

## Sumário

Escrevendo seu primeiro app Django, parte 1.....	3
Criando um projeto.....	3
O servidor de desenvolvimento .....	4
Criando a aplicação de enquetes: Polls .....	6
Escreva sua primeira view .....	6
<b>argumento de path(): route</b> .....	8
<b>argumento de path(): view</b> .....	8
<b>argumento de path(): kwargs</b> .....	8
<b>argumento de path(): name</b> .....	9
Escrevendo sua primeira aplicação Django, parte 2 .....	9
Configuração do banco de dados.....	9
Criando modelos.....	10
Ativando modelos .....	11
Brincando com a API .....	15
Escrevendo sua primeira aplicação Django, parte 7 .....	20
<b>Criando um usuário administrador</b> .....	20
<b>Inicie o servidor de desenvolvimento</b> .....	20
<b>Entre no site de administração</b> .....	21
<b>Torne a aplicação de enquetes editável no site de administração</b> .....	21
<b>Explore a funcionalidade de administração de graça</b> .....	22
Escrevendo sua primeira app Django, parte 3.....	24
Visão Geral.....	24
Escrevendo mais views.....	25
Escreva views que façam algo .....	26
<b>Um atalho: render()</b> .....	28
Levantando um erro 404 .....	29
<b>Um atalho: get_object_or_404()</b> .....	30
Use o sistema de template .....	30
Removendo URLs codificados nos templates .....	31
Namespacing nomes de URL .....	31
Escrevendo sua primeira aplicação Django, parte 4 .....	32
Write a minimal form.....	32
Use views genéricas: Menos código é melhor .....	36
<b>Corrija URLconf</b> .....	36
<b>Views alteradas</b> .....	37

Escrevendo sua primeira aplicação Django, parte 5 .....	39
Apresentando testes automatizados .....	39
O que são testes automatizados? .....	39
Porque você precisa criar testes .....	39
Estratégias básicas de testes .....	40
Escrevendo nosso primeiro teste .....	40
Nós identificamos um bug .....	40
Criar um teste para expor um bug .....	41
Executando o teste .....	42
Corrigindo o erro .....	43
Testes mais abrangentes .....	44
Teste a View .....	44
Um teste para uma view .....	45
O cliente de testes do Django .....	45
Melhorando a nossa view .....	46
Testando nossa nova view .....	47
Testando o <code>DetailView</code> .....	50
Ideias para mais testes .....	51
Quando estiver testando, mais é melhor .....	51
Mais testes .....	52
E agora? .....	52
Escrevendo sua primeira aplicação Django, parte 6 .....	52
Personalize a aparência da sua <i>aplicação</i> .....	52
Adicionar uma imagem de fundo .....	53
Escrevendo seu primeiro app Django, parte 7 .....	54
Personalize o formulário do site de administração .....	54
Adicionando objetos relacionados .....	56
Personalize a listagem da página de administração .....	61
Personalize a aparência do site de administração .....	63
Personalize os templates do seu <i>projeto</i> .....	64
Personalize os templates da sua <i>aplicação</i> .....	65
Personalize a página inicial do site de administração .....	65

# Escrevendo seu primeiro app Django, parte 1

Vamos aprender por um exemplo.

Através deste tutorial, nós vamos caminhar através da criação de uma aplicação básica de enquetes.

Ela irá consistir de duas partes:

- Uma página pública que permitirá que pessoas vejam as enquetes e votem nelas.
- Um site de administração que permite adicionar, alterar e deletar enquetes.

Assumiremos que você já tem o **Django instalado**. Você pode verificar se o Django está instalado e em qual versão rodando o seguinte comando em um prompt shell (indicado pelo prefixo \$):

```
$ python -m django --version
```

Se o Django estiver instalado, você verá a versão de sua instalação. Se não estiver, você receberá uma mensagem de erro dizendo “No module named django”.

Este tutorial foi escrito para Django 3.0, com suporte para Python 3.6 ou mais atual. Se a versão do Django não é a mesma, você pode acessar o tutorial da sua versão de Django usando o menu de versões no canto inferior desta página, ou atualizando o Django para uma versão mais recente. Se estiver usando uma versão mais antiga de Python, veja **Qual versão do Python eu posso usar com Django?** para encontrar uma versão compatível do Django.

Veja **Como instalar o Django** para recomendação sobre como remover versões antigas do Django e instalar uma mais recente.

## Onde obter ajuda:

Se tiver problemas enquanto caminha por este tutorial, por favor consulte a seção **Obtendo ajuda** da FAQ.

## Criando um projeto

Se esta é a primeira vez em que você está utilizando o Django, você terá que preocupar-se com algumas configurações iniciais. Isto é, você precisará auto-gerar um código para iniciar um **projeto** Django – uma coleção de configurações para uma instância do Django, incluindo configuração do banco de dados, opções específicas do Django e configurações específicas da aplicação.

A partir da linha de comando, execute **cd** para o diretório onde você gostaria de armazenar seu código, então execute o seguinte comando:

```
$ django-admin startproject mysite
```

Isto irá criar o diretório **mysite** no seu diretório atual. Se não funcionar, veja **Problemas ao rodar o django-admin**.

## Nota

Você precisará evitar dar nomes a projetos que remetam a componentes internos do Python ou do Django. Particularmente, isso significa que você deve evitar usar nomes como **django** (que irá conflitar com o próprio Django) ou **test** (que irá conflitar com um pacote interno do Python).

## Onde esse código deveria existir?

Se você tem experiência prévia em PHP (sem o uso de um framework moderno), você deve estar acostumado a colocar o código dentro do “document root” de seu servidor Web (em um lugar como **/var/www**). Com o Django você não fará

isto. Não é uma boa ideia colocar qualquer código Python no document root de seu servidor Web, porque existe o risco de pessoas conseguirem ver seu código através da Web. Isso não é bom para a segurança.

Coloque seu código em algum diretório **fora** do document root, como em **/home/mycode**.

Vamos ver o que o **startproject** criou:

```
mysite/
  manage.py
  mysite/
    __init__.py
    settings.py
    urls.py
    asgi.py
    wsgi.py
```

Esses arquivos são:

- O diretório **mysite/** exterior é um contêiner para o seu projeto. Seu nome não importa para o Django; você pode renomeá-lo para qualquer nome que você quiser.
- **manage.py**: um utilitário de linha de comando que permite a você interagir com esse projeto Django de várias maneiras. Você pode ler todos os detalhes sobre o **manage.py** em [django-admin and manage.py](#).
- O diretório **mysite/** interior é o pacote Python para o seu projeto. Seu nome é o nome do pacote Python que você vai precisar usar para importar coisas do seu interior (por exemplo, **mysite.urls**).
- **mysite/\_\_init\_\_.py**: um arquivo vazio que diz ao Python que este diretório deve ser considerado um pacote Python. Se você é um iniciante Python, leia [mais sobre pacotes](#) na documentação oficial do Python.
- **mysite/settings.py**: configurações para este projeto Django. [Configurações do Django](#) irá revelar para você tudo sobre o funcionamento do **settings**.
- **mysite/urls.py**: as declarações de URLs para este projeto Django; um “índice” de seu site movido a Django. Você pode ler mais sobre URLs em [Despachante de URL](#).
- **mysite/asgi.py**: um ponto de integração para servidores web compatíveis com ASGI usado para servir seu projeto. Veja [How to deploy with ASGI](#) para mais detalhes.
- **mysite/wsgi.py**: um ponto de integração para servidores web compatíveis com WSGI usado para servir seu projeto. Veja [Como implementar com WSGI](#) para mais detalhes.

## O servidor de desenvolvimento

Vamos verificar se ele funciona. Vá para o diretório **mysite**, se você ainda não estiver nele, e execute o seguinte comando:

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

Você verá a seguinte saída na sua linha de comando:

```
Performing system checks...
```

```
System check identified no issues (0 silenced).
```

```
You have unapplied migrations; your app may not work properly until they are applied.
```

```
Run 'python manage.py migrate' to apply them.
```

```
junho 08, 2020 - 15:50:53
```

```
Django version 3.0, using settings 'mysite.settings'
```

```
Starting development server at http://127.0.0.1:8000/
```

```
Quit the server with CONTROL-C.
```

### Nota

Por hora, ignore os avisos sobre as migrações do banco de dados não aplicadas; lidaremos com o banco de dados brevemente.

Você iniciou o servidor de desenvolvimento do Django, um servidor Web leve escrito puramente em Python. Nós incluímos ele com o Django, então você pode desenvolver coisas rapidamente, sem ter que lidar com a configuração de um servidor Web de produção – como o Apache – até que você esteja pronto para a produção.

Agora é uma boa hora para notar: **não** use esse servidor em nada relacionado a um ambiente de produção. Seu objetivo é ser utilizado apenas durante o desenvolvimento. (Nós estamos na atividade de criação de frameworks Web, não de servidores Web.)

Agora que o servidor está rodando, visite <http://127.0.0.1:8000/> no seu navegador Web. Você verá uma página de “Parabéns!”, com um foguete decolando. Funciona !

### Alterando a porta

Por padrão, o comando **runserver** inicia o servidor de desenvolvimento no IP interno na porta 8000.

Se você quer mudar a porta do servidor, passe ela como um argumento na linha de comando. Por exemplo, este comando iniciará o servidor na porta 8080:

```
$ python manage.py runserver 8080
```

Se você quer mudar o IP do servidor, passe junto com a porta. Por exemplo, para escutar em todos os IPs públicos disponíveis (o que é útil se estiver usando Vagrant ou quer mostrar seu trabalho para outros computadores na rede), use:

```
$ python manage.py runserver 0:8000
```

**o** é um atalho para **0.0.0.0**. A documentação completa para o servidor de desenvolvimento pode ser encontrada na referência do **runserver**.

### Recarregamento automática de runserver

O servidor de desenvolvimento recarrega o código Python para cada solicitação, conforme necessário. Você não precisa reiniciar o servidor para que alterações de código tenham efeito. No entanto, algumas ações como a adição de arquivos não disparam uma reinicialização, então você terá que reiniciar o servidor nestes casos.

## Criando a aplicação de enquetes: Polls

Agora que seu ambiente – um “projeto” – está configurado, você está pronto para começar o trabalho.

Cada aplicação que você escreve no Django consiste de um pacote Python que segue uma certa convenção. O Django vem com um utilitário que gera automaticamente a estrutura básica de diretório de uma aplicação, então você pode se concentrar apenas em escrever código em vez de ficar criando diretórios.

### Projetos versus aplicações

Qual é a diferença entre um projeto e uma aplicação? Uma aplicação é um conjunto de elementos web que faz alguma coisa - por exemplo, um sistema de blog, um banco de dados de registros públicos, ou uma pequena aplicação de enquetes. Um projeto é uma coleção de configurações e aplicações para um website particular. Um projeto pode conter múltiplas aplicações. Uma aplicação pode estar em múltiplos projetos. Em Django, chamamos uma aplicação de “app”.

Your apps can live anywhere on your **Python path**. In this tutorial, we'll create our poll app in the same directory as your **manage.py** file so that it can be imported as its own top-level module, rather than a submodule of **mysite**.

Para criar sua aplicação, certifique-se de que esteja no mesmo diretório que **manage.py** e digite o seguinte comando:

```
$ python manage.py startapp polls
```

Que irá criar o diretório **polls** com a seguinte estrutura:

```
polls/
  __init__.py
  admin.py
  apps.py
  migrations/
    __init__.py
  models.py
  tests.py
  views.py
```

Esta estrutura de diretório irá abrigar a aplicação de enquete.

## Escreva sua primeira view

Hora de criar a primeira view. Abra o arquivo **polls/views.py** e ponha o seguinte código em Python dentro dele:

```
polls/views.py
```

```
from django.http import HttpResponse
```

```
def index(request):
    return HttpResponse("Hello, world. You're at the polls index.")
```

Esta é a view mais simples possível no Django. Para chamar a view, nós temos que mapear a URL - e para isto nós precisamos de uma URLconf.

Para criar uma URLconf no diretório **polls**, crie um arquivo chamado **urls.py**. Agora seu diretório da aplicação deve ficar como:

```
polls/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    urls.py
    views.py
```

No arquivo **polls/urls.py** inclua o seguinte código:

**polls/urls.py**

```
from django.urls import path

from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

O próximo passo é apontar na raiz da URLconf para o módulo **polls.urls**. No arquivo **mysite/urls.py**, adicione uma importação de **django.urls.include** e insira um **include()** na lista **urlpatterns**, então você tem:

**mysite/urls.py**

```
from django.contrib import admin

from django.urls import include, path
```

```
urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

A função **include()** permite referenciar outras URLconfs. Qualquer lugar que o Django encontrar **include()**, irá recortar todas as partes da URL encontrada até aquele ponto e enviar a string restante para URLconf incluído para processamento posterior.

A idéia por trás do **include()** é facilitar plugar URLs. Uma vez que polls está em sua própria URLconf (**polls/urls.py**), ele pode ser colocado depois de `"/polls/"`, ou depois de `"/fun_polls/"`, ou depois de `"/content/polls/"`, ou qualquer outro início de caminho, e a aplicação ainda irá funcionar

### Quando usar include()

Deve-se sempre usar **include()** quando você incluir outros padrões de URL. **admin.site.urls** é a única exceção a isso.

Agora você amarrou uma view **index** no seu URLconf. Verique se funciona com o seguinte comando:

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

Acesse <http://localhost:8000/polls/> no seu navegador, e deverá ver o texto *"Hello, world. You're at the polls index."*, o qual foi definido na view `"index"`.

### Página não encontrada?

Se retornar uma página de erro, verifique se está acessando <http://localhost:8000/polls/> e não <http://localhost:8000/>.

A função **path()** são passado quatro argumentos, dois obrigatórios: **route** e **view**, e dois opcionais: **kwargs**, e **name**. Neste ponto, vale a pena revisar para o que estes argumentos servem.

### argumento de path(): route

**route** é uma string contendo uma descrição de uma URL. Quando processa uma requisição, o Django começa pela primeira descrição, e vai descendo a lista, comparando a URL requisitada com cada descrição até que encontre uma que combine.

Descrições não distinguem parâmetros GET e POST, ou o nome do domínio. Por exemplo, em uma requisição <https://www.example.com/myapp/>, o URLconf irá procurar por **myapp/**. Em uma requisição para <https://www.example.com/myapp/?page=3>, o URLconf também irá procurar por **myapp/**.

### argumento de path(): view

Quando o Django encontra uma descrição que combina, chama a função view especificada com um objeto **HttpRequest** como primeiro argumento e qualquer valor "capturado" da rota como argumentos keyword. Daremos um exemplo sobre isso logo.

### argumento de path(): kwargs

Argumentos nomeados arbitrários podem ser passadas em um dicionário para a view de destino. Nós não vamos usar este recurso do Django neste tutorial.



## argumento de `path()`: `name`

Nomear sua URL permite que você referencie ela de forma inequívoca de qualquer lugar no Django especialmente nos templates. Este poderoso recurso permite que você faça alterações globais nos padrões de URL de seu projeto enquanto modifica um único arquivo.

Quando estiver confortável com o básico da sequência de requisição e resposta, leia a [parte 2 desse tutorial](#) para começar a trabalhar com banco de dados.

# Escrevendo sua primeira aplicação Django, parte 2

Este tutorial inicia onde [Tutorial 1](#) termina. Iremos configurar um banco de dados, criar seu primeiro modelo, e ter uma rápida introdução ao site de administração do Django automaticamente gerado.

### Onde obter ajuda:

Se tiver problemas enquanto caminha por este tutorial, por favor consulte a seção [Obtendo ajuda](#) da FAQ.

## Configuração do banco de dados

Agora, abra o `mysite/settings.py`. Ele é um módulo normal do Python com variáveis de módulo representando as configurações do Django.

Por padrão, a configuração usa o SQLite. Se você é novo com banco de dados, ou se estiver somente interessado em experimentar o Django, esta é a maneira mais simples. SQLite está incluso no Python, de modo que você não precisa instalar nada a mais para ter o banco de dados. Contudo, quando iniciar seu primeiro projeto real, talvez queira usar um banco de dados mais escalável como o PostgreSQL, para evitar problemas com a troca do banco de dados no caminho.

Se você quiser usar outro banco de dados, instale o de acordo [database bindings](#) e altere os seguintes parâmetros no item `DATABASES` `'default'` de acordo com o seu banco de dados.

- **ENGINE** – ou mesmo `'django.db.backends.sqlite3'`, `'django.db.backends.postgresql'`, `'django.db.backends.mysql'`, ou `'django.db.backends.oracle'`. Outros backends são [also available](#).
- **NAME** – O nome do seu banco de dados, se você estiver usando SQLite, o banco de dados será um arquivo no seu computador; neste caso, **NAME** deve ser o caminho completo, incluindo o nome, para este arquivo. O valor padrão `os.path.join(BASE_DIR, 'db.sqlite3')`, irá criar este arquivo no diretório do seu projeto.

Caso não use o SQLite como banco de dados, configurações adicionais como **USER**, **PASSWORD**, e **HOST** deverão ser adicionadas. Para mais detalhes, veja a referência na documentação para **DATABASES**.

### Para fins diferentes da base de dados SQLite

Caso use um banco de dados além do SQLite, tenha certeza que o banco de dados foi criado até este ponto. Faça isso com `"CREATE DATABASE database_name;"` dentro do prompt interativo do seu banco de dados.

Também se certifique que o "database user" indicado em `mysite/settings.py` tenha a permissão "create database". Isso permite a criação automática de [test database](#) o qual será necessário e um tutorial mais adiante.

Se você está utilizando SQLite você não precisa criar nada de antemão - o arquivo do banco de dados será criado automaticamente quando ele for necessário.

Enquanto estiver editando `mysite/settings.py`, mude **TIME\_ZONE** para o seu fuso horário.

Também observe a configuração do **INSTALLED\_APPS** na parte superior do arquivo. Ela possui os nomes de todas as aplicações Django ativas para essa instância do Django. Aplicações podem ser usadas em múltiplos projetos, e você pode empacotá-las e distribuí-las para uso por outros em seus projetos.

Por padrão, o **INSTALLED\_APPS** contém as seguintes aplicações, que vêm com o Django:

- **django.contrib.admin** – O site de administração. Irá usar isso em breve.
- **django.contrib.auth** – Um sistema de autenticação.
- **django.contrib.contenttypes** – Um framework para tipos de conteúdo.
- **django.contrib.sessions** – Um framework de sessão.
- **django.contrib.messages** – Um framework de envio de mensagem.
- **django.contrib.staticfiles** – Um framework para gerenciamento de arquivos estáticos.

Estas aplicações são incluídas por padrão como uma conveniência para o caso comum.

Algumas dessas aplicações fazem uso de pelo menos uma tabela no banco de dados, assim sendo, nós precisamos criar as tabelas no banco de dados antes que possamos utilizá-las. Para isso rode o seguinte comando:

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

O comando **migrate** verifica a configuração em **INSTALLED\_APPS** e cria qualquer tabela do banco de dados necessária de acordo com as configurações do banco de dados no seu arquivo **mysite/settings.py** e as migrações que venham com a aplicação (cobriremos isso mais tarde). Você verá uma mensagem para cada migração que foi aplicada. Se lhe interessar, execute o cliente de linha de comando para seu banco de dados e digite **\dt** (PostgreSQL), **SHOW TABLES;** (MySQL), **.schema** (SQLite), ou **SELECT TABLE\_NAME FROM USER\_TABLES;** (Oracle) para mostrar as tabelas criadas pelo Django.

#### Para os minimalistas

Como dissemos acima, as aplicações padrão são incluídas para casos comuns, mas nem todo mundo precisa delas. Se você não precisa de nenhuma delas, sintá-se livre para comentar ou deletar a(s) linha(s) apropriada(s) do **INSTALLED\_APPS** antes de rodar o **migrate**. O comando **migrate** irá executar as migrações apenas para as aplicações que estiverem no **INSTALLED\_APPS**.

## Criando modelos

Agora vamos definir seus modelos – essencialmente, o layout do banco de dados, com metadados adicionais.

#### Filosofia

Um modelo é a única e definitiva fonte de verdade sobre seus dados. Ele contém os campos essenciais e os comportamentos para os dados que você estiver armazenando. O Django segue o **princípio DRY**. O objetivo é definir seu modelo de dados em um único local e automaticamente derivar coisas a partir dele.

Isso inclui as migrações - ao contrário de Ruby on Rails, por exemplo, as migrações são inteiramente derivada de seu arquivo de modelos, e são essencialmente apenas uma história que o Django pode avançar para atualizar o esquema de banco de dados para coincidir com seus modelos atuais.

Em nossa aplicação de enquete, nós iremos criar dois modelos: **Question** e **Choice**. Uma **Question** tem uma pergunta e uma data de publicação. Uma **Choice** tem dois campos: o texto da escolha e um totalizador de votos. Cada **Choice** é associada a uma **Question**.

Esses conceitos são representados por simples classes Python. Edite o arquivo **polls/models.py** de modo que fique assim:

**polls/models.py**

```
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
```

Aqui, cada modelo é representado por uma classe derivada da classe **django.db.models.Model**. Cada modelo possui alguns atributos de classe, as quais por sua vez representa um campo do banco de dados no modelo.

Cada campo é representado por uma instância de uma classe **Field** – por exemplo, **CharField** para campos do tipo caractere e **DateTimeField** para data/hora. Isto diz ao Django qual tipo de dado cada campo contém.

O nome de cada instância **Field** (por exemplo **question\_text** ou **pub\_date**) é o nome do campo, em formato amigável para a máquina. Você irá usar este valor no seu código Python, e seu banco de dados irá usá-lo como nome de coluna.

Você pode usar um argumento opcional na primeira posição de um **Field** para designar um nome legível para seres humanos o qual será usado em uma série de partes introspectivas do Django, e também servirá como documentação. Se esse argumento não for fornecido, o Django utilizará o nome legível para máquina. Neste exemplo nós definimos um nome legível para humanos apenas para **Question.pub\_date**. Para todos os outros campos neste modelo, o nome legível para máquina será utilizado como nome legível para humanos.

Algumas classes de **Field** possuem elementos obrigatórios. O **CharField**, por exemplo, requer que você informe a ele um **max\_length** que é usado não apenas no esquema do banco de dados, mas na validação, como nós veremos em breve.

Um **Field** pode ter vários argumentos opcionais; neste caso, definimos o valor **default** de **votes** para o.

Finalmente, note que uma relação foi criada, usando **ForeignKey**. Isso diz ao Django que cada **Choice** está relacionada a uma única **Question**. O Django oferece suporte para todos os relacionamentos comuns de um banco de dados: muitos-para-um, muitos-para-muitos e um-para-um.

## Ativando modelos

Aquela pequena parte do modelo fornece ao Django muita informação. Com ela o Django é capaz de:

- Criar um esquema de banco de dados (instruções **CREATE TABLE**) para a aplicação.
- Criar uma API de acesso a banco de dados para acessar objetos **Question** e **Choice**.

Mas primeiro nós precisamos dizer ao nosso projeto que a aplicação **polls** está instalada.

### Filosofia

Aplicações Django são "plugáveis": Você pode usar uma aplicação em múltiplos projetos, e você pode distribuir aplicações, porque elas não precisam ser atreladas a uma determinada instalação do Django.

Para incluir a aplicação no nosso projeto, precisamos adicionar a referência à classe de configuração da aplicação na definição do **INSTALLED\_APPS**. A

classe **PollsConfig** está no arquivo `:file:`polls/apps.py``, então seu caminho pontuado é `'polls.apps.PollsConfig'`. Edite o arquivo `mysite/settings.py` e adicione aquele caminho com notação de ponto a definição do **INSTALLED\_APPS**. Ficará parecido como aqui:

`mysite/settings.py`

```
INSTALLED_APPS = [
    'polls.apps.PollsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Agora o Django sabe que deve incluir a aplicação **polls**. Vamos rodar outro comando:

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

Você deverá ver algo similar ao seguinte:

```
Migrations for 'polls':
polls/migrations/0001_initial.py
- Create model Question
- Create model Choice
```

Ao executar **makemigrations**, você está dizendo ao Django que você fez algumas mudanças em seus modelos (neste caso, você fez novas modificações) e que você gostaria que as alterações sejam armazenadas como uma *migração*.

As migrações são como o Django armazena as alterações nos seus modelos (e, portanto, seu esquema de banco de dados) - eles são apenas arquivos no disco. Você pode ler a migração para seu novo modelo, se quiser; é o arquivo `polls/migrations/0001_initial.py`. Não se preocupe, você não precisa lê-los cada vez que o Django cria um, mas eles são projetados para serem editados no caso de você precisar ajustar manualmente como Django muda as coisas.

Existe um comando que vai rodar as migrações para você e gerenciar o esquema do banco de dados automaticamente - que é chamado **migrate**, e nós vamos chegar a ele em um momento - mas primeiro vamos ver qual SQL está migração vai rodar. O comando **sqlmigrate** recebe o nome da migração e o seu SQL.

```
$ python manage.py sqlmigrate polls 0001
```

Você deve ver algo semelhante ao seguinte (reformatamos para facilitar a leitura):

```
BEGIN;

--
-- Create model Question
--

CREATE TABLE "polls_question" (
    "id" serial NOT NULL PRIMARY KEY,
    "question_text" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);

--
-- Create model Choice
--

CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "choice_text" varchar(200) NOT NULL,
    "votes" integer NOT NULL,
    "question_id" integer NOT NULL
);

ALTER TABLE "polls_choice"
    ADD CONSTRAINT "polls_choice_question_id_c5b4b260_fk_polls_question_id"
        FOREIGN KEY ("question_id")
        REFERENCES "polls_question" ("id")
```

```
DEFERRABLE INITIALLY DEFERRED;

CREATE INDEX "polls_choice_question_id_c5b4b260" ON "polls_choice" ("question_id");

COMMIT;
```

Note o seguinte:

- A saída exata irá variar dependendo do banco de dados que você está utilizando.
- Os nomes das tabelas são gerados automaticamente combinando o nome da aplicação (**polls**) e o nome em minúsculas do modelo – **question** e **choice**. (Você pode alterar esse comportamento.)
- Chaves primárias (IDs) são adicionadas automaticamente. (Você também pode modificar isso.)
- Por convenção, o Django adiciona **"\_id"** ao nome do campo de uma chave estrangeira. (Sim, você pode alterar isso, como quiser.)
- The foreign key relationship is made explicit by a **FOREIGN KEY** constraint. Don't worry about the **DEFERRABLE** parts; it's telling PostgreSQL to not enforce the foreign key until the end of the transaction.
- Isto é atrelado ao banco que você está usando, então a utilização de campos específicos do banco de dados como **auto\_increment** (MySQL), **serial** (PostgreSQL), ou **integer primary key** (SQLite) é feita para você automaticamente. O mesmo ocorre com as aspas dos nomes de campos – e.g., usando aspas duplas ou aspas simples.
- The **sqlmigrate** command doesn't actually run the migration on your database - instead, it prints it to the screen so that you can see what SQL Django thinks is required. It's useful for checking what Django is going to do or if you have database administrators who require SQL scripts for changes.

Se você tiver interesse, você pode rodar **python manage.py check**; Ele checa por problemas no seu projeto sem criar migrations ou tocar seu banco de dados.

Agora rode o **migrate** novamente para criar essas tabelas dos modelos no seu banco de dados:

```
$ python manage.py migrate

Operations to perform:
  Apply all migrations: admin, auth, contenttypes, polls, sessions

Running migrations:
  Rendering model states... DONE
  Applying polls.0001_initial... OK
```

O comando **migrate** pega todas as migrações que ainda não foram aplicadas (Django rastreia quais foram aplicadas usando uma tabela especial em seu banco de dados chamada **django\_migrations**) e aplica elas no seu banco de dados - essencialmente, sincronizando as alterações feitas aos seus modelos com o esquema no banco de dados.

Migrações são muito poderosas e permitem que você altere seus modelos ao longo do tempo, à medida que desenvolve o seu projeto, sem a necessidade de remover o seu banco de dados ou tabelas e criar de novo - ele é especializado em atualizar seu banco de dados ao vivo, sem perda de dados. Nós vamos cobri-los com mais profundidade em uma parte posterior do tutorial, mas para agora, lembre-se deste guia de três passos para fazer mudanças nos seus modelos:

- Mude seus modelos (em `models.py`).
- Rode `python manage.py makemigrations` para criar migrações para suas modificações
- Rode `python manage.py migrate` para aplicar suas modificações no banco de dados.

The reason that there are separate commands to make and apply migrations is because you'll commit migrations to your version control system and ship them with your app; they not only make your development easier, they're also usable by other developers and in production.

Leia a [documentação do django-admin](#) para informações completas sobre o que o utilitário `manage.py` pode fazer.

## Brincando com a API

Agora vamos dar uma passada no shell interativo do Python e brincar um pouco com a API livre que o Django dá a você. Para invocar o shell do Python, use esse comando:

```
$ python manage.py shell
```

Nós usaremos isso em vez de simplesmente digitar "python", porque o `manage.py` configura a variável de ambiente `DJANGO_SETTINGS_MODULE`, O que dá ao Django o caminho de importação Python do arquivo `mysite/settings.py`.

Assim que você estiver no shell, explore a [API de banco de dados](#):

```
>>> from polls.models import Choice, Question # Import the model classes we just
wrote.

# No questions are in the system yet.

>>> Question.objects.all()

<QuerySet []>

# Create a new Question.

# Support for time zones is enabled in the default settings file, so
# Django expects a datetime with tzinfo for pub_date. Use timezone.now()
# instead of datetime.datetime.now() and it will do the right thing.

>>> from django.utils import timezone

>>> q = Question(question_text="What's new?", pub_date=timezone.now())

# Save the object into the database. You have to call save() explicitly.

>>> q.save()
```

```
# Now it has an ID.

>>> q.id
1

# Access model field values via Python attributes.

>>> q.question_text
"What's new?"

>>> q.pub_date
datetime.datetime(2012, 2, 26, 13, 0, 0, 775217, tzinfo=<UTC>)

# Change values by changing the attributes, then calling save().

>>> q.question_text = "What's up?"
>>> q.save()

# objects.all() displays all the questions in the database.

>>> Question.objects.all()
<QuerySet [<Question: Question object (1)>]>
```

Espere um pouco. **<Question: Question object (1)>** é uma representação totalmente inútil desse objeto. Vamos corrigir isso editando o modelo da **Question** (no arquivo **polls/models.py**) e adicionando um método **\_\_str\_\_()** a ambos **Question** e **Choice**:

**polls/models.py**

```
from django.db import models

class Question(models.Model):
    # ...

    def __str__(self):
        return self.question_text

class Choice(models.Model):
    # ...

    def __str__(self):
```



```
return self.choice_text
```

É importante adicionar métodos `__str__()` aos seus modelos, não apenas para sua própria conveniência quando estiver lidando com o prompt interativo, mas também porque representações de objetos são usadas por toda interface administrativa gerada automaticamente pelo Django.

Vamos também adicionar um método personalizado a este modelo:

`polls/models.py`

```
import datetime

from django.db import models
from django.utils import timezone

class Question(models.Model):
    # ...

    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)
```

Note a adição de `import datetime` e `from django.utils import timezone`, para referenciar o módulo padrão do Python `datetime` e o módulo Django utilitário de fuso horário `django.utils.timezone`, respectivamente. Se você não é familiar com o gerenciamento de fuso horário no Python, você pode aprender mais no [documentação de suporte a fuso horários](#).

Salve essas mudanças e vamos iniciar uma nova sessão do shell interativo do Python rodando `python manage.py shell` novamente:

```
>>> from polls.models import Choice, Question

# Make sure our __str__() addition worked.

>>> Question.objects.all()
<QuerySet [<Question: What's up?>]>

# Django provides a rich database lookup API that's entirely driven by
# keyword arguments.

>>> Question.objects.filter(id=1)
<QuerySet [<Question: What's up?>]>

>>> Question.objects.filter(question_text__startswith='What')
```

```

<QuerySet [<Question: What's up?>]>

# Get the question that was published this year.
>>> from django.utils import timezone
>>> current_year = timezone.now().year
>>> Question.objects.get(pub_date__year=current_year)
<Question: What's up?>

# Request an ID that doesn't exist, this will raise an exception.
>>> Question.objects.get(id=2)
Traceback (most recent call last):
...
DoesNotExist: Question matching query does not exist.

# Lookup by a primary key is the most common case, so Django provides a
# shortcut for primary-key exact lookups.
# The following is identical to Question.objects.get(id=1).
>>> Question.objects.get(pk=1)
<Question: What's up?>

# Make sure our custom method worked.
>>> q = Question.objects.get(pk=1)
>>> q.was_published_recently()
True

# Give the Question a couple of Choices. The create call constructs a new
# Choice object, does the INSERT statement, adds the choice to the set
# of available choices and returns the new Choice object. Django creates
# a set to hold the "other side" of a ForeignKey relation
# (e.g. a question's choice) which can be accessed via the API.

```

```

>>> q = Question.objects.get(pk=1)

# Display any choices from the related object set -- none so far.
>>> q.choice_set.all()
<QuerySet []>

# Create three choices.
>>> q.choice_set.create(choice_text='Not much', votes=0)
<Choice: Not much>
>>> q.choice_set.create(choice_text='The sky', votes=0)
<Choice: The sky>
>>> c = q.choice_set.create(choice_text='Just hacking again', votes=0)

# Choice objects have API access to their related Question objects.
>>> c.question
<Question: What's up?>

# And vice versa: Question objects get access to Choice objects.
>>> q.choice_set.all()
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>
>>> q.choice_set.count()
3

# The API automatically follows relationships as far as you need.
# Use double underscores to separate relationships.
# This works as many levels deep as you want; there's no limit.
# Find all Choices for any question whose pub_date is in this year
# (reusing the 'current_year' variable we created above).
>>> Choice.objects.filter(question__pub_date__year=current_year)
<QuerySet [<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]>

```

```
# Let's delete one of the choices. Use delete() for that.

>>> c = q.choice_set.filter(choice_text__startswith='Just hacking')

>>> c.delete()
```

Para mais informações sobre relacionamento de modelos, veja [Acessando objetos relacionados](#). Para mais informação em como usar sublinhados duplos para pesquisa usando campos da API, veja [Pesquisa com campos](#). Para um detalhamento completo da API de banco de dados, veja nossa [referência à API de Banco de Dados](#).

## Escrevendo sua primeira aplicação Django, parte 7

### Filosofia

Gerar um site de administração para sua equipe ou clientes para adicionar, alterar, e deletar conteúdo é uma tarefa tediosa que não requer muita criatividade. Por essa razão, o Django automatiza totalmente a criação da interface de administração para os modelos.

O Django foi desenvolvido em um ambiente de redação, onde havia uma clara separação entre “produtores de conteúdo” e o site “público”. Gerentes de site usam o sistema para adicionar notícias, eventos, resultado de esportes, etc, e o conteúdo é exibido no site público. O Django soluciona o problema de criar uma interface unificada para os administradores editarem o conteúdo.

A interface de administração não foi desenvolvida necessariamente para ser usada pelos visitantes do site, mas sim pelos gerentes.

### Criando um usuário administrador

Primeiro temos que criar um usuário que possa acessar o site de administração. Execute o seguinte comando:

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

Digite seu nome de usuário desejado e pressione enter.

```
Username: admin
```

Em seguida, será requisitado seu endereço de e-mail desejado:

```
Email address: admin@example.com
```

O último passo é digitar sua senha. Você será solicitado que digite sua senha duas vezes, a segunda vez como uma confirmação da primeira.

```
Password: *****
```

```
Password (again): *****
```

```
Superuser created successfully.
```

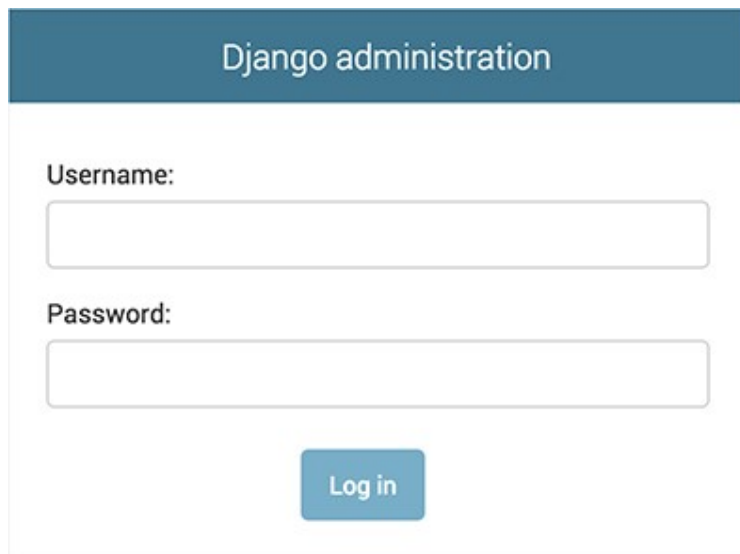
### Inicie o servidor de desenvolvimento

O site de administração vem ativado por padrão. Vamos iniciar o servidor de desenvolvimento e explorá-lo.

Se o servidor não estiver sendo executado, inicie-o da seguinte forma:

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

Agora, abra o navegador de internet e vá para “/admin/” no seu domínio local – ex.: <http://127.0.0.1:8000/admin/>. Você deverá ver a tela de acesso:



The image shows the Django administration login interface. It has a dark blue header with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:". Below the password field is a blue button labeled "Log in".

Since **translation** is turned on by default, if you set **LANGUAGE\_CODE**, the login screen will be displayed in the given language (if Django has appropriate translations).

## Entre no site de administração

Agora, tente acessar o sistema com a conta de super usuário que criou no passo anterior. Você deverá ver a página inicial do site de administração do Django:



The image shows the Django administration dashboard. At the top, there is a dark blue header with the text "Django administration" on the left and "WELCOME, ADMIN. VIEW SITE" on the right. Below the header, there is a section titled "Site administration". Under this section, there is a table with two rows: "Groups" and "Users". Each row has a "+ Add" link and a "Change" link. On the right side of the dashboard, there is a sidebar with the text "Recent" and "My Actions".

AUTHENTICATION AND AUTHORIZATION	
Groups	+ Add Change
Users	+ Add Change

Você deverá ver alguns outros tipos de conteúdos editáveis, incluindo grupos e usuários. Estas funcionalidades são fornecidas por `mod:django.contrib.auth`, o framework de autenticação fornecido pelo Django.

## Torne a aplicação de enquetes editável no site de administração

Mas onde está nossa aplicação de enquete? Ela não está visível na página principal do site de administração.

Only one more thing to do: we need to tell the admin that **Question** objects have an admin interface. To do this, open the `polls/admin.py` file, and edit it to look like this:

`polls/admin.py`

```
from django.contrib import admin

from .models import Question

admin.site.register(Question)
```

## Explore a funcionalidade de administração de graça

Agora que nós registramos **Question**, o Django sabe que ela deve ser exibida na página principal do site de administração:

### Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	<a href="#">+ Add</a>	<a href="#">Change</a>
Users	<a href="#">+ Add</a>	<a href="#">Change</a>
POLLS		
Questions	<a href="#">+ Add</a>	<a href="#">Change</a>

Recent /

My Actions

None avail

Clique em “Questions”. Agora você está na página “change list” para “questions”. Essa página mostra todas as “questions” em seu banco de dados e lhe deixa escolher uma para alterar. Lá está a pergunta “What’s up?” que criamos anteriormente.

Home > Polls > Questions

## Select question to change

Action:   0 of 1 selected

☐ QUESTION

☐ What's up?

1 question

Clique na enquete "What's up?" para editá-la:


Home > Polls > Questions > What's up?


## Change question

Question text:

What's up?

Date published:

Date: 2015-09-06 Today 

Time: 21:16:22 Now 

Delete

Save and add another

Save and continue

Coisas para observar aqui:

- O formulário é gerado automaticamente para o modelo **Question**.
- Os diferentes tipos de campos (**DateTimeField**, **CharField**) correspondem aos respectivos componentes HTML de inserção. Cada tipo de campo sabe como se exibir no site de administração do Django.
- Cada **DateTimeField** ganha um atalho JavaScript de graça. Datas possuem um atalho "Hoje" e um calendário popup, e horas têm um atalho "Agora" e um conveniente popup com listas de horas utilizadas comumente.

A parte inferior da página fornece uma série de opções:

- Salvar – Salva as alterações e retorna para a pagina lista de modificação para este tipo de objeto.
- Salvar e continuar editando – Salva as alterações e recarrega a página do site de administração para este objeto.

- Salvar e adicionar outro – Salva as informações e abre um novo formulário em branco para este tipo de objeto.
- Deletar – Exibe uma página de confirmação de exclusão.

Se o valor do “Date published” não bate com hora que a questão foi criada em [Tutorial 1](#), provavelmente quer dizer que foi esquecido de configurar o correto valor da configuração do **TIME\_ZONE**. Altere isso, e recarregue a página e verifique se o calor correto aparece.

Altere a “Data de publicação” clicando nos atalhos “Hoje” e “Agora”. Em seguida, clique em “Salvar e continuar editando.” Então clique em “Histórico” no canto superior direito. Você verá uma página exibindo todas as alterações feitas neste objeto pelo site de administração do Django, com a hora e o nome de usuário da pessoa que fez a alteração:

Home › Polls › Questions › What's up? › History		
Change history: What's up?		
DATE/TIME	USER	ACTION
Sept. 6, 2015, 9:21 p.m.	elky	Changed pub_date.

Quando estiver confortável com a API dos modelos e estiver familiarizado com o site de administração, leia [parte 3 desse tutorial](#) para aprender sobre como adicionar mais “views” para nossa aplicação “polls”.

## Escrevendo sua primeira app Django, parte 3

Este tutorial inicia-se onde o Tutorial 2 terminou. Vamos continuar a aplicação web de enquete e focaremos na criação da interface pública – “views”.

### Onde obter ajuda:

Se tiver problemas enquanto caminha por este tutorial, por favor consulte a seção [Obtendo ajuda](#) da FAQ.

## Visão Geral

Uma view é um “tipo” de página Web em sua aplicação Django que em geral serve a uma função específica e tem um template específico. Por exemplo, em uma aplicação de blog, você deve ter as seguintes views:

- Página inicial do blog - exibe os artigos mais recentes.
- Página de “detalhes” - página de vínculo estático permanente para um único artigo.
- Página de arquivo por ano - exibe todos os meses com artigos para um determinado ano.
- Página de arquivo por mês - exibe todos os dias com artigos para um determinado mês.
- Página de arquivo por dia - exibe todos os artigos de um determinado dia.
- Ação de comentários - controla o envio de comentários para um artigo.

Em nossa aplicação de enquetes, nós teremos as seguintes views:

- Página de “índice” de enquetes - exibe as enquetes mais recente.
- Question “detail” page – displays a question text, with no results but with a form to vote.



- Página de “resultados” de perguntas - exibe os resultados de uma pergunta em particular.
- Ação de voto - gerencia a votação para uma escolha particular em uma enquete em particular.

In Django, web pages and other content are delivered by views. Each view is represented by a Python function (or method, in the case of class-based views). Django will choose a view by examining the URL that’s requested (to be precise, the part of the URL after the domain name).

Now in your time on the web you may have come across such beauties as `ME2/Sites/dirmod.htm?sid=&type=gen&mod=Core+Pages&id=A6CD4967199A42D9B65B1B`. You will be pleased to know that Django allows us much more elegant *URL patterns* than that.

A URL pattern is the general form of a URL - for example: `/newsarchive/<year>/<month>/`.

To get from a URL to a view, Django uses what are known as ‘URLconfs’. A URLconf maps URL patterns to views.

This tutorial provides basic instruction in the use of URLconfs, and you can refer to [Despachante de URL](#) for more information.

## Escrevendo mais views

Agora vamos adicionar mais algumas views em `polls/views.py`. Estas views são um pouco diferentes, porque elas recebem um argumento:

`polls/views.py`

```
def detail(request, question_id):
    return HttpResponse("You're looking at question %s." % question_id)

def results(request, question_id):
    response = "You're looking at the results of question %s."
    return HttpResponse(response % question_id)

def vote(request, question_id):
    return HttpResponse("You're voting on question %s." % question_id)
```

Wire these new views into the `polls.urls` module by adding the following `path()` calls:

`polls/urls.py`

```
from django.urls import path

from . import views

urlpatterns = [
```

```

# ex: /polls/

path('', views.index, name='index'),

# ex: /polls/5/

path('<int:question_id>', views.detail, name='detail'),

# ex: /polls/5/results/

path('<int:question_id>/results/', views.results, name='results'),

# ex: /polls/5/vote/

path('<int:question_id>/vote/', views.vote, name='vote'),

]

```

Dê uma olhada no seu navegador, em `/polls/34/`. Ele vai executar o método `detail()` e mostrar o ID que você informou na URL. Tente `/polls/34/results/` e `/polls/34/vote/` também – Eles vão mostrar a pagina resultados e a pagina de votação.

Quando alguém requisita uma página do seu site Web – vamos dizer `/polls/34/`, o Django irá carregar o módulo Python `mysite.urls` para o qual ele aponta devido a configuração em `ROOT_URLCONF`. Ele encontra a variável nominada `urlpatterns` e atravessa as descrições na mesma ordem. Depois de encontrar a combinação `'polls/'`, ele reparte o texto encontrado (`"polls/"`) e envia o restante – `"34/"` – para o URLconf `'polls.urls'` para processamento posterior. Lá ele encontra `'<int:question_id>/'`, resultando em uma chamada para a view `detail()` como abaixo:

```
detail(request=<HttpRequest object>, question_id=34)
```

A parte `question_id=34` vem de `<int:question_id>`. Usando símbolos de “menor” e “maior” ele “captura” parte da URL e envia como um argumento de palavra-chave para a função view. A parte `:question_id` da string define o nome que será usado para identificar a descrição a ser encontrada, e a parte `<int:` é um conversor que determina qual descrição deve ser usada para encontrar essa parte do caminho da URL.

There’s no need to add URL cruft such as `.html` – unless you want to, in which case you can do something like this:

```
path('polls/latest.html', views.index),
```

Mas, não o faça isso, Isto é idiota.

## Escreva views que façam algo

Cada view é responsável por fazer uma das duas coisas: devolver um objeto `HttpResponse` contendo o conteúdo para a página requisitada, ou levantar uma exceção como `Http404`. O resto é com você.

Sua view pode ler registros do banco de dados, ou não. Ela pode usar um sistema de templates como o do Django - ou outro sistema de templates Python de terceiros - ou não. Ele pode gerar um arquivo PDF, saída em um XML, criar um arquivo ZIP sob demanda, qualquer coisa que você quiser, usando qualquer biblioteca Python você quiser.

Tudo que o Django espera é que a view devolva um `HttpResponse`. Ou uma exceção.

Porque é conveniente, vamos usar a própria API de banco de dados do Django, a qual cobrimos em [Tutorial 2](#). Aqui uma nova tentativa de view `index()`, a qual mostra as últimas 5 “poll questions” do sistema, separada por vírgulas, de acordo com sua data de publicação:

```
polls/views.py
```

```

from django.http import HttpResponseRedirect

from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([q.question_text for q in latest_question_list])
    return HttpResponseRedirect(output)

# Leave the rest of the views (detail, results, vote) unchanged

```

Há um problema aqui, no entanto: o design da página está codificado na view. Se você quiser mudar a forma de apresentação de sua página, você terá de editar este código diretamente em Python. Então vamos usar o sistema de templates do Django para separar o design do código Python:

Primeiro, crie um diretório chamado ``templates`` em seu diretório **polls**. O Django irá procurar templates lá.

A sua configuração de projeto **TEMPLATES** descreve como o Django vai carregar e renderizar templates. O arquivo de configuração padrão usa o backend **DjangoTemplates** do qual a opção **APP\_DIRS** é configurada como **True**. Por convenção **DjangoTemplates** procura por um subdiretório "templates" em cada uma das **INSTALLED\_APPS**.

Within the **templates** directory you have just created, create another directory called **polls**, and within that create a file called **index.html**. In other words, your template should be at **polls/templates/polls/index.html**. Because of how the **app\_directories** template loader works as described above, you can refer to this template within Django as **polls/index.html**.

### Namespacing de template

Now we *might* be able to get away with putting our templates directly in **polls/templates** (rather than creating another **polls** subdirectory), but it would actually be a bad idea. Django will choose the first template it finds whose name matches, and if you had a template with the same name in a *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the best way to ensure this is by *namespacing* them. That is, by putting those templates inside *another* directory named for the application itself.

Ponha o seguinte código neste template:

polls/templates/polls/index.html

```

{% if latest_question_list %}
    <ul>
        {% for question in latest_question_list %}
            <li><a href="/polls/{{ question.id }}">{{ question.question_text }}</a></li>
        {% endfor %}
    </ul>
{% endif %}

```

```

    </ul>

{% else %}

    <p>No polls are available.</p>

{% endif %}

```

#### Nota

To make the tutorial shorter, all template examples use incomplete HTML. In your own projects you should use [complete HTML documents](#).

Agora vamos atualizar nossa view **index** em **polls/views.py** para usar o template:

polls/views.py

```

from django.http import HttpResponse

from django.template import loader

from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    template = loader.get_template('polls/index.html')
    context = {
        'latest_question_list': latest_question_list,
    }
    return HttpResponse(template.render(context, request))

```

Esse código carrega o template chamado **polls/index.html** e passa um contexto para ele. O contexto é um dicionário mapeando nomes de variáveis para objetos Python.

Carregue a página apontando seu navegador para `/polls/`, e você deve ver uma lista contendo a questão “What’s up” do [Tutorial 2](#). O link aponta para página de detalhes das perguntas.

## Um atalho: `render()`

É um estilo muito comum carregar um template, preenchê-lo com um contexto e retornar um objeto **HttpResponse** com o resultado do template renderizado. O Django fornece este atalho. Aqui esta toda a view **index()** reescrita:

polls/views.py

```

from django.shortcuts import render

```

```

from .models import Question

def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    context = {'latest_question_list': latest_question_list}

    return render(request, 'polls/index.html', context)

```

Note que uma vez que você tiver feito isto em todas as views, nós não vamos mais precisar importar **loader** e **HttpResponse** (você vai querer manter **HttpResponse** se você ainda tiver os métodos criados para **detail**, **results**, e **vote**).

A função **render()** recebe o nome do template como primeiro argumento e um dicionário opcional como segundo argumento. Ele retorna um objeto **HttpResponse** do template informado renderizado com o contexto determinado.

## Levantando um erro 404

Agora, vamos abordar a view de detalhe de pergunta – a página que mostra as questões para uma enquete lançada. Aqui está a view:

polls/views.py

```

from django.http import Http404
from django.shortcuts import render

from .models import Question
# ...

def detail(request, question_id):
    try:
        question = Question.objects.get(pk=question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
    return render(request, 'polls/detail.html', {'question': question})

```

Um novo conceito aqui: A view levanta uma exceção **Http404** se a enquete com ID requisitado não existir.

Nós iremos discutir o que você poderia colocar no template **polls/detail.html** mais tarde, mas se você gostaria de ter o exemplo acima funcionando rapidamente, um arquivo contendo:

polls/templates/polls/detail.html

```
{{ question }}
```

irá ajudar a começar por enquanto.

## Um atalho: `get_object_or_404()`

É um estilo muito comum usar `get()` e levantar uma exceção **Http404** se o objeto não existir. O Django fornece um atalho para isso. Aqui está a view `detail()`, reescrita:

`polls/views.py`

```
from django.shortcuts import get_object_or_404, render

from .models import Question

# ...

def detail(request, question_id):
    question = get_object_or_404(Question, pk=question_id)

    return render(request, 'polls/detail.html', {'question': question})
```

A função `get_object_or_404()` recebe um modelo do Django como primeiro argumento e uma quantidade arbitrária de argumentos nomeados, que ele passa para a função do módulo `get()`. E levanta uma exceção **Http404** se o objeto não existir.

### Filosofia

Porquê usamos uma função auxiliar `get_object_or_404()` ao invés de automaticamente capturar as exceções **ObjectDoesNotExist** em alto nível ou fazer a API do modelo levantar **Http404** ao invés de **ObjectDoesNotExist**?

Porque isso seria acoplar a camada de modelo com a camada de visão. Um dos principais objetivo do design do Django é manter o baixo acoplamento. Alguns acoplamento controlado é introduzido no módulo `django.shortcuts`.

Existe também a função `get_list_or_404()`, que trabalha da mesma forma que `get_object_or_404()` – com a diferença de que ela usa `filter()` ao invés de `get()`. Ela levanta **Http404** se a lista estiver vazia.

## Use o sistema de template

De volta para a view `detail()` da nossa aplicação de enquetes. Dada a variável de contexto `question`, aqui está como o template `polls/detail.htm` deve ser:

`polls/templates/polls/detail.html`

```
<h1>{{ question.question_text }}</h1>

<ul>

{% for choice in question.choice_set.all %}

    <li>{{ choice.choice_text }}</li>

{% endfor %}
```

```
</ul>
```

O sistema de templates usa uma sintaxe separada por pontos para acessar os atributos da variável. No exemplo de `{{ question.question_text }}`, primeiro o Django procura por dicionário no objeto **question**. Se isto falhar, ele tenta procurar por um atributo – que funciona, neste caso. Se a procura do atributo também falhar, ele irá tentar uma chamada do tipo list-index.

A chamada do método acontece no laço `{% for %}`: **poll.choice\_set.all** é interpretado como código Python **poll.choice\_set.all()**, que retorna objetos **Choice** iteráveis que são suportado para ser usado na tag `{% for %}`.

Veja o [guia de templates](#) para maiores detalhes sobre templates.

## Removendo URLs codificados nos templates

Lembre-se, quando escrevemos o link para uma pergunta no template **polls/templates/index.html**, o link foi parcialmente codificado como este:

```
<li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
```

O problema com com essa abordagem acoplada, hardcoded é que torna muito difícil alterar URLs em projetos com muitos templates. Todavia, se definimos o argumento name nas funções **path()** no módulo **polls.urls**, você pode remover a dependência de um caminho de URL específico nas configurações de url usando a template tag `{% url %}`:

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

A maneira como isso funciona é, observando as definições de URL como especificado no módulo **polls.urls**. Você pode ver exatamente onde o nome de URL 'detalhe' é definido a seguir

```
...
# the 'name' value as called by the {% url %} template tag
path('<int:question_id>', views.detail, name='detail'),
...
```

Se você quiser alterar a URL das views de detalhe da enquete para outra coisa, talvez para algo como **polls/specifics/12/** em vez de fazê-lo no template (ou templates) você mudaria em **polls/urls.py**:

```
...
# added the word 'specifics'
path('specifics/<int:question_id>', views.detail, name='detail'),
...
```

## Namespacing nomes de URL

Este projeto tutorial tem apenas um aplicação, **polls**. Em projetos reais Django, pode haver cinco, dez, vinte ou mais aplicações. Como o Django diferencia os nomes de URL entre eles? Por exemplo, a aplicação **polls** tem uma view **detail**, e assim pode um aplicativo no mesmo projeto que é para um blog. Como é que faz para que o Django saiba qual view da aplicação será criada para a url ao usar a tag de template `'{% url %}'`?

A resposta é adicionar namespaces a seu URLconf. No arquivo `polls/urls.py`, continue e adicione um `app_name` para configurar o namespace da aplicação.

`polls/urls.py`

```
from django.urls import path

from . import views

app_name = 'polls'

urlpatterns = [
    path('', views.index, name='index'),
    path('<int:question_id>/', views.detail, name='detail'),
    path('<int:question_id>/results/', views.results, name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Agora mudo seu template ``polls/index.html`` de:

`polls/templates/polls/index.html`

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

para apontar para a view de detalhes d namespace:

`polls/templates/polls/index.html`

```
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```

When you're comfortable with writing views, read [part 4 of this tutorial](#) to learn the basics about form processing and generic views.

## Escrevendo sua primeira aplicação Django, parte 4

This tutorial begins where [Tutorial 3](#) left off. We're continuing the Web-poll application and will focus on form processing and cutting down our code.

### Onde obter ajuda:

Se tiver problemas enquanto caminha por este tutorial, por favor consulte a seção [Obtendo ajuda](#) da FAQ.

## Write a minimal form

Vamos atualizar nosso template de detalhamento da enquete ("polls/detail.html") do último tutorial, para que ele contenha um elemento HTML `<form>`:



polls/templates/polls/detail.html

```
<h1>{{ question.question_text }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}

{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{
choice.id }}">
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label><br>
{% endfor %}

<input type="submit" value="Vote">
</form>
```

Uma rápida explicação:

- O template acima exibe um botão radio para cada opção da enquete. O **value** de cada botão radio está associado ao ID da opção. O **name** de cada botão radio é a escolha "**choice**". Isso significa que, quando alguém escolhe um dos botões de radio e submete a formulário, ele vai enviar o dado ``choice=#`` por POST onde # é o ID da escolha selecionada. Este é o conceito básico sobre formulários HTML.
- We set the form's **action** to `{% url 'polls:vote' question.id %}`, and we set **method="post"**. Using **method="post"** (as opposed to **method="get"**) is very important, because the act of submitting this form will alter data server-side. Whenever you create a form that alters data server-side, use **method="post"**. This tip isn't specific to Django; it's good Web development practice in general.
- **forloop.counter** indica quantas vezes a tag `:ttag`for`` passou pelo laço.
- Since we're creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgeries. Thankfully, you don't have to worry too hard, because Django comes with a helpful system for protecting against it. In short, all POST forms that are targeted at internal URLs should use the `{% csrf_token %}` template tag.

Agora, vamos criar uma **view** Django que manipula os dados submetidos e faz algo com eles. Lembre-se, no Tutorial 3, criamos uma URLconf para a aplicação de enquete que inclui esta linha:

polls/urls.py

```
path('<int:question_id>/vote/', views.vote, name='vote'),
```

Nós também criamos uma implementação falsa da função **vote()**. Vamos criar a versão real. Adicione o seguinte em **polls/views.py**:

polls/views.py

```
from django.http import HttpResponseRedirect
```

```

from django.shortcuts import get_object_or_404, render
from django.urls import reverse

from .models import Choice, Question
# ...

def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)

    try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()

        # Always return an HttpResponseRedirect after successfully dealing
        # with POST data. This prevents data from being posted twice if a
        # user hits the Back button.

        return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))

```

Este código inclui algumas coisas que ainda não foram cobertas neste tutorial:

- **request.POST** é um objeto como dicionários que lhe permite acessar os dados submetidos pelos seus nomes chaves. Neste caso, **request.POST['choice']** retorna o ID da opção selecionada, como uma string. Os valores de **request.POST** são sempre strings.

Note que Django também fornece **request.GET** para acesar dados GET da mesma forma – mas nós estamos usando `attr=request.POST <django.http.HttpRequest.POST>` explicitamente no nosso código, para garantir que os dados só podem ser alterados por meio de uma chamada POST.

- **request.POST['choice']** irá levantar a exceção **KeyError** caso uma **choice** não seja fornecida via dados POST. O código acima checa por **KeyError** e re-exibe o formulário da enquete com as mensagens de erro se uma **choice** não for fornecida.

- Após incrementar uma opção, o código retorna um `class:~django.http.HttpResponseRedirect` em vez de um normal `HttpResponse`. `HttpResponseRedirect` recebe um único argumento: a URL para o qual o usuário será redirecionado (veja o ponto seguinte para saber como construímos a URL, neste caso).

As the Python comment above points out, you should always return an `HttpResponseRedirect` after successfully dealing with POST data. This tip isn't specific to Django; it's good Web development practice in general.

- Estamos usando a função `reverse()` no construtor da `HttpResponseRedirect` neste exemplo. Essa função nos ajuda a evitar de colocar a URL dentro da view de maneira literal. A ele é dado então o nome da "view" que queremos que ele passe o controle e a parte variável do padrão de formato da URL que aponta para a "view". Neste caso, usando o `URLconf` nós definimos em [Tutorial 3](#), esta chamada de `reverse()` irá retornar uma string como

```
• '/polls/3/results/'
```

onde o **3** é o valor para `question.id`. Esta URL redirecionada irá então chamar a view `'results'` para exibir a página final.

Como mencionado no [Tutorial 3](#), `request` é um objeto `HttpRequest` object. Para mais informações sobre objetos `HttpRequest`, veja [request and response documentation](#).

Depois que alguém votar em uma enquete, a view `vote()` redireciona para a página de resultados da enquete. Vamos escrever essa view:

`polls/views.py`

```
from django.shortcuts import get_object_or_404, render

def results(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    return render(request, 'polls/results.html', {'question': question})
```

Isto é quase exatamente o mesmo que a view `detail()` do [Tutorial 3](#). A única diferença é o nome do template. Iremos corrigir esta redundância depois.

Agora, crie o template `polls/results.html`:

`polls/templates/polls/results.html`

```
<h1>{{ question.question_text }}</h1>

<ul>

{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
{% endfor %}
```

```
</ul>

<a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```

Agora, vá para `/polls/1/` no seu navegador e vote em uma enquete. Você deverá ver uma página de resultados que será atualizado cada vez que você votar. Se você enviar o formulário sem ter escolhido uma opção, você deverá ver a mensagem de erro.

#### Nota

O código da nossa view `vote()` tem um pequeno problema. Ele primeiro pega o objeto de `selected_choice` do banco de dados, calcula o novo valor de `votes` e os salva novamente. Se dois usuários do seu site tentarem votar *ao mesmo tempo*: o mesmo valor, digamos 42, será obtido para `votes`. Então, para ambos usuários, o novo valor 43 é calculado e salvo, quando o valor esperado seria 44.

Isto é chamado de *condição de concorrência*. Se você estiver interessado, pode ler [Avoiding race conditions using F\(\)](#) para aprender como resolver este problema.

## Use views genéricas: Menos código é melhor

The `detail()` (from [Tutorial 3](#)) and `results()` views are very short – and, as mentioned above, redundant. The `index()` view, which displays a list of polls, is similar.

Estas views representam um caso comum do desenvolvimento Web básico: obter dados do banco de dados de acordo com um parâmetro passado na URL, carregar um template e devolvê-lo renderizado. Por isto ser muito comum, o Django fornece um atalho, chamado sistema de “views genéricas”.

Views genéricas abstraem padrões comuns para um ponto onde você nem precisa escrever código Python para escrever uma aplicação.

Let’s convert our poll app to use the generic views system, so we can delete a bunch of our own code. We’ll have to take a few steps to make the conversion. We will:

1. Converta o URLconf.
2. Delete algumas das views antigas, desnecessárias.
3. Introduz novas views baseadas em views genéricas Django’s.

Leia para obter mais detalhes.

#### Por que o código ficou embaralhado?

Geralmente, quando estiver escrevendo uma aplicação Django, você vai avaliar se views genéricas são uma escolha adequada para o seu problema e você irá utilizá-las desde o início em vez de refatorar seu código no meio do caminho. Mas este tutorial intencionalmente tem focado em escrever views “do jeito mais difícil” até agora, para concentrarmos nos conceitos fundamentais.

Você deve saber matemática básica antes de você começar a usar uma calculadora.

## Corrija URLconf

Em primeiro lugar, abra a URLconf `polls/urls.py` e modifique para ficar assim:

```
polls/urls.py
```

```

from django.urls import path

from . import views

app_name = 'polls'

urlpatterns = [
    path('', views.IndexView.as_view(), name='index'),
    path('<int:pk>/', views.DetailView.as_view(), name='detail'),
    path('<int:pk>/results/', views.ResultsView.as_view(), name='results'),
    path('<int:question_id>/vote/', views.vote, name='vote'),
]

```

Note que o nome da descrição encontrada nas strings de caminhos da segundo e da terceira descrição foi alterado de `<question_id>` para `<pk>`.

## Views alteradas

Em seguida, vamos remover a nossas velhas views `index`, `detail`, e `results` e usar views genéricas do Django em seu lugar. Para fazer isso, abra o arquivo `polls/views.py` e alterar para:

`polls/views.py`

```

from django.http import HttpResponseRedirect

from django.shortcuts import get_object_or_404, render

from django.urls import reverse

from django.views import generic

from .models import Choice, Question


class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

    def get_queryset(self):
        """Return the last five published questions."""

```

```

        return Question.objects.order_by('-pub_date')[:5]

class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'

class ResultsView(generic.DetailView):
    model = Question
    template_name = 'polls/results.html'

def vote(request, question_id):
    ... # same as above, no changes needed.

```

Nós estamos usando duas views genéricas aqui: **ListView** e **DetailView**. Respectivamente, essas duas **views** abstraem o conceito de exibir uma lista de objetos e exibir uma página de detalhe para um tipo particular de objeto.

- Cada “view” genérica precisa saber qual é o modelo que ela vai agir. Isto é fornecido usando o atributo **model**.
- A view genérica **DetailView** espera o valor de chave primaria capturado da URL chamada “pk”, então mudamos **question\_id** para **pk** para as views genérica.

Por padrão, a “view” genérica **DetailView** utiliza um template chamado `<app name>/<model name>_detail.html`. Em nosso caso, ela vai utilizar o template `polls/question_detail.html`. O atributo **template\_name** é usado para indicar ao Django para usar um nome de template em vez de auto gerar uma nome de template. Nós também especificamos **template\_name** para a view de listagem de **results** – isto garante que a view de detalhe tem uma aparência quando renderizada, mesmo que sejam tanto um `class ~django.views.generic.detail.DetailView` por trás das cenas.

Semelhantemente, a view genérica **ListView** utiliza um template chamado `<app name>/<model name>_list.html`; usamos **template\_name** para informar **ListView** para usar nossos templates `polls/index.html`.

Nas partes anteriores deste tutorial, os templates tem sido fornecidos com um contexto que contém as variáveis **question** e **latest\_question\_list**. Para a **DetailView** a variável **question** é fornecida automaticamente – já que estamos usando um modelo Django (**Question**), Django é capaz de determinar um nome apropriado para a variável de contexto. Contudo, para **ListView**, a variável de contexto gerada automaticamente é **question\_list**. Para sobrescrever nós fornecemos o atributo **context\_object\_name**, especificando que queremos usar **latest\_question\_list** no lugar. Como uma abordagem alternativa, você poderia mudar seus templates para casar o novo padrão das variáveis de contexto – mas é muito fácil dizer para o Django usar a variável que você quer.

Execute o servidor, e use sua nova aplicação de enquete baseada em generic views.

Para detalhes completos sobre views genéricas, veja a [documentação de generic views documentation](#).

Quando você estiver confortável com formulários e generic views, leia a [parte 5 deste tutorial](#) para aprender sobre como testar nossa aplicação de enquete.

## Escrevendo sua primeira aplicação Django, parte 5

Este tutorial começa onde o [Tutorial 4](#) parou. Nós construímos uma aplicação web de enquete, e agora nós vamos criar alguns testes automatizados para ela.

### Onde obter ajuda:

Se tiver problemas enquanto caminha por este tutorial, por favor consulte a seção [Obtendo ajuda](#) da FAQ.

## Apresentando testes automatizados

### O que são testes automatizados?

Tests are routines that check the operation of your code.

Testes funcionam em diferentes níveis. Alguns testes podem ser aplicados a um pequeno detalhe (*um determinado método de um modelo retorna o valor esperado?*) enquanto outros examinam o funcionamento global do software (*a sequência de entradas do usuário no site produz o resultado desejado?*). Isso não é diferente do teste que você fez anteriormente no [Tutorial 2](#), usando o **shell** para examinar o comportamento de um método, ou executar a aplicação e entrar com um dado para verificar como ele se comporta.

A diferença em testes *automatizados* é que o trabalho de testar é feito para você pelo sistema. Você cria um conjunto de testes uma vez, e então conforme faz mudanças em sua aplicação, você pode checar se o seu código continua funcionando como você originalmente pretendia, sem ter que gastar tempo executando o teste manualmente.

### Porque você precisa criar testes

Então, por que criar testes, e por que agora?

Você pode achar que já tem coisa suficiente no seu prato apenas aprendendo Python/Django, e tendo ainda outra coisa para aprender e fazer pode parecer exagerado e talvez desnecessário. Afinal, nossa aplicação de enquetes está funcionando muito bem agora; passar pelo trabalho de criar testes automatizados não vai fazê-la funcionar melhor. Se criar a aplicação de enquetes é a última coisa que você vai programar em Django, então realmente, você não precisa saber sobre como criar testes automatizados. Mas, se esse não é o caso, agora é uma excelente hora para aprender.

#### *Teste vão salvar seu tempo*

Até um certo ponto, 'verificar que parece funcionar' será um teste satisfatório. Em uma aplicação mais sofisticada, você pode ter dúzias de interações complexas entre componentes.

A change in any of those components could have unexpected consequences on the application's behavior. Checking that it still 'seems to work' could mean running through your code's functionality with twenty different variations of your test data to make sure you haven't broken something - not a good use of your time.

Isso é verdade especialmente quando testes automatizados poderiam fazer isso pra você em segundos. Se alguma coisa errada acontecer, os testes também ajudarão em identificar o código que está causando o comportamento inesperado.

Às vezes pode parecer uma tarefa para afastar você da sua produtividade e criatividade em programar para enfrentar o trabalho nada excitante e glamouroso de escrever testes, particularmente quando você sabe que seu código está funcionando corretamente.

No entanto, a tarefa de escrever testes é muito mais satisfatório do que gastar horas testando sua aplicação manualmente, ou tentando identificar a causa de um problema recém-introduzido.

*Testes não só identificam problemas, mas também os previnem*

É um erro pensar nos testes simplesmente como um aspecto negativo do desenvolvimento.

Sem testes, o objetivo ou comportamento desejado de uma aplicação pode não ser tão claro. Mesmo quando é seu próprio código, algumas vezes você vai se encontrar bisbilhotando nele tentando descobrir exatamente o que está fazendo.

Testes mudam isso; eles mostram seu código por dentro, e quando algumas coisas saem errado, eles revelam parte do erro.

*Testes fazem seu código mais atraente.*

You might have created a brilliant piece of software, but you will find that many other developers will refuse to look at it because it lacks tests; without tests, they won't trust it. Jacob Kaplan-Moss, one of Django's original developers, says "Code without tests is broken by design."

Assim, esses outros desenvolvedores procuram ver os testes do seu software antes de levar a sério, e isso é uma outra razão para você iniciar a escrita dos testes.

*Testes ajudam as equipes a trabalharem juntos*

Os pontos anteriores foram escritos do ponto de vista de um único desenvolvedor mantendo uma aplicação. Aplicações complexas serão mantidas por times. Testes garantem que seus colegas de trabalho não quebrem acidentalmente o seu código (e que você não quebre o deles sem saber). Se você deseja trabalhar como um programador Django, deve ser bom escrevendo testes!

## Estratégias básicas de testes

Há várias maneiras de abordar a escrita de testes.

Some programmers follow a discipline called "[test-driven development](#)"; they actually write their tests before they write their code. This might seem counter-intuitive, but in fact it's similar to what most people will often do anyway: they describe a problem, then create some code to solve it. Test-driven development formalizes the problem in a Python test case.

Comumente, um novato em testes irá criar algum código e depois decidir que ele deveria ser testado. Talvez seria melhor escrever alguns testes antes, mas nunca é tarde para começar.

Às vezes é difícil saber por onde devemos começar a escrever testes. Se você tiver escrito milhares de linhas em Python, escolher algo para testar pode não ser fácil. Nesse caso, é interessante escrever seu primeiro teste na próxima vez que você fizer alguma alteração, seja a adição de uma nova função ou correção de um bug.

Então vamos fazer isso a partir de agora.

## Escrevendo nosso primeiro teste

### Nós identificamos um bug



Felizmente, há um pequeno bug na aplicação **polls** para corrigirmos imediatamente: O método **Question.was\_published\_recently()** retorna **True** se a **Question** foi publicada dentro do último dia (o que está correto), mas também se o campo **pub\_date** de **Question** está no futuro (o que certamente não é o caso).

Confirme o bug utilizando **shell** para verificar o método numa questão em que a data encontra-se no futuro:

```
$ python manage.py shell

>>> import datetime

>>> from django.utils import timezone

>>> from polls.models import Question

>>> # create a Question instance with pub_date 30 days in the future

>>> future_question = Question(pub_date=timezone.now() + datetime.timedelta(days=30))

>>> # was it published recently?

>>> future_question.was_published_recently()

True
```

Desde que coisas no futuro não são 'recent', isto é claramente um erro.

## Criar um teste para expor um bug

O que nós fizemos no **shell** para testar o problema é exatamente o que podemos fazer em um teste automatizado, então vamos voltar para o teste automatizado

Um lugar convencional para os testes da aplicação é o arquivo **tests.py**; o sistema de testes irá encontrar automaticamente todos os testes que estiverem em qualquer arquivo cujo o nome comece com **test**.

Coloque o seguinte código no arquivo **tests.py** na aplicação **polls**:

polls/tests.py

```
import datetime

from django.test import TestCase
from django.utils import timezone

from .models import Question

class QuestionModelTests(TestCase):
```

```
def test_was_published_recently_with_future_question(self):
    """
    was_published_recently() returns False for questions whose pub_date
    is in the future.
    """
    time = timezone.now() + datetime.timedelta(days=30)
    future_question = Question(pub_date=time)
    self.assertIs(future_question.was_published_recently(), False)
```

Aqui criamos uma subclasse de `django.test.TestCase` com um método que cria uma instância de `Question` com uma `pub_date` no futuro. Então verificamos a saída de `was_published_recently()` - o qual *deve* ser `False`.

## Executando o teste

No terminal, podemos rodar nosso teste:

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

E você verá algo como:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
=====
FAIL: test_was_published_recently_with_future_question
(polls.tests.QuestionModelTests)
-----
Traceback (most recent call last):
  File "/path/to/mysite/polls/tests.py", line 16, in
test_was_published_recently_with_future_question
    self.assertIs(future_question.was_published_recently(), False)
AssertionError: True is not False
-----
Ran 1 test in 0.001s
```

```
FAILED (failures=1)

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

### Different error?

Se no lugar disso, você recebeu uma exceção **NameError**, você pode ter pulado algo passo da **Parte 2** onde nós adicionamos "imports" de `datetime` e `timezone` em `polls/models.py`. Copie os "imports" dessa seção, e tente executar seus testes novamente.

O que ocorreu foi o seguinte:

- `manage.py test polls` looked for tests in the `polls` application
- ele encontrou uma subclass da classe `django.test.TestCase`
- ele cria um banco de dados especial com o propósito de teste
- ele procurou por métodos de test - aquele cujo nome começam com `test`
- em `test_was_published_recently_with_future_question` é criado uma instância de `Question` na qual o campo `pub_date` está 30 dias no futuro
- ... e usando o método `assertIs()`, descobrimos que `was_published_recently()` retorna `True`, mas queremos que retorne `False`

O teste nos informa que teste falhou e até mesmo a linha na qual a falha ocorreu.

### Corrigindo o erro

Nós sabemos onde o problema está. "Question.was\_published\_recently()" deveria retornar "False" se "pub\_date" está no futuro. Corrija o método no "models.py", dessa forma ele só retornará "True" se a data também estiver no passado:

`polls/models.py`

```
def was_published_recently(self):
    now = timezone.now()
    return now - datetime.timedelta(days=1) <= self.pub_date <= now
```

execute e teste novamente

```
Creating test database for alias 'default'...

System check identified no issues (0 silenced).

.

-----

Ran 1 test in 0.001s

OK

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

Depois de identificar um erro, nós escrevemos um teste que demonstra e corrige o erro no código, então o nosso teste passa.

Many other things might go wrong with our application in the future, but we can be sure that we won't inadvertently reintroduce this bug, because running the test will warn us immediately. We can consider this little portion of the application pinned down safely forever.

## Testes mais abrangentes

Enquanto estamos aqui, nós podemos nos aprofundar no método `was_published_recently()`; De fato, seria certamente embaraçoso introduzir um novo bug enquanto arruma outro.

Adicione mais dois métodos de teste na mesma classe, para testá-la melhor.

`polls/tests.py`

```
def test_was_published_recently_with_old_question(self):
    """
    was_published_recently() returns False for questions whose pub_date
    is older than 1 day.
    """
    time = timezone.now() - datetime.timedelta(days=1, seconds=1)
    old_question = Question(pub_date=time)
    self.assertIs(old_question.was_published_recently(), False)

def test_was_published_recently_with_recent_question(self):
    """
    was_published_recently() returns True for questions whose pub_date
    is within the last day.
    """
    time = timezone.now() - datetime.timedelta(hours=23, minutes=59, seconds=59)
    recent_question = Question(pub_date=time)
    self.assertIs(recent_question.was_published_recently(), True)
```

E agora nós temos 3 testes que confirmam que `Question.was_published_recently()` retorna valores sensíveis de perguntas do passado, recente e futuro.

Again, `polls` is a minimal application, but however complex it grows in the future and whatever other code it interacts with, we now have some guarantee that the method we have written tests for will behave in expected ways.

## Teste a View

A aplicação de enquete é bastante indiscriminatória: irá publicar qualquer questão, incluindo aquelas que o campo **pub\_date** está no futuro. Devemos melhorar isso. Definindo um **pub\_date** no futuro deveria fazer com que a Question seja publicada naquele momento, mas que esteja invisível até lá.

## Um teste para uma view

When we fixed the bug above, we wrote the test first and then the code to fix it. In fact that was an example of test-driven development, but it doesn't really matter in which order we do the work.

Em nosso primeiro teste, focamos no comportamento interno do código. Para este teste, nós queremos checar seu comportamento como seria experienciado por um usuário através de um navegador web.

Antes de tentarmos consertar alguma coisa, vamos dar uma olhada nas ferramentas à nossa disposição.

## O cliente de testes do Django

o Django fornece um test **Client** para simular um usuário interagindo com o código no nível de view. Podemos usá-lo em **tests.py** ou mesmo no **shell**.

We will start again with the **shell**, where we need to do a couple of things that won't be necessary in **tests.py**. The first is to set up the test environment in the **shell**:

```
$ python manage.py shell

>>> from django.test.utils import setup_test_environment

>>> setup_test_environment()
```

O **setup\_test\_environment()** instala um renderizador de templates o qual permite que sejam examinados alguns atributos adicionais nas respostas http tal como um **response.context** que de outra maneira não estariam disponíveis. Note que este método *não* configura um banco de dados de teste, então o que se segue será executado no banco de dados existente e a saída pode ser diferente quanto as questões que você já criou. Você talvez tenha resultados inesperados se seu **TIME\_ZONE** no **settings.py** não estiver correto. Se você não lembrou de defini-lo antes, verifique-o antes de continuar.

A diante precisamos importar a classe cliente de teste (depois iremos usar a classe **django.test.TestCase** no **tests.py**, a qual traz seu próprio cliente, então isso não será necessário):

```
>>> from django.test import Client

>>> # create an instance of the client for our use

>>> client = Client()
```

Com isso pronto, podemos pedir que o cliente faça algum trabalho para a gente:

```
>>> # get a response from '/'

>>> response = client.get('/')

Not Found: /

>>> # we should expect a 404 from that address; if you instead see an

>>> # "Invalid HTTP_HOST header" error and a 400 response, you probably
```

```

>>> # omitted the setup_test_environment() call described earlier.

>>> response.status_code

404

>>> # on the other hand we should expect to find something at '/polls/'

>>> # we'll use 'reverse()' rather than a hardcoded URL

>>> from django.urls import reverse

>>> response = client.get(reverse('polls:index'))

>>> response.status_code

200

>>> response.content

b'\n    <ul>\n        \n            <li><a href="/polls/1/">What&#x27;s up?</a></li>\n    \n</ul>\n\n'

>>> response.context['latest_question_list']

<QuerySet [<Question: What's up?>]>

```

## Melhorando a nossa view

A lista de enquete mostra enquetes que não estão publicadas ainda (isto é aqueles que tem um `__pub_date` no futuro). Vamos concertar isso.

No **Tutorial 4** nós introduzimos o class-based views, baseado no **ListView**:

polls/views.py

```

class IndexView(generic.ListView):

    template_name = 'polls/index.html'

    context_object_name = 'latest_question_list'

    def get_queryset(self):

        """Return the last five published questions."""

        return Question.objects.order_by('-pub_date')[:5]

```

Precisamos fazer uma adição no método `get_queryset()` e alterá-lo de modo que ele verifique também a data comparando com `timezone.now()`. Antes precisamos adicionar uma importação:

polls/views.py

```

from django.utils import timezone

```

e então devemos fazer uma adição ao método `get_queryset` como a seguir:

polls/views.py

```
def get_queryset(self):
    """
    Return the last five published questions (not including those set to be
    published in the future).
    """
    return Question.objects.filter(
        pub_date__lte=timezone.now()
    ).order_by('-pub_date')[:5]
```

`Question.objects.filter(pub_date__lte=timezone.now())` retorna um queryset contendo **Questions** cujo o `pub_date` é menor ou igual a - que quer dizer, anterior ou igual a - `timezone.now`.

## Testando nossa nova view

Agora você pode se certificar se isso se comporta como esperado executando **runserver**, carregando o site no seu navegador, criando **Questions** com datas no passado e no futuro, e verificando que somente aqueles que foram publicados são listados. Você não quer ter que fazer isso *toda vez que fizer uma alteração que possa afetar isso* - então vamos também criar um teste, baseado na nossa sessão **shell** acima.

Adicione o seguinte a "polls/tests.py":

polls/tests.py

```
from django.urls import reverse
```

e nós vamos criar uma função de atalho para criar perguntas assim como uma nova classe de teste:

polls/tests.py

```
def create_question(question_text, days):
    """
    Create a question with the given `question_text` and published the
    given number of `days` offset to now (negative for questions published
    in the past, positive for questions that have yet to be published).
    """
    time = timezone.now() + datetime.timedelta(days=days)
    return Question.objects.create(question_text=question_text, pub_date=time)

class QuestionIndexViewTests(TestCase):
```

```

def test_no_questions(self):
    """
    If no questions exist, an appropriate message is displayed.
    """
    response = self.client.get(reverse('polls:index'))
    self.assertEqual(response.status_code, 200)
    self.assertContains(response, "No polls are available.")
    self.assertQuerysetEqual(response.context['latest_question_list'], [])

def test_past_question(self):
    """
    Questions with a pub_date in the past are displayed on the
    index page.
    """
    create_question(question_text="Past question.", days=-30)
    response = self.client.get(reverse('polls:index'))
    self.assertQuerysetEqual(
        response.context['latest_question_list'],
        ['<Question: Past question.>']
    )

def test_future_question(self):
    """
    Questions with a pub_date in the future aren't displayed on
    the index page.
    """
    create_question(question_text="Future question.", days=30)
    response = self.client.get(reverse('polls:index'))
    self.assertContains(response, "No polls are available.")
    self.assertQuerysetEqual(response.context['latest_question_list'], [])

```



```

def test_future_question_and_past_question(self):
    """
    Even if both past and future questions exist, only past questions
    are displayed.
    """
    create_question(question_text="Past question.", days=-30)
    create_question(question_text="Future question.", days=30)
    response = self.client.get(reverse('polls:index'))
    self.assertQuerysetEqual(
        response.context['latest_question_list'],
        ['<Question: Past question.>']
    )

def test_two_past_questions(self):
    """
    The questions index page may display multiple questions.
    """
    create_question(question_text="Past question 1.", days=-30)
    create_question(question_text="Past question 2.", days=-5)
    response = self.client.get(reverse('polls:index'))
    self.assertQuerysetEqual(
        response.context['latest_question_list'],
        ['<Question: Past question 2.>', '<Question: Past question 1.>']
    )

```

Vejamos algumas delas mais de perto.

O primeiro é uma função de atalho para questão, **create\_question**, para retirar algumas repetições do processo de criar questões.

**test\_no\_questions** não cria nenhuma questão, mas verifica a mensagem: “Nenhuma enquete disponível.” e verifica que **latest\_question\_list** está vazia. Note que a classe **django.test.TestCase** fornece alguns métodos adicionais de assertividade. Nestes exemplos nós usamos **assertContains()** e **assertQuerysetEqual()**.

Em **test\_past\_question**, criamos uma questão e verificamos se esta aparece na lista.

Em `test_future_question`, nós criamos uma `pub_date` no futuro. O banco de dados é resetado para cada método de teste, quer dizer que a primeira questão não está mais lá, então novamente o index não deve ter nenhuma questão.

E assim por diante. De fato, estamos usando os testes para contar uma história de entradas no site de administração e da experiência do usuário, e verificando se para cada estado e para cada nova alteração no estado do sistema, o resultado esperado é publicado.

## Testando o `DetailView`

O que temos funciona bem; Porém, mesmo quando questões futuras não aparecem no `* index*`, os usuários ainda podem acessá-las se eles conhecem ou adivinham a URL correta. Então precisamos adicionar uma regra similar ao `DetailView`:

`polls/views.py`

```
class DetailView(generic.DetailView):
    ...

    def get_queryset(self):
        """
        Excludes any questions that aren't published yet.
        """
        return Question.objects.filter(pub_date__lte=timezone.now())
```

E claro, adicionaremos alguns testes, para verificar que uma `Question` a qual seu `pub_date` está no passado pode ser mostrada, e que uma com `pub_date` no futuro não pode:

`polls/tests.py`

```
class QuestionDetailViewTests(TestCase):
    def test_future_question(self):
        """
        The detail view of a question with a pub_date in the future
        returns a 404 not found.
        """
        future_question = create_question(question_text='Future question.', days=5)
        url = reverse('polls:detail', args=(future_question.id,))
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)

    def test_past_question(self):
        """
```

```
The detail view of a question with a pub_date in the past
displays the question's text.
"""

past_question = create_question(question_text='Past Question.', days=-5)
url = reverse('polls:detail', args=(past_question.id,))
response = self.client.get(url)

self.assertContains(response, past_question.question_text)
```

## Ideias para mais testes

Devemos adicionar um método similar ao `get_queryset` ao `ResultsView` e criar uma nova classe de teste para essa “view”. Isso será muito parecido com o que criamos há pouco; defato terá muitas repetições.

Podemos também melhorar nossa aplicação de várias outras maneiras, adicionando testes no caminho. Por exemplo é estranho que **Questions** possam ser publicadas no site sem que tenham **Choices**. Então, nossas “views” podem verificar isso, e excluir tais **Questions**. Nossos testes podem criar uma **Question** sem **Choices** e então testar que não estão publicadas, também criar uma **Question** parecida com **Choices**, e testar se *estão* publicadas.

Talvez usuários registrados no admin devam ser permitidos a ver **Questions** não publicadas, mas não os visitantes comuns. Denovo: independente da necessidade a serem adicionadas ao software para que isso seja realizado, isso deve ser acompanhado por um teste, não importando se você escreve o teste primeiro e então faz o código para passar no teste, ou escreva o código da lógica primeiro e então escreva o teste para fazer a prova.

Em um certo ponto, você precisará olhar seus testes e se perguntar se o seu código está sofrendo de inchaço de testes, o que nos trás a:

## Quando estiver testando, mais é melhor

Pode parecer que nossos testes estão crescendo fora de controle. Nessa taxa teremos logo mais código nos testes do que em nossa aplicação, e a repetição não é muito estética, comparado a concisa elegância do resto do nosso código.

**Isso não importa.** Deixe eles crescerem. Por boa parte do tempo, você pode escrever um teste uma vez e então esquecer dele. Ele vai continuar a executar sua função útil enquanto você continua a escrever seu programa.

Algumas vezes os testes precisarão ser atualizados. Suponha que emendamos nossas “views” para que somente **Questions** com **Choices** são publicadas. Neste caso, muitos dos nossos testes existentes irão falhar - *nos dizendo exatamente quais testes precisam ter ajustes para que sejam atualizados*, assim que os próprios testes auxiliam na manutenção deles mesmos.

Na pior das hipóteses, ao continuar a desenvolver, você talvez encontre alguns tests que são redundantes agora. Mesmo isso não é um problema; em testes redundância é uma boa coisa.

Enquanto seus testes estiverem organizados sensatamente, eles não vão se tornar desorganizados. Boas práticas incluem ter:

- um “TestClass” separado para cada modelo ou view
- um método de teste separado para cada conjunto de condições que você quer testar
- nomes de métodos de teste que descrevem a sua função

## Mais testes

Esse tutorial apenas introduz alguns conceitos básicos de testes. Há muito mais do que você pode fazer, e uma série de ferramentas muito úteis à sua disposição para conseguir algumas coisas muito inteligentes.

Por exemplo, enquanto nossos testes aqui tem coberto algumas lógicas internas de um modelo e a informação publicada nas nossas “views”, você pode usar um framework para navegador como o [Selenium](#) para testar a maneira como seu HTML é processado de verdade pelo navegador. Isso permite a você testar não somente o comportamento do seu código Django, mas também, por exemplo, do seu JavaScript. É bem legal ver seus testes iniciarem um navegador, e começar a interagir com seu site, como se fosse um ser humano guiando! O Django inclui a **LiveServerTestCase** para facilitar a integração com ferramentas como o Selenium.

Se você tem uma aplicação complexa, você pode querer rodar testes automaticamente com cada commit pelo propósito de [integração contínua](#), e o próprio controle de qualidade - ao menos parcialmente - é automatizado.

Uma boa forma de encontrar partes não testadas de sua aplicação é verificar a cobertura de código. Isto também ajuda a identificar códigos frágeis ou mesmo morto. Se você não pode testar um pedaço do código, isso normalmente significa que o código precisa ser refatorado ou removido. Cobertura vai ajudar a identificar código morto. Veja [Integration with coverage.py](#) para mais detalhes.

Testes no django tem informações sobre testes.

## E agora?

Para maiores detalhes sobre testing, veja [Testando no Django](#).

Quando estiver confortável testando “views” Django, leia: [doc:parte 6 deste tutorial](#) para aprender sobre gerenciamento de arquivos estáticos.

## Escrevendo sua primeira aplicação Django, parte 6

Este tutorial começa onde o [Tutorial 5](#) parou. Nós construímos uma aplicação web de enquete testada, e agora nós vamos adicionar uma folha de estilos e uma imagem.

Além do HTML gerado pelo servidor, aplicações web normalmente precisam servir outros arquivos – como imagens, JavaScript, ou CSS – necessário para renderizar a página web completa. No Django, nós chamamos estes arquivos de “arquivos estáticos”.

For small projects, this isn't a big deal, because you can keep the static files somewhere your web server can find it. However, in bigger projects – especially those comprised of multiple apps – dealing with the multiple sets of static files provided by each application starts to get tricky.

É para isto que o **django.contrib.staticfiles** serve: ele coleciona os arquivos estáticos de cada uma de suas aplicações (e qualquer outro lugar que você especifique) em um único local que pode ser facilmente servido em produção.

### Onde obter ajuda:

Se tiver problemas enquanto caminha por este tutorial, por favor consulte a seção [Obtendo ajuda](#) da FAQ.

## Personalize a aparência da sua *aplicação*

Primeiro, crie um diretório chamado **static** no seu diretório **polls**. O Django vai procurar os arquivos estáticos ali, de maneira similar a como o Django encontra os templates dentro de **polls/templates/**.

A configuração **STATICFILES\_FINDERS** do Django contém uma lista de buscadores que sabem como descobrir os arquivos estáticos de diversas fontes. Um dos padrões é o **AppDirectoriesFinder** que procura por um subdiretório "static" em cada uma das **INSTALLED\_APPS**, como a **polls** que nós acabamos de criar. O site de administração usa a mesma estrutura de diretórios para os arquivos estáticos.

Within the **static** directory you have just created, create another directory called **polls** and within that create a file called **style.css**. In other words, your stylesheet should be at **polls/static/polls/style.css**. Because of how the **AppDirectoriesFinder** staticfile finder works, you can refer to this static file in Django as **polls/style.css**, similar to how you reference the path for templates.

### Arquivo estático namespace

Just like templates, we *might* be able to get away with putting our static files directly in **polls/static** (rather than creating another **polls** subdirectory), but it would actually be a bad idea. Django will choose the first static file it finds whose name matches, and if you had a static file with the same name in a *different* application, Django would be unable to distinguish between them. We need to be able to point Django at the right one, and the best way to ensure this is by *namespacing* them. That is, by putting those static files inside *another* directory named for the application itself.

Coloque o seguinte código na sua folha de estilo (**polls/static/polls/style.css**):

**polls/static/polls/style.css**

```
li a {
    color: green;
}
```

Em seguida, adicione o seguinte no topo de "polls/templates/polls/index.html":

**polls/templates/polls/index.html**

```
{% load static %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}">
```

O modelo tag **{% static %}** gera a URL absoluta para arquivos estáticos.

Isso é tudo que você precisa fazer para o desenvolvimento.

Inicie o servidor (ou reinicie se estiver em execução):

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

Reload **http://localhost:8000/polls/** and you should see that the question links are green (Django style!) which means that your stylesheet was properly loaded.

## Adicionar uma imagem de fundo

Após, nós vamos criar o subdiretório para imagens. Crie um subdiretório **images** no diretório **polls/static/polls**. Dentro deste diretório, coloque uma imagem chamada **background.gif**. Em outras palavras, coloque sua imagem em **polls/static/polls/images/background.gif**.

Então, adicione em seu stylesheet (**polls/static/polls/style.css**):

**polls/static/polls/style.css**

```
body {
    background: white url("images/background.gif") no-repeat;
}
```

Reload **http://localhost:8000/polls/** and you should see the background loaded in the top left of the screen.

#### Aviso

Claro que o `{% static %}` template tag não está disponível para uso em arquivos estáticos como o seu stylesheet que não são gerados pelo Django. Você deve sempre usar **caminhos relativos** para conectar seus arquivos estáticos entre cada um, porque assim, você pode mudar **STATIC\_URL** (usado pelo **static** template tag para gerar as URLs) sem ter que modificar um monte de caminhos em seus arquivos estáticos também.

Estes são os **básicos**. Para mais detalhes nas configurações e outras coisas incluídas no framework veja [o howto arquivos estáticos](#). Fazendo deploy de arquivos estáticos discute como usar arquivos estáticos em um servidor real.

Quando você estiver confortável com os arquivos estáticos, leia [parte 7 deste tutorial](#) para aprender como personalizar o site de administração gerado automaticamente pelo Django.

## Escrevendo seu primeiro app Django, parte 7

Esse tutorial começa onde o [Tutorial 6](#) parou. Estamos continuando a aplicação Web para enquetes e focaremos na personalização do site administrativo automaticamente gerado pelo Django que exploramos no [Tutorial 2](#).

#### Onde obter ajuda:

Se tiver problemas enquanto caminha por este tutorial, por favor consulte a seção [Obtendo ajuda](#) da FAQ.

## Personalize o formulário do site de administração

Ao registrarmos o modelo de **Question** através da linha `admin.site.register(Question)`, o Django constrói um formulário padrão para representá-lo. Comumente, você desejará personalizar a apresentação e o funcionamento dos formulários do site de administração do Django. Para isto, você precisará informar ao Django as opções que você quer utilizar ao registrar o seu modelo.

Vamos ver como isto funciona reordenando os campos no formulário de edição. Substitua a linha `admin.site.register(Question)` por:

**polls/admin.py**

```
from django.contrib import admin

from .models import Question

class QuestionAdmin(admin.ModelAdmin):
```

```
fields = ['pub_date', 'question_text']
```

```
admin.site.register(Question, QuestionAdmin)
```

Você seguirá este padrão – crie uma classe de “model admin”, em seguida, passe-a como o segundo argumento para o `admin.site.register()` – todas as vezes que precisar alterar as opções administrativas para um modelo.

Essa mudança específica no código acima faz com que a “Publication date” apareça antes do campo “Question”:

Home › Polls › Questions › What's up?

## Change question

Date published:

Date: 2015-09-06

Today 

Time: 21:16:20

Now 

Question text:

What's up?

Isso não é impressionante com apenas dois campos, mas para formulários com dúzias deles, escolher uma ordem intuitiva é um detalhe importante para a usabilidade.

E por falar em dúzias de campos, você pode querer dividir o formulário em grupos:

polls/admin.py

```
from django.contrib import admin

from .models import Question

class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question_text']}),
        ('Date information', {'fields': ['pub_date']}),
    ]

admin.site.register(Question, QuestionAdmin)
```

O primeiro elemento de cada tupla em **fieldsets** é o título do grupo. Aqui está como o nosso formulário se parece agora:

## Adicionando objetos relacionados

Ok, nós temos nossa página de administração Questão, mas a **Questão** tem múltiplas **Questões**, e a página de administração não exibe escolhas.

Ainda.

There are two ways to solve this problem. The first is to register **Choice** with the admin just as we did with **Question**:

`polls/admin.py`

```
from django.contrib import admin

from .models import Choice, Question

# ...



admin.site.register(Choice)
```

Agora “Choices” é uma opção disponível no site de administração do Django. O formulário de “Add choice” se parece com isto:



Home › Polls › Choices › Add choice

## Add choice

Question:	<input type="text" value="-----"/>  
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>

Nesse formulário, o campo “Question” é uma caixa de seleção contendo todas as enquetes no banco de dados. O Django sabe que uma **ForeignKey** deve ser representada no site de administração como um campo **<select>**. No nosso caso, só existe uma enquete até agora.

Observe também o link “Add Another” ao lado de “Question”. Todo objeto com um relacionamento de chave estrangeira para outro ganha essa opção gratuitamente. Quando você clica em “Add Another”, você terá uma janela popup com o formulário “Add question”. Se você adicionar uma enquete na janela e clicar em “Save”, o Django salvará a enquete no banco de dados e irá dinamicamente adicionar a opção já selecionada ao formulário “Add choice” que você está vendo.

Mas, sério, essa é uma maneira ineficiente de adicionar objetos **Choice** ao sistema. Seria muito melhor se você pudesse adicionar várias opções diretamente quando criasse um objeto **Question**. Vamos fazer isso acontecer.

Remova a chamada **register()** do modelo **Choice**. Então edite o código de registro de **Question** para que fique assim:

polls/admin.py

```
from django.contrib import admin

from .models import Choice, Question

class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [
```

```

    (None, {'fields': ['question_text']})),
    ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']})),
]
inlines = [ChoiceInline]

```

```
admin.site.register(Question, QuestionAdmin)
```

Isso informa ao Django: Objetos “**Choice**” são editados na mesma página de administração de **Question**. Por padrão, forneça campos suficientes para 3 escolhas.”

Carregue a página “Add question” para ver como está:

## Add question

Question text:

## Date information (Hide)

Date published:

Date:

Today | 

Time:

Now | 

## CHOICES

Choice: #1

Choice text:

Votes:

Choice: #2

Choice text:

Votes:

Choice: #3

Choice text:

Votes:

[+ Add another Choice](#)[Save and add another](#)[Save and continue](#)

Funciona assim: há três blocos para Escolhas relacionadas – como especificado em **extra** –, mas a cada vez que você retorna à página de “Alteração” para um objeto já criado, você ganha outros três blocos extra.

No final dos três blocos atuais você encontrará um link “Adicionar outra escolha”. Se você clicar nele, será adicionado um novo bloco. Se você deseja remover o bloco adicional, você pode clicar no X ao topo direito do bloco acrescentado. Observe que você não pode remover os três blocos originais. Esta imagem mostra um bloco acrescentado:

CHOICES	
Choice: #1	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #2	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #3	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
Choice: #4	
Choice text:	<input type="text"/>
Votes:	<input type="text" value="0"/>
<a href="#">+ Add another Choice</a>	

One small problem, though. It takes a lot of screen space to display all the fields for entering related **Choice** objects. For that reason, Django offers a tabular way of displaying inline related objects. To use it, change the **ChoiceInline** declaration to read:

polls/admin.py

```
class ChoiceInline(admin.TabularInline):
    #...
```

Com o **TabularInline** (em vez de **StackedInline**), os objetos relacionados são exibidos de uma maneira mais compacta, formatada em tabela:

CHOICES	
CHOICE TEXT	VOTES
<input type="text"/>	<input type="text" value="0"/>
<input type="text"/>	<input type="text" value="0"/>
<input type="text"/>	<input type="text" value="0"/>

[+ Add another Choice](#)

[Save and add another](#) [Save and continue](#)

Observe que há uma coluna extra “Delete?” que permite a remoção de linhas adicionadas usando o botão “Add Another Choice” e linhas que já foram salvas.

## Personalize a listagem da página de administração

Agora que a página de administração de Pergunta está bonita, vamos fazer algumas melhorias à página “change list” – aquela que exibe todas as enquetes do sistema.

Aqui é como ela está até agora:

Home > Polls > Questions

Select question to change

Action:   0 of 1 selected

<input type="checkbox"/>	QUESTION
<input type="checkbox"/>	What's up?

1 question

Por padrão, o Django mostra o `str()` de cada objeto. Mas algumas vezes seria mais útil se pudéssemos mostrar campos individuais. Para fazer isso, use a opção `list_display`, que é uma tupla de nomes de campos a serem exibidos, como colunas, na lista de mudança dos objetos:

```
polls/admin.py
```

```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date')
```

For good measure, let's also include the `was_published_recently()` method from [Tutorial 2](#):

polls/admin.py

```
class QuestionAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question_text', 'pub_date', 'was_published_recently')
```

Agora a página de edição de enquetes está assim:

Home › Polls › Questions

Select question to change

Action:   0 of 1 selected

<input type="checkbox"/>	QUESTION TEXT	DATE PUBLISHED	WAS PUBLISHED RECENTLY
<input type="checkbox"/>	What's up?	Sept. 3, 2015, 9:16 p.m.	False

1 question

Você pode clicar no cabeçalho da coluna para ordená-las por estes valores – exceto no caso do `was_published_recently`, porque a ordenação pelo resultado de um método arbitrário não é suportada. Também note que o cabeçalho da coluna para `was_published_recently` é, por padrão, o nome do método (com underscores substituídos por espaços), e que cada linha contém a representação da saída.

Você pode melhorar isso adicionando ao método (em `polls/models.py`) alguns atributos, como segue:

polls/models.py

```
class Question(models.Model):
    # ...

    def was_published_recently(self):
        now = timezone.now()
        return now - datetime.timedelta(days=1) <= self.pub_date <= now

    was_published_recently.admin_order_field = 'pub_date'
    was_published_recently.boolean = True
```

```
was_published_recently.short_description = 'Published recently?'
```

Para obter mais informações sobre essas propriedades de método, veja `:attr:`~django.contrib.admin.ModelAdmin.list_display``.

Edite o arquivo `polls/admin.py` de novo e adicione uma melhoria à página de lista de edição **Question**: Um filtro usando `list_filter`. Adicione a seguinte linha ao `QuestionAdmin`:

```
list_filter = ['pub_date']
```

Isso adiciona uma barra lateral “Filter” que permite às pessoas filtrarem a lista de edição pelo campo `pub_date`:

O tipo do filtro apresentado depende do tipo do campo pelo qual você está filtrando. Como `pub_date` é uma instância da classe `DateTimeField`, o Django consegue deduzir que os filtros apropriados são: “Qualquer data”, “Hoje”, “Últimos 7 dias”, “Esse mês”, “Esse ano”.

Isso está ficando bom. Vamos adicionar capacidade de pesquisa:

```
search_fields = ['question_text']
```

Isso adiciona um campo de pesquisa ao topo da lista de edição. Quando alguém informa termos de pesquisa, o Django irá pesquisar o campo `question_text`. Você pode usar quantos campos quiser – entretanto, por ele usar uma consulta **LIKE** internamente, limitar o número de campos de pesquisa a um número razoável, será mais fácil para o seu banco de dados para fazer pesquisas.

Agora é também uma boa hora para observar que a página de edição fornece uma paginação gratuitamente. O padrão é mostrar 100 itens por página. **Paginação da página de edição, campos de pesquisa, filters, hierarquia por data, e ordenação por cabeçalho de coluna** todos trabalham em sincronia como deveriam.

## Personalize a aparência do site de administração

Obviamente, ter “Django administration” no topo de cada página de administração é ridículo. Isso é só um texto de exemplo.

You can change it, though, using Django's template system. The Django admin is powered by Django itself, and its interfaces use Django's own template system.

## Personalize os templates do seu *projeto*

Criar um diretório **templates** no diretório do seu projeto (aquela que contém **manage.py**). Os modelos podem viver em qualquer lugar em seu sistema de arquivos que o Django possa acessar. (Django é executado como qualquer usuário que seu servidor é executado.) No entanto, manter seus templates dentro do projeto é uma boa convenção a seguir.

Abra seu arquivo de configurações (**mysite/settings.py**, lembre-se) e adicione uma opção **DIRS** na configuração **TEMPLATES** setting:

mysite/settings.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    ],
]
```

**DIRS** é uma lista de diretórios de arquivos que serão checados quando o Django carregar seus templates, ou seja, caminhos de busca.

### Organizando templates

Just like the static files, we *could* have all our templates together, in one big templates directory, and it would work perfectly well. However, templates that belong to a particular application should be placed in that application's template directory (e.g. **polls/templates**) rather than the project's (**templates**). We'll discuss in more detail in the [reusable apps tutorial](#) why we do this.

Agora crie um diretório chamado **admin** dentro de **templates**, e copie o template **admin/base\_site.html** de dentro do diretório padrão do site de administração do Django (**django/contrib/admin/templates**) para dentro deste diretório.



## Onde estão os arquivos de código-fonte do Django?

Se você tiver dificuldade em encontrar onde os arquivos do Django estão localizados em seu sistema, execute o seguinte comando:

```
$ python -c "import django; print(django.__path__)"
```

Então, edite o arquivo e substitua `{{ site_header|default:_('Django administration') }}` (incluindo as chaves) com nome do seu próprio site como desejar. Você deve acabar com uma seção de código como:

```
{% block branding %}

<h1 id="site-name"><a href="{% url 'admin:index' %}">Polls Administration</a></h1>

{% endblock %}
```

Nós usamos esta abordagem para ensinar-lhe como substituir templates. Em um projeto real, você provavelmente usaria o atributo `django.contrib.admin.AdminSite.site_header` para fazer mais facilmente essa personalização em particular.

Esse template contém uma série de textos como `{% block branding %}` e `{{ tittle }}`. As tags `{%}` e `{{}}` fazem parte da linguagem de templates do Django. Quando o Django renderiza o template "admin/base\_site.html", o seu conteúdo é processado por essa linguagem de templates para produzir a página HTML final, assim como vimos no [Tutorial 3](#).

Note que qualquer template padrão do site de administração pode ser sobrescrito. Para sobrescrever um template, apenas faça a mesma coisa que você fez com `base_site.html` – copie ele do diretório padrão para o seu próprio diretório, e faça as mudanças.

## Personalize os templates da sua aplicação

Leitores astutos irão se perguntar: Mas se **DIRS** estava vazio por padrão, como o Django pôde encontrar o diretório padrão dos templates do site de administração? A resposta é, Desde que **APP\_DIRS** seja **True**, o Django irá automaticamente procurar por um subdiretório **templates/** dentro de cada pacote de aplicação, para usar como fallback (Não esqueça que `django.contrib.admin` é uma aplicação)

Nossa aplicação de enquetes não é muito complexa e não precisa de templates personalizados para o site de administração. Mas se ele crescer e ficar mais sofisticado e necessária a modificação dos templates padrão de administração do Django para algumas de suas funcionalidades, seria mais sensato modificar os templates da *aplicação*, e não os do *projeto*. Dessa forma, você poderia incluir a aplicação de enquete em qualquer novo projeto e ter a certeza que iria encontrar os templates personalizados que você precisa.

Veja a [documentação de carga de template](#) - para mais informação sobre como o Django acha seus templates.

## Personalize a página inicial do site de administração

De maneira similar, você pode querer personalizar a aparência da página inicial do site de administração do Django.

Por padrão, ele exibe todas as aplicações em **INSTALLED\_APPS**, que estão registrados na aplicação de administração, em ordem alfabética. E você pode querer fazer alterações significativas no layout. Além do que, a página inicial é provavelmente a página mais importante da página de administração, e deve ser fácil de usar.

O template para personalizar é `admin/index.html`. (Faça o mesmo que `admin/base_site.html` na seção anterior – copie ele do diretório padrão para o seu diretório de template personalizado). Edite o arquivo e você verá que ele usa uma

variável de template chamada **app\_list**. Esta variável contém cada aplicação instalada no Django. Ao invés de usar isto, você pode criar links diretos para páginas do site de administração específicas em qualquer forma que você achar melhor.