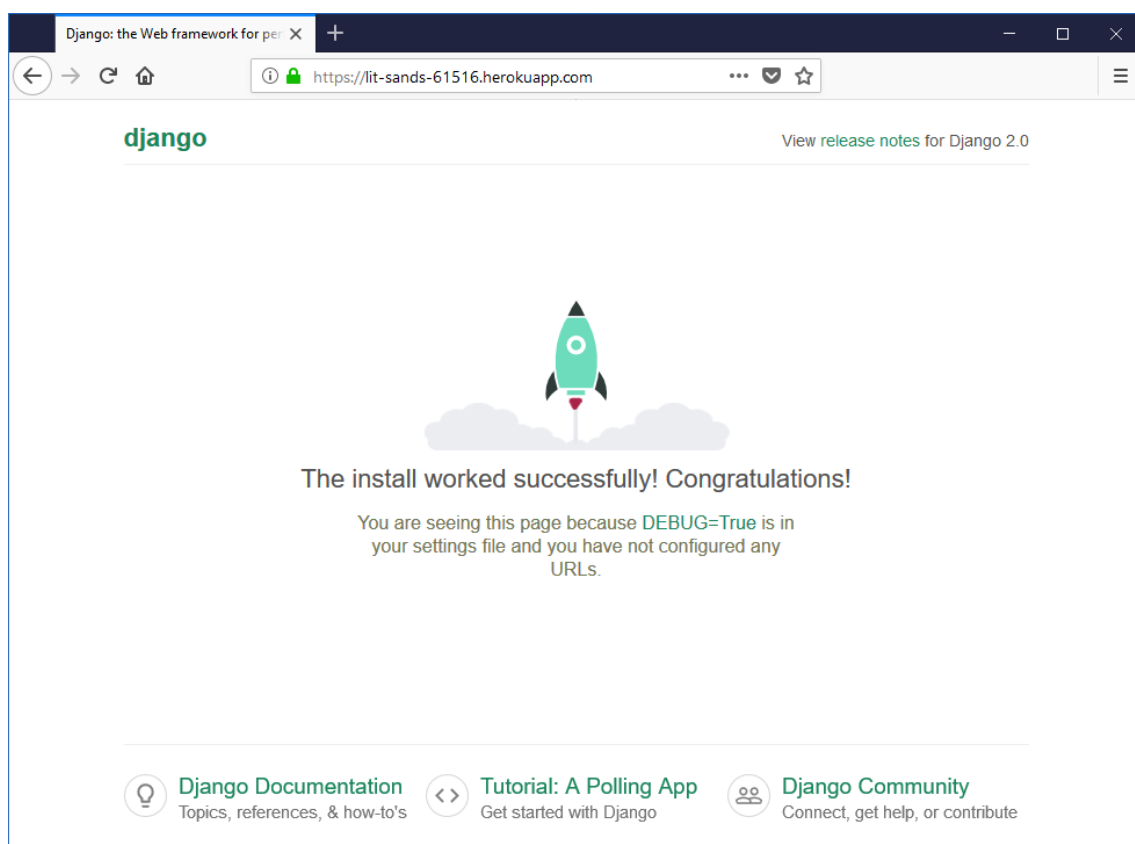


Figura 1.2: Janela do browser carregando o projeto django no ambiente remoto (produção)

Para verificar como seu aplicativo está configurado no Heroku, acesse sua *dashboard*, clique no aplicativo desejado e veja a área de configuração. A tela inicial (*Overview*) é semelhante ao que ilustra a Figura 1.3.

A Figura 1.3 ilustra que a tela *Overview* apresenta os *Add-ons* instalados e permite acessar a configuração deles, mostra as informações dos *Dynos* (no caso, utilizando o nível *free*), as últimas atividades e dá acesso a outras configurações do aplicativo, como *Deploy*.

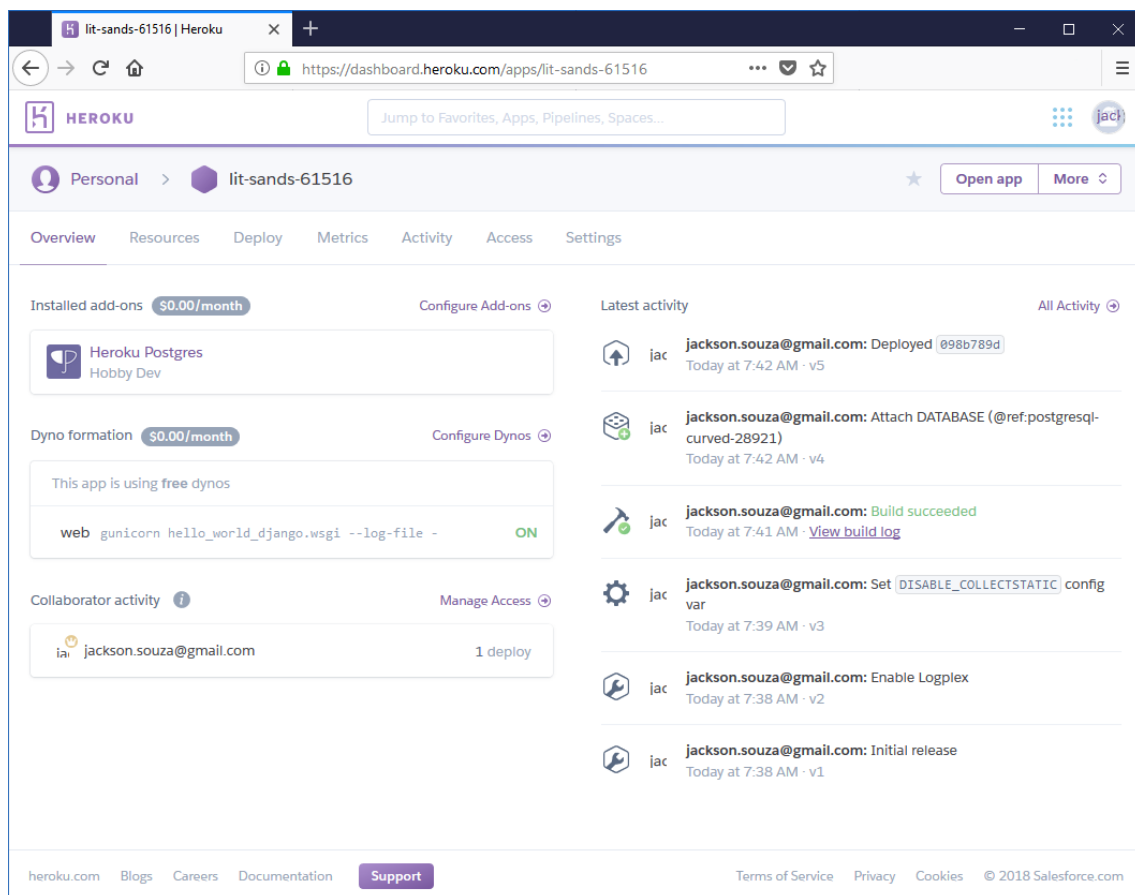


Figura 1.3: Tela da visão geral do aplicativo no Heroku

## 1.4 Conclusão

Esse capítulo apresentou informações sobre o Django, o Heroku e como funciona o *workflow* (fluxo de trabalho) para utilizar o **Git** e o **Heroku CLI** para manter os repositórios local e remoto, bem como fazer o *deploy* da aplicação.

A Figura 1.4 apresenta um resumo do *workflow* até aqui.

Como ilustra a Figura 1.4 o processo é baseado na utilização de uma ferramenta para gerenciando do ambiente do projeto e suas dependências (**virtualenv** ou **pipenv**), **Git** e **Heroku**. Esse *workflow* continuará sendo utilizado e detalhado no restante do livro.

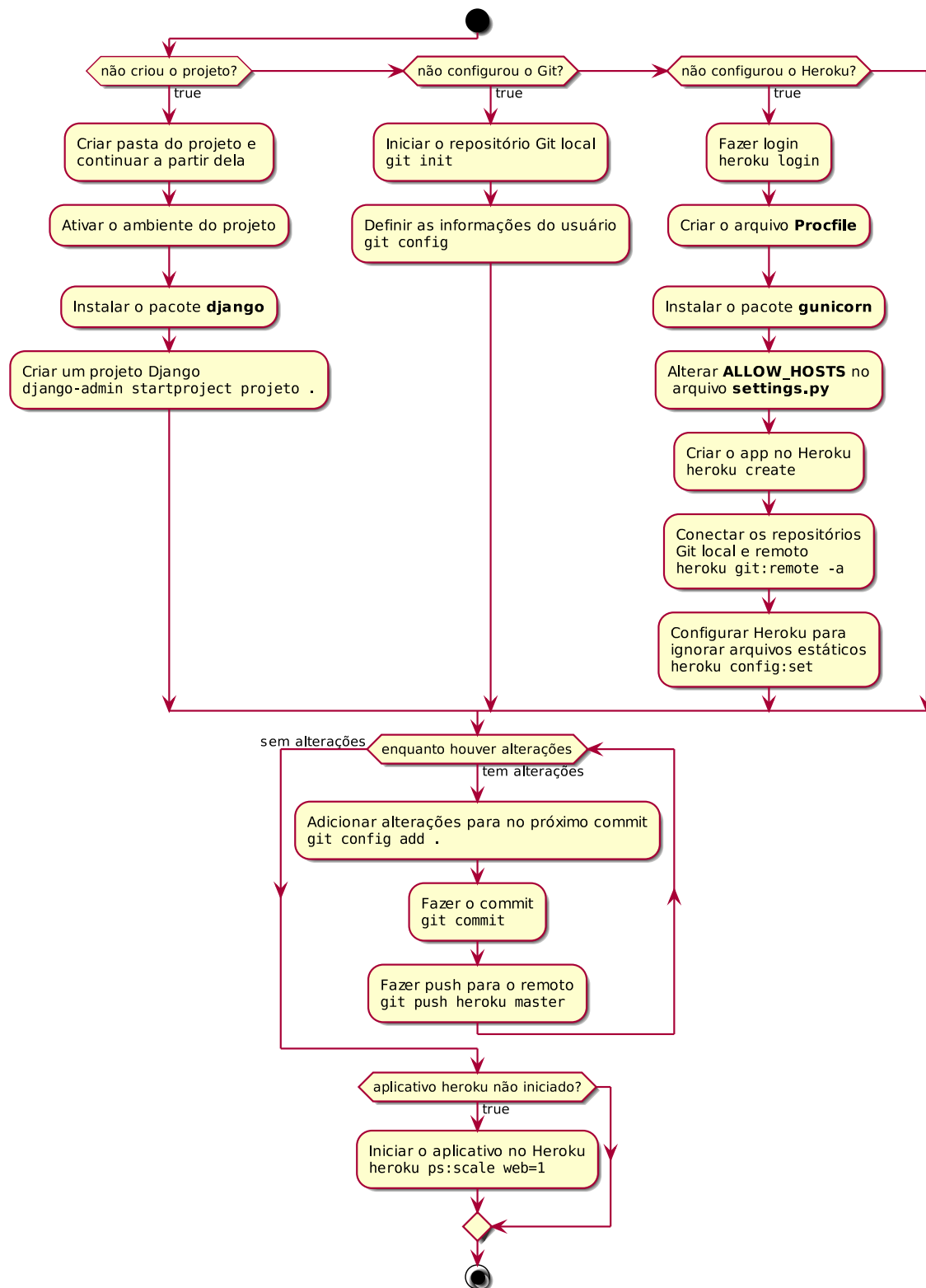


Figura 1.4: Workflow para o desenvolvimento com Django e Heroku

## Capítulo 2

# Aplicativo Notícias

O Capítulo 1 apresentou os conceitos e as ferramentas básicas para o desenvolvimento de aplicativos em Django. Entretanto, o aplicativo **hello-world-django** tem apenas o conteúdo padrão de um aplicativo Django e tem o objetivo de explorar as funcionalidades deste framework.

Este capítulo apresenta o aplicativo **noticias** e vai explorar conceitos de persistência de dados em bancos de dados, mapeador objeto-relacional (ORM) do Django e testes.

### 2.1 Configuração inicial

Siga o *workflow* apresentado no Capítulo 1 para criar a pasta `noticias`, configurar o ambiente do projeto com o pacote `django`, e criar o projeto Django `projeto_noticias`. Se estiver com dúvidas, não se preocupe, volta lá no Capítulo 1 sempre que precisar, depois continua.

### 2.2 Projeto e aplicativo Django

O Django organiza um software em duas unidades principais: **projeto** e **aplicativo**. Um software Django deve conter um projeto e um projeto pode conter nenhum ou muitos aplicativos. Tanto projeto quanto aplicativo podem ser redistribuídos e utilizados em outros softwares Django.

### 2.3 Criando o aplicativo

Para criar o aplicativo `app_noticias` execute o comando:

```
$ python manage.py startapp app_noticias
```

Altere o arquivo `projeto_noticias/settings.py`, modificando `INSTALLED_APPS` para incluir `app_noticias`:

```
1 ...
2 # Application definition
3
4 INSTALLED_APPS = [
5     'django.contrib.admin',
6     'django.contrib.auth',
7     'django.contrib.contenttypes',
8     'django.contrib.sessions',
9     'django.contrib.messages',
10    'django.contrib.staticfiles',
11    'app_noticias',
12 ]
13 ...
```

## 2.4 Criando o banco de dados

Por enquanto vamos utilizar um banco de dados **SQLite**, que não é indicado para ambiente de produção, mas funciona muito bem em ambiente de desenvolvimento. Para criar o banco de dados execute:

```
$ python manage.py migrate
```

Você deve ver uma saída parecida com a seguinte:

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... 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 auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying sessions.0001_initial... OK
```

Esse comando cria o arquivo `db.sqlite3`, que representa o banco de dados **SQLite**.

Esse processo utiliza o recurso de **migrations**, que representam uma forma de definir a estrutura do

banco de dados sem ter que lidar diretamente com um gerenciador ou com instruções em linguagem **SQL**. Isso funciona porque o Django disponibiliza uma ferramenta **ORM** (de *Object-Relational Mapper*).

Um ORM permite que o desenvolvedor Django mantenha o foco em código Python, apenas, e é responsável por toda a comunicação com o banco de dados (no caso o **SQLite**). Essa é uma abordagem conhecida como **model first**.

## 2.5 Estrutura do projeto

A estrutura do projeto já começa ficar maior e, para ajudar a esclarecer, a Figura 2.1 apresenta uma ilustração.

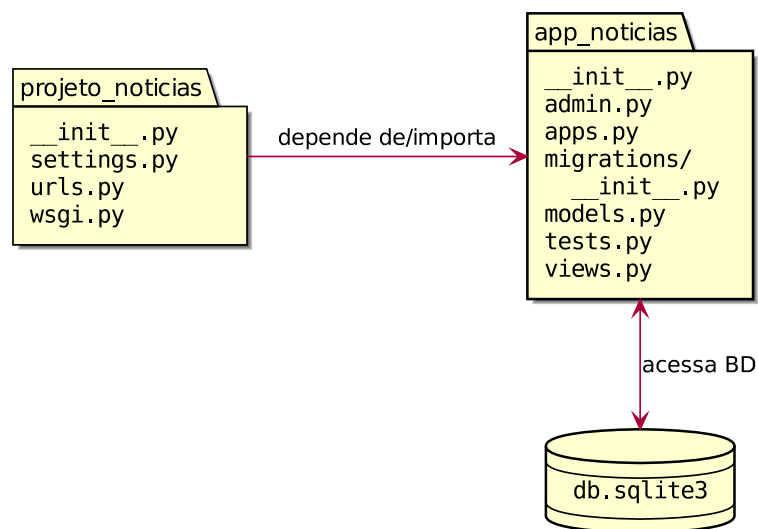


Figura 2.1: Estrutura do projeto Notícias

A Figura 2.1 omitiu os arquivos de dependências (`Pipfile` e `Pipfile.lock`) e o arquivo `manage.py`.

Como você já conhece a estrutura de um **projeto Django**, a composição de um **aplicativo Django** é a seguinte:

- `__init__.py`: indica que o conteúdo da pasta é de um pacote Django
- `admin.py`: contém definições da **interface administrativa**
- `apps.py`: contém configurações do aplicativo, como seu nome
- `migrations`: pasta que contém migrations
- `models.py`: contém definições de **Model**
- `tests.py`: contém definições de testes
- `views.py`: contém definições de **View**

A interface administrativa é fornecida por um pacote do Django chamado `admin` (está habilitado por padrão no arquivo `settings.py`, `INSTALLED_APPS`).

Os arquivos `models.py` e `views.py` representam uma parte importante do modelo MVC: `models.py` representa o modelo de dados do aplicativo, representado conforme regras do ORM do Django.



`views.py` representa as definições de views, que se tornarão formas de comunicação com o cliente. Detalhes destes elementos serão apresentados nas seções seguintes.

## 2.6 Criação do Model

Para o aplicativo `app_noticias` precisamos de um modelo de dados que permita representar uma notícia e seu conteúdo. Fazemos isso modificando o arquivo `app_noticias/models.py` para conter o seguinte:

Código-fonte 2.1: Código inicial do model Noticia

```
1 from django.db import models
2
3 class Noticia(models.Model):
4     conteudo = models.TextField()
```

O que (Código-fonte 2.1) apresenta é a classe `Noticia`, que herda de `models.Model` (`Model` é uma classe fornecida por `django.db.models`). O model `Noticia` contém o campo `conteudo`. Uma instância da classe `TextField` (fornecida por `django.db.models`) é utilizada para indicar que o campo `conteudo` é um campo de texto (contém texto). Mais especificamente, dizemos que é um campo do tipo `TextField`. Há mais tipos de campos para representar datas, números e outros, como mostra a Tabela 2.1<sup>1</sup>.

Tabela 2.1: Tabela com tipos de campos no model do Django

Tipo de campo	Descrição
<code>BigIntegerField</code>	Um número inteiro de 64-bit
<code>BinaryField</code>	Utilizado para armazenar dados binários
<code>BooleanField</code>	Um valor booleano (true/false)
<code>CharField</code>	Uma string
<code>DateField</code>	Uma data
<code>DateTimeField</code>	Uma data com representação de tempo
<code>DecimalField</code>	Um número real, com limite de casas e dígitos
<code>DurationField</code>	Utilizado para armazenar períodos de tempo
<code>EmailField</code>	Uma especialização de <code>CharField</code> para e-mail válido
<code>FileField</code>	Um campo que permite upload de arquivos
<code>FilePathField</code>	Permite selecionar um caminho de arquivo
<code>FloatField</code>	Um número real/float
<code>ImageField</code>	Uma especialização de <code>FileField</code> para imagens
<code>IntegerField</code>	Um número inteiro
<code>GenericIPAddressField</code>	Um endereço IPv4 ou IPv6
<code>NullBooleanField</code>	Um valor booleano que pode receber null

<sup>1</sup>A referência completa do Model do Django contém também os tipos de campos e está disponível em <https://docs.djangoproject.com/en/2.1/ref/models/fields/>

Tabela 2.1: Tabela com tipos de campos no model do Django

<code>PositiveSmallIntegerField</code>	Um número inteiro positivo pequeno
<code>SlugField</code>	Uma string geralmente gerada automaticamente a partir de outro campo
<code>SmallIntegerField</code>	Um número inteiro pequeno
<code>TextField</code>	Uma string longa
<code>TimeField</code>	Uma representação de tempo (horas, minutos e segundos)
<code>URLField</code>	Uma extensão de <code>CharField</code> para URL válida

O próximo passo é criar um **migration**:

```
$ python manage.py makemigrations
```

Esse comando cria um arquivo Python (com um nome gerado automaticamente) dentro da pasta `app_noticias/migrations`. Sem entrar em detalhes agora, o importante é saber que esse arquivo contém uma descrição utilizada pelo ORM do Django para criar ou atualizar o banco de dados de forma apropriada, conforme o **Model**.

Para aplicar a migration (criar a tabela para o Model no banco de dados) basta executar:

```
$ python manage.py migrate
```

Você também pode ver todas as migrations (aplicadas ou não) executando o comando:

```
$ python manage.py showmigrations
```

Neste momento a saída desse comando seria algo como o seguinte:

```
admin
[X] 0001_initial
[X] 0002_logentry_remove_auto_add
app_noticias
[X] 0001_initial
auth
[X] 0001_initial
[X] 0002_alter_permission_name_max_length
[X] 0003_alter_user_email_max_length
[X] 0004_alter_user_username_opts
[X] 0005_alter_user_last_login_null
[X] 0006_require_contenttypes_0002
[X] 0007_alter_validators_add_error_messages
[X] 0008_alter_user_username_max_length
[X] 0009_alter_user_last_name_max_length
```

```
contenttypes
[X] 0001_initial
[X] 0002_remove_content_type_name
sessions
[X] 0001_initial
```

Isso permite identificar quais migrations de quais aplicativos Django estão aplicadas (marcadas com [X]) ou não (marcadas com [ ]).

## 2.7 Interface Administrativa ou Django Admin

O **Django Admin** fornece uma interface administrativa que permite, entre outras coisas, o gerenciamento do banco de dados. Há vários componentes que fornecem funcionalidades padrão para executar quatro tarefas de software que acessa banco de dados que são conhecidas como **CRUD**, que vem de:

- **C - CREATE** representa a funcionalidade de cadastrar
- **R - RETRIEVE** representa a funcionalidade de consultar, recuperar dados
- **U - UPDATE** representa a funcionalidade de atualizar
- **D - DELETE** representa a funcionalidade de deletar, excluir

Crie o **super usuário** para o Django Admin utilizando o comando a seguir e seguindo as instruções da tela:

```
$ python manage.py createsuperuser
```

Agora inicie o servidor web local.

```
$ python manage.py runserver
```

Acesse o software no browser, direcionando para o caminho `/admin`) (ie. <http://localhost:8000/admin>). Você verá uma tela de autenticação semelhante à ilustrada pela figura Figura 2.2.

A Figura 2.2 mostra que a tela de autenticação apresenta um formulário com dois campos: “Username” e “Password”, além do botão “Log in”.

Preencha o formulário utilizando as informações que você forneceu ao criar o super usuário e clique no botão “Log in”. Se a autenticação for bem sucedida você verá a tela inicial da administração do site, como ilustra a Figura 2.3.

A Figura 2.3 mostra que a tela inicial da administração do site permite acessar diversas funcionalidades, de cima para baixo:

- na barra de menu global há links: para a página inicial do site (fora do Django admin), para a tela de atualização da senha e para sair do Django admin
- Na seção “Authentication and Authorization”: há links para o gerenciamento de grupos e usuários

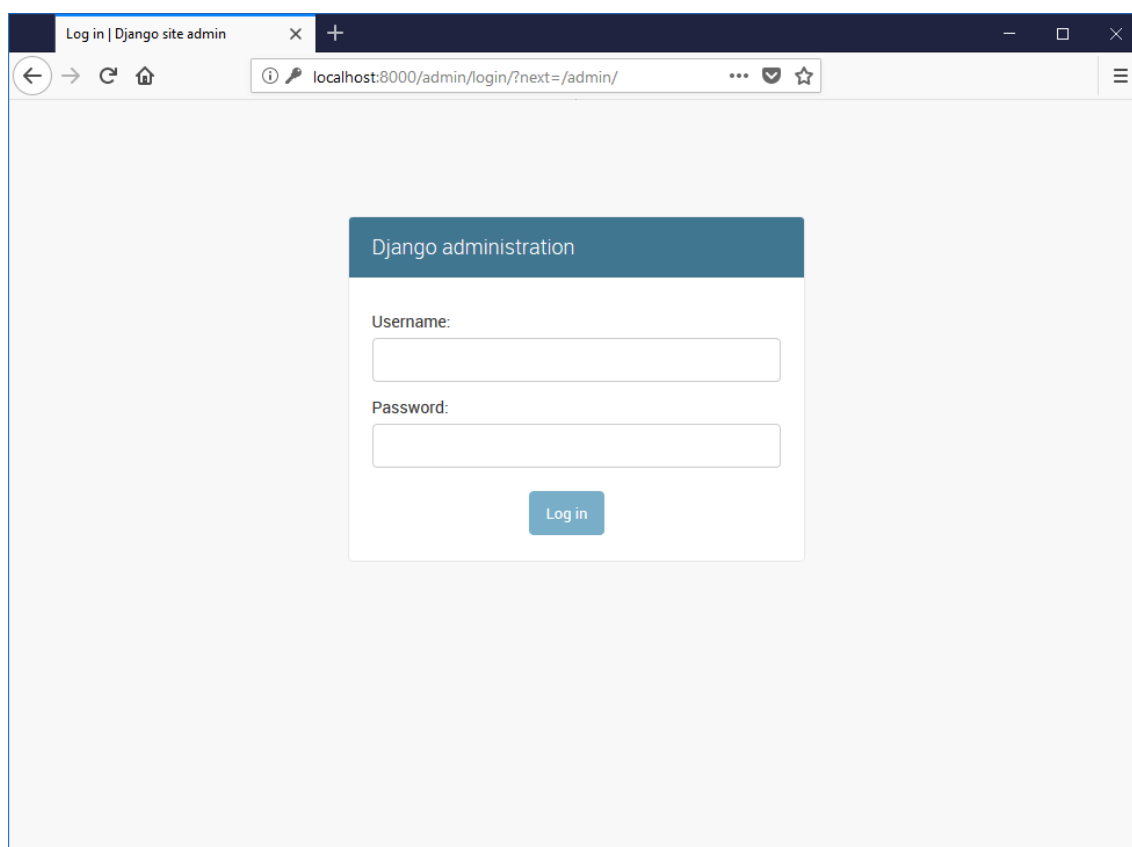


Figura 2.2: Tela de autenticação do Django Admin

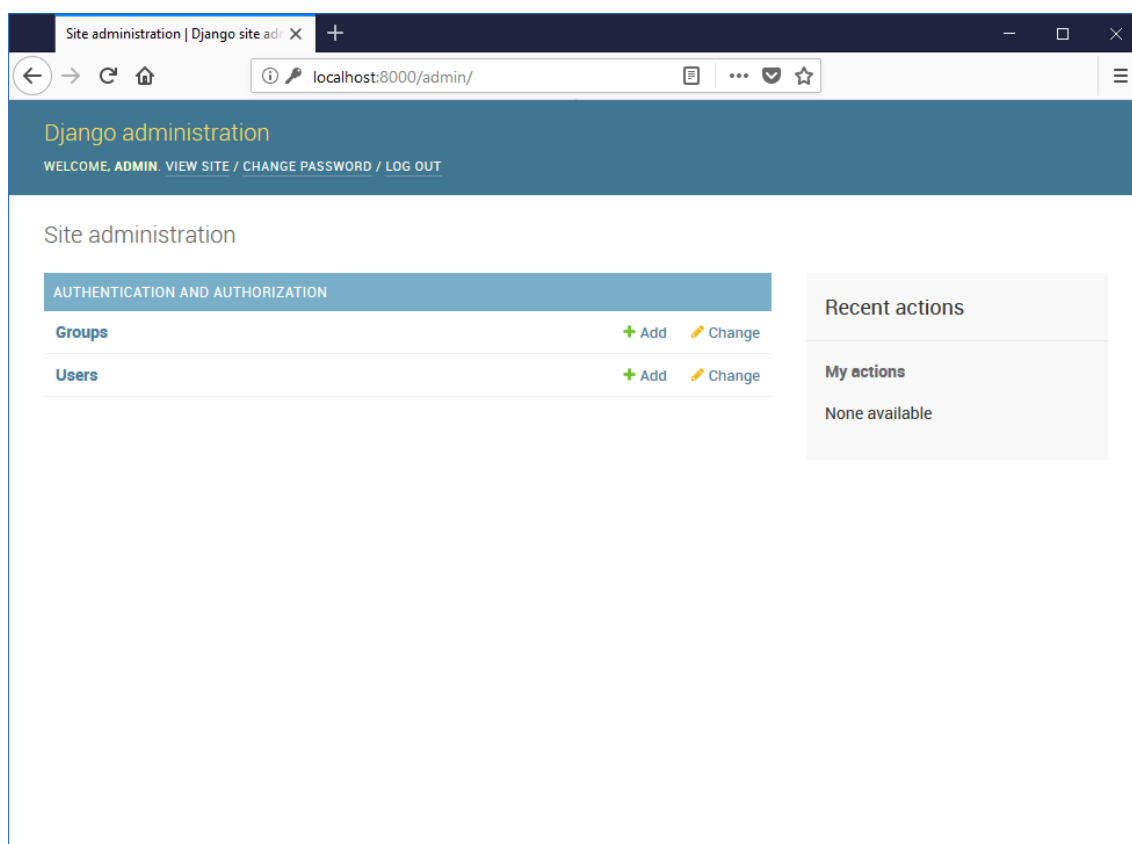


Figura 2.3: Tela inicial da administração do site no Django Admin

- Na seção “Recent actions” há uma lista das ações mais recentes do usuário (nesse caso, nenhuma ação mais recente)

## 2.8 Personalizando o idioma e o fuso horário

O Django foi desenvolvido com foco na **internacionalização**, a tarefa de adaptar a interface gráfica conforme o idioma e as necessidades do usuário.

Você deve ter percebido que as telas do Django admin estão com textos no idioma Inglês, mas seria muito melhor, considerando que o software de notícias seria utilizado por brasileiros, que o idioma fosse o Português. Para fazer isso altere o arquivo `projeto_noticias/settings.py` da seguinte forma:

```
1 ...
2 # Internationalization
3 # https://docs.djangoproject.com/en/2.0/topics/i18n/
4
5 LANGUAGE_CODE = 'pt-br'
6
7 TIME_ZONE = 'America/Araguaina'
8 ...
```

Perceba que foram alterados os valores de duas constantes: `LANGUAGE_CODE`, para `'pt-br'`, e `TIME_ZONE`, para `'America/Araguaina'`. Essas strings alteram o idioma e o fuso horário, respectivamente.

Se o servidor web local ainda estiver em execução, perceba que ele recarregou novamente o projeto Django, para ler as novas configurações. Senão, inicie o servidor web local. Por fim, perceba que o idioma da interface gráfica está como esperado. Por exemplo, agora você deve estar vendo “Administração do Django” bem no início da página, ao invés de “Django administration”.

## 2.9 Habilitando o aplicativo de notícias no Django Admin

Como você percebeu ainda não é possível cadastrar as notícias por meio do Django Admin. Para resolver isso, modifique o conteúdo do arquivo `app_noticias/admin.py` para o seguinte:

Código-fonte 2.2: Código inicial para configuração do Django admin

```
1 from django.contrib import admin
2 from .models import Noticia
3
4 @admin.register(Noticia)
5 class NoticiaAdmin(admin.ModelAdmin):
6     pass
```

Código-fonte 2.2 mostra a configuração do Django admin para o `app_noticias`:

1. importar o model `Noticia` (linha 4)
2. chamar a **função de anotação** `admin.register()` e indicar o model `Noticia`
3. declarar a classe `NoticiaAdmin`, que herda de `ModelAdmin` (disponível em `django.contrib.admin`).

Uma **função de anotação** (**decorator** ou **annotation function**) é um recurso do Python para adicionar metadados a uma classe, por exemplo. O código demonstra que é criada uma interface de administração para o model `Noticia`. Para ver, na prática, o que isso significa, veja a atualização na tela inicial do Django admin, como ilustra a Figura 2.4

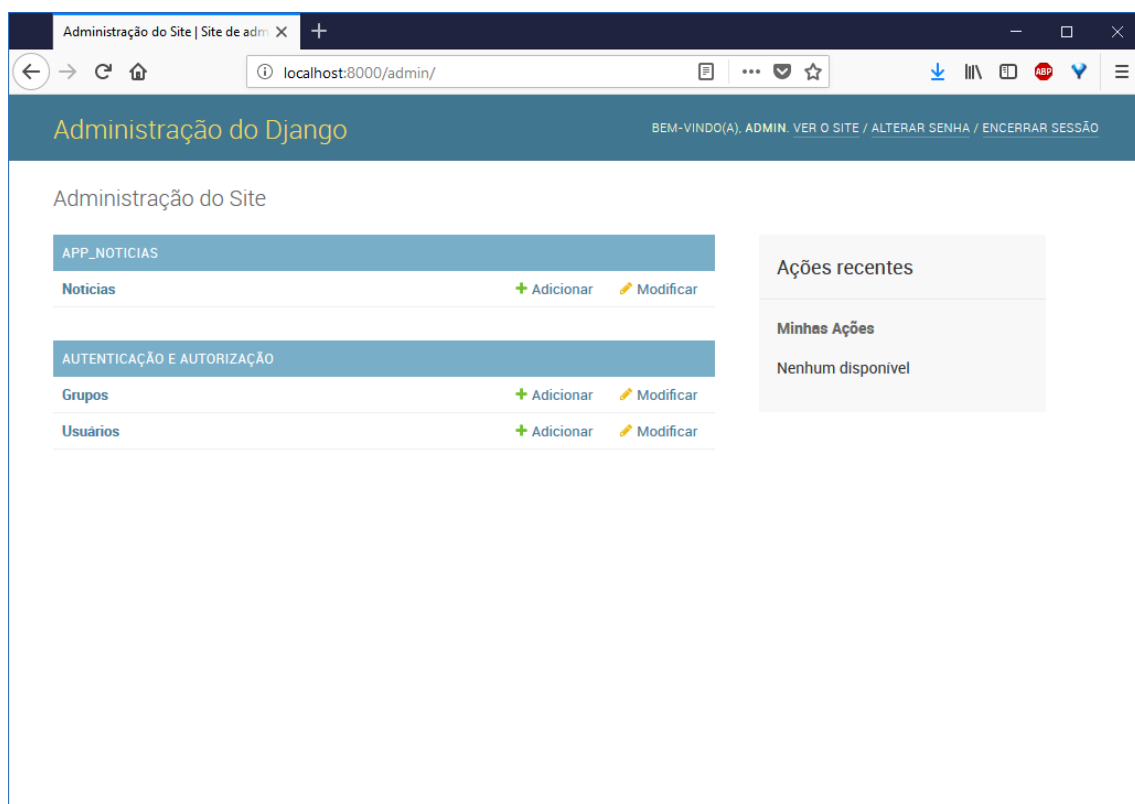


Figura 2.4: Tela inicial da administração do site no Django Admin mostrando o aplicativo “APP\_NOTICIAS”

A Figura 2.4 mostra que há uma nova seção de aplicativos chamada “APP\_NOTICIAS”, que permite gerenciar “Noticias”. Clique no link “Noticias” para acessar a tela da lista de notícias, ilustrada pela Figura 2.5.

A Figura 2.5 mostra a interface gráfica padrão para a lista de notícias. A tela mostra que não há notícias cadastradas e dá acesso ao formulário de cadastro, por meio do botão “Adicionar notícia”. Ao clicar no botão aparece a tela de cadastro de notícia, como mostra a Figura 2.6.

A Figura 2.6 apresenta um formulário contendo um campo “Conteúdo” (`textarea`). Além disso a interface padrão contém três botões: “Salvar e adicionar outro(a)”, “Salvar e continuar editando”

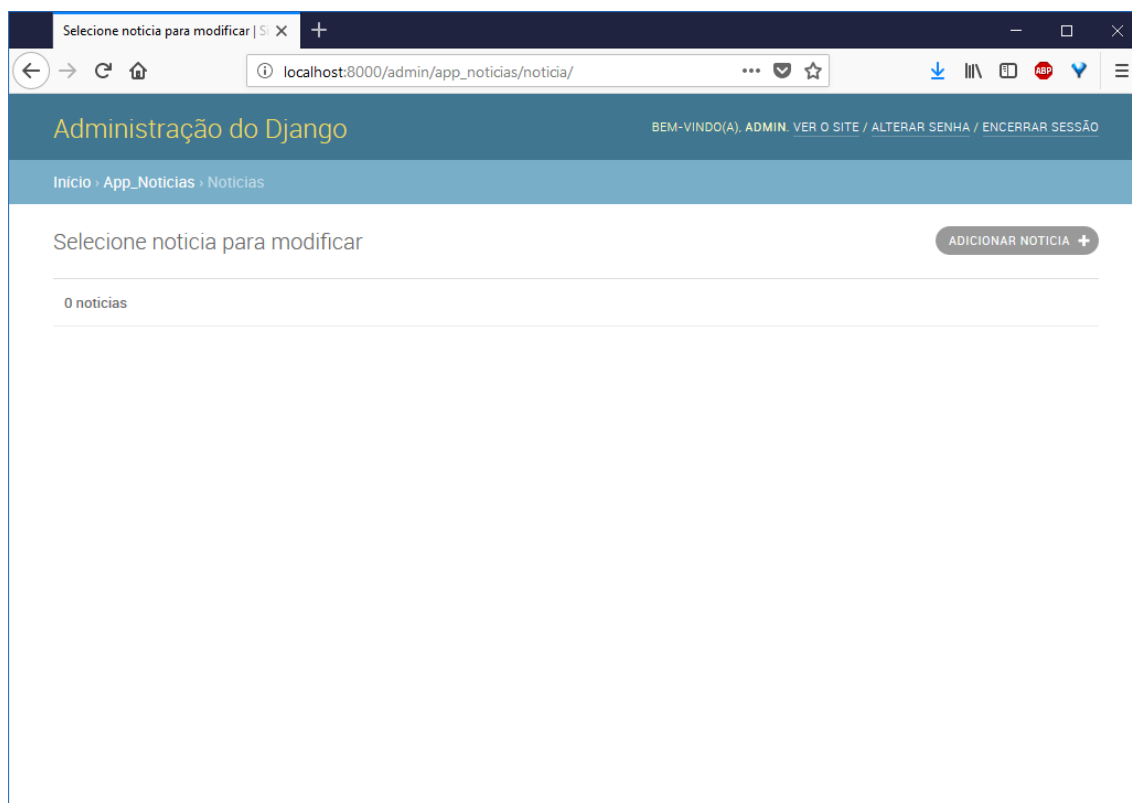


Figura 2.5: Tela da lista de notícias

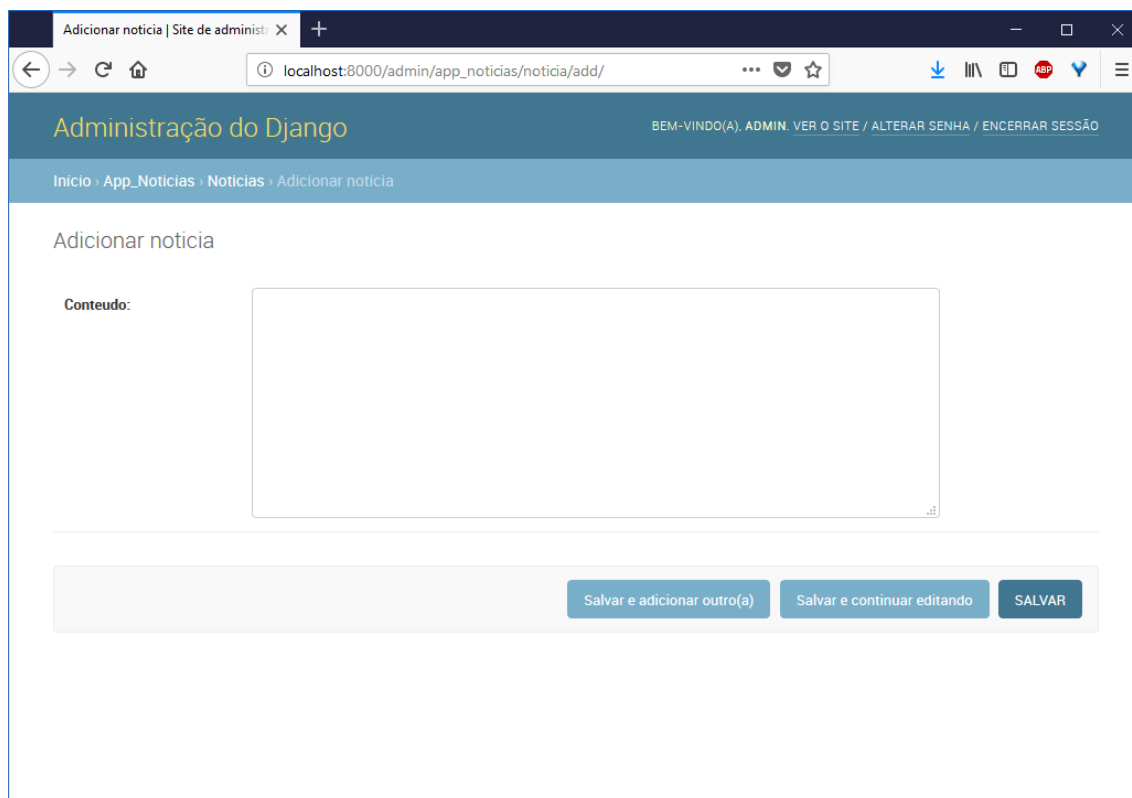


Figura 2.6: Tela do cadastro de notícia



e “Salvar”, que são autoexplicativos. Preencha o formulário para cadastrar uma notícia e clique em um dos botões. Ao clicar no botão “Salvar” o software apresentará a tela da lista de notícias, mas agora com registros, como ilustra a Figura 2.7.

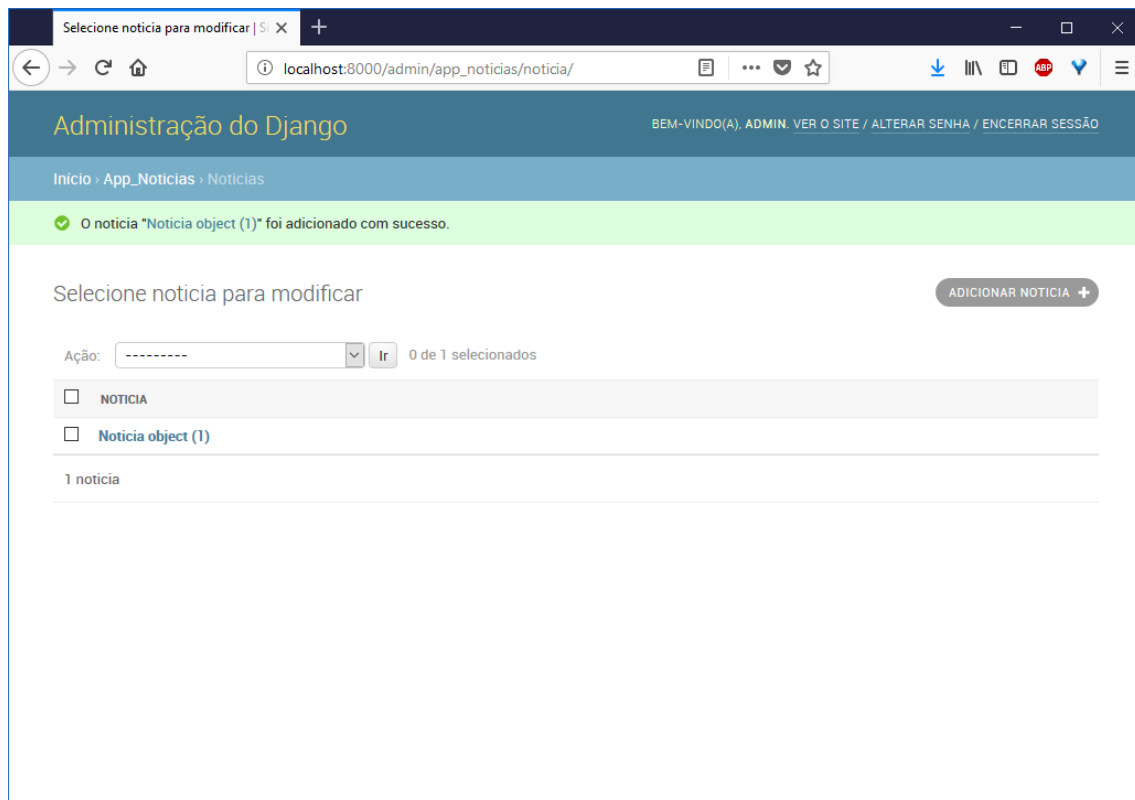


Figura 2.7: Tela da lista de notícias apresentando registros

A Figura 2.7 mostra uma notificação indicando que uma notícia foi cadastrada com sucesso (abaixo do *breadcrumbs*) e que é possível selecionar registros e clicar sobre um deles para editá-lo. Experimente brincar um pouco com essa interface de administração de dados.

## 2.10 Incrementando o modelo de dados

A interface administrativa funciona bem, mas não está muito interessante apresentar cada item da lista com “Noticia object(n)”, não é? Vamos melhorar isso. Modifique o arquivo `app_noticias/models.py` para o seguinte:

```
1 from django.db import models
2
3 class Noticia(models.Model):
4     class Meta:
5         verbose_name = 'Notícia'
6         verbose_name_plural = 'Notícias'
7
8     titulo = models.CharField('título', max_length=128)
```

```
9     conteudo = models.TextField('conteúdo')
10
11     def __str__(self):
12         return self.titulo
```

O código mostra três alterações principais: a inclusão da classe `Meta`, um novo campo `titulo` e o método `__str__()`. A classe `Meta` é utilizada pelo Django admin para configurar a interface administrativa. Já percebeu que a palavra “Notícia” está aparecendo sem o acento agudo? Então, para corrigir isso a classe `Meta` possui dois atributos:

- `verbose_name`: determina o nome literal do model no singular
- `verbose_name_plural`: determina o nome literal do model no plural

O campo `titulo` é do tipo `CharField` e a instanciação fornece mais parâmetros:

- `"Título"`, o primeiro parâmetro, determina o nome literal do campo
- `max_length` determina o tamanho máximo para um valor neste campo (no caso, 128)

A diferença entre os tipos `CharField` e `TextField` é que o primeiro é utilizado para representar uma string de uma linha, enquanto o segundo é utilizado para representar uma string com múltiplas linhas. Na interface gráfica do formulário de cadastro do model no Django admin o `TextField` é apresentado como um elemento `textarea`, enquanto o `CharField` é apresentado como um `input` com `type="text"`.

O campo `conteudo` também passa a ter um nome literal (primeiro parâmetro do construtor de `TextField`).

O método `__str__()` é utilizado para criar uma representação de string de um registro. O conteúdo do código indica, portanto, que a representação string do registro é o valor do campo `titulo`.

Para atualizar o banco de dados é preciso repetir o processo anterior: criar migration, aplicar migration. Entretanto, como já existe um banco de dados, a modificação no model pode gerar erros na sua estrutura. Para não ter problemas agora exclua os arquivos `db.sqlite3` e `app_noticias/migrations/0001_initial.py`.

Execute os comandos a seguir para criar a migration, aplicá-la e criar o super usuário:

```
$ python manage.py makemigrations
$ python manage.py migrate
$ python manage.py createsuperuser
```

Inicie o servidor web local e veja como o Django admin se comporta.

## 2.11 Personalizando o site

O Django chama de “site” a área do software que não utiliza o Django admin. Até o momento a página inicial do site mostra a página padrão do Django, como você já sabe. Vamos modificar isso para que a página inicial apresente a lista das notícias.

### 2.11.1 Criando a HomePageView

As seções anteriores mostraram como trabalhar com o **Model**. Agora é o momento de trabalhar com a **View**. Para isso, modifique o arquivo `app_noticias/views.py` para o seguinte conteúdo:

Código-fonte 2.3: Código inicial para as views do software Notícias

```
1 from django.shortcuts import render
2 from django.views.generic import ListView
3
4 from .models import Noticia
5
6 class HomePageView(ListView):
7     model = Noticia
8     template_name = 'app_noticias/home.html'
```

O Código-fonte 2.3 demonstra a importação da classe `ListView`, fornecida pelo pacote `django.views.generic` e da classe `Noticia`, que está no arquivo `models.py` (linha 5). Além disso, demonstra o conteúdo da classe `HomePageView`, que herda de `ListView` e é utilizada para representar uma **class based view** (view baseada em classe). A classe possui dois atributos importantes:

- `model`: indica o model utilizado na view (nesse caso, o model `Noticia`)
- `template_name`: indica o caminho do **template** utilizado na view

Quando o Django identifica que é necessário apresentar uma View do tipo `ListView` ele inicia um procedimento que passa pela identificação do model e do template. Identificar o model é necessário para saber quais dados devem ser obtidos (nesse caso, uma lista de notícias) e identificar o template para saber como, efetivamente, gerar o HTML que será fornecido como resposta para o browser.

### 2.11.2 Templates

Um **Template** é outro elemento importante do Django. Sua responsabilidade é, efetivamente, criar a interface gráfica de uma View. Embora o nome “view” dê a entender que a própria classe `HomePageView` teria também a descrição da interface gráfica, a maneira mais usual é associar um Template a uma view e a classe a um Controller. Nesse caso o template utilizado está em `app_noticias/templates/app_noticias/home.html`. Qual a razão desse caminho e por que na classe o valor de `template_name` é apenas `'app_noticias/home.html'`?

O Django determina uma estrutura padrão para o projeto e, em relação a templates de aplicativos Django, eles devem estar em uma pasta `templates` dentro do aplicativo. Além disso, os templates devem estar em outra pasta com o nome do aplicativo. Isso explica o caminho do arquivo do template.

Entretanto, na hora de informar para a view qual template utilizar deve-se informar apenas o caminho depois de `app_noticias/templates/`, por isso `app_noticias/home.html`. Veja o conteúdo desse arquivo:

Código-fonte 2.4: Código inicial para o template "home" do software Notícias

```
1 <h1>Notícias</h1>
2 {% for noticia in object_list %}
3 <div>
4     <h2>{{ noticia.titulo }}</h2>
5     <p>{{ noticia.conteudo | linebreaksbr }}</p>
6     <br>
7 </div>
8 {% endfor %}
```

O Código-fonte 2.4 mostra que o conteúdo do template é uma mistura de **HTML** com outra notação. Essa notação, chamada de **linguagem de template do Django** (ou DTL, de *Django Template Language*), usa a **engine** de template padrão do Django, chamada **DjangoTemplates**. Perceba que essa notação tem alguns elementos importantes:

- **tags**: código que está entre {% e %}
- **variáveis**: código que está entre {{ e }}
- **filtros**: modifica a saída de uma variável e utiliza o símbolo “pipe” (|)

Uma **tag** permite, por exemplo, controle de fluxo. No Código-fonte 2.4 a tag **for** cria um bloco de repetição, cujo conteúdo está entre as linhas 4 e 8, e é finalizado pela tag **endfor** (da linha 9). A sintaxe é:

```
{% for objeto in lista %}
... conteudo do bloco ...
{% endfor %}
```

O template possui a variável `object_list`, que representa a lista de notícias cadastradas (isso é informado pela `HomePageView`). Assim, o conteúdo do bloco é repetido para cada elemento de `object_list`, ou seja, o objeto `noticia`.

O conteúdo do bloco utiliza a sintaxe de variável para apresentar o título (linha 5) e o conteúdo da notícia (linha 6). Em especial, a linha 6 utiliza o filtro `linebreaksbr`, que é utilizado para converter quebras de linha em elementos `<br>` do HTML.

### 2.11.3 URLs

Embora o software tenha a `HomePageView` e seu template, é necessário informar ao Django como chegar até essa view, definindo URLs do aplicativo `app_noticias`. Para isso, crie o arquivo `app_noticias/urls.py` com o seguinte conteúdo:

Código-fonte 2.5: Código inicial para as URLs do aplicativo Notícias

```
1 from django.urls import path
```

```
2 from . import views
3
4 urlpatterns = [
5     path('', views.HomePageView.as_view(), name='home'),
6 ]
```

Primeiro, o Código-fonte 2.5 importa as views definidas no aplicativo (linha 2), depois, cria a variável `urlpatterns`, uma lista com definições de URLs. Cada item da lista é criado por meio da função `path()`. Os parâmetros da função são:

- caminho, neste caso `''` representa a raiz
- view, neste caso `views.HomePageView.as_view()`
- nome da URL, neste caso `home`

Cada view do aplicativo `app_noticias` deve ter um caminho registrado no seu arquivo `urls.py`.

Por fim, é necessário incluir as URLs do `app_noticias` no `projeto_noticias`. Para isso, modifique o arquivo `projeto_noticias/urls.py` para que tenha o conteúdo a seguir.

Código-fonte 2.6: URLs do projeto Notícias

```
1 ...
2 from django.contrib import admin
3 from django.urls import path, include
4
5 urlpatterns = [
6     path('admin/', admin.site.urls),
7     path('', include('app_noticias.urls'))
8 ]
```

O Código-fonte 2.6 começa incluindo a função `include()` (do pacote `django.urls`) e, na definição da variável `urlpatterns`, declara um novo caminho (linha 7) que é responsável por incluir as URLs do `app_noticias` no caminho raiz. Dessa forma, ao acessar `http://127.0.0.1:8000/` o Django fornecerá a view `HomePageView` (ou `home`) definida no `app_noticia`. Acesse o software no browser. O resultado deverá ser semelhante ao ilustrado pela Figura 2.8.

A Figura 2.8 ilustra a tela inicial do software apresentando as notícias cadastradas.

## 2.12 Fazendo deploy no Heroku

Seguindo o *workflow* o próximo passo é configurar o repositório local do Git e fazer o deploy no Heroku. Essa etapa fica como exercício. Se tiver dúvidas, volte para o Capítulo 1.

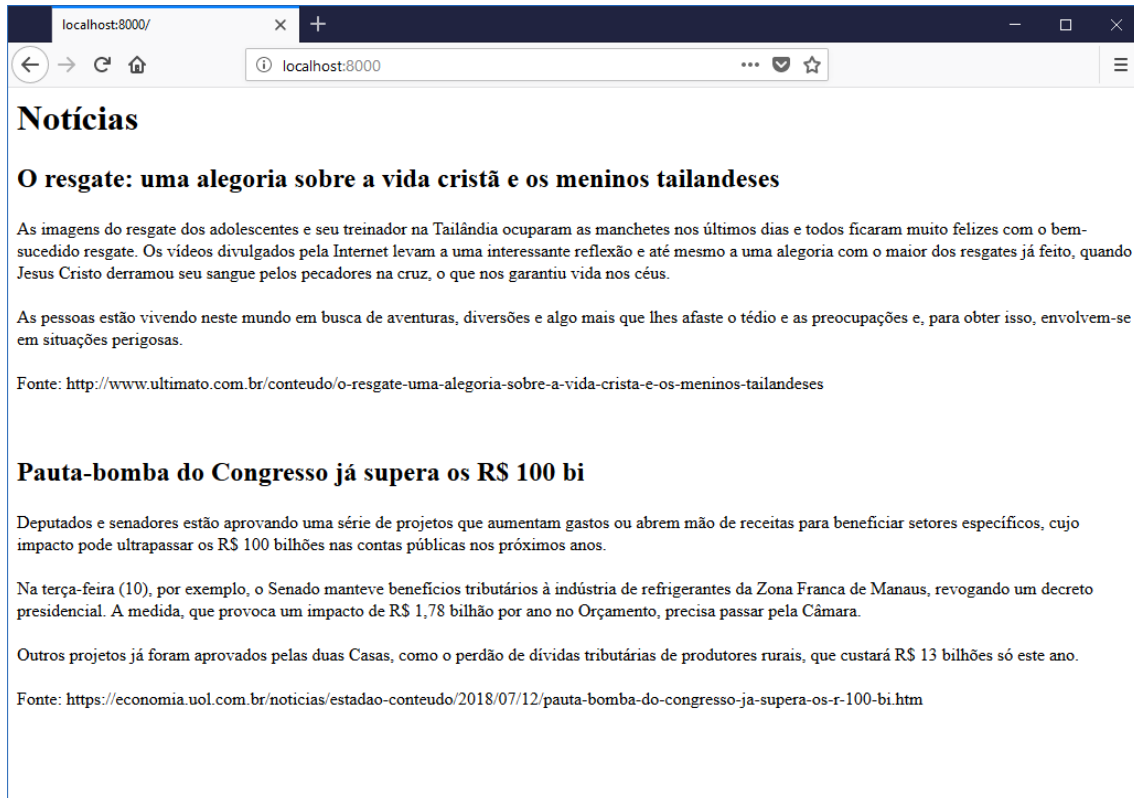


Figura 2.8: Tela inicial do software Notícias

## 2.13 Django Admin no Heroku

Se você chegou até aqui e acessou a interface administrativa do Django (Django Admin) provavelmente ficou frustrado porque a interface está incompleta porque não carregou os arquivos estáticos.

Independentemente de o ambiente de produção estar no Heroku o servidor **gunicorn** geralmente não é utilizado para entregar os arquivos estáticos. Geralmente utiliza-se outro serviço para isso (como o **Amazon S3** ou o **nginx**). Entretanto, é possível instruir o gunicorn a servir os arquivos estáticos realizando algumas configurações. Primeiro, instale o pacote **whitenoise**. Por exemplo:

```
pipenv install whitenoise
```

Depois, configure a constante `MIDDLEWARE` do `settings.py` do projeto Django para incluir `whitenoise.middleware.WhiteNoiseMiddleware`. Exemplo:

```
1 ...
2 MIDDLEWARE = [
3     'django.middleware.security.SecurityMiddleware',
4     'django.contrib.sessions.middleware.SessionMiddleware',
5     'django.middleware.common.CommonMiddleware',
6     'django.middleware.csrf.CsrfViewMiddleware',
7     'django.contrib.auth.middleware.AuthenticationMiddleware',
8     'django.contrib.messages.middleware.MessageMiddleware',
```

```
9     'django.middleware.clickjacking.XFrameOptionsMiddleware',
10     'whitenoise.middleware.WhiteNoiseMiddleware',
11 ]
12 ...
```

Na sequência, ainda no `settings.py`, informe a constante `STATICFILES_STORAGE`:

```
1 ...
2 STATICFILES_STORAGE = 'whitenoise.storage.
    CompressedManifestStaticFilesStorage'
3 ...
```

Por fim, ainda no `settings.py`, configure os valores das constantes `STATIC_ROOT` e `STATICFILES_DIR`:

```
1 ...
2 STATIC_ROOT = os.path.join(BASE_DIR, 'static')
3
4 # Extra places for collectstatic to find static files.
5 STATICFILES_DIRS = (
6     os.path.join(BASE_DIR, 'static'),
7 )
```

Não se esqueça de seguir o **workdlow de deploy** e realizar as etapas de commit e push.

---

## Django, Heroku e arquivos estáticos

Geralmente o desenvolvimento de software Django envolve a utilização de mais de um servidor web: um servido web para o softwer Django, em si (como o **gunicorn**) e outro para os arquivos estáticos (como o **nginx**).

Como esse capítulo demonstrou, é possível utilizar o pacote **whitenoise** para fazer com que o **gunicorn** também entregue arquivos estáticos.

Mas onde estão esses arquivos estáticos, quem os cria e como isso acontece? O Django contém, por padrão, vários arquivos estáticos. O servidor web local não se preocupa com isso porque ele consegue encontrar e entregar os arquivos estáticos automaticamente, mas o Django fornece um comando para gerar esses arquivos:

```
python manage.py collectstatic
```

Esse comando lê as configurações do arquivo `settings.py`, identifica onde os arquivos estáticos do software e os salva no local adequado (veja, principalmente, o valor da constante `STATIC_ROOT`).

No Heroku, o comportamento padrão é executar esse mesmo comando a cada **deploy**. Entretanto, no Capítulo 1 você mudou esse comportamento ao executar:

```
heroku config:set DISABLE_COLLECTSTATIC=1
```

A documentação do Heroku não é muito clara em relação a isso porque deixa como uma situação de exceção desabilitar a execução do comando, mas, se estiver criando o projeto Heroku, tente não executar esse comando (`heroku config:set...`) ou, se já tiver criado o projeto e executado esse comando anteriormente, execute:

```
heroku config:set DISABLE_COLLECTSTATIC=0
```

Isso fará com que o Heroku retorne ao comportamento normal de executar o comando `python manage.py collectstatic` ao fazer um **deploy**.

---

## 2.14 Ciclo do app no Heroku e o banco de dados SQLite

Até agora estamos utilizando o banco de dados SQLite. Já comentei que ele é mais útil no ambiente de desenvolvimento do que na produção e, no caso do Heroku, há uma razão que justifica essa afirmação. Os **dynos**, as unidades de execução do aplicativo no Heroku, são controlados pela estrutura de **PaaS**, o que significa que eles podem ser reiniciados de tempos em tempos e, até mesmo, são reiniciados toda vez que você faz um deploy.

Esse comportamento faz com que o arquivo do banco de dados SQLite seja **sobrescrito com o conteúdo do último commit** toda vez que os **dynos** são iniciados. Então, se você testar seu software durante algum tempo, pode perceber que dados foram perdidos. Isso não é o comportamento adequado para a produção mas, por enquanto, não vamos resolver esse problema. Fica para outros capítulos. Aguenta aí.

## 2.15 Testes

Lembra que comentei no Capítulo 1 que no *workflow* é muito bom garantir que o software esteja funcionando corretamente antes de fazer o deploy? Então, essa seção apresenta uma forma sistemática de garantir isso.

Quando é necessário verificar que um software está funcionando como esperado, geralmente são conduzidos **testes unitários**, que verificam, por exemplo, se a lógica do acesso ou gerenciamento dos dados (Model) está funcionando. Para conduzir esse tipo de teste o Django fornece recursos como a classe `TestCase` (pacote `django.test`).

Para escrever testes deve-se herdar da classe `django.test.TestCase` e declarar um ou mais métodos cujos nomes comecem com `test`. Para exemplificar crie um teste para verificar se os models estão funcionando, se é possível criar e recuperar um registro, por exemplo. Para isso, crie o arquivo `app_noticias/tests.py` com o seguinte conteúdo:



```
1 from django.test import TestCase
2 from .models import Noticia
3
4 class NoticiaModelTest(TestCase):
5     def setUp(self):
6         Noticia.objects.create(titulo='Noticia X', conteudo='Conteudo')
7
8     def test_deve_encontrar_noticia_x(self):
9         noticia = Noticia.objects.get(titulo='Noticia X')
10        self.assertEqual(noticia.titulo, 'Noticia X')
11
12    def test_deve_encontrar_noticia_com_id_1(self):
13        noticia = Noticia.objects.get(id=1)
14        self.assertEqual(noticia.id, 1)
15
16    def test_deve gerar_excecao_para_encontrar_noticia_com_id_2(self):
17        with self.assertRaisesMessage(Noticia.DoesNotExist, 'Noticia
18            matching query does not exist'):
19            noticia = Noticia.objects.get(id=2)
```

Perceba que a classe `NoticiaModelTest` herda de `TestCase` e tem quatro métodos:

- `setUp()`: executado pelo Django antes do primeiro teste. Nesse caso (linha 6) cadastra uma notícia
- `test_deve_encontrar_noticia_x()`: testa positivo para encontrar uma notícia cujo título é “Noticia X”
- `test_deve_encontrar_noticia_com_id_1()`: testa positivo para encontrar uma notícia cujo atributo `id` é igual a 1
- `test_deve_gerar_excecao_para_encontrar_noticia_com_id_2()`: testa positivo para gerar uma exceção ao tentar encontrar uma notícia cujo atributo `id` tenha valor 2

Quando o software utiliza um banco de dados o Django cria automaticamente um banco de dados para teste. Então, não se preocupe com seu banco de dados de desenvolvimento ou de produção.

A classe `NoticiaModelTest` é chamada também de **suíte de testes** (contém vários testes). O Django executa cada teste na ordem em que são encontrados. Os nomes dos testes são realmente longos, mas a expectativa é que sejam autoexplicativos. Vamos analisar cada teste.

O `test_deve_encontrar_noticia_x()` realiza uma consulta no model `Noticia` buscando uma notícia com atributo `titulo` igual a 'Noticia X' (linha 9). Na sequência usa o método `assertEqual()` para realizar uma **asserção**.

Uma asserção é uma comparação entre um valor esperado e um valor obtido. Por exemplo, considere que o valor esperado seja 1, se o valor obtido for igual a 1, então a asserção **passa**, se o valor obtido for igual a 2, então a asserção **falha**. Um teste pode ver mais de uma asserção. Um teste passa se todas as suas asserções também passam. Um teste falha se uma de suas asserções falha.

Como o método `setUp()` criou uma notícia com o título 'Noticia X', é esperado que realmente haja uma notícia cadastrada com atributo `titulo` igual a 'Noticia X', ou seja, é esperado que o

teste passe.

O `test_deve_encontrar_noticia_com_id_1()` realiza uma consulta no model `Noticia` buscando uma notícia com atributo `id` igual a 1.

O `test_deve gerar_excecao_para_encontrar_noticia_com_id_2()` usa um contexto criado pelo método `assertRaisesMessage()` para gerar com a expectativa de gerar uma exceção do tipo `Noticia.DoesNotExist` ao tentar encontrar um notícia com atributo `id` igual a 2.

Use o comando a seguir para executar os testes:

```
$ python manage.py test
```

A saída vai ser parecida com o seguinte:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...
-----
Ran 3 tests in 0.004s

OK
Destroying test database for alias 'default'...
```

É importante interpretar essa saída:

- foram executados 3 testes (`Ran 3 tests`)
- os testes executaram em um tempo total de 0.004 segundos (`in 0.004s`)
- todos os testes passaram (`OK`)

Para gerar uma falha proposital nos testes, modifique o `test_deve_encontrar_noticia_com_id_1()` e altere a asserção para `self.assertEqual(noticia.id, 2)`. Claro, você já sabe que o teste vai falhar, mas faça isso mesmo assim para entender o que muda na saída. Seria mais ou menos assim:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F..
=====
FAIL: test_deve_encontrar_noticia_com_id_1 (app_noticias.tests.NoticiaModelTest
)
-----
Traceback (most recent call last):
  File "/mnt/d/Developer/djangobook-noticias/app_noticias/tests.py", line 17,
    in test_deve_encontrar_noticia_com_id_1
    self.assertEqual(noticia.id, 2)
AssertionError: 1 != 2
-----
```

```
Ran 3 tests in 0.005s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Interpretando a saída temos:

- o teste `test_deve_encontrar_noticia_com_id_1` falhou porque esperava que 1 fosse igual a 2 (o que não é verdade)
- foram executados 3 testes, em 0.005 segundos
- a suíte de testes falhou (**FAILED**)
- 1 de 3 testes falharam

Enquanto a suíte `NoticiaModelTest` testa o Model, também é possível escrever teste para testar a View, veja o trecho de código a seguir, que mostra a classe `HomePageViewTest`, também declarada no arquivo `app_noticias/tests.py`:

```
1 from django.test import TestCase
2 from django.urls import reverse
3
4 class HomePageViewTests(TestCase):
5     def setUp(self):
6         Noticia.objects.create(titulo='Noticia X', conteudo='Conteudo')
7
8     def test_home_status_code_deve_ser_200(self):
9         response = self.client.get('/')
10        self.assertEqual(response.status_code, 200)
11
12    def test_deve_encontrar_url_por_nome(self):
13        response = self.client.get(reverse('home'))
14        self.assertEqual(response.status_code, 200)
15
16    def test_view_deve_usar_template_correto(self):
17        response = self.client.get(reverse('home'))
18        self.assertEqual(response.status_code, 200)
19        self.assertTemplateUsed(response, 'app_noticias/home.html')
```

Essa suíte possui três testes:

- `test_home_status_code_deve_ser_200()`: testa positivo para o código de retorno de uma requisição para a view ser 200
- `test_deve_encontrar_url_por_nome()`: testa positivo para o código de retorno de uma requisição para a URL chamada "home" ser 200
- `test_view_deve_usar_template_correto()`: testa positivo para o código de retorno de uma requisição para a URL chamada "home" ser 200 e o template usado na view ser `app_noticias/home.html`

Como esses são testes para a View eles utilizam o método `get()` do objeto `client` para fazer

uma requisição HTTP GET para uma URL. No `test_home_status_code_deve_ser_200` é feita uma requisição GET para a URL `/`, ou seja a raiz do software. No `test_deve_encontrar_url_por_nome` primeiro é obtida a URL a partir do nome, usando a função `reverse()`, do pacote `django.urls`, depois é feita uma requisição para a URL encontrada. Além disso, os testes demonstram o foco no teste do código de status da resposta (HTTP). Quando o código de status é 200 significa que foi possível fazer a requisição de forma correta.

O último teste da suíte usa o método `assertTemplateUsed()` para fazer uma asserção de que o template usado na view seja o esperado.

Ao executar os testes completos, com as duas suítes, o resultado é como apresentado a seguir:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
-----
Ran 6 tests in 0.043s

OK
Destroying test database for alias 'default'...
```

Ou seja:

- foram executados 6 testes, em 0.043 segundos
- todos os testes passaram

Também é possível aumentar o detalhamento da saída dos testes utilizando a opção `-v 2`, por exemplo:

```
$ python manage.py test -v 2
```

A saída seria mais ou menos assim:

```
Creating test database for alias 'default' ('file:memorydb_default?mode=memory&
    cache=shared')...
Operations to perform:
  Synchronize unmigrated apps: messages, staticfiles
  Apply all migrations: admin, app_noticias, auth, contenttypes, sessions
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying app_noticias.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 auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying sessions.0001_initial... OK
System check identified no issues (0 silenced).
test_deve_encontrar_url_por_nome (app_noticias.tests.HomePageViewTests) ... ok
test_home_status_code_deve_ser_200 (app_noticias.tests.HomePageViewTests) ...
    ok
test_view_deve_usar_template_correto (app_noticias.tests.HomePageViewTests) ...
    ok
test_deve_encontrar_noticia_com_id_1 (app_noticias.tests.NoticiaModelTest) ...
    ok
test_deve_encontrar_noticia_x (app_noticias.tests.NoticiaModelTest) ... ok
test_deve_gerar_excecao_para_encontrar_noticia_com_id_2 (app_noticias.tests.
    NoticiaModelTest) ... ok

-----
Ran 6 tests in 0.032s

OK
Destroying test database for alias 'default' ('file:memorydb_default?mode=
memory&cache=shared')...
```

Perceba que a saída mostra mais informações, como a execução das migrations para a criação do banco de dados e o resultado individual de cada teste.

## 2.16 Conclusão

A estrutura do projeto começa a aumentar bastante à medida que vão sendo adicionadas funcionalidades no projeto e no aplicativo Django, como ilustra a Figura 2.9.

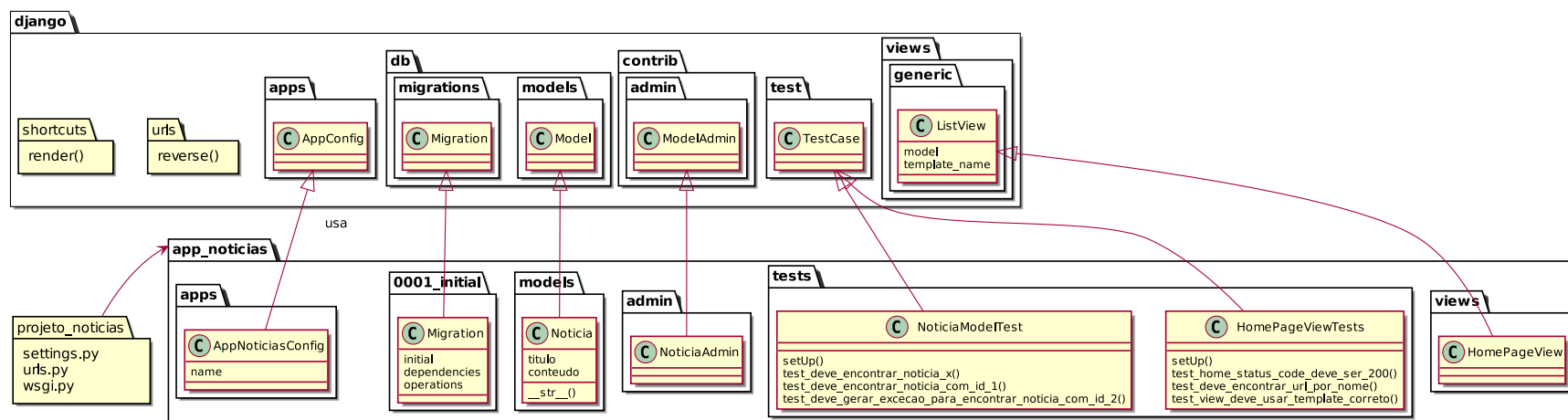


Figura 2.9: Tela da lista de notícias apresentando registros

A Figura 2.9 apresenta uma visão do projeto e suas dependências com classes do Django, bem como a sua complexidade, que, novamente, tende a aumentar conforme vão sendo adicionados recursos e funcionalidades. O importante é identificar que essa é a estrutura padrão de um aplicativo Django, que os capítulos seguintes continuarão apresentando de forma mais detalhada, apresentando novos recursos e conceitos. Além disso a figura procura apresentar uma visão mais estruturada dos elementos do projeto (pacotes e classes) ao invés de somente arquivos.

A Figura 2.10 apresenta uma alteração no *workflow* incluindo a criação de um aplicativo Django e a etapa de testes.

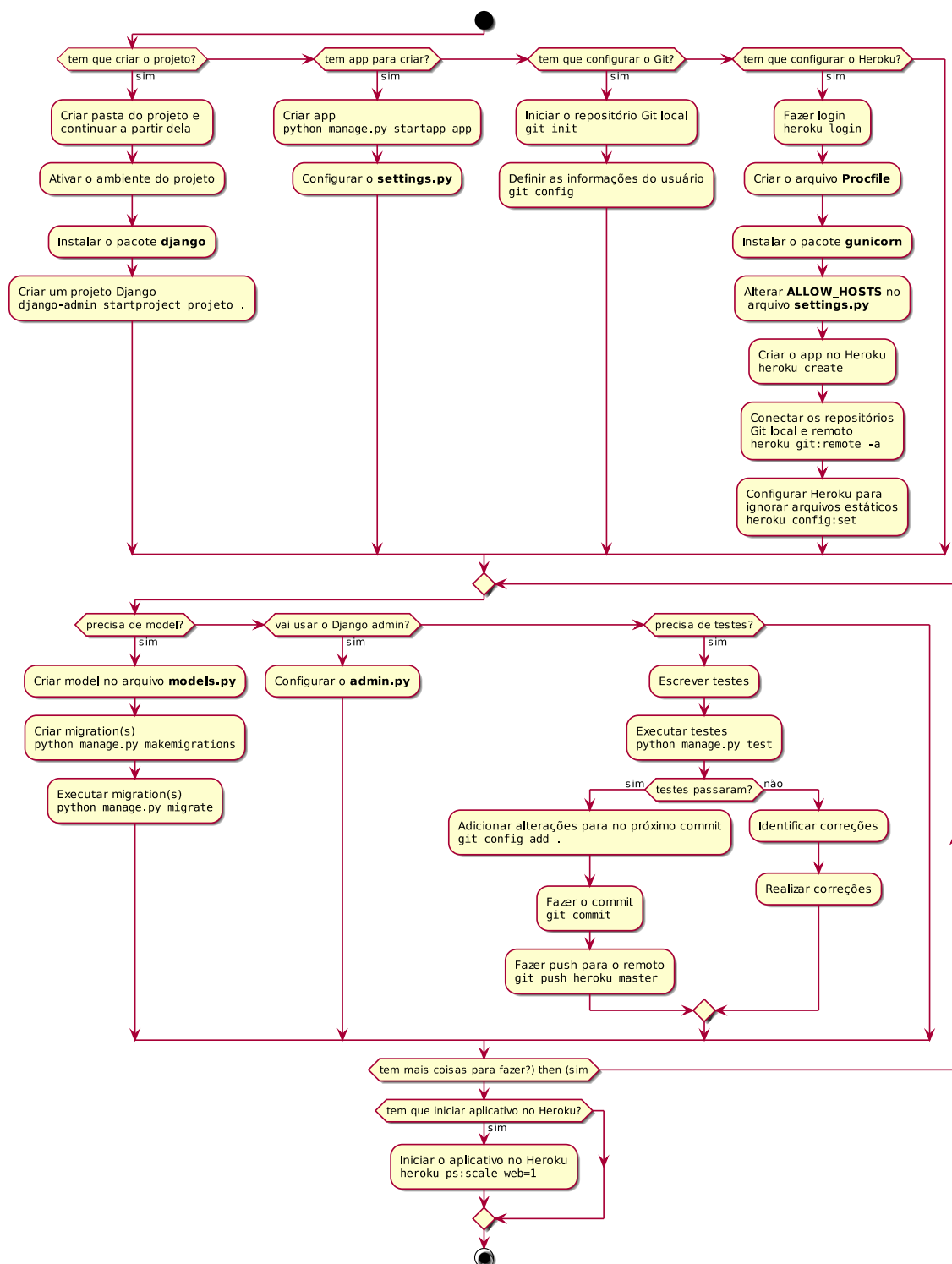


Figura 2.10: Workflow para o desenvolvimento com Testes



## Capítulo 3

# Django ORM

Este capítulo trata de como realizar operações no banco de dados por meio do ORM, como consultas e manipulações dos dados.

### 3.1 Shell do Django

Aprender a utilizar o ORM do Django é uma tarefa primordial. Pode ser interessante, antes de testes e da programação (desenvolvimento do software, em si) desenvolver a habilidade de lidar com a API do ORM de forma mais direta. Para isso o Django fornece um shell, que pode ser obtido executando o comando:

```
python manage.py shell
```

O resultado é um shell interativo (semelhante ao prompt interativo do Python) já carregado com o Django. As seções a seguir podem ser executadas no shell.

### 3.2 Criando objetos

Para criar um objeto (um registro no banco de dados) há duas formas. A primeira delas é criar uma instância do model e depois chamar o método `save()`, como no exemplo a seguir:

```
from app_noticia.models import Noticia
n = Noticia(
    titulo='Alta do dólar',
    conteudo='Dólar sobe para maior preço em dez anos'
)
n.save()
```

A outra forma é chamar o método `create()` de `objects`:

```
from app_noticia.models import Noticia
n = Noticia.objects.create(
    titulo='Alta do dólar',
    conteudo='Dólar sobe para maior preço em dez anos'
)
```

Cada model possui o atributo `objects`, que é chamado de **Manager**, e é muito importante porque dá acesso a operações de consulta no banco de dados.

Em ambos os casos, tanto para `save()` quanto para `create()`, perceba que os parâmetros são os campos do model – cuja ordem não é relevante.

Assim que a instância é criada (o registro é salvo no banco de dados) o Django cria automaticamente o atributo `id`, um número inteiro incrementado automaticamente (verifique).

A criação de instâncias funciona de modo semelhante à criação de registros em tabelas com a instrução `INSERT INTO` da linguagem SQL.

### 3.3 Atualizando objetos

Uma vez que você estiver de posse de uma instância, seus atributos podem ser alterados e, para atualizar o banco de dados, deve ser utilizado o método `save()`. Por exemplo:

```
n.conteudo = 'Dólar sobe para maior patamar nos últimos dez anos'
n.save()
```

O método `save()` precisa ser chamado explicitamente para que o banco de dados seja atualizado, funcionando como a instrução `UPDATE`, em linguagem SQL.

### 3.4 Recuperando objetos

Para recuperar objetos do banco de dados você utiliza um **Manager** para criar **QuerySet**, que representa uma coleção de objetos do banco de dados. O **Manager** é acessado a partir do atributo de classe `objects`. Exemplo: `Noticia.objects`.

As seções a seguir demonstram as operações mais comuns e consideram o model `Noticia`.

#### 3.4.1 Recuperar todos os objetos

A maneira mais simples de recuperar todos os objetos de um model é usar o método `all()` do **Manager**:

```
todas = Noticia.objects.all()
```

O método `all()` é semelhante à instrução `SELECT * FROM tabela` em linguagem SQL.

Também é possível limitar a quantidade de elementos do `QuerySet` utilizando a sintaxe de **slicing** de array em python:

```
primeiras_5 = Noticia.objects.all()[:5]
ultimas_5   = Noticia.objects.all()[5:]
entre_2_e_5 = Noticia.objects.all()[2:5]
```

A sintaxe geral de **slicing** é `OFFSET:LIMIT`, onde:

- **OFFSET** é a quantidade de registros a deslocar (padrão é zero)
- **LIMIT** é a quantidade de registros a considerar (padrão é a quantidade de registros no `QuerySet`)

portanto (suponha que haja 5 instâncias):

- `:5` significa `OFFSET=0, LIMIT=5` (primeiras 5 instâncias)
- `5:` significa `OFFSET=5, LIMIT=5` (desloca 5 instâncias, pega 5 instâncias)
- `2:5` significa `OFFSET=2, LIMIT=5` (desloca 2 instâncias, pega 5 instâncias)

### 3.4.2 Recuperar objetos específicos usando filtros

A maneira mais comum de refinar um `QuerySet` é adicionar filtros. Isso pode ser feito usando os métodos `filter()` e `exclude()`. Por exemplo: para retornar todas as notícias cujo título seja “Alta do dólar” podemos usar:

```
noticias_alta_do_dolar = Noticia.objects.filter(titulo='Alta do dólar')
```

Essa consulta seria semelhante à seguinte em SQL:

```
SELECT * FROM Noticia WHERE titulo='Alta do dólar'
```

Para retornar todas as notícias publicadas no ano de 2018 (considerando um campo `data_de_publicacao` do tipo `DateField`):

```
noticias_alta_do_dolar = Noticia.objects.filter(data_de_publicacao__year=2018)
```

Em SQL, seria semelhante ao seguinte:

```
SELECT * FROM Noticia WHERE data_de_publicacao <= '2018-01-01'
```

Os parâmetros de `filter()` e `exclude()` seguem a sintaxe chamada de **Field lookups**.

Também é possível fazer um encadeamento de filtros, como no exemplo:

```
noticias_ao_dolar_2018 = Noticia.objects.exclude(
    titulo__contains='dólar'
).filter(
    data_de_publicacao__year=2018
)
```

A consulta obtém todas as notícias que não cotêm “dólar” no título e cujo ano da data de publicação seja 2018.

### 3.4.3 QuerySets são lazy

O Django não executa uma consulta no banco de dados quando um QuerySet é criado, apenas quando ele é **avaliado**. Veja o exemplo:

```
noticias_2018 = Noticias.objects.filter(data_de_publicacao__year=2018)
noticias_2018 = noticias_2018.exclude(titulo__contains='dólar')
print(noticias_2018)
```

Apenas na chamada da função `print()` é que a consulta é realizada no banco de dados.

## 3.5 Recuperar um objeto único

Como você viu, o método `filter()` é utilizado para retornar um `QuerySet`. Se você deseja retornar um único objeto pode utilizar o método `get()`, que também aceita a mesma sintaxe de **Field lookups**. Por exemplo:

```
noticia_1 = Noticias.objects.get(id=1)
```

A consulta busca a instância cujo campo `id` seja igual a 1 e seria semelhante à consulta SQL:

```
SELECT * FROM Notica WHERE id = 1
```

Quando a consulta não encontra registro o Django dispara uma exceção criada automaticamente para cada model. Por exemplo, se não houver uma instância com `id=1` então será disparada a exceção `Noticia.DoesNotExist`. Por isso, é importante usar o recurso de tratamento de exceção:

```
1 try:
2     noticia_1 = Noticias.objects.get(id=1)
3     print(noticia_1)
4 except Noticias.DoesNotExist as erro:
5     print('Erro ao tentar encontrar notícia.', erro)
```

## 3.6 Field lookup

**Field lookup** (que poderíamos traduzir como “expressão de busca”) permitem adicionar critérios de busca às consultas e podem ser aplicados aos métodos `get()`, `filter()` e `exclude()`.

A sintaxe é `campo__pesquisa=valor` onde:

- `campo` é o nome do campo no model
- `pesquisa` é o tipo da pesquisa ou expressão de pesquisa
- `valor` é o valor da expressão de pesquisa

As pesquisas mais comuns são apresentadas na tabela a seguir.

Pesquisa	Descrição
<code>exact</code>	Uma comparação exata entre o campo e um valor
<code>iexact</code>	Uma comparação exata, desconsiderando maiúsculas e minúsculas
<code>contains</code>	Verifica se um campo contém um valor
<code>startswith</code>	Verifica se um campo começa com um valor
<code>endswith</code>	Verifica se um campo termina com um valor
<code>year</code>	Verifica se um campo do tipo <code>DateTimeField</code> tem ano igual ao valor
<code>month</code>	Verifica se um campo do tipo <code>DateTimeField</code> tem mês igual ao valor
<code>lt</code>	Verifica se o campo é menor que o valor
<code>lte</code>	Verifica se o campo é menor ou igual ao valor
<code>gt</code>	Verifica se o campo é maior que o valor
<code>gte</code>	Verifica se o campo é maior ou igual ao valor

## 3.7 Excluindo objetos

Uma vez que você estiver de posse de uma instância basta utilizar o método `delete()` para excluí-la. Por exemplo:

```
n = Noticia.objects.get(id=1)
n.delete()
```

Se precisar, também pode chamar o método `delete()` de um `QuerySet` para excluir várias instâncias ao mesmo tempo. Por exemplo:

```
noticias_2018 = Noticia.objects.filter(data_de_publicacao__year=2018)
noticias_2018.delete()
```

O método `delete()` opera de forma semelhante à instrução `DELETE FROM` em linguagem SQL.

## Capítulo 4

# Melhorando o modelo de dados do Aplicativo Notícias

Este capítulo mostra como melhorar e evoluir o modelo de dados do aplicativo notícias utilizando recursos do model do Django.

Até agora o modelo de dados do `app_noticias` contém apenas o model `Noticia` (e seus atributos `titulo` e `conteudo`). Vamos incrementar esse modelo considerando a seguinte situação:

- notícia está associada a tags: tags são como marcadores atribuídos às notícias pelos autores
- notícia tem um autor: um autor é quem escreve a notícia
- pessoa tem um usuário: o usuário permite que a pessoa acesse o Django Admin para cadastrar notícias

Para alcançar estes objetivos será necessário utilizar o recurso de relacionamentos, como apresentam as seções a seguir.

### 4.1 Relacionamentos

Da mesma forma que bancos de dados relacionais o ORM do Django permite o recurso de relacionamentos entre models e isso pode ser feito de três formas: **muitos-para-um**, **muitos-para-muitos** e **um-para-um**.

#### 4.1.1 Relacionamento um-para-um

O relacionamento **um-para-um** é definido utilizando um campo do tipo `django.db.models.OneToOneField`. Na situação apresentada anteriormente, um autor está associado (relacionado) a um usuário. Para fazer isso, primeiro, deve-se considerar que não é necessário criar um model para usuário, mas utilizar `User`, fornecido por `django.contrib.auth.models`. Assim, podemos definir o model `Pessoa` como demonstra o trecho de código a seguir:

```

1 from django.contrib.auth.models import User
2 from django.db import models
3
4 class Pessoa(models.Model):
5     usuario = models.OneToOneField(User, on_delete=models.CASCADE,
6         verbose_name='Usuário')
7     nome = models.CharField('Nome', max_length=128)
8     data_de_nascimento = models.DateField(
9         'Data de nascimento', blank=True, null=True)
10    telefone_celular = models.CharField('Telefone celular', max_length=15,
11        help_text='Número do telefone
12            celular no formato (99)
13            99999-9999',
14        null=True, blank=True,
15    )
16    telefone_fixo = models.CharField('Telefone fixo', max_length=14,
17        help_text='Número do telefone fixo no
18            formato (99) 9999-9999',
19        null=True, blank=True,
20    )
21    email = models.EmailField('E-mail', null=True, blank=True)
22
23    def __str__(self):
24        return self.nome

```

Perceba que o campo `usuario` é do tipo `OneToOneField` e os parâmetros do construtor são, nesta ordem:

- o tipo `User`, para representar o relacionamento **um-para-um**, em si
- `on_delete=models.CASCADE` para indicar que quando a instância de `User` for excluída a instância de `Pessoa` relacionada também deve ser excluída (exclusão em cascata)
- `verbose_name='Usuário'` para definir o nome literal do campo

#### 4.1.2 Relacionamentos muitos-para-um

Para definir um relacionamento **muitos-para-um** use a classe `django.db.models.ForeignKey` ao definir seu model. Para exemplificar, considere que muitas `Noticia` possam ser escritas por um autor (ou um autor possa escrever muitas notícias) então teríamos um relacionamento **muitos-para-um** entre o model `Pessoa` e `Noticia` (o código a seguir tem apenas os trechos mais importantes para o contexto):

```

1 from django.db import models
2
3 class Pessoa(models.Model):
4     # ...
5     pass

```

```

6
7
8 class Noticia(models.Model):
9     autor = models.ForeignKey(Pessoa, on_delete=models.CASCADE)
10    # ...

```

Dessa forma o model `Noticia` tem um campo `autor` que **referencia** o model `Pessoa`. Os parâmetros do construtor de `ForeignKey` são:

- `Pessoa`, para representar o relacionamento, em si
- `on_delete=models.CASCADE` para definir exclusão em cascata

Outro valor para o parâmetro `on_delete` é `models.SET_NULL`, para atribuir o valor `null` ao campo `autor` quando a instância relacionada for excluída. Nesse caso, o campo precisa aceitar `null` – com os atributos `null=True` e `blank=True`.

### 4.1.3 Relacionamentos muitos-para-muitos

Para definir um relacionamento **muitos-para-muitos** use a classe `django.db.models.ManyToManyField` ao definir seu model. Para exemplificar, considere que `Noticia` tem muitas `Tag` e `Tag` tem muitas `Noticia`:

```

1 from django.db import models
2
3 class Tag(models.Model):
4     nome = models.CharField(max_length=64)
5     slug = models.SlugField(max_length=64)
6
7     def __str__(self):
8         return self.nome
9
10
11 class Noticia(models.Model):
12     # ...
13     tags = models.ManyToManyField(Tag)
14     # ...

```

O parâmetro para o construtor de `ManyToManyField` é `Tag`, para representar o relacionamento, em si.

Assim, temos os seguintes relacionamentos:

- `Pessoa` **um-para-um** `User`
  - `Noticia` **muitos-para-um** `Pessoa`
  - `Noticia` **muitos-para-muitos** `Tag`
-



## Migrations

Como os models foram modificados não se esqueça de executar os comandos `makemigrations` e `migrate`.

### migrate zero

Há casos em que você já tem um banco de dados e já executou migrations anteriores, mas não está conseguindo progredir por causa de incompatibilidades que geram erros ou situações que você não consegue resolver no momento. Se você não precisar se importar com perda de dados ou se tiver feito um **backup dos seus dados** pode executar `migrate app_noticias zero` (substituindo `app_noticias` pelo nome do seu app) para desfazer todas as migrations, a depois apagar todos os arquivos de migrations da pasta `migrations` e então executar `makemigrations` novamente. Isso é uma forma de *começar do zero*, por assim dizer (mas use com sabedoria!).

### Voltando no tempo

Há casos em que você precisa voltar uma migration. Quando precisar fazer isso não apague o arquivo da migration diretamente. O Django controla as migrations executadas no banco de dados, então você precisa fazer isso da forma certa. Se não percebeu ainda, o Django cria as migrations como uma linha do tempo, uma migration dependendo de outra anterior. Depois de criar uma migration, volte para uma migration anterior com o comando `migrate app_name migration_name` (substituindo `app_name` pelo nome do aplicativo e `migration_name` pelo nome da migration – o nome do arquivo). Depois disso, pode excluir o arquivo da migration desejada.

### migrate app\_name e makemigrations app\_name

Outra boa prática é, à medida em que o projeto começa a ficar maior, com vários aplicativos, começar a adicionar o nome do aplicativo como parâmetro dos comandos `makemigrations` e `migrate` para diferenciar de outros aplicativos.

## 4.2 Configurando o Django Admin

Para permitir o gerenciamento dos dados configure o módulo `app_noticias.admin` para conter o seguinte:

```

1  @admin.register(Pessoa)
2  class PessoaAdmin(admin.ModelAdmin):
3      pass
4
5
6  @admin.register(Tag)
7  class TagAdmin(admin.ModelAdmin):
8      prepopulated_fields = {'slug': ('nome',)}
9
10
```

```

11 @admin.register(Noticia)
12 class NoticiaAdmin(admin.ModelAdmin):
13     pass

```

Como já foi feito, o padrão é criar uma classe para administrar cada model, herdando de `django.contrib.admin.ModelAdmin` e registrá-la usando `django.contrib.admin.register()`:

- classe que herda de `django.contrib.admin.ModelAdmin`
- usar a função de anotação `register()` passando como argumento o nome do model que será gerenciado pela classe

A classe `TagAdmin` tem uma configuração adicional: `prepopulated_fields` é utilizado para indicar ao Django Admin que um campo deve ser preenchido a partir de outro. Nesse caso, o campo `slug` deve ser preenchido a partir do campo `nome`, mas isso é feito de uma forma diferenciada, por causa do campo do tipo `SlugField`: caracteres especiais e espaços são removidos ou substituídos.

Há mais configurações do Django Admin, mas por enquanto isso já é suficiente para permitir as funcionalidades que são objetivo deste capítulo. Começando pelo cadastro de notícia (como ilustra a Figura 4.1) agora há um formulário mais completo, com uma interface que permite entrada de dados nos campos e seleção dos registros relacionados.

Administração do Django BEM-VINDO(A) [AUTOR](#) [VER O SITE](#) [ALTERAR SENHA](#) [ENCERRAR SESSÃO](#)

Início > App\_Noticias > Notícias > Adicionar Notícia

### Adicionar Notícia

**Título:**

**Conteúdo:**

pais. Em conjunto, somam 15% do total de pessoas aptas a votar nas eleições de 2018 – um número que só é inferior ao do estado de São Paulo, que responde por 22%.

A campanha eleitoral começou em 16 de agosto. Desde então, e até quarta-feira (5), 11 dos 13 presidenciais tiveram agendas de campanha. Eles realizaram um total de 232 visitas a alguma cidade do país, e visitaram 20 estados e o DF.

O principal destino é São Paulo. O estado – que, além de principal colégio eleitoral do país, concentra entidades e veículos de comunicação com alcance nacional e é berço político de dois dos candidatos a Presidência – recebeu os 11 candidatos e teve o maior número de visitas na agenda de todos eles.

**Data de publicação:**

Data:  Hoje

Hora:  Agora

**Autor:**

**Tags:**

Mantenha pressionado o "Control", ou "Command" no Mac, para selecionar mais de uma opção.

Figura 4.1: Adicionar notícia com autor e tags

A Figura 4.1 mostra que botões (ícones) ao lado do campo “Autor” permitem acessar as funcionalidades, nesta ordem:

- editar pessoa selecionada (se houver uma pessoa selecionada)
- adicionar/cadastrar pessoa
- excluir pessoa selecionada (se houver uma pessoa selecionada)

Ao clicar no botão “adicionar”, por exemplo, aparece uma “popup”, uma janela que permite cadastrar a pessoa, como ilustra a Figura 4.2.

Adicionar pessoa

Usuário: ----- ▾ ✎ +

Nome:

Data de nascimento:  Hoje

Telefone celular:   
Número do telefone celular no formato (99) 99999-9999

Telefone fixo:   
Número do telefone fixo no formato (99) 9999-9999

E-mail:

**SALVAR**

Figura 4.2: Adicionar notícia - popup de adicionar pessoa

A Figura 4.2 mostra que é possível cadastrar a pessoa. Interessante observar que ao salvar os dados o Django Admin fará com que a pessoa seja selecionada como autor na tela de cadastro da notícia (o mesmo vale para a tela de edição).

De forma semelhante, o mesmo acontece com o campo “Tags”, embora haja apenas o botão (ícone) para adicionar, permitindo cadastrar uma tag, como ilustra a Figura 4.3.

Adicionar tag

Nome:

Slug:

**SALVAR**

Figura 4.3: Adicionar notícia - popup de adicionar tag

A Figura 4.3 mostra o formulário de cadastro da tag. O comportamento ao salvar é o mesmo observado anteriormente para o cadastro de pessoa: a tag passa a ser parte das tags selecionadas da notícia que está sendo cadastrada. Importante observar que a interface usa um componente de formulário que permite a seleção de múltiplos elementos (há uma descrição na tela sobre como usar a interface para selecionar mais de uma tag ou tirar uma tag da seleção).

Por fim, a Figura 4.4 ilustra o formulário preenchido.

Figura 4.4: Adicionar notícia com autor e tags completo

## 4.3 Conclusão

Este capítulo apresentou os conceitos de relacionamentos usando os tipos de campos `OneToOneField`, `ForeignKey` e `ManyToManyField`. Além disso, mostrou como o Django Admin adapta a interface de formulários para permitir a seleção de registros relacionados e manter os relacionamentos.

## Capítulo 5

# Acesso público: views e templates

Agora que o `app_noticias` tem uma interface administrativa que permite um gerenciamento dos dados mais completo podemos incrementar o acesso público, permitindo funcionalidades para leitura das notícias.

Você já sabe que o Django é um framework **MVC** (Model-View-Controller) onde **model** representa o modelo de dados, **view** representa interface gráfica e **controller** contém a lógica de negócio. Entretanto, os componentes do Django são traduzidos para **MTV** (**Model-Template-View**) ou seja, o **template** representa a interface gráfica, enquanto a **view** contém a lógica. Essa distinção é importante. Além disso, é importante entender que o Django utiliza URLs para indicar como views devem ser acessadas. Esse capítulo apresenta esses conceitos.

Views podem ser descritas de duas formas: usando funções ou usando classes. Independentemente da forma de descrever as views é necessário utilizar URLs para acessá-las. As seções a seguir detalham as formas de descrever as views.

### 5.1 Views baseadas em funções

As views descritas por funções são as mais simples de escrever. Para exemplificar, considere uma view que apresenta um sumário (resumo) com a quantidade total de notícias:

```
1 from django.http import HttpResponse
2 from .models import Noticia
3
4 def noticias_resumo(request):
5     total = Noticia.objects.count()
6     html = """
7     <html>
8     <body>
9     <h1>Resumo</h1>
10    <p>A quantidade total de notícias é {}.</p>
11    </body>
```

```
12     </html>
13     """.format(total)
14     return HttpResponse(html)
```

A função `noticias_resumo()` recebe o parâmetro `request`, obrigatório como primeiro parâmetro de toda view definida por função.

A variável `total` recebe o resultado de `Noticia.objects.count()` (o total de instâncias do model `Noticia`).

A variável `html` é uma string que contém elementos (tags) do HTML, mas foi definida manualmente e diretamente no código python. Embora isso não esteja totalmente errado, você já deve perceber que não é uma boa prática fazer isso com uma página longa. Voltaremos nisso daqui a pouco.

O retorno da função `noticias_resumo()` é uma instância de `HttpResponse()` que recebe como argumento a variável `html`.

## 5.2 Configurações de URLs (acessando views)

Para acessar a view é necessário configurar as URLs do aplicativo, ou seja, o módulo `app_noticias.urls`:

```
1 from django.urls import path
2 from app_noticias.views import noticias_resumo, HomePageView
3
4 urlpatterns = [
5     path('', HomePageView.as_view(), name='home'),
6     path('noticias/resumo/', noticias_resumo, name='resumo'),
7 ]
```

A variável `urlpatterns` é uma lista de instâncias de `django.urls.path()` ou `django.urls.re_path()`. Durante o processamento da requisição o Django começa a procurar nas URLs do aplicativo aquela que mais combina com a URL da requisição e, quando a encontra, identifica qual view deve ser importada e chamada. No momento de chamar a view, o Django passa os seguintes argumentos:

- uma instância de `HttpRequest`
- parâmetros de rota e valores-padrão (se definidos) – isso será visto mais adiante

O construtor de `path()` recebe os parâmetros:

- **caminho** (ou **rota**): uma string que representa o caminho da URL
- **view**
- outros parâmetros nomeados, como `name`, que indica o nome da URL

**Importante:** como estamos definindo as URLs em um aplicativo é fundamental que o projeto as importe corretamente. Para isso, verifique o módulo `projeto_noticias.urls`:

---

```
1 from django.contrib import admin
2 from django.urls import path, include
3
4 urlpatterns = [
5     path('admin/', admin.site.urls),
6     path('', include('app_noticias.urls'))
7 ]
```

O segundo elemento do array `urlpatterns` é uma chamada para `path()` com os parâmetros:

- `''`: indicando a **raiz do site**
- uma chamada à função `include()` passando como argumento `'app_noticias.urls'`, a string que indica as URLs do `app_noticias`

Assim, se a requisição for para a URL `/` o Django fará a busca pelas URLs na sequência:

- em `urlpatterns` de `projeto_noticias.urls` (não encontra, continua procurando)
- em `urlpatterns` de `app_noticias.urls` (encontra a URL `home` para a view `HomePageView`)

Se a requisição for para a URL `/noticias/resumo/` a busca será feita na sequência:

- em `urlpatterns` de `projeto_noticias.urls` (não encontra, continua procurando)
- em `urlpatterns` de `app_noticias.urls` (encontra a URL `resumo` para a view `noticias_resumo`)

Se a requisição for para a URL `/noticias/teste/` a busca será feita na sequência:

- em `urlpatterns` de `projeto_noticias.urls` (não encontra, continua procurando)
- em `urlpatterns` de `app_noticias.urls` (não encontra, continua procurando)
- não tem mais onde procurar, retorna erro 404 (página não encontrada)

## 5.3 Templates

A view `noticias_resumo()`, embora funcione, não é muito eficiente do ponto-de-vista de código e boas práticas Django porque ela define código HTML dentro de strings no código Python. Uma maneira de separar isso, mantendo código separado, é utilizar templates. Para exemplificar, veja a view `noticias_resumo_template()`:

```
1 from django.shortcuts import render
2 def noticias_resumo_template(request):
3     total = Noticia.objects.count()
4     return render(request, 'app_noticias/resumo.html', {'total': total})
```

A view `noticias_resumo_template()` é semelhante à `noticias_resumo()`, mas retorna uma instância de `render()`, um atalho para `HttpResponse` que aceita os parâmetros:

- uma instância de `HttpRequest`
- o caminho para o template
- um dicionário de valores a serem adicionados ao *contexto do template*

O *contexto do template* contém alguns objetos ou valores disponibilizados automaticamente pelo Django no template. No caso da função `render()` o dicionário de valores **permite passar valores da view para o template**.

O código-fonte do template para apresentar o resumo das notícias (`app_noticias/templates/app_noticias/resumo.html`):

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3 <head>
4     <meta charset="UTF-8">
5     <title>Resumo das notícias</title>
6 </head>
7 <body>
8 <h1>Resumo das notícias</h1>
9 <p>Quantidade de notícias: {{ total }}.</p>
10 </body>
11 </html>
```

Perceba que a view `noticias_resumo_template()` definiu um dicionário com a chave `total` para ser fornecido ao template. Por isso o template pode usar a variável de template `total`.

## 5.4 Views e URLs com parâmetros

Suponha que a página inicial (*home*) do `app_noticias` apresente uma lista com títulos das notícias recentes e cada item da lista seja um link para uma página que permita ler o conteúdo da notícia (bem como ver outras informações, como autor, data e tags). Uma forma de permitir ler uma notícia específica, digamos por meio do seu identificador, é utilizar uma URL com parâmetro. Primeiro, a view `noticia_detalhes()`:

```
1 from django.http import Http404
2 from django.shortcuts import render
3 from .models import Noticia
4
5 def noticia_detalhes(request, noticia_id):
6     try:
7         noticia = Noticia.objects.get(pk=noticia_id)
8     except Noticia.DoesNotExist:
9         raise Http404('Notícia não encontrada')
10    return render(request, 'app_noticias/detalhes.html', {'noticia':
        noticia})
```

Como você já sabe, a view `noticia_detalhes()` deve ter o primeiro parâmetro `request`. O segundo parâmetro `noticia_id` é novidade (voltaremos nele daqui a pouco). Por enquanto, assumo que `noticia_id` contém o identificador da notícia que deve ser apresentada. Assim, a primeira tarefa



é encontrar a instância de `Noticia`. Para isso o código utiliza `try...except`: tenta encontrar a notícia utilizando `Noticia.objects.get(pk=noticia_id)`; se não encontrar (`except` do tipo `Noticia.DoesNotExist`) dispara uma exceção criada por `Http404` (uma página de erro indicando que a notícia indicada por `noticia_id` não foi encontrada); se encontrar, retorna um `HttpResponse` utilizando como template `app_noticias/detalhes.html`.

O código-fonte do template:

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3 <head>
4     <meta charset="UTF-8">
5     <title>{{ noticia.titulo }}</title>
6 </head>
7 <body>
8 <h1>{{ noticia.titulo }}</h1>
9 <p>Por <strong>{{ noticia.autor }}</strong> em {{ noticia.
    data_de_publicacao }}</p>
10 <p>Tags:
11     {% for tag in noticia.tags.all %}
12         <a href="{% url 'noticias_da_tag' tag.slug %}">{{ tag }}</a>{% if
            not forloop.last %}, {% endif %}
13     {% endfor %}
14 </p>
15 <div>
16     {{ noticia.conteudo | linebreaks }}
17 </div>
18
19 </body>
20 </html>
```

O template apresenta as informações da notícia, como título, autor, data de publicação, tags e conteúdo. Atenção especial para a apresentação das tags da notícia. O model `Noticia` define um relacionamento **muitos-para-muitos** com `Tag` por meio do campo `tags`. Para obter a lista das tags da notícia é necessário utilizar `noticia.tags.all` na tag `for`.

Agora, a configuração da URL:

```
1 from django.urls import path
2 from app_noticias.views import noticia_detalhes
3
4 urlpatterns = [
5     # ...
6     path('noticias/<int:noticia_id>/', noticia_detalhes, name='detalhes'),
7     # ...
8 ]
```

A URL `detalhes` contém o caminho: `noticias/<int:noticia_id>/`. A parte entre `<>` representa a

definição do **parâmetro de rota** chamado `noticia_id` e a parte `int`: é um conversor de caminho, o que significa que o parâmetro de rota será tratado como um número inteiro positivo. Outros conversores de caminho são: `str`, `slug`, `uuid` e `path`. O nome do parâmetro de rota `noticia_id` ser o mesmo do parâmetro `noticia_id` da view `noticia_detalhes()` não é por acaso.

Suponha que o Django trate uma requisição para a URL `/noticias/1/`. O processamento da URL ocorre nesta sequência:

- procura em `urlpatterns` de `projeto_noticias.urls` (não encontra, continua procurando)
- procura em `urlpatterns` de `app_noticias.urls` (encontra a URL `detalhes` para a view `noticia_detalhes` e extrai o parâmetro de rota `noticias_id` com valor 1 (um número inteiro positivo))
- executa a view `noticia_detalhes`

Está curioso para saber o que acontece se a URL for `/noticias/a/`? Esta seria a sequência:

- procura em `urlpatterns` de `projeto_noticias.urls` (não encontra, continua procurando)
- procura em `urlpatterns` de `app_noticias.urls` (não encontra, continua procurando)
- não tem mais onde procurar, retorna erro 404 (página não encontrada)

Por que ocorreu o erro 404? Porque mesmo com uma URL com um padrão de caminho semelhante o parâmetro `noticia_id` deve ser tratado como um número inteiro positivo. Não é o caso aqui.

O que acontece se a URL for `/noticias/2/` (supondo que não exista uma notícia com identificador igual a 2)? Esta seria a sequência:

- procura em `urlpatterns` de `projeto_noticias.urls` (não encontra, continua procurando)
- procura em `urlpatterns` de `app_noticias.urls` (encontra a URL `detalhes` para a view `noticia_detalhes` e extrai o parâmetro de rota `noticias_id` com valor 2 (um número inteiro positivo))
- executa o código da view `noticias_detalhes` e, como não encontra a notícia com `pk` igual a 2, gera uma página de erro 404, indicando que a notícia em questão não foi encontrada

Há uma diferença sutil, mas muito importante na forma como o Django executa o processamento dessas requisições.

## 5.5 Gerando URLs nos templates

A URL `detalhes` tem o parâmetro de rota `noticia_id` (um inteiro positivo). Gerar um elemento `a` (link) para a página de detalhe de uma notícia específica deve ser bastante simples, certo? Sim e não. Sim porque é, a princípio, uma tarefa simples, mas não porque a maior complicação é não saber onde o software estará disponível. Aqui entra em ação o nome da URL. Veja um trecho do template da URL `home`:

```
1 <h2>Notícias recentes</h2>
2 {% for noticia in object_list %}
3 <div>
4     <div><a href="{% url 'detalhes' noticia.pk %}">{{ noticia.titulo }}</a>
```

```
        ></div>
5  </div>
6  {% endfor %}
```

O atributo `href` do `a` que representa o link para a página de detalhes da notícia é gerado pela tag `url` do Django, que recebe dois parâmetros:

- o nome da URL (`'detalhes'`)
- o identificador da notícia (`noticia.pk`)

Isso faz com que seja gerada uma saída para o browser semelhante à seguinte:

```
1  <div>
2  <a href="/noticias/1/">...</a>
3  </div>
```

Perceba que o atributo `href` tem um valor gerado especificamente para situação da execução do software no momento (ele está hospedado na raiz `/`, por exemplo).

---

### Utilização de conversores de caminho

Embora os exemplos deste capítulo tenham demonstrado a utilização de **conversor de caminho** isso não é sempre necessário. Por exemplo, a rota da URL `detalhes` poderia ser `noticias/<noticia_id>/`, sem utilizar um conversor de caminho. Isso modificaria o processamento da URL `/noticias/a/`.

---

## 5.6 Views baseadas em classes

As views baseadas em classes são um mecanismo mais complexo do Django para descrever views e grande parte dessa complexidade se deve ao fato da hierarquia de tipos de views, que começa com a classe `View`.

### 5.6.1 View

Implementar uma view baseada em classe significa criar uma classe que usa uma das classes da hierarquia que começa com `View` e utilizar os métodos específicos para cada situação. Cada classe especializa uma maneira de construir a view. A classe `View` fornece a implementação mais básica. Para alcançar um resultado semelhante ao obtido anteriormente com views baseadas em funções poderíamos implementar a classe `NoticiasResumoView` da seguinte forma:

```
1  from django.views import View
2  from django.shortcuts import render
```

```
3 from .models import Noticia
4
5 class NoticiasResumoView(View):
6     template_name = 'app_noticias/resumo.html'
7
8     def get(self, request, *args, **kwargs):
9         total = Noticia.objects.count()
10        return render(request, self.template_name, {'total': total})
```

O método `get()` da classe `NoticiasResumoView` atende requisições HTTP com o verbo GET e, da mesma forma, há métodos para atender requisições de outros verbos do HTTP, como POST e HEAD. O método `get()` retorna o valor do método `render()` que, como já vimos, é um objeto `HttpResponse`.

A classe também possui um atributo de classe chamado `template_name`, que contém o nome do template. Se você está se perguntando: não, não tem nada de obrigatório ou especial com esse nome; é apenas o nome do atributo de classe. Perceba a semelhança com a view baseada em função `noticias_resumo_template()`.

A configuração da URL passa por uma pequena mudança:

```
1 from django.urls import path
2 from app_noticias.views import NoticiasResumoView
3
4 urlpatterns = [
5     # ...
6     path('noticias/resumo/', NoticiasResumoView.as_view(), name='resumo'),
7     # ...
8 ]
```

O segundo argumento para a função `path()`, ao invés de ser o nome da função, é o retorno do método de classe `as_view()`. Essa é a forma padrão de configurar a URL para toda classe que herda de `View`.

### 5.6.2 TemplateView

A classe `TemplateView` implementa uma view que *renderiza* um template e pode ter contexto capturado da URL. Essa classe herda de:

- `django.views.generic.base.TemplateResponseMixin` que fornece, principalmente, o atributo de classe `template_name`
- `django.views.generic.base.ContextMixin` que fornece, principalmente, o método `get_context_data()` e o atributo de classe `extra_context`
- `django.views.generic.base.View`

A seguir, o código-fonte da classe `NoticiasResumoView` modificada para herdar de `TemplateView`:

```
1 from django.views.generic import TemplateView
2 from .models import Noticia
3 from django.shortcuts import render
4
5 class NoticiasResumoView(TemplateView):
6     template_name = 'app_noticias/resumo.html'
7
8     def get_context_data(self, **kwargs):
9         context = super().get_context_data(**kwargs)
10        context['total'] = Noticia.objects.count()
11        return context
```

O atributo de classe `template_name` contém o caminho para o template. O método `get_context_data()` é uma sobrescrita do método da superclasse e define o contexto do template com a chave `total` (que contém a quantidade total de notícias).

O *contexto do template* é um dicionário passado para template e serve como uma forma de transportar dados da view para o template. No exemplo de código, o contexto contém a chave `total` e a torna disponível para ser utilizada no template.

As classes `View` e `TemplateView` são consideradas pelo Django como **classes base** por fornecerem as implementações de recursos mais fundamentais de views. Entretanto, o Django também fornece implementações de tarefas mais corriqueiras, como views que acessam model.

O Django categoriza as views da seguinte forma:

- views base
  - `View`
  - `TemplateView`
  - `RedirectView`
- views genéricas de visualização
  - `DetailView`
  - `ListView`
- views genéricas de edição
  - `FormView`
  - `CreateView`
  - `UpdateView`
  - `DeleteView`
- views genéricas de datas
  - `ArchiveIndexView`
  - `YearArchiveView`
  - `MonthArchiveView`
  - `WeekArchiveView`
  - `DayArchiveView`
  - `TodayArchiveView`
  - `DateDetailView`

As seções seguintes apresentam essas views.

### 5.6.3 DetailView

A classe `DetailView` implementa uma view que está vinculada a um model e apresenta detalhes de uma instância. Mais do que isso, a view possui recursos que tornam sua implementação bastante simples. Vamos utilizar a `DetailView` para implementar a view que mostra os detalhes de uma notícia.

```
1 from django.views.generic import DetailView
2 from .models import Noticia
3
4 class NoticiaDetalhesView(DetailView):
5     model = Noticia
6     template_name = 'app_noticias/detalhes.html'
```

A classe `NoticiaDetalhesView` herda de `DetailView` e possui dois atributos de classe:

- `model` indica o model associado à view (`Noticia`)
- `template_name` indica o template utilizado (`app_noticias/detalhes.html`)

Um pequeno detalhe está na configuração da URL:

```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     # ...
6     path('noticias/<int:pk>/', views.NoticiaDetalhesView.as_view(), name='
       detalhes'),
7     # ...
8 ]
```

O parâmetro de rota do caminho da URL passa a se chamar `pk` (porque é o padrão para o nome do parâmetro de rota para a `DetailView`). Esse nome não é por acaso. O ORM do Django considera `pk` como um “atalho” para a chave primária do model que, por padrão, é `id`. Então, quando encontra o parâmetro de rota `pk` o Django fará uma busca pelo campo `id`.

A view trata a URL e busca a instância do model no banco de dados, sem a necessidade de implementar esse processo manualmente, como fizemos anteriormente. Isso não é de surpreender, porque a implementação segue um procedimento padrão para qualquer model. Além de buscar a instância pelo parâmetro de rota `pk` outra forma é encontrar pelo campo `slug` (se o model tiver um campo com esse nome).

Não é necessário alterar o template para que o software funcione: o template está baseado na variável `noticia` que é criada automaticamente pela `DetailView` no contexto do template com base no nome do model. Se precisar alterar isso utilize o atributo de classe `context_object_name`. Além

de estar disponível no template a instância do model também está disponível na view por meio do atributo de classe `object`.

Geralmente não é necessário alterar mais alguma coisa mas, se precisar, você pode sobrescrever um ou mais destes métodos:

- `get_queryset()`
- `get_object()`
- `get_context_data()`
- `get()`

#### 5.6.4 ListView

A classe `ListView` implementa uma view que está vinculada a um model e apresenta uma lista de instâncias de um model. Vamos revisitar a implementação da classe `HomePageView`, que mostra a lista das notícias recentes, e modificá-la para que mostre apenas as cinco notícias mais recentes que estejam publicadas:

```
1 class HomePageView(ListView):
2     model = Noticia
3     context_object_name = 'noticias'
4     template_name = 'app_noticias/home.html'
5
6     def get_queryset(self):
7         return Noticia.objects.exclude(data_de_publicacao=None).order_by('
            -data_de_publicacao')[:5]
```

A classe `HomePageView` herda de `ListView` e possui os atributos de classe:

- `model` indica o model associado à view (`Noticia`)
- `context_object_name` indica o nome da lista de instâncias no contexto do template (`noticias`)
- `template_name` indica o nome do template

Além disso a classe sobrescreve o método `get_queryset()`. Por padrão, a `ListView` retorna todas as instâncias do model. Ao fornecer uma implementação própria o método `get_queryset()` a classe `HomePageView` consegue modificar esse comportamento para retornar apenas as cinco notícias mais recentes (pelo campo `data_de_publicacao`) cujo campo `data_de_publicacao` não seja nulo (igual a `None`) – isso serviria para garantir, por exemplo, que a notícia esteja publicada.

A lista das instâncias (notícias) está disponível na view por meio do atributo de classe `object_list`.

## 5.7 Conclusão

Este capítulo apresenta de forma mais detalhada como o Django trabalha com o padrão MVC ou MTV, ao implementar template e view. Além disso, o capítulo mostrou como criar views baseadas em funções e views baseadas em classes, como ligar views e URLs e alguns detalhes sobre

a hierarquia de tipos da view começa com a classe `View` e tem também `TemplateView`, `DetailView` e `ListView`.



## Capítulo 6

# Formulários e mais sobre templates

Uma das funcionalidades mais importantes de software web é permitir a interação com o usuário por meio da entrada de dados. O HTML fornece o recurso de formulários para entrada de dados e o Django também tem uma maneira específica de tratar esse recurso.

Embora você já tenha visto o Django Admin tratar entrada de dados de uma forma praticamente automática, não é sempre que será útil ou suficiente disponibilizá-lo para seu usuário. Por exemplo, considere que é necessário fornecer um formulário de contato no `app_noticias`. Não é possível utilizar o Django Admin porque ele requer autenticação e isso é incompatível com essa funcionalidade porque ela não deve requerer autenticação: o usuário deve fornecer seu nome, a mensagem de contato e, opcionalmente, seu e-mail.

Este capítulo demonstra como utilizar formulários do Django para implementar essa funcionalidade.

### 6.1 Model MensagemDeContato

O model `MensagemDeContato` possui quatro campos para armazenar: nome, email, mensagem e data do contato:

```
1 class MensagemDeContato(models.Model):
2     class Meta:
3         verbose_name = 'Mensagem de contato'
4         verbose_name_plural = 'Mensagens de contato'
5
6     nome = models.CharField(max_length=128)
7     email = models.EmailField('E-mail', null=True, blank=True)
8     mensagem = models.TextField()
9     data = models.DateTimeField(auto_now_add=True)
10
```

```
11     def __str__(self):
12         return self.nome
```

O campo `email` é de preenchimento opcional. O campo `data`, do tipo `DateTimeField`, tem o parâmetro `auto_now_add=True`, o que faz com que seu valor seja gerado automaticamente no momento em que o registro é criado e não permite que seu valor seja alterado no Django Admin.

## 6.2 Django Admin

O admin do model `MensagemDeContato` deve ser registrado de forma semelhante aos anteriores:

```
1  @admin.register(MensagemDeContato)
2  class MensagemDeContatoAdmin(admin.ModelAdmin):
3      readonly_fields = ('data',)
```

A diferença agora é que o model tem o campo `data`, do tipo `DateTimeField` e que tem o parâmetro `auto_now_add=True`, então o Django Admin não pode editá-lo. Isso torna o campo oculto por padrão (já que o formulário, por padrão, mostra apenas os campos que podem ser editados). Para apresentar o campo em formato apenas para leitura é utilizado o atributo de classe `readonly_fields`.

## 6.3 Formulários

O Django tem uma mania: as coisas são bem estruturadas. Com formulários não é diferente. Embora você possa fazer tudo em uma view – esse capítulo não vai seguir por esse caminho – o Django também permite criar uma estrutura para formulários. Isso significa criar uma classe para representar o formulário de contato com três atributos: nome, email e mensagem. Além disso, a classe é responsável por fazer a validação dos dados. Nesse caso, considere que o aplicativo `app_noticias` não aceita:

- mensagem de contato de e-mail do domínio “gmail.com”
- mensagem de contato que contenha alguma das palavras “problema”, “defeito” ou “erro”

A classe `ContatoForm` está em `app_noticias/forms.py`:

```
1  from django import forms
2
3
4  class ContatoForm(forms.Form):
5      nome = forms.CharField(max_length=128, min_length=12)
6      email = forms.EmailField(required=False)
7      mensagem = forms.CharField(widget=forms.Textarea)
8
9      def clean(self):
10         dados = super().clean()
```

```
11         # não aceita e-mail do gmail
12         email = dados.get('email', None)
13         mensagem = dados.get('mensagem')
14         if '@gmail.com' in email:
15             self.add_error('email', 'Provedor de e-mail não suportado (
                gmail.com)')
16         # testa palavras não permitidas na mensagem
17         palavras = ['problema', 'defeito', 'erro']
18         for palavra in palavras:
19             if palavra in mensagem.lower():
20                 self.add_error('mensagem', 'Mensagem contém palavra não
                    permitida')
21         return dados
```

A classe `ContatoForm` herda de `django.forms.Form` e possui três atributos de classe:

- `nome` do tipo `forms.CharField`
- `email` do tipo `forms.EmailField`
- `mensagem` do tipo `forms.CharField`

Vamos chamar esses atributos de “campos”. O campo `nome` tem algumas informações adicionais: aceita de 12 a 128 caracteres, o que é definido, respectivamente, pelos parâmetros nomeados `min_length=12` e `max_length=128`. O campo `email` não é de preenchimento obrigatório (`required=False` – o valor padrão é `True`). O campo `mensagem` tem um “widget” diferente: `widget=forms.TextArea`. O “widget” é a forma de mudar a aparência do campo.

Além dos atributos a classe sobreescreve o método `clean()`, que é utilizado para implementar a lógica da validação. O método começa obtendo os valores dos campos, por meio da chamada `super().clean()`. A partir de então, utiliza `dados.get()` para obter valores dos campos e tratar cada situação.

O método `add_error()` é utilizado para adicionar uma mensagem de erro para um campo. O primeiro parâmetro é o nome do campo e o segundo parâmetro é a mensagem de erro.

Por fim, o método retorna `dados`.

**Muito útil:** Se for necessário gerar um erro não associado a um campo específico pode-se disparar uma exceção `forms.ValidationError`.

É importante observar que a classe `ContatoForm` declara não apenas os campos, mas também a lógica de validação do formulário. Isso é muito útil como arquitetura de software por separar e organizar código de outras partes do software.

## 6.4 FormView

Com o formulário pronto é hora de criar uma view para apresentá-lo. O Django fornece a view `FormView` para apresentar formulários. A seguir, o código que implementa duas classes `ContatoView` e `ContatoSucessoView`:

```
1 class ContatoView(FormView):
2     template_name = 'app_noticias/contato.html'
3     form_class = ContatoForm
4
5     def form_valid(self, form):
6         dados = form.clean()
7         mensagem = MensagemDeContato(nome=dados['nome'], email=dados['
            email'], mensagem=dados['mensagem'])
8         mensagem.save()
9         return super().form_valid(form)
10
11     def get_success_url(self):
12         return reverse('contato_sucesso')
13
14
15 class ContatoSucessoView(TemplateView):
16     template_name = 'app_noticias/contato_sucesso.html'
```

A classe `ContatoView` é uma `FormView` e, por isso, tem dois atributos de classe:

- `template_name` o nome do template (`app_noticias/contato.html`)
- `form_class` a classe do formulário (`ContatoForm`)

Além disso a classe implementa dois métodos: `form_valid()` e `get_success_url()`. O primeiro é executado quando o formulário está válido, ou seja, quando não há erro de validação. O código obtém os dados do formulário por meio do método `form.clean()`, cria uma instância do model `MensagemDeContato` e chama o método `save()` para salvar no banco de dados.

Um comportamento padrão de views herdam de `FormView` é entregar uma URL para o browser quando ocorre sucesso na execução do formulário. É por isso que a classe sobrescreve o método `get_success_url()` e seu código usa a função `reverse()` para traduzir o nome da URL `contato_sucesso` em uma URL completa.

A URL `contato_sucesso` está vinculada à view `ContatoSucessoView`, que é uma `TemplateView` bem simples (posso dizer isso agora, sem medo) que apenas apresenta uma tela de confirmação para o usuário.

## 6.5 Template para o formulário

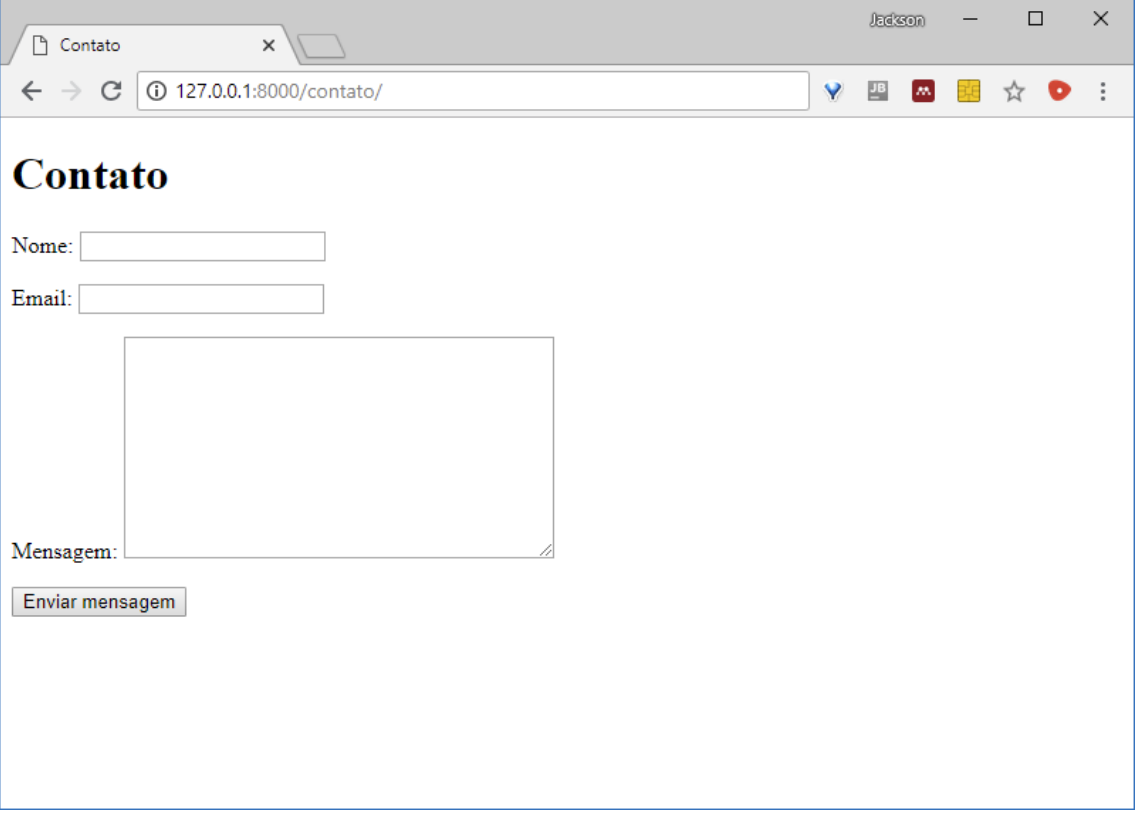
O template para o formulário (`app_noticias/contato.html`) requer uma estrutura padrão para views que herdam de `FormView`:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
```

```
5     <title>Contato</title>
6     <style type="text/css">
7         .errorlist {
8             color: red;
9         }
10    </style>
11 </head>
12 <body>
13 <h1>Contato</h1>
14 <form method="post">
15     {% csrf_token %}
16     {{ form.as_p }}
17     <button type="submit">Enviar mensagem</button>
18 </form>
19 </body>
20 </html>
```

A view `FormView` entrega o formulário para o template por meio da variável `form` mas, para apresentar o formulário, é preciso seguir algumas regras. Uma das formas de fazer isso é inserir manualmente o elemento `form`, a tag `csrf_token` (uma tag de segurança do Django), apresentar o formulário utilizando `form.as_p` e, por fim, incluir botões para permitir ao usuário enviar os dados (no exemplo, é utilizado o elemento `button`).

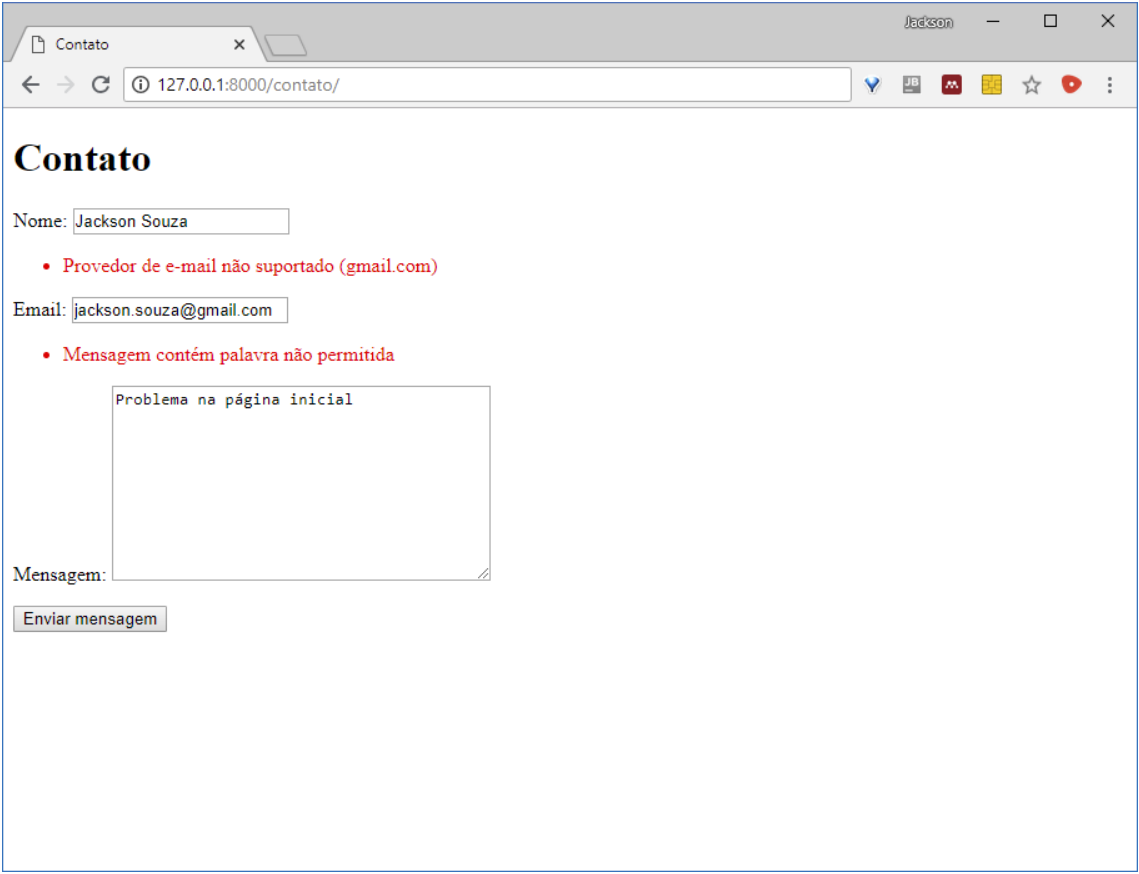
A Figura 6.1 mostra o formulário de contato.



The screenshot shows a web browser window with the title 'Contato'. The address bar displays '127.0.0.1:8000/contato/'. The page content includes a heading 'Contato', followed by input fields for 'Nome:' and 'Email:'. Below these is a large text area for 'Mensagem:'. At the bottom, there is a button labeled 'Enviar mensagem'.

Figura 6.1: Formulário de contato

A Figura 6.2 mostra o formulário de contato com as mensagens de erro.



The screenshot shows a web browser window with the title 'Contato'. The address bar displays '127.0.0.1:8000/contato/'. The form contains the following elements:

- Nome:** A text input field containing 'Jackson Souza'.
- Email:** A text input field containing 'jackson.souza@gmail.com'.
- Messages:** Two red error messages are displayed:
  - Provedor de e-mail não suportado (gmail.com)
  - Mensagem contém palavra não permitida
- Mensagem:** A large text area containing the text 'Problema na página inicial'.
- Enviar mensagem:** A button at the bottom of the form.

Figura 6.2: Formulário de contato com as mensagens de erro

A Figura 6.3 mostra a tela de sucesso depois de um preenchimento válido.

Por fim, a Figura 6.4 mostra o Django admin apresentando os dados da mensagem de contato.

## 6.6 Se tem URL de sucesso, tem URL de erro?

É possível que essa pergunta tenha surgido naturalmente nesse ponto: se a `FormView` possui uma `success_url` então também teria algo semelhante a uma `error_url`? A resposta curta é: Não. A resposta longa é: vamos olhar um pouco mais para a filosofia do Django sobre formulários.

A filosofia do Django para formulários é separar código. Isso não funciona apenas para validação, mas também para lógica de negócio. Isso quer dizer que o exemplo que este capítulo apresentou poderia ser feito de outra forma: o código que está no método `form_valid()` da view deveria estar em um método do formulário, que deveria ser chamado em `form_valid()`. Se algo de errado ocorresse, o método deveria disparar uma exceção (usando `forms.ValidationError`), então o browser continuaria na mesma página e seria apresentada uma mensagem de erro.

Por esse motivo o Django não fornece uma URL de erro.

Há ainda outro recurso para interagir com o usuário para apresentar mensagens curtas, mas isso



Figura 6.3: Página de confirmação da mensagem de contato

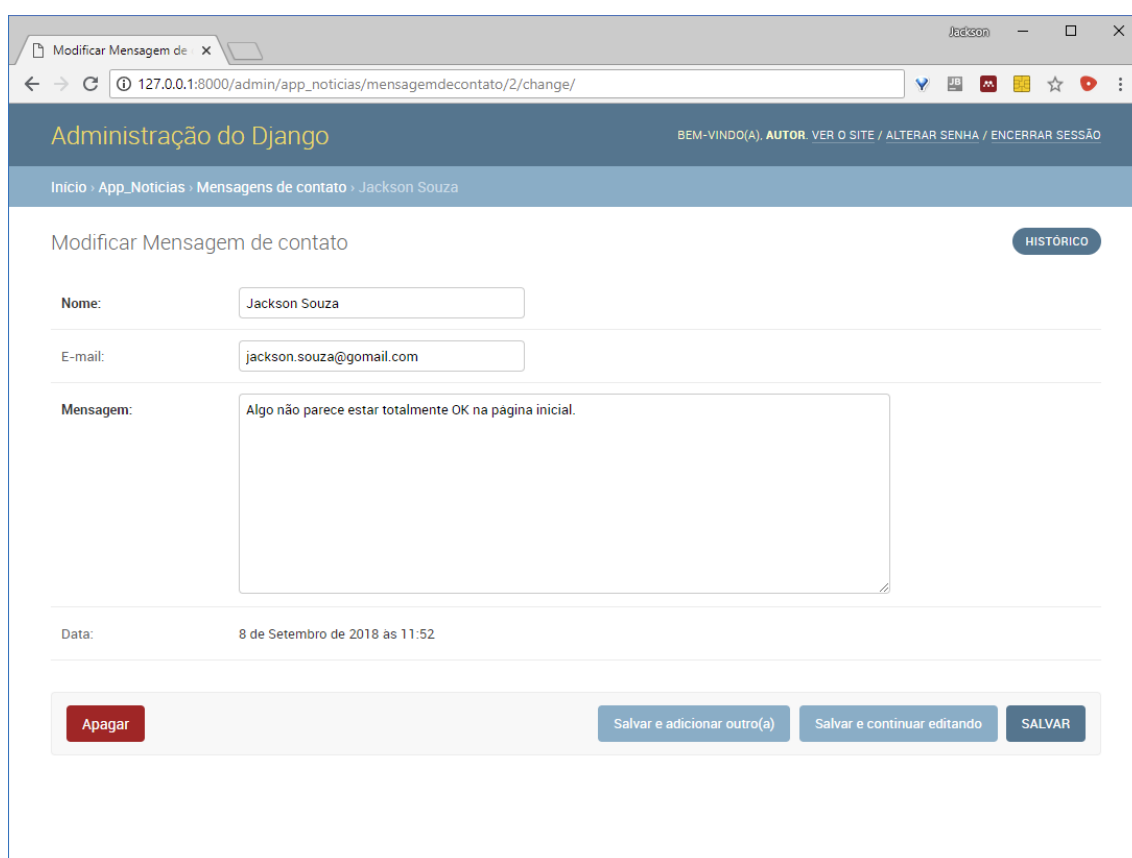


Figura 6.4: Formulário de contato com as mensagens de erro

será visto em outro capítulo.

## 6.7 Conclusão

O capítulo apresentou como o Django separa código entre views e formulários. Mostrou também como é a estrutura de uma classe que herda de `Form` para definir campos e a lógica da validação.

O capítulo também mostrou como a view interage com o formulário e como um formulário é apresentado no template.



## Apêndice A

# Configuração do ambiente Python

### A.1 Windows

Faça download do instalador do Python na [página de releases para windows](https://www.python.org/downloads/windows/)<sup>1</sup>.

Depois de concluir o download execute o instalador e siga os passos apresentados nas telas.

Verifique a instalação obtendo a versão do Python, executando o seguinte comando:

```
$ python --version
```

O comando apresenta a versão instalada.

### A.2 Linux (Ubuntu)

Antes de continuar, atualize seus repositórios `apt` executando o comando:

```
$ sudo apt-get update
```

Execute o comando a seguir para instalar o python 3:

```
$ sudo apt-get install build-essential python3 python3-pip python3-dev python3-setuptools
```

Esse comando instala: **Python**, **pip** e pacotes para um ambiente completo de desenvolvimento Python.

Geralmente a instalação deste pacote tornará disponíveis os programas `python3` e `pip3`. Lembre-se disso porque distribuições Linux costumam usar esse recurso diferenciar o **Python 2.x** do **Python**

---

<sup>1</sup>Acesse: <https://www.python.org/downloads/windows/>

3.x.

## A.3 Ambiente com permissões restritas

Se você estiver utilizando um ambiente com restrições de permissões (ie. não tem acesso root ou administrator) adicione a opção `--user` toda vez que utilizar o comando `pip` antes de habilitar um ambiente do projeto. Isso fará com que os pacotes sejam instalados no diretório do seu usuário e não haverá problemas com permissões. Por exemplo, para instalar o `pipenv`:

```
$ pip install pipenv --user
```

## A.4 Usando o virtualenv

Instale o `virtualenv` utilizando `pip`:

```
$ pip install virtualenv
```

### A.4.1 Criação de um ambiente do projeto

A criação de um ambiente do projeto permite diferenciar pacotes e versões de pacotes do ambiente global do python.

A partir da pasta do projeto execute:

```
$ virtualenv env
```

Nesse caso será criado um ambiente python para o projeto local chamado `env` e estará na pasta `./env`, contendo os programas principais: `python`, `pip`, `activate` e `deactivate`. Os dois últimos são responsáveis, respectivamente, por ativar e desativar o ambiente local. Você pode mudar o nome do ambiente, se preferir.

### A.4.2 Ativação do ambiente

A ativação do ambiente é um pouco diferente entre Windows e Linux. A partir da pasta do projeto, execute:

no Windows:

```
$ env\Scripts\activate
```

no Linux:

```
$ source env/bin/activate
```

A indicação de que o comando foi alterado com sucesso é a presença de `(env)` no prompt e, além disso, você pode executar o comando a seguir para obter a lista de pacotes instalados no ambiente local:

```
$ pip list
```

Se tudo estiver correto, você verá uma lista com:

- `pip`
- `setuptools`
- `wheel`

Perceba que não é mais necessário usar os programas `pip` ou `pip3` para diferenciar a versão do Python. Apenas `pip` é necessário.

Na prática, o programa `activate` configura o ambiente do projeto definindo, principalmente, variáveis de ambiente.

### A.4.3 Desativação do ambiente

Para desativar o ambiente do projeto e retornar ao ambiente global do Python execute o programa `deactivate`:

```
$ deactivate
```

### A.4.4 Instalação de pacotes

Uma vez que o ambiente do projeto esteja ativado é possível instalar pacotes utilizando o `pip`, como o exemplo a seguir, que mostra como instalar o `django`:

```
$ pip install django
```

É importante lembrar que os pacotes são instalados apenas no ambiente do projeto.

É uma prática comum utilizar o arquivo `requirements.txt` para listar as dependências (os pacotes) do ambiente. Se o projeto não tiver esse arquivo, é possível gerá-lo utilizando o `pip`, como mostra o exemplo:

```
$ pip freeze > requirements.txt
```

O comando obtém a lista dos pacotes instalados no ambiente do projeto e cria o arquivo `requirements.txt`.

Também é possível instalar os pacotes a partir de um arquivo `requirements.txt`, também utilizando `pip`:

```
$ pip install -r requirements.txt
```

Nesse caso o `pip` obtém os pacotes e suas especificações de versões do arquivo `requirements.txt` e instala no ambiente do projeto.

## A.5 Usando o `pipenv`

Para instalar o `pipenv` utilize o comando:

```
$ pip install pipenv
```

Para detalhes da instalação leia a [documentação oficial do `pipenv`](#)<sup>2</sup>.

Depois da instalação do `pipenv` você poderá utilizá-lo para criar um ambiente Python de forma semelhante ao `virtualenv`.

### A.5.1 Ativação do ambiente

Para ativar o ambiente do projeto utilize o comando a partir da pasta do projeto:

```
$ pipenv shell
```

Esse processo é semelhante ao utilizado no `virtualenv` e faz a mesma coisa: configura variáveis de ambiente e modifica o prompt para mostrar uma identificação do ambiente.

### A.5.2 Desativação do ambiente

A desativação do ambiente do projeto é feita com o programa `exit`, portanto basta executá-lo:

```
$ exit
```

### A.5.3 Instalação de pacotes

A instalação de pacotes é feita com o programa `pipenv`:

```
$ pipenv install django
```

---

<sup>2</sup>Acesse: <https://docs.pipenv.org/>

Nesse caso o programa `pipenv` instala o pacote `django` no ambiente do projeto.

O **pipenv** mantém dois arquivos para o gerenciamento das dependências (pacotes) do projeto:

- `Pipfile`
- `Pipfile.lock`

Esses arquivos armazenam as informações sobre o ambiente do projeto e sobre os pacotes.

Para instalar pacotes a partir de um arquivo `requirements.txt` use o comando:

```
$ pipenv install -r requirements.txt
```

## Apêndice B

### Git

```
1 $ git init
2 $ git add .
3 $ git commit -m "mensagem do commit"
```

## Apêndice C

# Utilizando Heroku CLI

### C.1 Fazendo login

Execute o comando:

```
1 $ heroku login
```

Siga as instruções para fornecer o e-mail e senha da conta de usuário no Heroku.

# Referências

DJANGO SOFTWARE FOUNDATION. **The Web framework for perfectionists with deadlines | Django**, [s.d.].

GIT COMMUNITY. **Git**, [s.d.]. Disponível em: <<https://git-scm.com/>>. Acesso em: 22 jul. 2018

MICROSOFT. **Visual Studio Code - Code Editing. Redefined**, [s.d.]. Disponível em: <<https://code.visualstudio.com/>>. Acesso em: 22 jul. 2018

PYPA. **pip – pip 1.8.0 documentation**, [s.d.]. Disponível em: <<https://pip.pypa.io/en/stable/>>. Acesso em: 24 jul. 2018a

PYPA. **Virtualenv – virtualenv 16.0.0 documentation**, [s.d.]. Disponível em: <<https://virtualenv.pypa.io/en/stable/>>. Acesso em: 24 jul. 2018b

PYPA. **Pipenv: Python Dev Workflow for Humans**, [s.d.]. Disponível em: <<https://docs.pipenv.org/>>. Acesso em: 24 jul. 2018c

SALESFORCE.COM. **Cloud Application Platform | Heroku**, [s.d.]. Disponível em: <<https://www.heroku.com/>>. Acesso em: 24 jul. 2018

WIKIPEDIA CONTRIBUTORS. **Model–view–controller — Wikipedia, The Free Encyclopedia**, 2018. Disponível em: <<https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller&oldid=849850595>>. Acesso em: 23 jul. 2018

WIKIPÉDIA. **Cliente-servidor — Wikipédia, a enciclopédia livre**, 2018. Disponível em: <<https://pt.wikipedia.org/w/index.php?title=Cliente-servidor&oldid=52116277>>. Acesso em: 24 jul. 2018