

Esta é a tradução da documentação oficial do Django, mantida pelo grupo de localização do Django para o Português.

Django Documentation

Release 1.0

Django Software Foundation

September 12, 2017

Contents

I	Primeiros passos	1
1	Django num relance	5
2	Guia de instalação rápida	11
3	Escrevendo sua primeira aplicação Django, parte 1	13
4	Escrevendo sua primeira aplicação Django, parte 2	23
5	Escrevendo sua primeira aplicação Django, parte 3	35
6	Escrevendo sua primeira aplicação Django, parte 4	43
7	What to read next	49
II	Usando o Django	53
8	Como instalar o Django	57
9	Models e bancos de dados	61
10	Manipulando requisições HTTP	99
11	Trabalhando com formulários	129
12	The Django template language	157
13	Views genéricas	167
14	Gerenciando arquivos	175
15	Testando aplicações Django	179
16	Autenticação de Usuário no Django	195
17	O framework de cache do Django	215
18	Sending e-mail	227
19	Internacionalização	233
20	Paginação	249

21	Seritando objetos do Django	255
22	Django settings	259
23	Sinais (Signals)	263
III	Guias “how-to”	267
24	Autenticando no Apache usuários do banco de dados do Django	271
25	Escrevendo commando customizados para o django-admin	273
26	Writing custom model fields	275
27	Tags e filtros de template personalizados	285
28	Escrevendo um sistema de armazenamento customizado	299
29	Implantando o Django	301
30	Reporte de erros via e-mail	313
31	Provendo dados iniciais para models	315
32	Rodando Django no Jython	319
33	Integrando o Django com um banco de dados legado	321
34	Exportando CSV com o Django	323
35	Exportando PDFs com o Django	327
36	Como servir arquivos estáticos	331
IV	FAQ do Django	335
37	FAQ: Geral	337
38	FAQ: Instalação	341
39	FAQ: Usando o Django	343
40	FAQ: Obtendo ajuda	345
41	FAQ: Bancos de dados e modelos	347
42	FAQ: A administração	349
43	FAQ: Contribuindo com código	353
V	Referência da API	355
44	Os add-ons “django.contrib”	357
45	Notes about supported databases	433
46	django-admin.py e manage.py	441
47	Referência de manipulação de arquivos	455

48	Formulários	459
49	Generic views (Visões genéricas)	491
50	Referência dos middlewares embutidos	507
51	Modelos	511
52	Objetos de requisição e resposta	553
53	Configurações disponíveis	561
54	Referência de sinais embutidos	579
55	Referência do Template	583
56	Dados Unicode no Django	615
VI	Meta-documentação e miscelânea	621
57	Filosofia de design	625
58	Estabilidade de API	631
59	Distribuições de terceiros do Django	635
VII	Glossário	637
VIII	Notas de Lançamento	641
60	Django version 0.95 release notes	645
61	Django version 0.96 release notes	647
62	Django 1.0 alpha notas de lançamento	651
63	Django 1.0 alpha 2 release notes	655
64	Django 1.0 beta 1 release notes	657
65	Notas de lançamento do Django 1.0 beta 2	661
66	Django 1.0 release notes	663
67	Notas de lançamento do Django 1.0.1	679
68	Notas de lançamento do Django 1.0.2	681
IX	Django internamente	683
69	Contributing to Django	687
70	Como a documentação do Django funciona	703
71	Django committers	707
72	Django's release process	711
73	Agenda de Obsolescência do Django	715

X	Índices, glossário e tabelas	717
XI	Documentação depreciada/obsoleta	721
74	Documentação depreciada/obsoleto	725
	Python Module Index	729

Part I

Primeiros passos

Novo no Django? Ou em desenvolvimento em geral? Bem, você veio ao lugar certo: leia este material para entender e começar o quanto antes.

Django num relance

Como o Django foi desenvolvido no ambiente movimentado de uma redação, ele foi desenhado para tornar as tarefas comuns do desenvolvimento Web rápidas e fáceis. Aqui está uma visão informal sobre como escrever uma aplicação Web dirigida a banco de dados com Django.

O objetivo deste documento é dar a você as especificações técnicas necessárias para entender como o Django funciona, mas este texto não pretende ser um tutorial ou uma referência – mas possuímos os dois! Quando você estiver pronto para iniciar um projeto, você pode *iniciar com o tutorial* ou *mergulhar direto em uma documentação mais detalhada*.

Projete seu model

Embora você possa usar o Django sem um banco de dados, ele vem com um mapeador objeto-relacional no qual você descreve o layout da sua base de dados em código Python.

A *sintaxe do modelo de dados* oferece muitas formas ricas de representar seus models – portanto, ela está resolvendo cerca de dois anos de problemas de esquemas de base de dados. Aqui está um exemplo rápido:

```
class Reporter(models.Model):
    full_name = models.CharField(max_length=70)

    def __unicode__(self):
        return self.full_name

class Article(models.Model):
    pub_date = models.DateTimeField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)

    def __unicode__(self):
        return self.headline
```

Instale-o

Em seguida, rode o utilitário de linha de comando do Django para criar as tabelas da base de dados automaticamente:

```
manage.py syncdb
```

O comando `syncdb` procura por todos os seus modelos disponíveis e cria as tabelas na sua base de dados caso as tabelas ainda não existam.

Aproveite a API livre

Com ela você irá ter uma *API Python* livre e rica para acessar seus dados. A API é criada em tempo de execução, nenhuma geração de código é necessária:

```
>>> from mysite.models import Reporter, Article

# Nenhum jornalista no sistema ainda.
>>> Reporter.objects.all()
[]

# Crie um novo jornalista.
>>> r = Reporter(full_name='John Smith')

# Salve o objeto na base de dados. Você terá que chamar o save()
# explicitamente.
>>> r.save()

# Agora ele tem um ID.
>>> r.id
1

# Agora o novo jornalista está na base de dados.
>>> Reporter.objects.all()
[<Reporter: John Smith>]

# Campos são representados como atributos de objetos Python.
>>> r.full_name
'John Smith'

# O Django fornece uma rica API de pesquisa à base de dados.
>>> Reporter.objects.get(id=1)
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__startswith='John')
<Reporter: John Smith>
>>> Reporter.objects.get(full_name__contains='mith')
<Reporter: John Smith>
>>> Reporter.objects.get(id=2)
Traceback (most recent call last):
...
DoesNotExist: Reporter matching query does not exist.

# Crie um artigo.
>>> from datetime import datetime
>>> a = Article(pub_date=datetime.now(), headline='Django is cool',
...             content='Yeah.', reporter=r)
>>> a.save()

# Agora o artigo está na base de dados.
>>> Article.objects.all()
```

```
[<Article: Django is cool>]

# Objetos Article tem acesso via API a objetos Reporter relacionados.
>>> r = a.reporter
>>> r.full_name
'John Smith'

# E vice-versa: Objetos Reporter tem acesso via API a objetos Article.
>>> r.article_set.all()
[<Article: Django is cool>]

# A API segue relacionamentos da forma que você precisar, realizando JOINS
# eficientes para você por baixo dos panos.
# Isto encontra todos os artigos de um jornalista cujo nome começa com "John".
>>> Article.objects.filter(reporter__full_name__startswith="John")
[<Article: Django is cool>]

# Modifique um objeto alterando seus atributos e chamando save().
>>> r.full_name = 'Billy Goat'
>>> r.save()

# Delete um objeto com delete().
>>> r.delete()
```

Uma interface de administração dinâmica: não é apenas o andaime – é a casa inteira

Uma vez definido seus models, o Django poderá criar automaticamente uma *interface administrativa* profissional pronta para produção – um Web site que permite aos usuários autenticados adicionar, alterar e deletar objetos. E isso é tão fácil como registrar seu modelo no site de administracao:

```
# In models.py...

from django.db import models

class Article(models.Model):
    pub_date = models.DateTimeField()
    headline = models.CharField(max_length=200)
    content = models.TextField()
    reporter = models.ForeignKey(Reporter)

# In admin.py in the same directory...

import models
from django.contrib import admin

admin.site.register(models.Article)
```

A filosofia aqui é que seu site é editado por uma equipe, ou um cliente, ou talvez apenas por você – e você não quer ter que lidar com criação de interfaces backend apenas para gerenciar conteúdo.

Um workflow típico na criação de aplicações Django é criar os models e ter os sites administrativos rodando o mais rápido possível, assim sua equipe (ou cliente) poderá começar a inserir os dados. Em seguida, desenvolve-se a forma com que os dados serão apresentados ao público.

Projete suas URLs

Um esquema limpo e elegante de URLs é um detalhe importante em uma aplicação Web de alta qualidade. O Django encoraja o desenho de belíssimas URLs e não coloca nenhuma sujeira nelas, como `.php` ou `.asp`.

Para desenhar URLs para uma aplicação, você cria um módulo Python chamado *URLconf*. Uma tabela de conteúdos para sua aplicação, contendo um mapeamento simples entre padrões de URL e funções Python de retorno (funções de callback). URLconfs também servem para desacoplar as URLs do código Python.

Aqui está como um URLconf se apresentaria para o exemplo `Reporter/Article` acima:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^articles/(\d{4})/$', 'mysite.views.year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'mysite.views.month_archive'),
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'mysite.views.article_detail'),
)
```

O código acima mapeia URLs, como simples expressões regulares, para a localização das funções Python de retorno (“views”). As expressões regulares usam parênteses para “capturar” valores das URLs. Quando um usuário requisita uma página, o Django percorre cada padrão, em ordem, e para no primeiro que combina com a URL requisitada. (Se nenhum deles combinar o Django chama uma view de exceção 404.) Isso é incrivelmente rápido, porque as expressões regulares são compiladas em tempo de execução.

Uma vez que uma das expressões regulares case, o Django importa e chama a view indicada, que é uma simples função Python. Cada view recebe um objeto request – que contém os metadados da requisição – e os valores capturados na expressão regular.

Por exemplo, se um usuário requisitou a URL `“/articles/2005/05/39323/”`, o Django deverá chamar a função `mysite.views.article_detail(request, '2005', '05', '39323')`.

Escreva suas views

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

Geralmente, uma view recupera dados de acordo com os parâmetros, carrega um template e renderiza o template com os dados recuperados. Aqui está uma view para o `year_archive` do exemplo acima:

```
def year_archive(request, year):
    a_list = Article.objects.filter(pub_date__year=year)
    return render_to_response('news/year_archive.html', {'year': year, 'article_
↪list': a_list})
```

Esse exemplo usa o *sistema de template* do Django, o qual possui diversos recursos poderosos mas prima por permanecer simples o suficiente para que não-programadores possam usá-lo.

Desenhe seus templates

O código acima carrega o template `news/year_archive.html`.

O Django tem um caminho de pesquisa para templates, o qual permite a você minimizar a redundância entre templates. Nas configurações do Django, você especifica uma lista de diretórios para procurar por templates. Se um template não existir no primeiro diretório, ele verifica o segundo e assim por diante.

Vamos dizer que o template `news/article_detail.html` foi encontrado. Aqui está como ele deve se parecer:

```
{% extends "base.html" %}

{% block title %}Artigos de {{ year }}{% endblock %}

{% block content %}
<h1>Artigos de {{ year }}</h1>

{% for article in article_list %}
    <p>{{ article.headline }}</p>
    <p>Por {{ article.reporter.full_name }}</p>
    <p>Publicado em {{ article.pub_date|date:"F j, Y" }}</p>
{% endfor %}
{% endblock %}
```

Variáveis são cercadas por chaves duplas. `{{ article.headline }}` que significa “retorne o valor do atributo `headline` do artigo”, mas pontos não são usados apenas para acessar atributos: eles também podem acessar chaves de dicionários, acessar índices e chamar funções.

Note que `{{ article.pub_date|date:"F j, Y" }}` usa um “pipe” no estilo Unix (o caractere “|”). Isso é chamado de filtro de template, e é uma forma de filtrar o valor de uma variável. Nesse caso, o filtro “date” formata um objeto Python `datetime` para um formato dado (como o encontrado na função `date` do PHP; sim, aqui está uma boa idéia do PHP).

Você pode atrelar tantos filtros quanto você quiser. Você pode escrever filtros personalizados. Você pode escrever tags de template personalizadas, que irão rodar código Python personalizado por baixo dos panos.

Finalmente, o Django usa o conceito de “herança de templates”: É isso que o `{% extends "base.html" %}` faz. Isso significa “primeiro carregue o template chamado ‘base’, o qual define uma série de blocos, e preencha os blocos com os seguintes blocos”. Resumindo, isso permite a você reduzir drasticamente a redundância em seus templates: cada template tem que definir apenas aquilo que é único para aquele template.

Aqui está como o template “base.html” deve se parecer:

```
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>
    
    {% block content %}{% endblock %}
</body>
</html>
```

Simplesmente, ele define a aparência do site (com o logotipo do site), e fornece “buracos” para templates filhos preencherem. Isso torna o redesign do site tão fácil quanto modificar um único arquivo – o template base.

Isso também permite a você criar múltiplas versões de um site, com templates base diferentes, enquanto reutiliza templates filhos. Os criadores do Django utilizaram essa técnica para criar versões de sites para celulares totalmente diferentes – simplesmente criando um novo template base.

Note que você não tem que usar o sistema de templates do Django se você preferir outro. Enquanto o sistema de templates do Django é particularmente bem integrado com a camada de modelos do Django, nada força você a usá-lo. Da mesma maneira que, você não tem que usar a API de banco de dados do Django, também. Você pode usar outras camadas de abstração de banco de dados, você pode ler arquivos XML, você pode ler arquivos do disco, ou qualquer coisa que desejar. Cada parte do Django – `models`, `views` e `templates` – é desacoplada da próxima.

Isto é apenas a superfície

Essa foi apenas uma visão rápida das funcionalidades do Django. Alguns outros recursos úteis:

- Um *framework para caching* que se integra com o memcached e outros backends.
- Um *syndication framework* que torna a criação de RSS e Atom uma tarefa tão fácil quanto escrever uma classe Python.
- Mais recursos sexy da administração gerada automaticamente – essa introdução mal arranhou a superfície.

Os próximos passos óbvios para você fazer é [baixar o Django](#), ler [o tutorial](#) e juntar-se [a comunidade](#). Obrigado pelo seu interesse!

Guia de instalação rápida

Antes que você possa utilizar o Django, você precisará instalá-lo. Nós temos um [guia completo de instalação](#) que abrange todas as possibilidades; este guia irá guiar você em uma instalação simples e mínima que será suficiente durante a leitura da introdução.

Instalando o Python

Sendo um framework web Python, o Django requer Python. Ele funciona com qualquer versão desde a 2.3 até a 2.6 (devido a questões de incompatibilidade retroativa no Python 3.0, o Django atualmente não funciona com o Python 3.0; veja [a FAQ do Django](#) para mais informações sobre as versões suportadas do Python e a transição para o 3.0), mas nós recomendamos instalar o Python 2.5 ou posterior. Se fizer isso, você não precisará configurar um servidor de banco de dados: o Python 2.5 ou posterior inclui uma base de dados leve chamada [SQLite](#).

Baixe o Python em <http://www.python.org>. Se você estiver rodando Linux ou Mac OS X, você provavelmente já possui ele instalado.

Django com Jython

Se você usa [Jython](#) (Uma implementação Python para a plataforma Java), você irá precisar seguir alguns passos adicionais. Veja [Rodando Django no Jython](#) para detalhes.

you can verify if Python is installed by typing `python` in your terminal; you should see something like:

```
Python 2.5.1 (r251:54863, Jan 17 2008, 19:35:17)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Configurando um banco de dados

If you installed Python 2.5 or later, you can skip this step now.

If not, or if you would like to work with a "big" database engine like PostgreSQL, MySQL, or Oracle, consult [the database installation information](#).

Remova qualquer versão antiga do Django

Se você está atualizando sua instalação do Django, você vai precisar *desinstalar a versão antiga do Django antes de instalar a nova versão*.

Instale o Django

Você tem três opções fáceis para instalar Django:

- Instale a versão do Django *provida pelo distribuidor do seu sistema operacional*. Esta é a opção mais rápida para aqueles que tem sistemas operacionais que distribuem o Django.
- *Instale uma release oficial*. Esta é a melhor abordagem para usuários que querem um número de versão estável e não estão preocupados em rodar uma versão ligeiramente mais antiga do Django.
- *Instalar a versão de desenvolvimento mais recente*. Esta é melhor para usuários que querem as mais recentes e melhores funcionalidades e não tem medo de executar códigos novos.

Warning: Se você fizer um dos dois primeiro passos, mantenha seus olhos longe das partes da documentação marcadas como **novo na versão de desenvolvimento**. Esta frase marca características que estão disponíveis apenas na versão de desenvolvimento do Django; Se você tentar utilizar elas com uma release oficial elas não irão funcionar.

É isso!

É isso! – Agora você pode *avançar para o tutorial*.

Escrevendo sua primeira aplicação Django, parte 1

Vamos aprender através de um exemplo.

Ao longo deste tutorial nós iremos guiá-lo na criação de uma aplicação básica de enquete.

Ela consistirá em duas partes:

- Um site público em que as pessoas poderão visualizar e votar em enquetes.
- Um site de administração que você poderá adicionar, alterar e deletar enquetes.

Iremos assumir que você já possui o *Django* *instalado*. Você pode saber se o Django está instalado rodando o interpretador interativo do Python e digitando `import django`. Se esse comando rodar com sucesso, sem nenhum erro, o Django está instalado.

Onde conseguir ajuda:

Se você estiver tendo problemas no decorrer desse tutorial, por favor envie uma mensagem para a [django-brasil](#) ou para a [django-users](#) (em inglês) ou deixe-a em `#django-br` ou `#django` no `irc.freenode.net` para bater-papo com outros usuários Django dispostos a ajudar.

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, `cd` para o diretório onde você gostaria de armazenar seu código, então execute o comando `django-admin.py startproject mysite`. Isto irá criar o diretório `mysite` no seu diretório atual.

Permissões do Mac OS X

Se você estiver usando Mac OS X, você poderá ver a mensagem “permissão negada” quando você tentar rodar o `django-admin.py startproject`. Isto porque, em sistemas baseados em Unix como o OS X, um arquivo precisa ser marcado como “executável” antes que possa ser executado como um programa. Para fazer isso, abra o

Terminal.app e navegue (usando o comando `cd`) até o diretório onde o `django-admin.py` está instalado, então rode o comando `chmod +x django-admin.py`.

Note: 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).

O `django-admin.py` deverá estar no “path” do sistema se você instalou o Django via `python setup.py`. Se ele não estiver no path, você poderá encontrá-lo em `site-packages/django/bin`, onde `site-packages` é um diretório dentro da sua instalação do Python. Considere a possibilidade de criar um link simbólico para o `django-admin.py` em algum lugar em seu path, como em `/usr/local/bin`.

Onde esse código deve ficar?

Se você tem experiência prévia em PHP, 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 idéia 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 olhar o que o `startproject` criou:

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

Estes arquivos são:

- `__init__.py`: Um arquivo vazio que diz ao Python que esse diretório deve ser considerado como um pacote Python. (Leia [mais sobre pacotes](#) na documentação oficial do Python se você for iniciante em Python.)
- `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.py e manage.py](#).
- `settings.py`: Configurações para este projeto Django. [Django settings](#) Irá revelar para você tudo sobre como o settings funciona.
- `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 [URL dispatcher](#).

O servidor de desenvolvimento

Vamos verificar se ele funciona. Vá para o diretório `mysite`, se você ainda não estiver nele, e rode o comando `python manage.py runserver`. Você irá ver a seguinte saída na sua linha de comando:

```
Validating models...
0 errors found.

Django version 1.0, using settings 'mysite.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

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 anotar: 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/> com seu navegador Web. Você irá ver a página “Welcome to Django”, em agradáveis tons de azul claro pastel. Ele funcionou!

Mudando 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-o junto com a porta. Para então escutar em todos os IPs públicos (útil se você quer mostrar seu trabalho em outros computadores), use:

```
python manage.py runserver 0.0.0.0:8000
```

A documentação completa para o servidor de desenvolvimento está na referência `runserver`.

Configuração do banco de dados

Agora, edite o `settings.py`. Ele é um módulo normal do Python com variáveis representando as configurações do Django. Altere essas configurações para que atendam aos parâmetros de conexão do seu banco de dados:

- `DATABASE_ENGINE` – Um entre ‘`postgresql_psycopg2`’, ‘`mysql`’ ou ‘`sqlite3`’. Outros backends também *estão disponíveis*.
- `DATABASE_NAME` – O nome do seu banco de dados, se você estiver usando SQLite, o banco de dados será uma arquivo no seu computador; neste caso, `DATABASE_NAME` deve ser o caminho completo, incluindo o nome, para este arquivo. Se este arquivo não existir, ele sera automaticamente criado quando você sincronizar o banco de dados pela primeira vez (veja abaixo).

Quando estiver especificando o caminho, sempre use a barra normal até mesmo no Windows (e.g. `C:/homes/user/mysite/sqlite3.db`).

- `DATABASE_USER` – Usuário do seu banco de dados (não utilizado para o SQLite).
- `DATABASE_PASSWORD` – Senha do seu banco de dados (não utilizado para o SQLite).
- `DATABASE_HOST` – O host onde seu banco de dados está. Deixe isso em branco se o seu servidor de banco de dados está na mesma máquina física (não utilizado para o SQLite).

Se você é novo em banco de dados, nós recomendamos simplesmente utilizar o SQLite (configurando `DATABASE_ENGINE` para ‘`sqlite3`’). SQLite é incluído como parte do Python 2.5 e posterior, então você não precisará instalar qualquer outra coisa.

Nota

Se você estiver utilizando PostgreSQL ou MySQL, certifique-se de que já tenha criado o banco de dados a partir deste ponto. Faça isso com “`CREATE DATABASE database_name;`” no prompt interativo do seu banco de dados.

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 edita o `settings.py`, observe a configuração do `INSTALLED_APPS` próximo ao final 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.auth` – Um sistema de autenticação.
- `django.contrib.contenttypes` – Um framework para tipos de conteúdo.
- `django.contrib.sessions` – Um framework de sessão.
- `django.contrib.sites` – Um framework para gerenciamento de múltiplos sites com uma instalação do Django.

Essas aplicações são incluídas por padrão como uma conveniência para os casos comuns.

Cada uma dessas aplicações faz 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 syncdb
```

O comando `syncdb` verifica a configuração em `INSTALLED_APPS` e cria todas as tabelas necessárias de acordo com a configuração do banco de dados em seu arquivo `settings.py`. Você irá ver uma mensagem para cada tabela do banco de dados que ele criar, e você irá receber um prompt perguntando se gostaria de criar uma conta de super-usuário para o sistema de autenticação. Vá em frente e faça isso.

Se você estiver interessado, rode o cliente de linha de comando do seu banco de dados e digite `\dt` (PostgreSQL), `SHOW TABLES;` (MySQL), ou `.schema` (SQLite) para mostrar as tabelas que o Django criou.

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, sintase livre para comentar ou deletar a(s) linha(s) apropriadas do `INSTALLED_APPS` antes de rodar o `syncdb`. O comando `syncdb` irá criar apenas as tabelas para as aplicações que estiverem no `INSTALLED_APPS`.

Criando models

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, em algum lugar no [path do Python](#), que seguirá 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 vs. aplicações

Qual é a diferença entre um projeto e uma aplicação? Uma aplicação é uma aplicação Web que faz alguma coisa – ex., um sistema de blog, um banco de dados de registros públicos ou uma simples aplicação de enquete. Um projeto é uma coleção de configurações e aplicações para um determinado Web site. Um projeto pode conter múltiplas aplicações. Uma aplicação pode estar em múltiplos projetos.

Neste tutorial nós iremos criar nossa aplicação de enquete no diretório `mysite`, por simplicidade. Consequentemente, a aplicação irá ser acoplada ao projeto – no caso, o código Python da aplicação de enquete irá se referir a `mysite.polls`. Posteriormente, neste tutorial, nós iremos discutir como desacoplar suas aplicações para serem distribuídas.

Para criar sua aplicação, certifique-se de que esteja no diretório `mysite` e digite o seguinte comando:

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

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

```
polls/  
  __init__.py  
  models.py  
  views.py
```

Esta estrutura de diretório irá abrigar a aplicação de enquete.

O primeiro passo ao escrever um banco de dados de uma aplicação Web no Django é definir seus `models` – essencialmente, o layout do banco de dados, com metadados adicionais.

Filosofia

Um `model` é a única e definitiva fonte de dados 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.

Em nossa simples aplicação de enquete, nós iremos criar dois `models`: `polls` e `choices`. Uma enquete (`poll`) tem uma pergunta e uma data de publicação. Uma escolha (`choice`) tem dois campos: o texto da escolha e um totalizador de votos. Cada escolha é associada a uma enquete.

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

```
from django.db import models  
  
class Poll(models.Model):  
    question = models.CharField(max_length=200)  
    pub_date = models.DateTimeField('date published')  
  
class Choice(models.Model):  
    poll = models.ForeignKey(Poll)  
    choice = models.CharField(max_length=200)  
    votes = models.IntegerField()
```

Erros sobre `max_length`

Se o Django retornar para você uma mensagem de erro dizendo que o `max_length` não é um argumento válido, você provavelmente está usando uma versão antiga do Django. (Esta versão do tutorial foi escrita para a última versão de desenvolvimento do Django.) Se você estiver usando um checkout de Subversion da versão de desenvolvimento do Django (veja [a documentação de instalação](#) para mais informações), você não deveria estar tendo nenhum problema.

Se você quiser continuar com uma versão antiga do Django, você deverá mudar para o [tutorial do Django 0.96](#), porque este tutorial cobre vários recursos que só existem na versão de desenvolvimento do Django.

O código é auto-explicativo. Cada `model` é representado por uma classe derivada da classe `django.db.models.Model`. Cada `model` possui um número de atributos de classe, as quais por sua vez representam campos de banco de dados no `model`.

Cada campo é representado por uma instância de uma classe `Field` – e.g., `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` (e.g. `question` 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 por 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 pela máquina. Neste exemplo nós definimos um nome legível por humanos apenas para `Poll.pub_date`. Para todos os outros campos neste model, o nome legível pela máquina será utilizado como nome legível por humanos.

Algumas classes `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.

Finalmente, note a definição de um relacionamento, usando `ForeignKey`, que diz ao Django que cada Choice (Escolha) está relacionada a uma única Poll (Enquete). O Django suporta todos os relacionamentos de banco de dados comuns: muitos-para-um, muitos-para-muitos e um-para-um.

Ativando os models

Aquela pequena parte do model 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 `Poll` 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.

Edite o arquivo `settings.py` novamente e altere o valor de `INSTALLED_APPS` para incluir `'mysite.polls'` de modo que se pareça com isso:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'mysite.polls'
)
```

Agora o Django sabe que o `mysite` inclui a aplicação `polls`. Vamos rodar outro comando:

```
python manage.py sql polls
```

Você deverá ver algo similar ao seguinte (as instruções SQL `CREATE TABLE` para a aplicação de enquetes):

```
BEGIN;
CREATE TABLE "polls_poll" (
    "id" serial NOT NULL PRIMARY KEY,
    "question" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "poll_id" integer NOT NULL REFERENCES "polls_poll" ("id"),
    "choice" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
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 model – `poll` e `choice`. (Você pode alterar esse comportamento.)
- Chaves primárias (id's) 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.
- O relacionamento da chave estrangeira é feito explicitamente por uma instrução `REFERENCES`.
- 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. O autor deste tutorial roda PostgreSQL, então a saída do exemplo está na sintaxe do PostgreSQL.
- O comando `sql` na realidade não roda o SQL no seu banco de dados - ele apenas o exibe na tela para que você saiba qual SQL o Django acha que é necessário. Se você quiser, você pode copiar e colar esse SQL no prompt do seu banco de dados. Entretanto, como nós iremos ver em breve, o Django fornece uma maneira mais fácil de submeter o SQL ao banco de dados.

Se tiver interesse, rode também os seguintes comandos:

- `python manage.py validate` – Verifica por quaisquer erros na construção dos seus models.
- `python manage.py sqlcustom polls` – Mostra quaisquer *instruções SQL personalizadas* (como modificações em tabelas ou restrições) que tenham sido definidas para a aplicação.
- `python manage.py sqlclear polls` – Mostra as instruções `DROP TABLE` necessárias para a aplicação, de acordo com as tabelas que já existem no seu banco de dados (se houver alguma).
- `python manage.py sqlindexes polls` – Mostra as instruções `CREATE INDEX` necessárias para a aplicação.
- `python manage.py sqlall polls` – Uma combinação de todo SQL dos comandos `sql`, `sqlcustom`, e `sqlindexes`.

Olhar para a saída desses comandos irá ajudar a entender o que está acontecendo por baixo dos panos.

Agora rode o `syncdb` novamente para criar essas tabelas dos models no seu banco de dados:

```
python manage.py syncdb
```

O comando `syncdb` roda o SQL do '`sqlall`' em seu banco de dados para todas as aplicações presentes no *INSTALLED_APPS* que ainda não existam em seu banco de dados. Isso criará todas as tabelas, dados iniciais e índices para quaisquer aplicações que você tenha adicionado ao seu projeto desde a última vez que você rodou o `syncdb`. O `syncdb` pode ser chamado sempre que você quiser, e ele irá apenas criar as tabelas que não existirem.

Leia a *documentação do `django-admin.py`* 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 o ambiente de projeto para você. A "configuração do ambiente" envolve duas coisas:

- Colocar o `mysite` no `sys.path`. Buscando flexibilidade, várias partes do Django referem-se aos projetos usando a notação de caminho pontuado (dotted-path) do Python (e.g. `'mysite.polls.models'`). Para que isso funcione, o pacote `mysite` precisa estar no `sys.path`.

Nós já vimos um exemplo disso: a configuração `INSTALLED_APPS` é uma lista de pacotes na notação de caminho pontuado.

- Configurar a variável de ambiente `DJANGO_SETTINGS_MODULE`, que fornece ao Django o caminho para o seu arquivo `settings.py`.

Ignorando o `manage.py`

Se você optar por não usar o `manage.py`, não há problemas. Apenas certifique-se de que o `mysite` está no nível raiz do caminho do Python (i.e., `import mysite` funciona) e configure a variável de ambiente `DJANGO_SETTINGS_MODULE` para `mysite.settings`.

Para mais informações sobre isso tudo, veja a [documentação do `django-admin.py`](#).

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

```
# Importe as classes de model que acabamos de criar.
>>> from mysite.polls.models import Poll, Choice

# Não há nenhuma enquete no sistema ainda.
>>> Poll.objects.all()
[]

# Crie uma nova enquete.
>>> import datetime
>>> p = Poll(question="What's up?", pub_date=datetime.datetime.now())

# Salve o objeto na base de dados. Você tem que chamar o save() explicitamente.
>>> p.save()

# Agora ele tem uma ID. Note que ele irá mostrar "1L" em vez de "1",
# dependendo de qual banco de dados você está usando. Não é nada demais; isso
# apenas significa que o backend do seu banco de dados prefere retornar
# inteiros como objetos long integers do Python.
>>> p.id
1

# Acesse colunas do banco de dados via atributos do Python.
>>> p.question
"What's up?"
>>> p.pub_date
datetime.datetime(2007, 7, 15, 12, 00, 53)

# Modifique os valores alterando os atributos e depois chamando o save()..
>>> p.pub_date = datetime.datetime(2007, 4, 1, 0, 0)
>>> p.save()

# objects.all() mostra todas as enquetes do banco de dados..
>>> Poll.objects.all()
[<Poll: Poll object>]
```

Espere um pouco. `<Poll: Poll object>` é uma representação totalmente inútil desse objeto. Vamos corrigir isso editando o model da enquete (no arquivo `polls/models.py`) e adicionando um método `__unicode__()` a ambos `Poll` e `Choice`:

```
class Poll(models.Model):
    # ...
    def __unicode__(self):
```

```

        return self.question

class Choice(models.Model):
    # ...
    def __unicode__(self):
        return self.choice

```

Se `__unicode__()` não parecer funcionar

Se você adicionar o método `__unicode__()` aos seus models e não observar nenhuma mudança na forma com que eles são representados, você deve estar usando uma versão antiga do Django (esta versão do tutorial foi escrita para a última versão de desenvolvimento do Django). Se você estiver usando um checkout do Subversion do Django (veja [a documentação de instalação](#) para mais informações), você não deveria estar tendo nenhum problema.

Se você quiser seguir com uma versão antiga do Django, você deverá mudar para o [tutorial do Django 0.96](#), porque este tutorial cobre vários recursos que existem apenas na versão de desenvolvimento do Django.

É importante adicionar métodos `__unicode__()` aos seus models, não apenas para sua própria sanidade quando estiver lidando com o prompt interativo, mas também porque representações de objetos são usadas em toda a administração gerada automaticamente pelo Django.

Por que `__unicode__()` e não `__str__()`?

Se você está familiarizado com o Python, você deve ter o hábito de adicionar métodos `__str__()` às suas classes, não métodos `__unicode__()`. Nós usamos `__unicode__()` aqui porque os models do Django lidam com Unicode por padrão. Todos os dados armazenados nos seus bancos de dados são convertidos para Unicode quando são retornados.

Os models do Django têm um método padrão `__str__()` que chama o `__unicode__()` e converte os resultados para uma bytestring UTF-8. Isso significa que `unicode(p)` irá retornar uma string Unicode, e que `str(p)` irá retornar uma string normal, com caracteres codificados como UTF-8.

Se tudo isso é grego para você, apenas lembre-se de adicionar métodos `__unicode__()` aos seus models. Com alguma sorte, as coisas deverão simplesmente funcionar para você.

Note que esses são métodos Python normais. Vamos adicionar um método personalizado, apenas por demonstração:

```

import datetime
# ...
class Poll(models.Model):
    # ...
    def was_published_today(self):
        return self.pub_date.date() == datetime.date.today()

```

Note a adição de `import datetime` para referenciar o módulo padrão do Python `datetime`.

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

```

>>> from mysite.polls.models import Poll, Choice

# Certifique-se de que a sua adição do __unicode__() funcionou.
>>> Poll.objects.all()
[<Poll: What's up?>]

# O Django fornece uma rica API de procura em banco de dados inteiramente
# controlada por argumentos formados por palavras-chave.
>>> Poll.objects.filter(id=1)

```

```
[<Poll: What's up?>]
>>> Poll.objects.filter(question__startswith='What')
[<Poll: What's up?>]

# Pegue a enquete cujo ano é 2007. Claro que se você estiver seguindo este
# tutorial em outro ano, altere como for apropriado.
>>> Poll.objects.get(pub_date__year=2007)
<Poll: What's up?>

>>> Poll.objects.get(id=2)
Traceback (most recent call last):
...
DoesNotExist: Poll matching query does not exist.

# Buscar por uma chave primária é o caso mais comum, então o Django fornece
# um atalho para buscas por chaves primárias específicas.
# O que se segue é idêntico a Poll.objects.get(id=1).
>>> Poll.objects.get(pk=1)
<Poll: What's up?>

# Certifique-se de que o método personalizado funcionou.
>>> p = Poll.objects.get(pk=1)
>>> p.was_published_today()
False

# Dê à Poll algumas Choices. A chamada create constrói um novo objeto
# choice, executa a instrução INSERT, adiciona a choice ao conjunto de
# choices disponíveis e retorna o novo objeto choice.
>>> p = Poll.objects.get(pk=1)
>>> p.choice_set.create(choice='Not much', votes=0)
<Choice: Not much>
>>> p.choice_set.create(choice='The sky', votes=0)
<Choice: The sky>
>>> c = p.choice_set.create(choice='Just hacking again', votes=0)

# Objetos Choice possuem acesso via API aos seus objetos Poll relacionados.
>>> c.poll
<Poll: What's up?>

# E vice-versa: Objetos Poll possuem acesso aos objetos Choice.
>>> p.choice_set.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
>>> p.choice_set.count()
3

# A API segue automaticamente os relacionamentos da forma como você precisa.
# Use underscores duplos para separar relacionamentos.
# Isso funciona a tantos níveis abaixo quanto precisar; não há limite.
# Encontre todas as opções para uma enquete cuja data de publicação for de
# 2007.
>>> Choice.objects.filter(poll__pub_date__year=2007)
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]

# Vamos apagar uma das escolhas. Use delete() para isso.
>>> c = p.choice_set.filter(choice__startswith='Just hacking')
>>> c.delete()
```

Para um detalhamento completo da API de banco de dados, veja nossa [referência à API de Banco de Dados](#).

Quando você se sentir confortável com a API, leia a [parte 2 deste tutorial](#) para colocar a administração automática do Django em funcionamento.

Escrevendo sua primeira aplicação Django, parte 2

Este tutorial inicia-se onde o *Tutorial 1* terminou. Vamos continuar a aplicação web de enquete focando agora no site de administração automática do Django.

Filosofia

Gerar sites de administração para sua equipe ou clientes adicionarem, editarem ou excluïrem conteúdo é um trabalho entediante que não requer muita criatividade. Por essa razão, o Django automatiza toda a criação da interface de administração para os models.

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

Ative o site de administração do Django

O site de administração não vem ativado por padrão – ele é opcional. Para ativá-lo para sua instalação, siga estes três passos:

- Adicione `"django.contrib.admin"` às suas configurações de `INSTALLED_APPS`.
- Execute `python manage.py syncdb`. Já que uma nova aplicação foi adicionada ao `INSTALLED_APPS`, as tabelas do banco de dados precisam ser atualizadas.
- Edite seu arquivo `meusite/urls.py` e retire o comentário das linhas abaixo de “Uncomment the next two lines...”. Esse arquivo é um URLconf; entraremos em detalhes sobre URLconfs no próximo tutorial. Por enquanto, tudo o que você tem que saber é que ele mapeia as URLs principais para as aplicações. No final você deve ter um `urls.py` parecido com este:

```
from django.conf.urls.defaults import *

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()
```

```
urlpatterns = patterns('',
    # Example:
    # (r'^mysite/', include('mysite.foo.urls')),

    # Uncomment the admin/doc line below and add 'django.contrib.
    ↪admin docs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admin docs.urls')),

    # Uncomment the next line to enable the admin:
    (r'^admin/(.*)', admin.site.root),
)
```

(As linhas em negrito são as que precisam ser descomentadas.)

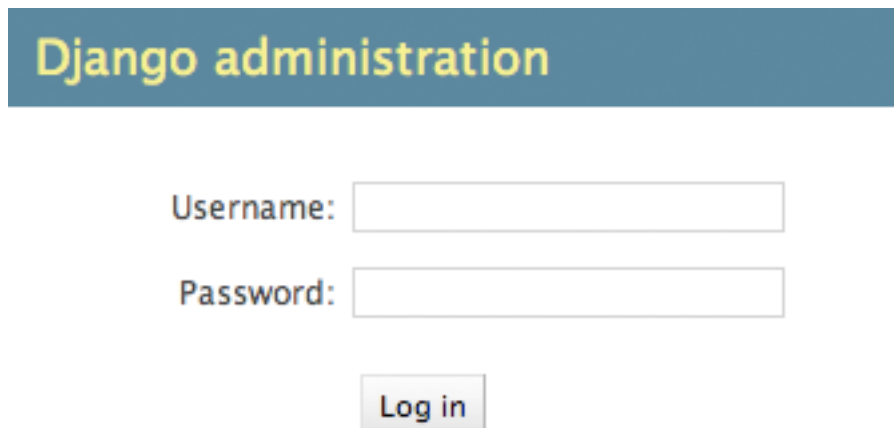
Inicie o servidor de desenvolvimento

Vamos iniciar o servidor de desenvolvimento e explorar o site de administração.

Lembre-se do Tutorial 1 onde você iniciou o servidor com:

```
python manage.py runserver
```

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



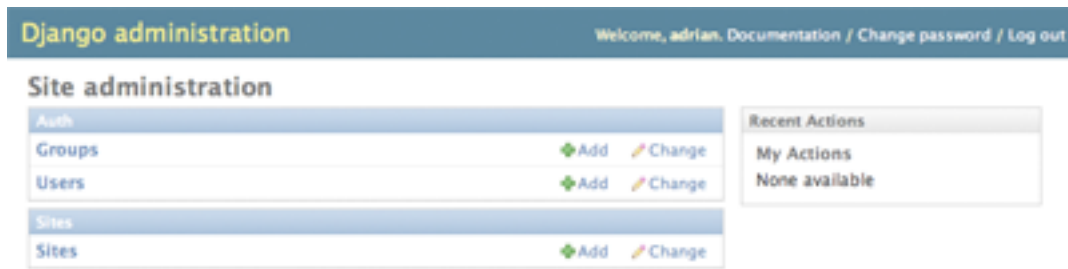
Django administration

Username:

Password:

Entre no site de administração

Agora tente acessar o sistema. (Você criou uma conta de superusuário na primeira parte deste tutorial, lembra? Se você não criou ou esqueceu a senha você pode *criar novamente*.) Você deverá ver a página inicial do admin do Django:



Você deverá ver alguns outros tipos de conteúdos editáveis, incluindo grupos, usuários e sites. Essas são as funcionalidades centrais que o Django inclui por padrão.

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 admin.

Apenas uma coisa a ser feita: Nós temos que dizer para o site de administração que os objetos “Poll” possuem uma interface de administração. Para fazer isto, crie um arquivo chamado `admin.py` em seu diretório `polls`, e edite para que se pareça com isto:

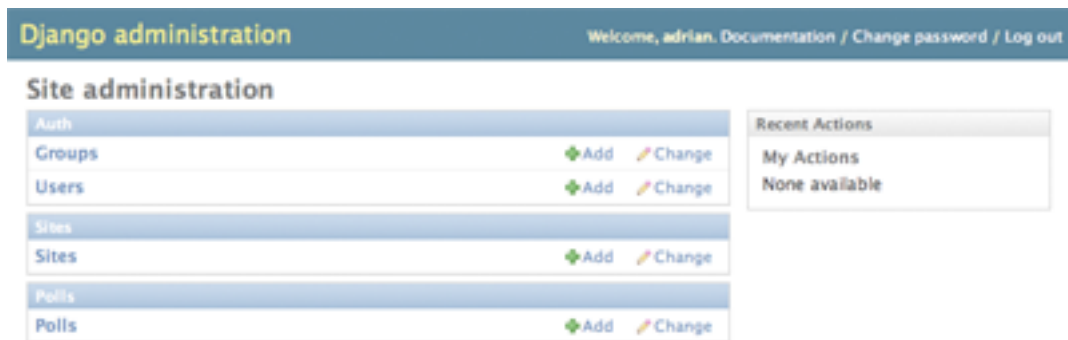
```
from mysite.polls.models import Poll
from django.contrib import admin

admin.site.register(Poll)
```

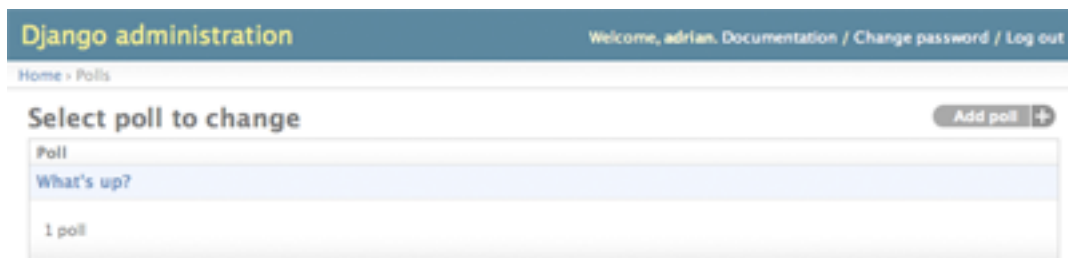
Você precisará reiniciar o servidor de desenvolvimento para ver suas modificações. Normalmente, o servidor recarrega automaticamente toda a vez que você modifica um arquivo, mas a ação de criar um novo arquivo não dispara a lógica de recarga automática.

Explore de graça a funcionalidade de administração

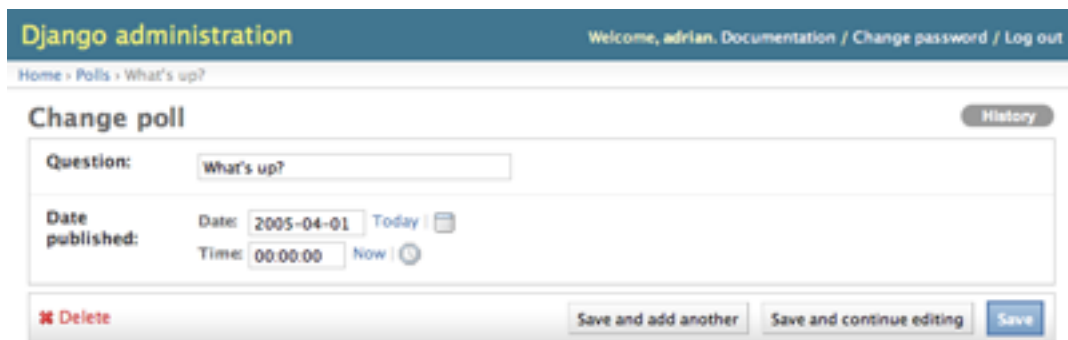
Agora que nós registramos `Poll`, o Django sabe que ela deve ser exibida na página principal do site de administração:



Clique em “Polls”. Agora você está na página “change list” (lista de edição) para as enquetes. Essa página exhibe todas as enquetes do banco de dados e deixa que você escolha uma para alterar. Existe a enquete “What’s up?” criada no primeiro tutorial:



Clique na enquete “What’s up?” para editá-la:



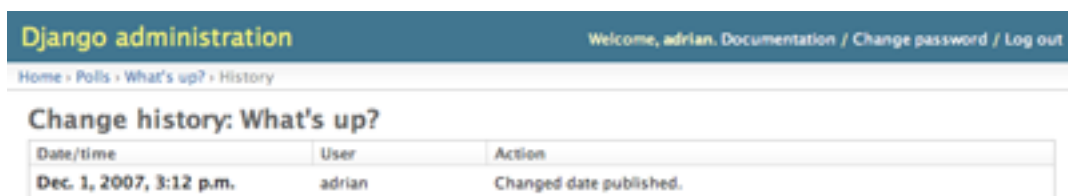
Note que:

- O formulário é gerado automaticamente para o model `Poll`.
- Os diferentes tipos de campos (`DateTimeField`, `CharField`) correspondem aos respectivos widgets 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 à change list para este tipo de objeto.
- Salvar e continuar editando – Salva as alterações e retorna à página para este objeto.
- Salvar e adicionar outro – Salva as informações e abre um formulário em branco para este tipo de objeto.
- Deletar – Exibe uma página de confirmação de exclusão.

Altere a “Publication date” 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:



Personalize o formulário de administração

Tire alguns minutos para apreciar todo o código que você não teve que escrever. Ao registrar o model `Poll` com `admin.site.register(Poll)`, o Django foi capaz de construir uma representação de formulário padrão.

Frequentemente, você desejará controlar como o site de administração se parecerá e funcionará. Você fará isso informando ao Django sobre as opções que você quer quando registra o objeto.

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

```
class PollAdmin(admin.ModelAdmin):
    fields = ['pub_date', 'question']

admin.site.register(Poll, PollAdmin)
```

Você seguirá este padrão – crie um objeto “model admin” e então o passe como segundo argumento para o `admin.site.register()` – sempre que precisar alterar as opções do admin para um objeto.

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

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 (fieldsets):

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

admin.site.register(Poll, PollAdmin)
```

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

Django administration

Welcome, adrian. Documentation / Change password / Log out

Home > Polls > What's up?

Change poll History

Question:

Date information

Date published: Date: Today
Time: Now

✖ Delete Save and add another Save and continue editing Save

Você pode atribuir classes HTML arbitrárias para cada grupo. O Django fornece uma classe "collapse" que exibe um grupo particular inicialmente recolhido. Isso é útil quando você tem um formulário longo que contém um grupo de campos que não são comumente utilizados:

```
class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
```

Django administration

Home > Polls > What's up?

Change poll

Question:

Date information (Show)

✖ Delete

Adicionando objetos relacionados

OK, temos nossa página de administração de Enquetes. Mas uma Poll tem múltiplas Choices, e a página de administração não exibe as opções.

Ainda.

Há duas formas de solucionar esse problema. A primeira é registrar Choice no site de administração, assim como fizemos com Poll. Isto é fácil:

```
from mysite.polls.models import Choice


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:

Django administration

Home > Choices > Add choice

Add choice

Poll:	<input type="text" value="What's up?"/> 
Choice:	<input type="text"/>
Votes:	<input type="text"/>

Nesse formulário, o campo “Poll” 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 “Poll”. 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 poll”. 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 Poll. Vamos fazer isso acontecer.

Remova a chamada `register()` do model Choice. Então edite o código de registro de Poll para que fique assim:

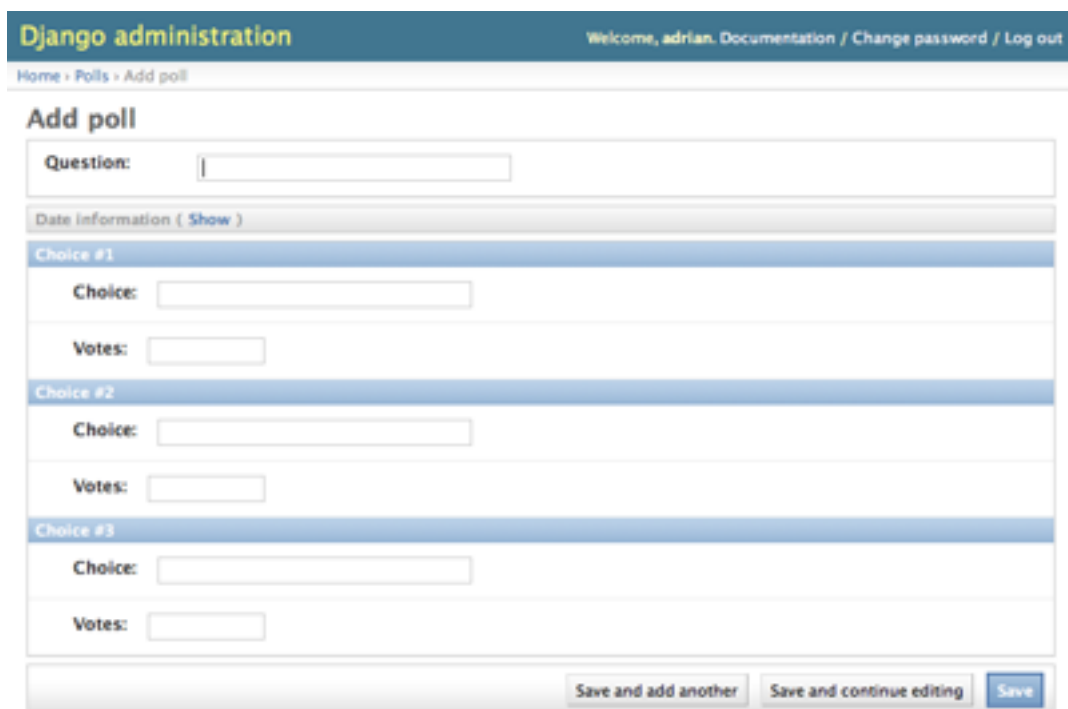
```
class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]

admin.site.register(Poll, PollAdmin)
```

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

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



The screenshot shows the Django administration interface for adding a new poll. The header bar is blue with the Django logo and the text 'Django administration'. On the right, it says 'Welcome, adrian. Documentation / Change password / Log out'. Below the header, there's a breadcrumb trail: 'Home > Polls > Add poll'. The main form is titled 'Add poll'. It has a 'Question' field with a text input. Below that is a 'Date information' section with a 'Show' link. There are three 'Choice' sections, each with a 'Choice' field and a 'Votes' field. At the bottom of the form, there are three buttons: 'Save and add another', 'Save and continue editing', and 'Save'.

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

No entanto, há um pequeno problema. Um monte de espaço na tela é tomado para exibir todos os três objetos Choice relacionados a serem inseridos. Por essa razão, o Django oferece uma maneira alternativa para exibir cada objeto relacionado em uma única linha; você só precisa alterar a declaração `ChoiceInline` para que seja:

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

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

Add poll

Question:

Date information ([Show](#))

Choices	
Choice	Votes
<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>

Personalize a página “change list”

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

Aqui como ela está até agora:

Django administration Welcome, [adrian](#). [Documentation](#) / [Change password](#) / [Log out](#)

[Home](#) » [Polls](#)

Select poll to change [Add poll](#) [+](#)

Poll
What's up?
1 poll

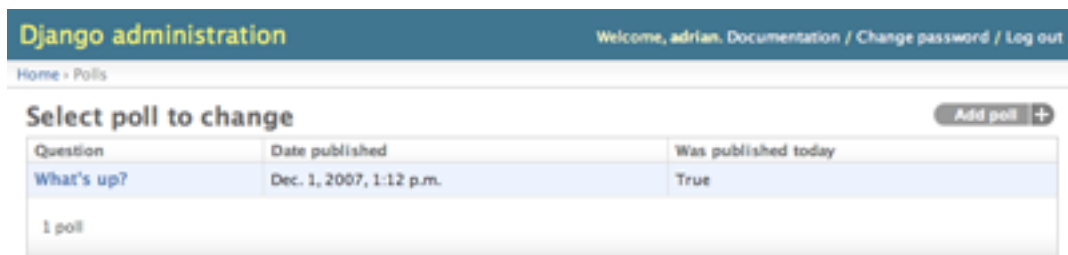
Por padrão, o Django mostra o `str()` de cada objeto. Mas algumas vezes seria mais útil se pudéssemos mostrar campos individuais. Para fazer isso, use a opção `list_display`, que é uma tupla de nomes de campos a serem exibidos, como colunas, na change list dos objetos:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date')
```

Apenas para facilitar, vamos incluir o método personalizado `was_published_today` do Tutorial 1:

```
class PollAdmin(admin.ModelAdmin):
    # ...
    list_display = ('question', 'pub_date', 'was_published_today')
```

Agora a página de edição de enquetes está assim:



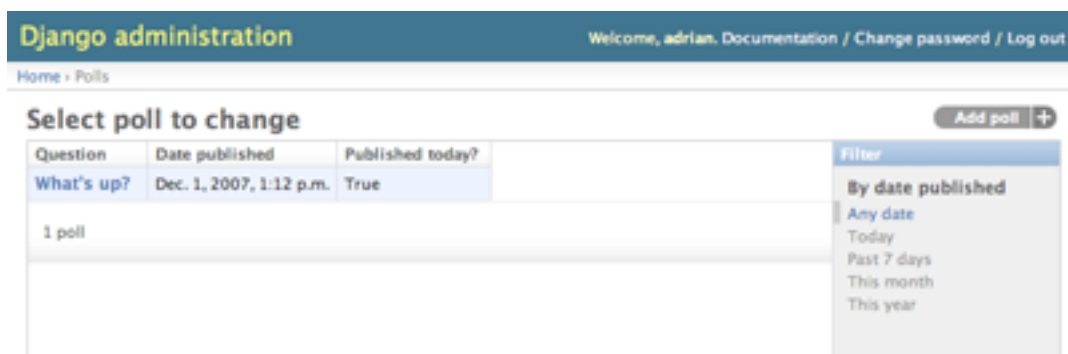
Você pode clicar no cabeçalho da coluna para ordená-las por estes valores – exceto no caso do `was_published_today`, porque a ordenação pelo resultado de um método arbitrário não é suportada. Também note que o cabeçalho da coluna para `was_published_today` é, por padrão, o nome do método (com underscores substituídos por espaços). Mas você pode alterar isso fornecendo ao método (em `models.py`) um atributo `short_description`:

```
def was_published_today(self):
    return self.pub_date.date() == datetime.date.today()
was_published_today.short_description = 'Published today?'
```

Vamos adicionar uma outra melhoria à lista de edição: Filtros. Adicione a seguinte linha ao `PollAdmin`:

```
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 de filtro exibido depende do tipo de campo que você está filtrando. Devido ao `pub_date` ser um `Date-TimeField`, o Django sabe dar as opções de filtro para `Date-TimeFields`: “Any date,” “Today,” “Past 7 days,” “This month,” “This year.”

Isso está tomando uma boa forma. Vamos adicionar alguma capacidade de pesquisa:

```
search_fields = ['question']
```

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`. Você pode usar quantos campos quiser – entretanto, por ele usar um comando `LIKE` internamente, seja moderado para manter seu banco de dados feliz.

Finalmente, porque os objetos `Poll` possuem datas, é conveniente acompanhar por data. Adicione esta linha:

```
date_hierarchy = 'pub_date'
```

Isso adiciona uma navegação hierárquica, por data, no topo da change list. No nível mais acima, ele exibe todos os anos disponíveis. Então desce para os meses e, por último, os dias.

Agora é também uma boa hora para observar que a change list fornece uma paginação gratuitamente. O padrão é mostrar 50 itens por página. A change list, campos de pesquisa, filtros, hierarquia por data, 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 admin é ridículo. Isso é só um texto de exemplo.

É fácil de editar, no entanto, usando o sistema de templates do Django. O Django admin é feito com o próprio Django e conseqüentemente sua interface usa o sistema de templates nativo do Django. (How meta!)

Abra seu arquivo de configurações (`mysite/settings.py`, lembre-se) e veja a configuração `TEMPLATE_DIRS`. `TEMPLATE_DIRS` é uma tupla de diretórios de arquivos que serão checados quando carregar os templates do Django, ou seja, caminhos de busca.

Por padrão, `TEMPLATE_DIRS` vem vazio. Portanto, vamos adicionar uma linha para dizer onde nossos templates Django estão:

```
TEMPLATE_DIRS = (
    "/home/my_username/mytemplates", # Change this to your own directory.
)
```

Agora copie o template `admin/base_site.html` de dentro do diretório padrão do Django admin (`django/contrib/admin/templates`) em um subdiretório admin onde quer que esteja o diretório que você está usando em `TEMPLATE_DIRS`. Por exemplo, se seu `TEMPLATE_DIRS` inclui `"/home/my_username/mytemplates"`, como acima, então copie `django/contrib/admin/templates/admin/base_site.html` para `"/home/my_username/mytemplates/admin/base_site.html`. Não se esqueça do subdiretório admin.

Então simplesmente edite o arquivo e substitua o texto genérico do Django com o nome do seu próprio site como desejar.

Note que qualquer template padrão do admin 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.

Leitores astutos irão se perguntar: Mas se `TEMPLATE_DIRS` estava vazio por padrão, como o Django pôde encontrar o diretório padrão dos templates do admin? A resposta é, por padrão, o Django irá automaticamente procurar por um subdiretório `templates/` dentro de cada pacote de aplicação, para usar como fallback. Veja a [documentação do template loader](#) - para a informação completa.

Personalize a página inicial de administração

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

Por padrão, ele exibe todas as aplicações em `INSTALLED_APPS`, que estão registrados na aplicação admin, 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 a ser personalizado é o `admin/index.html` (faça o mesmo que foi feito com o `admin/base_site.html` na seção anterior – copie ele do diretório padrão para o seu próprio diretório de templates). Edite o arquivo, e você verá que ele usa uma variável de template chamada `app_list`. Esta variável contém todas as aplicações instaladas no Django. Em vez de usá-la, você pode explicitamente criar os links para os objetos específicos na página de administração da maneira que você achar mais apropriado.

Quando você se sentir confortável com o site de administração, leia a [parte 3 deste tutorial](#) para começar a trabalhar com parte pública da enquete.

Escrevendo sua primeira aplicação 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”.

Filosofia

Uma view é um “tipo” de página em sua aplicação Django que em geral serve 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 detalhamento de um único artigo;
- Página de arquivo por ano - exibe todos os meses com artigos de um determinado ano;
- Página de arquivo por mês - exibe todos os dias com artigos de 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 “arquivo” de enquetes - exibe as enquetes mais recentes;
- Página de “detalhes” da enquete - exibe questões para a enquete, sem os resultados mas com um formulário para votar;
- Página de “resultados” de enquetes - exibe os resultados de uma enquete em particular;
- Ação de voto - permite a votação para uma escolha particular em uma enquete em particular.

No Django, cada view é representada por uma função simples em Python.

Monte suas URLs

O primeiro passo para escrever views é montar sua estrutura de URLs. Você faz isso criando um módulo em Python, chamado de URLconf. URLconfs são como o Django associa uma URL a um código em Python.

Quando um usuário requisita uma página construída em Django, o sistema verifica a variável `ROOT_URLCONF`, que contém uma string do caminho de um módulo Python. O Django carrega esse módulo e verifica se o mesmo possui uma variável chamada `urlpatterns`, que é uma sequência de tuplas no seguinte formato:

```
(expressão regular, função Python de resposta [, dicionário opcional])
```

O Django começa pela primeira expressão regular e percorre até o final da lista, comparando a URL requisitada contra cada expressão regular até encontrar uma que combine.

Quando encontra uma que combine, o Django chama a função Python de resposta, com um objeto `HttpRequest` como seu primeiro argumento, quaisquer valores “capturados” da expressão regular como argumentos chave e opcionalmente, os elementos chaves do dicionário informado na URL (um terceiro item opcional da tupla).

Para saber mais sobre objetos `HttpRequest`, veja a documentação sobre *Objetos de requisição e resposta*. Para maiores detalhes sobre a URLconf, veja *URL dispatcher*.

Quando você executou `django-admin.py startproject mysite` no início do Tutorial 1, ele criou uma URLconf padrão em `mysite/urls.py`. E também fixou automaticamente a variável `ROOT_URLCONF` (em `settings.py`) para apontar para este arquivo:

```
ROOT_URLCONF = 'mysite.urls'
```

Pausa para um exemplo. Edite `mysite/urls.py` e deixe como abaixo:

```
from django.conf.urls.defaults import *

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^polls/$', 'mysite.polls.views.index'),
    (r'^polls/(?P<poll_id>\d+)/$', 'mysite.polls.views.detail'),
    (r'^polls/(?P<poll_id>\d+)/results/$', 'mysite.polls.views.results'),
    (r'^polls/(?P<poll_id>\d+)/vote/$', 'mysite.polls.views.vote'),
    (r'^admin/(.*)', admin.site.root),
)
```

Aqui vale uma revisão. Quando alguém solicita uma página do seu web site – diz, “/polls/23/”, o Django carregará este módulo em Python, porque ele foi apontado pela variável `ROOT_URLCONF`. Ele acha a variável chamada `urlpatterns` e percorre as expressões regulares na ordem. Quando ele encontra uma expressão regular que combine – `r'^polls/(?P<poll_id>\d+)/$'` – ele carrega a função `detail()` de `mysite.polls.views.detail`. Que corresponde à função `detail()` em `mysite/polls/views.py`. Finalmente, ele chama a função `detail()` como a seguir:

```
detail(request=<HttpRequest object>, poll_id='23')
```

A parte `poll_id='23'` vem de `(?P<poll_id>\d+)`. Usando parênteses em torno de um padrão “captura” o texto casado por este padrão e envia-o como um argumento da função; a `?P<poll_id>` define o nome que será usado para identificar o padrão casado; e `\d+` é a expressão regular para casar uma sequência de dígitos (ex., um número).

Como os padrões de URL são expressões regulares, realmente não há limites para o que você possa fazer com elas. E também não é necessário adicionar extensão na URL como `.php` – a menos que você tenha um sinistro senso de humor, neste caso você pode fazer algo como:

```
(r'^polls/latest\.php$', 'mysite.polls.views.index'),
```

Mas, não o faça isso, Isto é idiota.

Note que aquelas expressões regulares não procuram por parâmetros de GET e POST ou nome de domínios. Por exemplo, em uma requisição para `http://www.example.com/myapp/`, a URLconf irá procurar por `/myapp/`. Numa requisição para `http://www.example.com/myapp/?page=3`, a URLconf irá procurar por `/myapp/`.

Se você precisar de ajuda com expressões regulares, veja nesta [pagina da Wikipedia](#) e na [documentação do Python](#). Também o livro da “O’Reilly “Mastering Regular Expressions”, de Jeffrey Friedl que é fantástico.

Finalmente, uma nota de performance: essas expressões regulares são compiladas na primeira vez que o módulo URLconf é carregado. Elas são super rápidas.

Escreva sua primeira view

Bem, nós não criamos nenhuma view ainda – nós só temos a URLconf. Mas vamos ter certeza de que o Django está seguindo a URLconf apropriadamente.

Rode o servidor web de desenvolvimento do Django:

```
python manage.py runserver
```

Agora vá para “<http://localhost:8000/polls/>” no seu domínio em seu navegador web. Você deverá ver uma amigavelmente colorida página de erro com a seguinte mensagem:

```
ViewDoesNotExist at /polls/

Tried index in module mysite.polls.views. Error was: 'module'
object has no attribute 'index'
```

Este erro ocorreu porque você não escreveu uma função `index()` no módulo `mysite/polls/views.py`.

Tente “[/polls/23/](#)”, “[/polls/23/results/](#)” e “[/polls/23/vote/](#)”. As mensagens de erro dirão a você qual view o Django tentou (e não conseguiu encontrar, porque você não a escreveu nenhuma view ainda).

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

```
from django.http import HttpResponseRedirect

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

Esta é a view mais simples possível. Vá para `/polls/` em seu navegador, e você irá ver seu texto.

Agora adicione a seguinte view. Esta é ligeiramente diferente, porque ela recebe um argumento (que, lembre-se, é passado com o que foi capturado pela expressão regular na URLconf):

```
def detail(request, poll_id):
    return HttpResponseRedirect("You're looking at poll %s." % poll_id)
```

Dê uma olhada no seu navegador, em `/polls/34/`. Ele vai mostrar o ID que você informou na URL.

Escreva views que façam algo

Cada view é responsável por fazer uma das duas coisas: retornar 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 retorne um [HttpResponse](#). Ou uma exceção.

Por conveniência, vamos usar a própria API de banco de dados do Django, que nós vimos no [Tutorial 1](#). Aqui está uma tentativa para a view `index()`, que mostra as últimas 5 enquetes no sistema, separadas por vírgulas, de acordo com a data de publicação:

```
from mysite.polls.models import Poll
from django.http import HttpResponse

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    output = ', '.join([p.question for p in latest_poll_list])
    return HttpResponse(output)
```

Há um problema aqui, todavia: o design da página esta 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:

```
from django.template import Context, loader
from mysite.polls.models import Poll
from django.http import HttpResponse

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    t = loader.get_template('polls/index.html')
    c = Context({
        'latest_poll_list': latest_poll_list,
    })
    return HttpResponse(t.render(c))
```

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

Recarregue a página. Agora você verá um erro:

```
TemplateDoesNotExist at /polls/
polls/index.html
```

Ah. Não há template ainda. Primeiro, crie um diretório, em algum lugar de seu sistema de arquivos, Cujo o conteúdo o Django possa acessar. (O Django roda com qualquer usuário que seu servidor rodar.) Portanto não o crie dentro da raiz de documentos do seu servidor. Você provavelmente não gostaria torna-lo público, apenas por questões de segurança. Agora edite a variável `TEMPLATE_DIRS` em seu `settings.py` para dizer ao Django onde ele pode encontrar os templates - exatamente como você fez na seção “Customize a aparência do site de administração” do Tutorial 2.

Quando você tiver feito isto, crie um diretório `polls` em seu diretório de templates. Dentro, crie um arquivo chamado `index.html`. Note que nosso código `loader.get_template('polls/index.html')` acima direciona para “[`template_directory`]/`polls/index.html`” no sistema de arquivos.

Ponha o seguinte código no arquivo do template:

```
{% if latest_poll_list %}
<ul>
  {% for poll in latest_poll_list %}
    <li>{{ poll.question }}</li>
  {% endfor %}
</ul>
{% else %}
  <p>No polls are available.</p>
{% endif %}
```

Carregue a página em seu navegador, e você verá uma lista contendo a enquete “What’s up” do Tutorial 1.

Um atalho: `render_to_response()`

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:

```
from django.shortcuts import render_to_response
from mysite.polls.models import Poll

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    return render_to_response('polls/index.html', {'latest_poll_list': latest_poll_list})
```

Note que uma vez que você tiver feito isto em todas as views, nós não vamos mais precisar importar `loader`, `Context` e `HttpResponse`.

A função `render_to_response()` 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 exceção 404

Agora, vamos atacar a view de detalhe da enquete - a página que mostra as questões para uma enquete lançada. Aqui esta a view:

```
from django.http import Http404
# ...
def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render_to_response('polls/detail.html', {'poll': p})
```

O novo conceito aqui: a view levanta a 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 rapidamente ter o exemplo acima funcionando, é só:

```
{{ poll }}
```

irá ajudar por agora.

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 esta a view `detail()`, reescrita:

```
from django.shortcuts import render_to_response, get_object_or_404
# ...
def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/detail.html', {'poll': p})
```

A função `get_object_or_404()` recebe um modelo do Django como primeiro argumento e um número arbitrário de argumentos chave, 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 model 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.

Há também a função `get_list_or_404()`, que trabalhada 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.

Escreva uma view do código 404 (página não encontrada)

Quando você levanta `Http404` de dentro de uma view, o Django carregará uma view especial devotada para manipulação de erros 404. Ela é encontrada procurando pela variável `handler404`, que é uma string em Python com o caminho da view separada por pontos - o mesmo formato que a função de chamada das `URLconf` usa. Uma view 404 por si só não tem nada de especial: é apenas uma view normal.

Você normalmente não terá de se incomodar em escrever views de 404. Por padrão, as `URLconfs` têm a seguinte linha no começo do arquivo:

```
from django.conf.urls.defaults import *
```

Que se encarrega de definir `handler404` no módulo corrente. Como você pode ver em `django/conf/urls/defaults.py`, `handler404` é definido para `django.views.defaults.page_not_found()` por padrão.

Mais quatro coisas a se notar sobre views 404:

- Se o `DEBUG` for definido como `True` (no seu módulo settings) sua view 404 nunca será usada (e assim o seu template `404.html` nunca será renderizado) porque o traceback será exibido em seu lugar.
- A view 404 também é chamada se o Django não conseguir encontrar uma combinação depois de checar todas as expressões regulares da `URLconf`;
- Se você não definir sua própria view 404 e simplesmente usar a padrão, que é recomendável – você terá uma obrigação: criar um template `404.html` na raiz de seu diretório de templates. A view padrão de 404 irá usar este template para todos os erros 404;
- Se o `DEBUG` for definido como `False` (no seu módulo settings) e se você não criou um arquivo `404.html`, um `Http500` será levantado em seu lugar. Então lembre-se de criar um `404.html`.

Escreva uma view do código 500 (erro no servidor)

Similarmente, `URLconfs` pode definir uma `handler500`, que aponta para uma view a ser chamada em caso de erro no servidor. Erros de servidor acontecem quando você tem erros de tempo de execução no código da view.

Use o sistema de templates

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

```
<h1>{{ poll.question }}</h1>
<ul>
{% for choice in poll.choice_set.all %}
    <li>{{ choice.choice }}</li>
{% endfor %}
</ul>
```

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

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.

Simplificando as URLconfs

Vamos tomar um tempo para brincar em torno das views e o sistema de templates. A medida que você edita a URLconf, você poderá notar uma leve redundância nela:

```
urlpatterns = patterns('',
    (r'^polls/$', 'mysite.polls.views.index'),
    (r'^polls/(?P<poll_id>\d+)/$', 'mysite.polls.views.detail'),
    (r'^polls/(?P<poll_id>\d+)/results/$', 'mysite.polls.views.results'),
    (r'^polls/(?P<poll_id>\d+)/vote/$', 'mysite.polls.views.vote'),
)
```

Isto é, `mysite.polls.views` está em todas as chamadas.

Porque este é um caso comum, o framework de URLconf provê um atalho para prefixos comuns. Você pode decompor em fatores os prefixos comuns e adicioná-los como primeiro argumento de `patterns()`, como abaixo:

```
urlpatterns = patterns('mysite.polls.views',
    (r'^polls/$', 'index'),
    (r'^polls/(?P<poll_id>\d+)/$', 'detail'),
    (r'^polls/(?P<poll_id>\d+)/results/$', 'results'),
    (r'^polls/(?P<poll_id>\d+)/vote/$', 'vote'),
)
```

Isto é funcionalmente idêntico a formatação anterior. Somente um pouco mais arrumado.

Desacoplando as URLconfs

Já que nós estamos aqui, nós devemos tomar um tempo para desacoplar as URLs da aplicação `poll` da configuração do nosso projeto Django. As aplicações no Django devem ser escritas para serem plugáveis – que, cada aplicação em particular deve ser transferível para qualquer outra instalação Django com o mínimo de esforço.

Nossa aplicação `poll` é agradavelmente desacoplada neste ponto, graças à estrita estrutura de diretórios que o comando `python manage.py startapp` criou, mas uma parte dela está acoplada às configurações do Django: A URLconf.

Nós vínhamos editando as URLs em `mysite/urls.py`, mas o projeto de URLs de uma aplicação é específicas para ela, não para a instalação do Django – vamos então mover as URLs para dentro do diretório da aplicação.

Copie o arquivo `mysite/urls.py` para `mysite/polls/urls.py`. Então, modifique o `mysite/urls.py` para remover as URLs específicas da aplicação `poll` e insira um `include()`:

```
...
urlpatterns = patterns('',
    (r'^polls/', include('mysite.polls.urls')),
    ...
)
```

O `include()` simplesmente referencia outra URLconf. Observe que a expressão regular não tem um `$` (caracter que combina com o fim da string) mas possui uma barra. Em qualquer momento o Django encontra o

`include()`, ele decepta fora qualquer parte da URL que tiver combinado ate este ponto e envia a string restante para a URLconf incluída para um tratamento posterior.

Aqui o que acontece se um usuário vai para `"/polls/34/"` neste sistema:

- O Django irá encontrar uma combinacao em `'^polls/'`
- Ele irá separar o texto combinado (`"polls/"`) e enviar o restante do texto – `"34/"` – para a URLconf `'mysite.polls.urls'` para um tratamento posterior.

Agora que nós desacoplamos isto, nós precisamos desacoplar a URLconf `'mysite.polls.urls'` removendo o trecho `"polls/"` inicial de cada linha, e removendo as linhas registrando o site administrativo:

```
urlpatterns = patterns('mysite.polls.views',
    (r'^$', 'index'),
    (r'^(?P<poll_id>\d+)/$', 'detail'),
    (r'^(?P<poll_id>\d+)/results/$', 'results'),
    (r'^(?P<poll_id>\d+)/vote/$', 'vote'),
)
```

A idéia por trás do `include()` e o desacoplamento da URLconf é criar facilmente URLs plug-and-play. Agora que as polls possuem sua própria URLconf, elas podem ser colocadas abaixo de `"/polls/"`, ou abaixo de `"/fun_polls/"`, ou abaixo de `"/content_polls/"`, ou qualquer outra raiz de URL, que a aplicação irá continuar funcionando.

Toda a aplicação poll preocupa-se com URLs relativas, e não com URLs absolutas.

Quando você estiver confortável em escrever views, leia a [a parte 4 deste tutorial](#) para aprender sobre processamento de formulários simples e sobre as views genéricas.

Escrevendo sua primeira aplicação Django, parte 4

Este tutorial inicia-se onde o *Tutorial 3* terminou. Estamos continuando a aplicação web de enquete e concentraremos em uma forma simples de processamento de formulário e a redução de nosso código.

Escreva um simples formulário

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

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

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

<form action="/polls/{{ poll.id }}/vote/" method="post">
{% for choice in poll.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ _
↪choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice }}</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 `choice=3` por POST. Este são os formulários HTML 101.
- Nós definimos o parâmetro `action` do formulário para `/polls/{{ poll.id }}/vote/`, e definimos `method=post`. Usando `method=post` (em vez de `method=get`) é muito importante, porque o ato de enviar este formulário irá alterar dados do lado servidor. Sempre que criar um formulário que modifique os dados do lado do servidor, utilize `method="post"`. Esta dica não é específica do Django; mais sim uma boa prática para o desenvolvimento é Web.
- `forloop.counter` indica quantas vezes a tag `:tag‘for‘` atravessou o seu ciclo.

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:

```
(r'^(?P<poll_id>\d+)/vote/$', 'vote'),
```

Então, vamos criar um função `vote()` em `mysite/polls/views.py`:

```
from django.shortcuts import get_object_or_404, render_to_response
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from mysite.polls.models import Choice, Poll
# ...
def vote(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    try:
        selected_choice = p.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the poll voting form.
        return render_to_response('polls/detail.html', {
            'poll': p,
            '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('mysite.polls.views.results', args=(p.
↪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 pelas suas chaves. Neste caso, `request.POST['choice']` retorna o ID da opção selecionada, tal 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 `request.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 POST. O código acima checka 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 `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).

Como o comentário Python acima salienta, você deve sempre retornar uma `HttpResponseRedirect` depois de lidar com sucesso com dados POST. Esta dica não é específica do Django; mais sim uma boa prática para o desenvolvimento Web.

- Estamos usando a função `reverse()` no construtor do `HttpResponseRedirect` neste exemplo. Essa função ajuda a evitar ter que escrever na mão a URL na função de view. É dado o nome da view de que queremos passar o controle e uma porção da variável do padrão de URL que aponta para essa view. Neste caso, usando o URLConf criado no Tutorial 3, esta chamada a `reverse()` irá retornar uma string como:

```
'/polls/3/results/'
```

... onde o 3 é o valor de `p.id`. Esta URL redirecionada irá então chamar a view `'results'` afim de exibir a página final. Note que você precisará usar o nome completo da view aqui (incluindo o prefixo).

Como mencionado no Tutorial 3, `request` é um objeto `HttpRequest`. Para mais informações sobre o objeto `HttpRequest`, veja a [documentação do request e response](#).

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

```
def results(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/results.html', {'poll': p})
```

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 um template `results.html`:

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

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

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.

Use views genéricas: Menos código é melhor

As views `detail()` (do [Tutorial 3](#)) e `results()` são estupidamente simples – e, como já mencionado acima, são redundantes. A view, `index()` (também do [Tutorial 3](#)), que exibe uma lista de enquetes, é semelhante.

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.

Vamos converter a nossa aplicação de enquete para utilizar o sistema de views genéricas, por isso podemos excluir um monte do nosso próprio código. Iremos apenas ter que executar alguns passos para fazer a conversão. Nós iremos:

1. Converter a URLconf.
2. Renomear alguns templates.
3. Remover algumas views antigas e desnecessárias.
4. Corrigir a manipulação de URL para as novas views.

Leia a respeito para obter mais detalhes.

Por que o código se arrastou?

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.

Em primeiro lugar, abra a URLconf `polls/urls.py`. Ela esta assim, de acordo com o tutorial até o momento:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.polls.views',
    (r'^$', 'index'),
    (r'^(?P<poll_id>\d+)/$', 'detail'),
    (r'^(?P<poll_id>\d+)/results/$', 'results'),
    (r'^(?P<poll_id>\d+)/vote/$', 'vote'),
)
```

Modifique para ficar assim:

```
from django.conf.urls.defaults import *
from mysite.polls.models import Poll

info_dict = {
    'queryset': Poll.objects.all(),
}

urlpatterns = patterns('',
    (r'^$', 'django.views.generic.list_detail.object_list', info_dict),
    (r'^(?P<object_id>\d+)/$', 'django.views.generic.list_detail.object_detail',
    ↪info_dict),
    url(r'^(?P<object_id>\d+)/results/$', 'django.views.generic.list_detail.object_
    ↪detail', dict(info_dict, template_name='polls/results.html'), 'poll_results'),
    (r'^(?P<poll_id>\d+)/vote/$', 'mysite.polls.views.vote'),
)
```

Nós estamos usando duas views genéricas aqui: `object_list()` e `object_detail()`. 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 tipo de dado ela vai agir em cima. Esses dados são fornecidos em um dicionário. A chave `queryset` neste dicionário aponta para a lista de objetos a serem manipulados pela view genérica.
- A view genérica `object_detail()` espera o valor ID capturado da URL em `"object_id"`, de modo que alteramos `poll_id` para `object_id` para as views genérica.
- Nós adicionamos um nome, `poll_results`, para os resultados da view assim nós temos uma maneira de se referir à sua URL depois (veja a documentação sobre [naming URL patterns](#) para informações). Também estamos usando a função `url()` de `django.conf.urls.defaults` aqui. É um bom hábito usar `url()` quando você estiver fornecendo um padrão de nome como este.

Por padrão, a view genérica `object_detail()` utiliza um template chamado `<app name>/<model name>_detail.html`. Em nosso caso, ela vai utilizar o template `"polls/poll_detail.html"`. Assim, renomeie o seu template `polls/detail.html` para `polls/poll_detail.html` e altere a linha `render_to_response` em `vote()`.

Semelhantemente, a view genérica `object_list()` utiliza um template chamado `<app name>/<model name>_list.html`. Assim, renomeie `polls/index.html` para `polls/poll_list.html`.

Por termos mais de uma entrada na URLconf que usa `object_detail()` para a aplicação de enquete, nós especificamos manualmente um nome para a view de resultados: `template_name='polls/results.html'`. Caso contrário, as duas views utilizariam o mesmo template. Note que nós usamos `dict()` para retornar um dicionário alterado no lugar.

Note: `django.db.models.QuerySet.all()` é preguiçosa

Poderá parecer um pouco assustador ver `Poll.objects.all()` ser utilizado na view de detalhe que necessita apenas de um objeto `Poll` mas não se preocupe; `Poll.objects.all()` é na verdade um objeto especial chamado de *QuerySet*, que é “preguiçosa” e não toca no seu banco de dados a menos que seja absolutamente necessário. No momento em que a consulta ao banco de dados acontecer, a view genérica `object_detail()`

terá limitado o seu escopo para um único objeto, de modo que a eventual consulta só irá selecionar uma linha da base de dados.

Se você quiser saber mais sobre seu funcionamento, a documentação de banco de dados do Django [explica a natureza preguiçosa dos abjetos *QuerySet*](#).

Nas partes anteriores deste tutorial, os templates tem sido fornecidos com um `context` que contém as variáveis de contexto `poll` e `latest_poll_list`. No entanto, as views genérica fornecem as variáveis `object` e `object_list` como contexto. Portanto, você precisa mudar seus templates para combinar com as novas variáveis de contexto. Vá através de seus templates, e altere qualquer referência a `latest_poll_list` para `object_list` e altere qualquer referência de `poll` para `object`.

Agora você pode deletar as views, `index()`, `detail()` e `results()` do arquivo `polls/views.py`. Não precisamos delas mais – elas foram substituídas por views genéricas.

A view `vote()` ainda é necessária. No entanto, ela deve ser modificada para corresponder ao novo contexto de variáveis. Na chamada `render_to_response()`, renomeie a variável de contexto de `poll` para `object`.

A última coisa a fazer é corrigir o manipulador da URL para levar em conta a utilização de views genéricas. Na view `vote()` acima, usamos a função `reverse()` para evitar escrever na mão nossas URLs. Agora que mudamos para view genérica, nós vamos precisar alterar a chamada `reverse()` para apontar de volta para a nossa nova view genérica. Nós não podemos simplesmente utilizar a função `view` mais – views genéricas podem ser (e são) utilizada várias vezes - mas podemos usar o nome que foi dado:

```
return HttpResponseRedirect(reverse('poll_results', args=(p.id,)))
```

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

Para maiores detalhes sobre views genéricas, consulte a [documentação sobre views genéricas](#).

Brevemente

O tutorial acaba aqui por enquanto. Futuras partes deste tutorial cobrirão:

- Processamento avançado de formulário
- Usando o framework de RSS
- Usando o framework de cache
- Usando o framework de comentários
- Características avançadas da interface de administração: Permissões
- Características avançadas da interface de administração: JavaScript personalizado

Por enquanto, você pode querer verificar alguns pontos em [para onde ir a partir daqui](#)

CHAPTER 7

What to read next

So you’ve read all the *introductory material* and have decided you’d like to keep using Django. We’ve only just scratched the surface with this intro (in fact, if you’ve read every single word you’ve still read less than 10% of the overall documentation).

So what’s next?

Well, we’ve always been big fans of learning by doing. At this point you should know enough to start a project of your own and start fooling around. As you need to learn new tricks, come back to the documentation.

We’ve put a lot of effort into making Django’s documentation useful, easy to read and as complete as possible. The rest of this document explains more about how the documentation works so that you can get the most out of it.

(Yes, this is documentation about documentation. Rest assured we have no plans to write a document about how to read the document about documentation.)

Finding documentation

Django’s got a *lot* of documentation – almost 200,000 words – so finding what you need can sometimes be tricky. A few good places to start are the search and the genindex.

Or you can just browse around!

How the documentation is organized

Django’s main documentation is broken up into “chunks” designed to fill different needs:

- The *introductory material* is designed for people new to Django – or to web development in general. It doesn’t cover anything in depth, but instead gives a high-level overview of how developing in Django “feels”.
- The *topic guides*, on the other hand, dive deep into individual parts of Django. There are complete guides to Django’s *model system*, *template engine*, *forms framework*, and much more.

This is probably where you’ll want to spend most of your time; if you work your way through these guides you should come out knowing pretty much everything there is to know about Django.

- Web development is often broad, not deep – problems span many domains. We’ve written a set of *how-to guides* that answer common “How do I ...?” questions. Here you’ll find information about *generating PDFs with Django*, *writing custom template tags*, and more.

Answers to really common questions can also be found in the *FAQ*.

- The guides and how-to’s don’t cover every single class, function, and method available in Django – that would be overwhelming when you’re trying to learn. Instead, details about individual classes, functions, methods, and modules are kept in the *reference*. This is where you’ll turn to find the details of a particular function or whathaveyou.
- Finally, there’s some “specialized” documentation not usually relevant to most developers. This includes the *release notes*, *documentation of obsolete features*, *internals documentation* for those who want to add code to Django itself, and a *few other things that simply don’t fit elsewhere*.

How documentation is updated

Just as the Django code base is developed and improved on a daily basis, our documentation is consistently improving. We improve documentation for several reasons:

- To make content fixes, such as grammar/typo corrections.
- To add information and/or examples to existing sections that need to be expanded.
- To document Django features that aren’t yet documented. (The list of such features is shrinking but exists nonetheless.)
- To add documentation for new features as new features get added, or as Django APIs or behaviors change.

Django’s documentation is kept in the same source control system as its code. It lives in the `django/trunk/docs` directory of our Subversion repository. Each document online is a separate text file in the repository.

Where to get it

You can read Django documentation in several ways. They are, in order of preference:

On the Web

The most recent version of the Django documentation lives at <http://docs.djangoproject.com/en/dev/>. These HTML pages are generated automatically from the text files in source control. That means they reflect the “latest and greatest” in Django – they include the very latest corrections and additions, and they discuss the latest Django features, which may only be available to users of the Django development version. (See “Differences between versions” below.)

We encourage you to help improve the docs by submitting changes, corrections and suggestions in the [ticket system](#). The Django developers actively monitor the ticket system and use your feedback to improve the documentation for everybody.

Note, however, that tickets should explicitly relate to the documentation, rather than asking broad tech-support questions. If you need help with your particular Django setup, try the [django-users mailing list](#) or the [#django IRC channel](#) instead.

In plain text

For offline reading, or just for convenience, you can read the Django documentation in plain text.

If you’re using an official release of Django, note that the zipped package (tarball) of the code includes a `docs/` directory, which contains all the documentation for that release.

If you're using the development version of Django (aka the Subversion "trunk"), note that the `docs/` directory contains all of the documentation. You can `svn update` it, just as you `svn update` the Python code, in order to get the latest changes.

You can check out the latest Django documentation from Subversion using this shell command:

```
$ svn co http://code.djangoproject.com/svn/django/trunk/docs/ django_docs
```

One low-tech way of taking advantage of the text documentation is by using the Unix `grep` utility to search for a phrase in all of the documentation. For example, this will show you each mention of the phrase "max_length" in any Django document:

```
$ grep -r max_length /path/to/django/docs/
```

As HTML, locally

You can get a local copy of the HTML documentation following a few easy steps:

- Django's documentation uses a system called [Sphinx](#) to convert from plain text to HTML. You'll need to install Sphinx by either downloading and installing the package from the Sphinx website, or by Python's `easy_install`:

```
$ easy_install Sphinx
```

- Then, just use the included `Makefile` to turn the documentation into HTML:

```
$ cd path/to/django/docs
$ make html
```

You'll need [GNU Make](#) installed for this.

- The HTML documentation will be placed in `docs/_build/html`.

Warning: At the time of this writing, Django's using a version of Sphinx not yet released, so you'll currently need to install Sphinx from the source. We'll fix this shortly.

Differences between versions

As previously mentioned, the text documentation in our Subversion repository contains the "latest and greatest" changes and additions. These changes often include documentation of new features added in the Django development version – the Subversion ("trunk") version of Django. For that reason, it's worth pointing out our policy on keeping straight the documentation for various versions of the framework.

We follow this policy:

- The primary documentation on djangoproject.com is an HTML version of the latest docs in Subversion. These docs always correspond to the latest official Django release, plus whatever features we've added/changed in the framework *since* the latest release.
- As we add features to Django's development version, we try to update the documentation in the same Subversion commit transaction.
- To distinguish feature changes/additions in the docs, we use the phrase: "New in version X.Y", being X.Y the next release version (hence, the one being developed).
- Documentation for a particular Django release is frozen once the version has been released officially. It remains a snapshot of the docs as of the moment of the release. We will make exceptions to this rule in the case of retroactive security updates or other such retroactive changes. Once documentation is frozen, we

add a note to the top of each frozen document that says “These docs are frozen for Django version XXX” and links to the current version of that document.

- The [main documentation Web page](#) includes links to documentation for all previous versions.

See also:

Se você é novo em [Python](#), você pode querer começar obtendo uma idéia do que é a linguagem. Django é 100% Python, então se você tiver um mínimo de conforto com Python você provavelmente irá tirar mais proveito do Django.

Se você é inteiramente novo em programação, você pode se interessar por esta [lista de materiais sobre Python para não programadores](#).

Se você já conhece outras linguagens e quer aprender Python rapidamente, nós recomendamos [Dive Into Python](#) (disponível também na *versão impressa*). Se não te agradar muito, existe muitos outros [livros sobre Python](#).

Part II

Usando o Django

Introduções para todos os partes-chaves do Django, você precisa saber:

Como instalar o Django

Este documento colocará o seu Django para funcionar.

Instalando o Python

Por ser um framework escrito em Python, a instalação do Django precisa da instalação do Python.

Ele roda com Python desde a versão 2.3 a 2.6 (devido a questões de incompatibilidade retroativa no Python 3.0, o Django não funciona atualmente com o Python 3.0; veja [a FAQ do Django](#) para mais informação das versões suportadas do Python e a transição para o 3.0).

Baixe o Python de <http://python.org>. Se você estiver usando Linux ou Mac OS X, você provavelmente já o tem instalado.

Django on Jython

Se você usa o [Jython](#) (uma implementação do Python para plataforma Java), você precisará seguir alguns passos adicionais. Veja [Rodando Django no Jython](#) para mais detalhes.

Instalando Apache e mod_wsgi

Se você quiser apenas experimentar o Django, avance para a próxima sessão. O Django vem com um servidor Web bem leve que você pode usar para testes, de forma que você não precise instalar e configurar o Apache antes de colocar o Django em produção.

Se você quer usar o Django em um site em produção, use o Apache com [mod_wsgi](#). `mod_wsgi` é similar ao `mod_perl`, ele embute o Python dentro do Apache e carrega na memória códigos-fonte escritos em Python quando o servidor é iniciado. Este código fica na memória durante toda a vida do processo do Apache, o que leva a um ganho de desempenho significativo sobre outras configurações de servidor. Garanta que você tem o Apache instalado e que o módulo `mod_wsgi` esteja ativo. O Django irá funcionar com qualquer versão do Apache que suporte o `mod_wsgi`.

Leia [Como usar o Django com mod_wsgi](#) para mais informações sobre como configurar o `mod_wsgi` uma vez que você o tenha instalado.

Se você não puder usar o `mod_wsgi` por qualquer motivo, não se desespere: o Django suporta muitas outras opções de implantação. Uma grande segunda escolha é *mod_python*, o predecessor do `mod_wsgi`. Além disso, o Django segue a especificação WSGI, o que permite que ele rode em uma variedade de plataformas de servidor. Leia o wiki [server-arrangements](#) para informações específicas de instalação em cada plataforma.

Ponha seu banco de dados para funcionar

Se você pretende usar a API de banco de dados do Django, você vai precisar ter certeza de que seu servidor de banco de dados está funcionando. Django suporta muitos backends de bancos de dados e é oficialmente suportado por [PostgreSQL](#), [MySQL](#), [Oracle](#) e [SQLite](#) (apesar do SQLite não precisar de um servidor separado para rodar).

Além dos bancos de dados oficialmente suportados, há backends de terceiros que permite você usar outros bancos de dados com Django:

- [Sybase SQL Anywhere](#)
- [IBM DB2](#)
- [Microsoft SQL Server 2005](#)
- [Firebird](#)
- [ODBC](#)

As versões do Django e funcionalidades do ORM suportadas por estes backends não oficiais variam consideravelmente. Questões específicas relativas a estes backends não oficiais, juntamente com qualquer suporte a consultas, devem ser direcionadas para os canais de suporte fornecidos pelos projetos terceiros.

Em adição ao banco de dados, você vai precisar ter certeza de que as suas bibliotecas Python de acesso ao banco de dados estão instaladas.

- Se você está usando PostgreSQL, você vai precisar do pacote [psycopg](#), o Django suporta as versões 1 e 2. (Quando você for configurar a camada de banco de dados do Django, escolha `postgresql` para a versão 1 ou `postgresql_psycopg2` para a versão 2.)

Se você estiver usando Windows, olhe a [versão não-oficial compilada para Windows](#).

- Se você estiver usando MySQL, você vai precisar da biblioteca [MySQLdb](#) (versão 1.2.1p2 ou superior). Você também vai querer ler as observações específicas de bancos de dados para o *backend MySQL*.
- Se você estiver usando SQLite e Python 2.3 ou Python 2.4, vai precisar da biblioteca [pysqlite](#). Use a versão 2.0.3 ou superior. A versão 2.5 do Python já vem com um wrapper `sqlite` na biblioteca padrão, então você não precisa instalar mais nada neste caso. Please read the SQLite backend [notes](#).
- Se você estiver usando Oracle, você precisará de uma cópia do [cx_Oracle](#), mas por favor, leia as notas específicas do banco de dados para o *backend Oracle* para informações importantes.
- Se você está usando um backend não oficial de terceiros, por favor consulte a documentação fornecida para quaisquer requerimentos adicionais.

Se você pretende usar o comando `manage.py syncdb` do Django para criar automaticamente as tabelas para os seus modelos, você precisa garantir que o Django tem permissão para criar tabelas no banco de dados que você está usando. Se você pretende criar as tabelas manualmente, precisa apenas dar permissão de `SELECT`, `INSERT`, `UPDATE` e `DELETE` para o usuário do Django (configurado em `settings.py`, atributo `DATABASE_USER`). O Django não usa comandos `ALTER TABLE`, então não precisa de permissão para isso.

Se você pretende usar o *framework de teste* do Django com ‘data fixtures’, o Django também vai precisar de permissão para criar um bando de dados de teste temporário.

Remova qualquer versão antiga do Django

Se você está atualizando sua instalação do Django, vai precisar desinstalar a versão antiga antes de instalar a nova.

Se você instalou o Django usando `setup.py install`, o processo de desinstalação é simples: basta apagar o diretório `django` de seu diretório `site-packages`.

Se você instalou o Django usando um egg Python, remova o arquivo `.egg` do Django e remova a referência a ele no arquivo `easy-install.pth` (dentro do diretório `site-packages`).

Onde fica o diretório `site-packages` ?

A localização do diretório `site-packages` depende de seu sistema operacional e o local em que o Python foi instalado. Para descobrir a localização de seu `site-packages`, execute o seguinte comando:

```
.. code-block:: bash
```

```
python -c "from distutils.sysconfig import get_python_lib; print get_python_lib()"
```

(Observe que este comando deve ser executado em um terminal, não no prompt interativo do Python)

Instalando o Django

As instruções de instalação são um tanto diferentes, dependendo se você está instalando um pacote de uma distribuição específica, baixando a última versão oficial lançada ou usando a versão em desenvolvimento.

O processo é simples, independentemente de qual você escolher.

Instalando um pacote de uma distribuição específica

Verifique as *observações específicas de distribuições* para ver se a sua plataforma/distribuição fornece pacotes/instaladores oficiais do Django. Pacotes fornecidos por distribuições geralmente permitem a instalação automática de dependências e atualizações fáceis.

Instalando uma versão oficial

1. Baixe a última versão oficial lançada da [página de download](#) do Django.
2. Descompacte o arquivo baixado (por exemplo: `tar xzvf Django-NNN.tar.gz`, onde `NNN` é o número da última versão lançada). Se você estiver usando Windows, você pode baixar a ferramenta linha de comando `bsdtar` para fazer isso, ou você usar uma ferramenta com interface gráfica como a [7-zip](#).
3. Entre no diretório criado pelo passo 2 (por exemplo: `cd Django-NNN`).
4. Se você estiver usando Linux, Mac OS X ou algum outro tipo de Linux, execute o comando `sudo python setup.py install` no terminal. Se você estiver usando Windows, abra um prompt (DOS) com privilégios de administrador e execute o comando `python setup.py install`.

Estes comandos instalarão o Django em seu diretório `site-packages` da instalação do Python.

Instalando a versão em desenvolvimento

Tracking Django development

Se você decidiu usar a última versão de desenvolvimento do Django, você terá que prestar muita atenção na [linha do tempo do desenvolvimento](#) e você terá que manter um olho sobre a [lista de mudanças de retro-compatibilidade](#). Isto irá ajudá-lo a estar atualizado quanto a qualquer funcionalidade que você queira usar, bem como qualquer mudança que você precisará fazer em seu código quando atualizar suas instalações do Django. (Para versões estáveis, qualquer mudança necessária está documentada nas notas de lançamento.)

Se você gostaria de poder atualizar seu Django ocasionalmente com as últimas correções de bugs e melhorias, siga estas instruções:

1. Certifique-se de que você tem o [Subversion](#) instalado e que você pode executar seus comandos em um terminal. (Digite `svn help` no terminal para testar.)
2. Baixe o branch principal de desenvolvimento do Django (o 'trunk'):

```
.. code-block:: bash
```

```
svn co http://code.djangoproject.com/svn/django/trunk/ django-trunk
```

3. Depois, certifique-se de que o interpretador Python consegue carregar o código Django. Existem várias maneiras de fazer isso. Uma das mais convenientes, em Linux, Mac OS X ou outros sistemas Unix é usar um link simbólico:

```
ln -s `pwd`/django-trunk/django SITE_PACKAGES_DIR/django
```

(Na linha acima, troque `SITE_PACKAGES_DIR` pela localização do diretório `site-packages` no seu sistema, como explicado em “Onde fica o diretório `site-packages`?”).

Alternativamente, você pode definir sua variável de ambiente `PYTHONPATH` para que ela inclua o diretório `django - sub-diretório de django-trunk`. Talvez essa seja a solução mais conveniente para o Windows, que não oferece suporte a links simbólicos. (Variáveis de ambiente podem ser definidas no Windows [pelo Painel de Controle](#))

E o Apache e `mod_python`?

Se você escolher a solução de definir sua `PYTHONPATH` você precisa lembrar de fazer o mesmo na sua configuração do Apache quando for colocar seu site em produção. Faça isso definindo `PythonPath` no seu arquivo de configuração do Apache.

Mais informações sobre implementações em produção, é lógico, estão disponíveis na documentação [Como usar o Django com `mod_python`](#).

4. Em sistemas Unix, crie um link simbólico para o arquivo `django-trunk/django/bin/django-admin.py` para um diretório no seu 'system path', por exemplo: `/usr/local/bin`:

```
.. code-block:: bash
```

```
ln -s `pwd`/django-trunk/django/bin/django-admin.py /usr/local/bin
```

Isso permite a você simplesmente digitar `django-admin.py` de qualquer diretório, em vez de ter que escrever o caminho inteiro para o script.

Quando em Windows, o mesmo resultado pode ser obtido copiando o arquivo `django-trunk/django/bin/django-admin.py` para algum lugar no seu 'system path'. Por exemplo: `C:\Python24\Scripts`.

Você *não* precisa executar `python setup.py install`, pois você já fez o equivalente nos passos 3 e 4.

Quando você quiser atualizar a sua cópia do código-fonte do Django, apenas execute `svn update` dentro do diretório `django-trunk`. Ao fazer isso, o Subversion vai automaticamente baixar as atualizações.

Models e bancos de dados

A model is the single, definitive source of data about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

Um modelo é uma fonte singular e definitiva de dado sobre seu seus dados. Ele contém os campos e comportamentos essenciais dos dados que você está armazenando. Geralmente, cada modelo mapeia uma única tabela do banco de dados.

Models

Um modelo é a fonte única e definitiva de dados sobre os seus dados. Ele contém os campos e comportamentos essenciais dos dados que você está gravando. Geralmente, cada modelo mapeia para uma única tabela no banco de dados.

O básico:

- Cada modelo é uma classe Python que estende `django.db.models.Model`.
- Cada atributo do modelo representa uma coluna do banco de dados.
- Com tudo isso, o Django lhe dá uma API de acesso a banco de dados gerada automaticamente, o que é explicado em *Fazendo consultas*.

See also:

Um companheiro para esse documento é o [repositório oficial de exemplos de modelo](#). (Na distribuição do fonte do Django, esses exemplos estão no diretório `tests/modeltests`.)

Exemplo rápido

Esse modelo de exemplo define uma `Person`, que tem um `first_name` e um `last_name`:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` e `last_name` são *campos* do modelo. Cada campo é especificado como um atributo de classe, e cada atributo é mapeado para uma coluna no banco de dados.

O modelo `Person` acima criaria uma tabela assim:

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Algumas notas técnicas:

- O nome da tabela, `myapp_person`, é automaticamente derivado de alguns metadados do modelo, no entanto isto pode ser sobrescrito. Veja *Nomes de tabelas* abaixo.
- Um campo `id` é adicionado automaticamente, mas esse comportamento também pode ser alterado. Veja *Campos de chave primária automáticos* abaixo.
- O comando SQL `CREATE TABLE` nesse exemplo é formatado usando a sintaxe do PostgreSQL, mas é digno de nota que o Django usa o SQL adaptado ao banco de dados especificado no seu *arquivo de configurações*.

Usando modelos

Uma vez que já tenha criado seus modelos, o passo final é dizer ao Django para usar estes modelos. Para isto, basta editar seu arquivo `settings.py` e mudar o `INSTALLED_APPS` adicionando o nome do módulo que contém seu `models.py`.

Por exemplo, se os modelos de sua aplicação ficam no módulo `mysite.myapp.models` (a estrutura de pacote que é criada para uma aplicação pelo script `manage.py startapp`), o `INSTALLED_APPS` deve ler, em parte:

```
INSTALLED_APPS = (
    #...
    'mysite.myapp',
    #...
)
```

Quando você adicionar novas aplicações ao `INSTALLED_APPS`, assegure-se de rodar o `manage.py syncdb`.

Campos

A parte mais importante do modelo – e a única obrigatória – é a lista de campos do banco de dados que ele define. Campos são especificados por meio de atributos de classe.

Exemplo:

```
class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Tipos de campos

Cada campo no seu modelo deve ser uma instância da classe `Field` apropriada. O Django usa os tipos das classes para determinar algumas coisas:

- O tipo de coluna no banco de dados (ex: `INTEGER`, `VARCHAR`).
- O widget a ser usado na interface administrativa do Django, se você a utilizar (ex: `<input type="text">`, `<select>`).
- Os requisitos mínimos para validação, usados no site de administração do Django e nos formulários automaticamente gerados.

O Django é disponibilizado com dezenas de tipos de campos embutidos; você pode encontrar a lista completa em [referência de campos do model](#). Você também pode facilmente escrever seus próprios tipos de campos se os que acompanham o Django não lhe servirem; veja [Writing custom model fields](#).

Field options

Cada tipo de campo recebe um certo conjunto de argumentos específicos (documentados na [referência de campos do model](#)). Por exemplo, `CharField` (e suas subclasses) requerem um argumento `max_length` que especifica o tamanho do campos `VARCHAR` que será usado para armazenar os dados.

Também há um conjunto de argumentos comuns disponíveis para todos os tipos de campos. todos são opcionais. Eles são totalmente explicados na [referência](#), mas aqui há um pequeno sumário dos mais frequentemente usados:

`null` Se `True`, o Django irá gravar valores vazios como `NULL` no banco de dados. O padrão é `False`.

`blank` Se `True`, o campo pode ser vazio. o padrão é `False`.

Note que isso é diferente de `null`. `null` é puramente relacionado ao banco de dados, e `blank` é relacionado com validação. Se um campo tem `blank=True`, a validação na administração do Django irá permitir a entrada de um valor vazio. Se um campo tem `blank=False`, o campo será obrigatório.

`choices` Um iterável(e.g., uma lista ou tupla) de tupla duplas para usar como escolhas para esse campo. Se fornecido, a administração do Django usará uma caixa de seleção no lugar de um campo de texto padrão e irá limitar as escolhas as opções dadas.

Uma lista de opções é parece com isso:

```
YEAR_IN_SCHOOL_CHOICES = (
    (u'FR', u'Freshman'),
    (u'SO', u'Sophomore'),
    (u'JR', u'Junior'),
    (u'SR', u'Senior'),
    (u'GR', u'Graduate'),
)
```

O primeiro elemeno de cada tupla é o verdadeiro valor a ser gravado. O segundo elemento será mostrado pela interface de adminsitração, ou em um `ModelChoiceField`. Dada uma instância de um objeto de model, o valor mostrado pelo campo `choices` pode ser acessado usando o método `get_FOO_display`. Por exemplo:

```
from django.db import models

class Person(models.Model):
    GENDER_CHOICES = (
        (u'M', u'Male'),
        (u'F', u'Female'),
    )
    name = models.CharField(max_length=60)
    gender = models.CharField(max_length=2, choices=GENDER_CHOICES)
```

```
>>> p = Person(name="Fred Flinstone", gender="M")
>>> p.save()
>>> p.gender
u'M'
>>> p.get_gender_display()
u'Male'
```

default O valor padrão para o campo. Pode ser também um objeto chamável. Se for um chamável, será chamado a cada vez que um novo objeto for criado.

help_text Um texto de “ajuda” extra para ser mostrado sob o campo no formulário de objetos do admin. É útil para documentação, mesmo que seu objeto não tenha um formulário administrativo.

primary_key Se True, esse campo será a chave primária para o modelo.

Se você não especificar `primary_key=True` `<Field.primary_key>` para nenhum campo no seu modelo, o Django adicionará automaticamente um campo `IntegerField` para ser a chave primária, desta forma, você não precisa configurar o `primary_key=True` `<Field.primary_key>` em qualquer um dos seus campos a menos que você queira sobrescrever o comportamento padrão de chaves primárias. Para saber mais, [Campos de chave primária automáticos](#).

unique Se True, esse campo deve ser único na tabela.

Novamente, estes são somente descrições curtas das opções mais comuns dos campos. Detalhes completos podem ser encontrados na [referência de opções dos campos comuns do model](#).

Campos de chave primária automáticos

Por padrão, o Django dá a cada model o seguinte campo:

```
id = models.AutoField(primary_key=True)
```

Esta é uma chave primária auto incremental.

Se você gostaria de especificar uma chave primária customizada, somente especifique `primary_key=True` em um dos seus campos. Se o Django ver que você especificou uma `:attr:Field.primary_key`, ele não adicionará a coluna `id` automaticamente.

Cada model requer que exatamente um campo tenha `primary_key=True`.

Nomes de campos por extenso

Cada tipo de campo, exceto `ForeignKey`, `ManyToManyField` e `OneToOneField`, recebem um primeiro argumento opcional – um nome por extenso. Se o nome por extenso não é informado, o Django criará automaticamente a partir do atributo `name` do campo, convertendo underscores em espaços.

Nesse exemplo, o nome por extenso é "Person's first name":

```
first_name = models.CharField("Person's first name", max_length=30)
```

Nesse exemplo, o nome por extenso é "first name":

```
first_name = models.CharField(max_length=30)
```

`ForeignKey`, `ManyToManyField` e `OneToOneField` requerem que o primeiro argumento seja uma classe do modelo, assim usa o argumento `verbose_name` como um argumento nomeado:

```
poll = models.ForeignKey(Poll, verbose_name="the related poll")
sites = models.ManyToManyField(Site, verbose_name="list of sites")
place = models.OneToOneField(Place, verbose_name="related place")
```

A convenção é não colocar em caixa alta a primeira letra do `verbose_name`. O Django irá automaticamente capitalizar a primeira letra quando for necessário.

Relacionamentos

Claramente, o poder dos bancos relacionais reside na capacidade de relacionar tabelas umas as outras. O Django oferece formas de definir os três tipos de relacionamento mais comuns: muitos-para-um, muitos-para-muitos e um-para-um.

Relacionamentos muitos-para-um

Para definir um relacionamento muitos para um, use `ForeignKey`. Você o usa como qualquer outro `Field`: incluindo-o como um atributo de classe no seu modelo.

O `ForeignKey` requer um argumento posicional: a classe a qual esse modelo é relacionado.

Por exemplo, se um modelo `Car` tem um `Manufacturer` – isso é, um `Manufacturer` faz múltiplos carros, mas cada `Car` somente tem um `Manufacturer` – use as seguintes definições:

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

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer)
    # ...
```

Para criar um *relacionamento recursivo* – (um objeto que tem um relacionamento muitos para um consigo mesmo) e *relacionamentos com models que ainda não foram definidos*; veja *a referência de campos do model* para mais detalhes.

É sugerido, mas não obrigatório, que o nome de um campo `ForeignKey` (`manufacturer` do exemplo acima) seja o nome do model, em minúsculo. Você pode, é claro, chamar o campo como você quiser. Por exemplo:

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(Manufacturer)
    # ...
```

See also:

Veja o [exemplo de relacionamento Muitos-para-um](#) para um exemplo completo.

Os campos `ForeignKey` também aceitam um número extra de argumentos que são explicados na *referência de campos do model*. Estas opções ajudam a definir como o relacionamento deve funcionar; todos são opcionais.

Relacionamentos muitos-para-muitos

Para definir um relacionamento muitos-para-muitos, use o `ManyToManyField`. Você o utiliza como qualquer outro tipo de `Field`: incluindo ele como um atributo de classe do seu model.

O `ManyToManyField` requer um argumento posicional: a classe à qual esse modelo está relacionado.

Por exemplo, se uma `Pizza` tem múltiplos objetos `Topping` – isto é, um `Topping` pode estar em múltiplas pizzas e cada `Pizza` tem várias sobremesas – aqui está como representar isso:

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

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```


Como com *ForeignKey*, você pode também criar *relacionamentos recursivos* (um objeto com um relacionamento muitos-para-um para si mesmo) e *relacionamentos para models ainda não definidos*; veja *a referência de campos do model* para mais detalhes.

É sugerido, mas não obrigatório, que o nome de um *ManyToManyField* (toppings no exemplo acima) esteja no plural, descrevendo o conjunto de objetos model relacionados.

Não importa qual model recebe o *ManyToManyField*, mas você somente precisa dele em um dos models – não em ambos.

Geralmente, instâncias de *ManyToManyField* devem ir no objeto que sera editado na interface do admin, se você estiver usando o admin do Django. No exemplo acima, toppings está em Pizza (ao invés de Topping ter um *ManyToManyField* pizzas) porque é mais natural pensar sobre pizzas tendo sobremesas do que sobremesas tendo várias pizzas. A forma que é mostrada acima, o formulário de Pizza no admin, deixaria os usuários selecionar sobremesas.

See also:

Veja o [exemplo de relacionamento de model Muitos-para-muitos](#) para um exemplo completo.

Os campos *ManyToManyField* também aceitam um número extra de argumentos que são explicados na *referência de campos do model*; Estas opções ajudam a definir como o relacionamento deve funcionar; todos são opcionais.

Campos extra sobre relacionamentos muitos-para-muitos

Please, see the release notes Quando você está somente lidando com relacionamentos muitos-para-muitos simples assim como misturando ou combinando pizzas e sobremesas, um *ManyToManyField* padrão é tudo que você precisa. Entretanto, algumas vezes você pode precisar associar dados com o relacionamento entre dois models.

Por exemplo, considere o caso de uma aplicação que monitora grupos musicais dos quais músicos pertencem. Há um relacionamento muitos-para-muitos entre uma pessoa e os grupos dos quais ela é um membro, então você poder usar um *ManyToManyField* para representar este relacionamento. No entanto, há um monte de detalhes sobre a filiação que você pode querer coletar, como a data em que a pessoa se juntou a um grupo.

Para estas situações, o Django permite você especificar o model que será usado para governar o relacionamento muitos-para-muitos. Você pode então colocar campos extra sobre o model intermediário. O model mediador é associado com o *ManyToManyField* usando o argumento *through* para apontar o model que agirá como um mediador. Para o nosso exemplo dos músicos, o código pareceria com algo assim:

```
class Person(models.Model):
    name = models.CharField(max_length=128)

    def __unicode__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __unicode__(self):
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person)
    group = models.ForeignKey(Group)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

Quando você configura o model intermediário, você explicitamente especifica as chaves estrangeiras para o models que estão envolvidos no relacionamento ManyToMany. Esta declaração explicita define como os dois models serão relacionados.

Há umas poucas restrições no model intermediário:

- Seu model mediador deve conter uma - e *somente* uma - chave estrangeira para o model alvo (este seria o `Person` em nosso exemplo). Se você tiver mais de uma chave estrangeira, um erro de validação será gerado.
- Seu model mediador deve conter uma - e *somente* uma - chave estrangeira para o model fonte (este seria o `Group` no nosso exemplo). Se você tiver mais de uma chave estrangeira, um erro de validação será gerado.
- A única exceção para isto é o model que tem relacionamento muitos-para-muitos, consigo mesmo através de um model intermediário. Neste caso, duas chaves estrangeiras para o mesmo model é permitida, mas elas serão tratadas como dois (diferentes) lados do muitos-para-muitos.
- Quando se define um relacionamento muitos-para-muitos de um model para ele mesmo, usando um model intermediário, você *deve* usar `symmetrical=False` (veja [referencia de campos do model](#)).

Agora que você tem configurado seu `ManyToManyField` para usar seu model mediador (`Membership`, neste caso), você está pronto para começar a criar alguns relacionamentos muitos-para-muitos. Você faz isto criando instâncias do model intermediário:

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason= "Needed a new drummer.")
>>> m1.save()
>>> beatles.members.all()
[<Person: Ringo Starr>]
>>> ringo.group_set.all()
[<Group: The Beatles>]
>>> m2 = Membership.objects.create(person=paul, group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason= "Wanted to form a band.")
>>> beatles.members.all()
[<Person: Ringo Starr>, <Person: Paul McCartney>]
```

Diferentemente de campos muitos-para-muitos normais, você *não pode* usar `add`, `create`, ou atribuição (i.e., `beatles.members = [...]`) para criar relacionamentos:

```
# ISTO NÃO FUNCIONARÁ
>>> beatles.members.add(john)
# NEM ISSO IRÁ
>>> beatles.members.create(name="George Harrison")
# E NEM MESMO ISSO
>>> beatles.members = [john, paul, ringo, george]
```

Porquê? você não pode simplesmente criar um relacionamento entre um `Person` e um `Group` - você precisa especificar todos os detalhes para o relacionamento requerido pelo model `Membership`. As chamadas simples `add`, `create` e atribuição não provem uma forma de especificar estes detalhes a mais. Como um resultado, eles são desabilitados pelos relacionamentos muitos-para-muitos que usam um model mediador. A única forma de criar este tipo de relacionamento é criando instâncias do model intermediário.

O método `remove` é desabilitado por razões similares. No entanto, o método `clear()` pode ser usado para remover todo relacionamento muitos-para-muitos para uma instância:

```
# Beatles have broken up
>>> beatles.members.clear()
```

Uma vez que você estabeleça o relacionamento muitos-para-muitos criando instâncias de seus models intermediários, você pode emitir consultas. Só que como um relacionamento muitos-para-muitos normal, você pode consultar usando os atributos do model relacionado com o muitos-para-muitos:

```
# Encontra todos os grupos com o membro cujo nome começa com 'Paul'
>>> Group.objects.filter(members__name__startswith='Paul')
[<Group: The Beatles>]
```

Como você está usando um model intermediário, você pode também consultar seus atributos:

```
# Encontre todos os membro do Beatles que entraram depois de 1 Jan 1961
>>> Person.objects.filter(
...     group__name='The Beatles',
...     membership__date_joined__gt=date(1961,1,1))
[<Person: Ringo Starr>]
```

Relacionamentos um-para-um

Para definir um relacionamento um-para-um, use o *OneToOneField*. Você deve utilizá-lo como qualquer outro *Field*: incluindo-o como um atributo de classe em seu model.

Este é mais útil sobre a chave primária de um objeto, quando este objeto “estende” outro objeto de alguma forma.

O *OneToOneField* requer um argumento posicional: a classe para qual o model está relacionado.

Por exemplo, se você construiu um banco de dados de “lugares”, você iria construir um belo padrão, com coisas tipo endereço, telefone, etc. no banco de dados. Então se você esperava construir um banco de dados de restaurantes sobre os lugares, ao invés de repetir e replicar todos esses campos no model *Restaurant*, você poderia fazer *Restaurant* ter um *OneToOneField* para *Place* (porque um restaurante “é um” lugar; em fato, para manipular isto você normalmente usaria *herança*, que envolve uma relação um-para-um implícita.)

Como com *ForeignKey*, um *relacionamento recursivo* pode ser definido e *referenciar models ainda indefinidos*; veja *a referência de campos do model* para mais detalhes.

See also:

Veja o *exemplo de relacionamento de model um-para-um* para um exemplo completo.

Please, see the release notes Os campos *OneToOneField* também aceitam um argumento opcional descrito na *referência de campos do model*.

OneToOneField classes used to automatically become the primary key on a model. This is no longer true (although you can manually pass in the *primary_key* argument if you like). Thus, it’s now possible to have multiple fields of type *OneToOneField* on a single model.

As classes *OneToOneField* são usadas para automaticamente tornar-se a chave primária de um model. Isto já não é verdade (embora você possa manualmente passar em um argumento *primary_key* se você quiser). Assim, agora é possível ter vários campos do tipo *OneToOneField* em um único model.

Models em todos os arquivos

É perfeitamente normal relacionar um model com um de outra aplicação. Para fazer isto, importe o model relacionado no topo do arquivo que contém seu model. Então, é só referenciar para a outra classe model onde você quiser. Por exemplo:

```
from mysite.geography.models import ZipCode

class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(ZipCode)
```

Restrições de nome de campos

O Django impõe apenas duas restrições aos nomes de campos do modelo:

1. Um nome de campo não pode ser uma palavra reservada do Python, porque isso resultaria num erro de sintaxe do Python. Por exemplo:

```
class Example(models.Model):
    pass = models.IntegerField() # 'pass' é uma palavra reservada!
```

2. Um nome de campo não pode conter mais de um underscore em uma linha, devido à forma que a sintaxe de busca de consultas do Django funciona. Por exemplo:

```
class Example(models.Model):
    foo__bar = models.IntegerField() # 'foo__bar' tem dois underscores!
```

Essas limitações podem ser trabalhadas, já que o nome do seu campo não precisa necessariamente ser igual ao nome da coluna no seu banco de dados. Veja a opção `db_column`.

Palavras reservadas do SQL, como `join`, `where` ou `select`, são permitidas como nomes de campos no modelo, porque o Django escapa todos os nomes de tabelas e colunas em cada consulta SQL. Ele usa a sintaxe de quoting do seu banco de dados em particular.

Tipos de campos customizados

Please, see the release notes Se um dos campos existentes no model não pode ser usado para o que você propõe, ou se você deseja ter vantagem sobre alguma coluna de banco de dados menos comum, você pode criar suas próprias classes de campos. A cobertura total da criação de seus próprios campos é fornecida em *Writing custom model fields*.

Opções Meta

Forneça seus metadados de model usando uma classe interna `class meta`, desta forma:

```
class Ox(models.Model):
    horn_length = models.IntegerField()

    class Meta:
        ordering = ["horn_length"]
        verbose_name_plural = "oxen"
```

O metadado do model é “qualquer coisa que não seja um campo”, assim como opções de ordenamento (attr:~*Options.ordering*), nome de tabelas do banco de dados (*db_table*), ou nomes legíveis-por-humanos no singular ou plural (*verbose_name_plural*). Nenhum é obrigatório, e adicionar `class Meta` ao model é completamente opcional.

Uma lista completa de todas as opções possíveis do *Meta* podem ser encontradas na *referência e opções de model*.

Métodos do model

Define métodos customizados sobre um model para adicionar funcionalidades a “nível de linha” para seus objetos. Considerando que os métodos *Manager* são destinados para fazer coisas “table-wide”, os métodos de model devem agir sobre uma instância particular de model.

Esta é uma técnica valiosa para manter a lógica de negócio em um só lugar – o model.

Por exemplo, este model tem uns poucos métodos customizados:

```
from django.contrib.localflavor.us.models import USStateField

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()
```

```
address = models.CharField(max_length=100)
city = models.CharField(max_length=50)
state = USStateField() # Yes, this is America-centric...

def baby_boomer_status(self):
    "Returns the person's baby-boomer status."
    import datetime
    if datetime.date(1945, 8, 1) <= self.birth_date <= datetime.date(1964, 12, 31):
        return "Baby boomer"
    if self.birth_date < datetime.date(1945, 8, 1):
        return "Pre-boomer"
    return "Post-boomer"

def is_midwestern(self):
    "Returns True if this person is from the Midwest."
    return self.state in ('IL', 'WI', 'MI', 'IN', 'OH', 'IA', 'MO')

def _get_full_name(self):
    "Returns the person's full name."
    return '%s %s' % (self.first_name, self.last_name)
full_name = property(_get_full_name)
```

O último método deste exemplo é um *property*. Leia mais sobre propriedades.

A *referência de instância de model* tem uma lista completa de *métodos automaticamente dados para cada model*. Você pode sobrescrever quase todos eles – veja *sobrescrevendo métodos de model predefinidos*, abaixo – mas há alguns que você irá quase sempre querer definir:

`__unicode__()` Um “método mágico” do Python que retorna uma “representação” unicode de qualquer objeto. Isto é o que o Python e o Django irá usar sempre que uma instância de model precisar ser mostrada como uma string. Mais notavelmente, isto acontece quando você mostra um objeto em um console interativo ou no admin.

Você sempre irá querer definir este método; o padrão não é muito útil.

`get_absolute_url()` Este diz ao Django como calcular a URL para um objeto. O Django o usa na sua interface de administração, e toda vez que precisa descobrir a URL de um objeto.

Qualquer objeto que tem uma URL que o identifica exclusivamente, deve definir este método.

Sobrescrevendo métodos de model predefinidos

Há outro conjunto de *métodos do model* que encapsulam um monte de comportamentos de banco de dados que você deseja customizar. Em particular você frequentemente vai querer mudar a forma de funcionamento do `save()` e `delete()`.

Você é livre para sobrescrever estes métodos (e qualquer outro método do model) para alterar comportamento.

Um caso de uso clássico para sobrecarga de métodos embutidos é se você deseja que algo aconteça sempre que você salva o objeto. Por exemplo (veja `save()` para documentação e parâmetros aceitos):

```
class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, force_insert=False, force_update=False):
        do_something()
        super(Blog, self).save(force_insert, force_update) # Call the "real"
        ↪ save() method.
        do_something_else()
```

Você pode também prevenir o salvamento:

```
class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, force_insert=False, force_update=False):
        if self.name == "Yoko Ono's blog":
            return # Yoko não deve nunca ter seu próprio blog!
        else:
            super(Blog, self).save(force_insert, force_update) # Chama o método_
↪save() "real".
```

É importante lembrar de chamar o método da superclasse – esse é que é o negócio `super(Blog, self).save()` – para assegurar que o objeto ainda será salvo no banco de dados. Se você esquecer de chamar o método da superclasse, o comportamento padrão não acontecerá e o banco de dados não será tocado.

Executando SQL customizado

Outra prática comum é escrever consultas SQL personalizadas em métodos do modelo a nível módulo. Para mais detalhe sobre como usar SQL puro, veja a documentação em [usando SQL puro](#).

Herança de modelos

Please, see the release notes Herança de modelos no Django funciona quase igual à forma normal de herança entre classes do Python. A única decisão que você tem de tomar é se você quer que os modelos pai sejam modelos no seu próprio contexto (com suas próprias tabelas de banco de dados), ou se os pais são somente mantenedores de informações comuns que somente serão visíveis através da herança entre modelos.

Freqüentemente, você só quer usar a classe pai para manter informações que você não deseja escrever para cada modelo filho. Esta classe nunca será usada de forma isolada, então *classes abstratas de base* são o que estiver depois. Entretanto, se você está estendendo uma subclasse de modelo existente (talvez algo de outra aplicação), ou quer que cada modelo tenha sua própria tabela no banco de dados, *herança com multi-tabelas* é o caminho a seguir.

Classes Abstratas de Base

Uma classe abstrata de base é útil quando você quer colocar alguma informação comum à disposição de vários modelos. Você escreve sua classe de base e coloca `abstract=True` dentro da classe *Meta*. Este modelo não será usado para criar qualquer tabela no banco de dados. Em vez disso, quando ele for usado como uma classe abstrata de base por outro modelo, seus campos serão adicionados aos seus modelos filho. É um erro ter campos na classe abstrata de base com o mesmo nome daqueles que a herda (o Django irá lançar uma exceção).

Um exemplo:

```
class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

    class Student(CommonInfo):
        home_group = models.CharField(max_length=5)
```

O modelo `Student` possuirá três campos: `name`, `age` e `home_group`. O modelo `CommonInfo` não pode ser usado como um modelo normal do Django, já que ele é uma classe abstrata de base. `CommonInfo` não gera uma tabela no banco de dados e nem pode ser instanciado ou salvo diretamente.

Para muitos usos, este tipo de herança de modelo será exatamente o que você quer. Ele fornece uma maneira de fatorar informações comuns ao nível do Python, enquanto ao nível do banco de dados só serão criadas tabelas para os modelos filhos.

Herança de Meta

Quando uma classe abstrata de base é criada, o Django torna qualquer *Meta* interno da classe que você criou, disponível como um atributo a classe de base. Se uma classe filha não declara sua própria classe *Meta*, ela irá herdar de seu pai. Se a classe filha quer estender a classe *Meta* do pai, ela pode. Por exemplo:

```
class CommonInfo(models.Model): ...

    class Meta: abstract = True ordering = ['name']

class Student(CommonInfo): ...

    class Meta(CommonInfo.Meta): db_table = 'student_info'
```

O Django faz um ajuste para a classe *Meta* de uma classe abstrata de base: antes instalando o atributo *Meta*, e configurando `abstract=False`. Isto significa que os filhos da classe abstrata de base não se tornam automaticamente classes abstratas. É claro, você pode fazer uma classe abstrata de base que herda outra classe abstrata de base. Você só precisa lembrar de setar explicitamente `abstract=True` toda vez.

Alguns atributos não fazem sentido serem incluídos na classe *Meta* de uma classe abstrata de base. Por exemplo, incluindo `db_table` poderia significar que todos os filhos (que não especificarem seus próprios *Meta*) poderiam usar a mesma tabela no banco de dados, que é provavelmente o que você não quer.

Seja cuidadoso com `related_name`

Se você esta usando o atributo `related_name` sobre um `ForeignKey` ou `ManyToManyField`, você sempre deve especificar um **reverse name** *único* para o campo. Isto normalmente poderia causar um problema nas classes abstratas de base, já que os campos desta classe são incluídos em cada classe filha, com exatamente os mesmos valores para os atributos (incluindo `related_name`) toda vez.

Para contornar este problema, quando você está usando o `related_name` em uma classe abstrata de base (soamente), parte do nome deve ser a string `'%(class)s'`. Isto é substituído por um nome em minúsculo da classe filha onde o campo é usado. Desde que cada classe tenha um nome diferente, cada nome relacionado irá acabar sendo diferente. Por exemplo:

```
class Base(models.Model):
    m2m = models.ManyToManyField(OtherModel, related_name="%(class)s_related")

    class Meta:
        abstract = True

class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

O **reverse name** do campo `ChildA.m2m` será `childa_related`, enquanto que o **reverse name** do campo `ChildB.m2m` será `childb_related`. Cabe a você como usar a porção `'%(class)s'` para contruir seu nome relacionado, mas se você se esquecer de usá-lo, o Django irá lançar erros quando você validar seus modelos (ou ao rodar o `syncdb`).

Se você não especificar um atributo `related_name` para um campo em uma classe abstrata de base, o **reverse name** padrão será o nome da classe filho seguido por `'_set'`, do mesmo modo que normalmente seria se você tivesse declarado na classe filha. Por exemplo, no código acima, se o atributo `related_name` fosse omitido, o **reverse name** para o campo `m2m` seria `childa_set` no caso do `ChildA` e `childb_set` para o campo `ChildB`.

Herança com Multi-Tabelas

O segundo tipo de herança de modelos suportado pelo Django é quando cada modelo na hierarquia é um modelo em si mesmo. Cada modelo corresponde a sua própria tabela no banco de dados e pode ser consultado e criado individualmente. A relação de herança introduz links entre o modelo filho e cada um de seus pais (por meio de um campo `OneToOneField` criado automaticamente). Por exemplo:

```
class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField()
    serves_pizza = models.BooleanField()
```

Todos os campos de `Place` também estarão disponíveis no `Restaurant`, apesar de o dado residir em uma tabela diferente no banco de dados. Então ambos são possíveis:

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

Se você tem um `Place` que também é um `Restaurant`, você pode obter do objeto `Place` o objeto `Restaurant` usando a versão minúscula do nome do model:

```
>>> p = Place.objects.filter(name="Bob's Cafe")
# Se "Bob's Cafe" é um objeto Restaurant, isto irá retornar uma classe filho:
>>> p.restaurant
<Restaurant: ...>
```

Entretanto, se `p` no exemplo acima *não* for um `Restaurant` (que havia sido criado diretamente como um objeto `Place` ou foi pai de alguma outra classe), referir-se a `p.restaurant` poderia gerar um erro.

Meta e herança com multi-tabelas

Nesta situação de herança com multi-tabelas, não faz sentido a classe filha herdar a classe *Meta* de seus pais. Todas as opções em *Meta* já teriam sido aplicadas para a classe e aplicá-las de novo normalmente levaria a um comportamento contraditório (isto está em contraste com o caso da classe abstrata de base, onde a classe de base não existe em seu próprio direito).

Então o modelo filho não tem acesso às classes *Meta* de seus pais. Entretanto, existem uns poucos casos limitados onde o filho herda o comportamento do pai: se o filho não especificar um atributo `django.db.models.Options.ordering` ou um atributo `django.db.models.Options.get_latest_by`, ele os herdará de seus pais.

Se o pai tem um ordenamento e você não quer que o filho o tenha, você pode explicitar com uma lista vazia:

```
class ChildModel(ParentModel):
    ...
    class Meta:
        # Remove o ordenamento dos pais
        ordering = []
```

Herança e relações reversas

Porque a herança com multi-tabelas usa um campo implícito `OneToOneField` para ligar o filho ao pai, é possível mover propriedades do pai para o filho, como no exemplo acima. Entretanto, isso usa o nome que é o valor padrão `related_name` para relações `django.db.models.fields.ForeignKey` e `django.db.models.fields.ManyToManyField`. Se você está colocando estes tipos de relações sobre uma subclasse

de outro modelo, você **deve** especificar o atributo `related_name` em cada campo. Se você esquecer, o Django gerará um erro quando você rodar o `validate` ou `syncdb`.

Por exemplo, usando a classe `Place` novamente, vamos criar outra subclasse com um `ManyToManyField`:

```
class Supplier(Place):
    # Você deve especificar o related_name em todas as relações.
    customers = models.ManyToManyField(Restaurant,
                                       related_name='provider')
```

Especificando o campo `parent_link`

Como mencionado, o Django criará automaticamente um `OneToOneField` ligando sua classe filha com qualquer modelo pai não-abstrato. Se você deseja controlar o nome dos atributos de ligação, você pode criar seu próprio `OneToOneField` e setar `parent_link=True`, para indicar que seu campo é o link para a classe pai.

Herança múltipla

Da mesma forma que as subclasses do Python, é possível para um modelo do Django ter herança de múltiplos modelos pais. Mantenha em mente a aplicação da resolução normal de nomes do Python. A primeira classe de base em que um nome particular aparece (ex: *Meta*) será utilizada. Nós paramos de procurar assim que encontramos um nome. Isso significa que se os múltiplos pais possuem uma classe *Meta*, somente o primeiro será usado. Todos os outros serão ignorados.

Geralmente você não necessita de herança múltipla. O principal caso de uso onde isso se torna comum é para classes “mix-in”: adicionando um campo ou método extra em particular para cada classe que herdar do mix-in. Tente manter suas hierarquias o mais simples possível, para que você não trave batalhas para descobrir de onde vem uma certa informação.

“Esconder” nomes de campos não é permitido

Numa classe normal do Python, é admissível para uma classe filha sobrescrever qualquer atributo de sua classe pai. No Django, isto não é permitido para atributos que são instâncias de `Field` (pelo menos, não neste momento). Se a classe base tem um filho chamado `author`, você não pode criar outro campo para o model chamado `author` em qualquer classe que herde da classe base.

Sobrescrever campos no model pai conduz a dificuldade em áreas como inicialização de novas instâncias (especificando que os campos são inicializados no `Model.__init__`) e serialização. Estas são funcionalidades que a herança de classes normal do Python não consegue lidar da mesma forma, então a diferença entre a herança de model do Django e herança de classes do Python não é meramente arbitrária.

Esta restrição somente se aplica aos atributos que são instâncias do `Field`. Atributos normais do Python podem ser sobrescritos se você desejar. Ele também se aplica apenas ao nome do atributo como Python o vê: se você está manualmente especificando o nome da coluna do banco de dados, você pode ter o nome da coluna aparecendo em ambas uma filha e um model antecessor para herança multi-tabela (elas são colunas em duas tabelas diferentes no banco de dados).

Django will raise a `FieldError` exception if you override any model field in any ancestor model. O Django irá lançar uma exceção `FieldError` se você sobrescrever qualquer campo em qualquer model antecessor.

Fazendo consultas

Uma vez que você tenha criado seus *data models*, o Django automaticamente lhe dá uma API de abstração de banco de dados que permite você criar, receber, atualizar e deletar objetos. Este documento explica como usar esta API. Visite a *referência do model* para detalhes completos de todas opções de model.

Por todo esse guia (e na referência), nós iremos referir aos seguintes models, que compreendem um aplicação de weblog:

```
class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __unicode__(self):
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=50)
    email = models.EmailField()

    def __unicode__(self):
        return self.name

class Entry(models.Model):
    blog = models.ForeignKey(Blog)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateTimeField()
    authors = models.ManyToManyField(Author)

    def __unicode__(self):
        return self.headline
```

Criando objetos

Para representar dados de uma tabela de banco de dados em objetos Python, o Django usa um sistema intuitivo: Uma classe model representa uma tabela de banco de dados, e uma instância dessa classe representa um dado em particular dentro da tabela.

Para criar um objeto, instancie-o usando argumentos nomeados para a classe model, e então chame `save()` para salvá-lo no banco de dados.

You import the model class from wherever it lives on the Python path, as you may expect. (We point this out here because previous Django versions required funky model importing.)

Assumimos que os models estão em `mysite/blog/models.py`, aqui tem um exemplo:

```
>>> from mysite.blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

Isso executa uma consulta SQL INSERT por trás das cenas. O Django não executa nada no banco de dados até que você, explicitamente `save()`.

O método `save()` não retorna valores.

See also:

O `save()` recebe algumas opções avançadas não descritas aqui. Veja a documentação do `save()` para detalhes completos.

Para criar um objeto e salvá-lo em um passo veja o método `create()`.

Salvando mudanças de objetos

Para salvar mudanças de um objeto que já existe no banco de dados, use `save()`.

Dado uma instância `b5` do `Blog` que já está salvo no banco de dados, este exemplo muda seu nome e o atualiza no banco de dados:

```
>> b5.name = 'New name'
>> b5.save()
```

Isso executa uma consulta SQL `UPDATE` por trás das cenas. O Django não acessa o banco de dados até que você, explicitamente, chame `save()`.

Salvando campos `ForeignKey` e `ManyToManyField`

Atualizar um campo `ForeignKey` funciona exatamente da mesma forma como salvar um campo normal; simplesmente atribuindo um objeto do tipo certo ao campo em questão:

```
>>> cheese_blog = Blog.objects.get(name="Cheddar Talk")
>>> entry.blog = cheese_blog
>>> entry.save()
```

Atualizar um `ManyToManyField` funciona um pouco diferente; usa o método `add()` sobre o campo para adicionar um dado a relação:

```
>> joe = Author.objects.create(name="Joe")
>> entry.authors.add(joe)
```

O Django se queixará se você tentar atribuir ou adicionar um objeto do tipo errado.

Recebendo objetos

Para receber objetos de seu banco de dados, você constrói uma `QuerySet` através do `Manager` de sua classe `model`.

Um `QuerySet` representa uma coleção de objetos do banco de dados. Ela pode ter nenhum, um ou muitos *filtros* – critérios que apurem a coleção baseado nos parâmetros dados. Em termos SQL, um `QuerySet` equipara-se a uma consulta `SELECT`, e um filtro é uma cláusula limitadora como `WHERE` ou `LIMIT`.

Você obtém um `QuerySet` usando o `Manager` de seu `model`. Cada `model` tem pelo menos um `Manager`, e ele é chamado `objects` por padrão. Acesse-o diretamente via classe `model`, desta forma:

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objects
Traceback:
...
AttributeError: "Manager isn't accessible via Blog instances."
```

Note: Os `Managers` são acessíveis somente via classes, ao invés de instâncias, para reforçar a separação entre operações a nível de tabela e operações a nível de dado.

O `Manager` é a principal fonte de `QuerySets` para um `model`. Ele age como uma `QuerySet` “raiz” que descreve todos os objetos de uma tabela de banco de dados. Por exemplo, `Blog.objects` é o `QuerySet` inicial que contém todos os objetos do `Blog` no banco de dados.

Recebendo todos os objetos

A forma mais simples de receber objetos da tabela é obtendo todos eles. Para fazer isso, use o método `all()` de um `Manager`:

```
>>> all_entries = Entry.objects.all()
```

O método `all()` retorna um *QuerySet* de todos os objetos do banco de dados.

(Se `Entry.objects` é um *QuerySet*, porquê nós fizemos um `Entry.objects`? Isso é porque `Entry.objects`, o *QuerySet* raiz, é um caso especial que não pode ser avaliado. O método `all()` retorna um *QuerySet* que *pode* ser avaliado.)

Recebendo objetos específicos com filtros

O *QuerySet* raiz fornecido pelo *Manager* descreve todos os objetos na tabela de banco de dados. Geralmente, contudo, você precisará selecionar somente um conjunto de objetos.

Para criar um subconjunto, você precisa refinar o *QuerySet* inicial, adicionando condições ao filtro. As duas formas mais comuns de refinar um *QuerySet* são:

```
``filter(**kwargs)``
    Retorna um novo ``QuerySet`` contendo objetos que combinam com os
    parâmetros fornecidos.

``exclude(**kwargs)``
    Retorna um novo ``QuerySet`` contendo objetos que *não* combinam com os
    parâmetros fornecidos.
```

Os parâmetros (`**kwargs` na definição de função acima) devem estar no formato descrito em *Campos de pesquisa* abaixo.

Por exemplo, para obter um *QuerySet* de entradas de blog do ano 2006, use o `filter()` desta forma:

```
Entry.objects.filter(pub_date__year=2006)
```

Nós não temos que adicionar um `all()` – `Entry.objects.all().filter(...)`. Que continua funcionando, mas você somente precisa de `all()` quando deseja obter todos os objetos do *QuerySet* raiz.

Filtros encadeados

O resultado de refinar uma *QuerySet* é, em si, um *QuerySet*, então é possível encadear refinamentos juntos. Por exemplo:

```
>>> Entry.objects.filter(
...     headline__startswith='What '
... ).exclude(
...     pub_date__gte=datetime.now()
... ).filter(
...     pub_date__gte=datetime(2005, 1, 1)
... )
```

Isso recebe o *QuerySet* inicial de todas as entradas do banco de dados, adiciona um filtro, então uma exclusão, e então um outro filtro. O resultado final é um *QuerySet* contendo todas as entradas com um cabeçalho que começa com “What”, que foi publicado entre Janeiro 1, 2005 e a data atual.

QuerySets filtrados são únicos

Cada vez que você refina um *QuerySet*, você tem um *QuerySet* novo que não é vinculado ao *QuerySet* anterior. Cada refinamento cria um *QuerySet* separado e distinto que pode ser armazenado, usado e reusado.

Exemplo:

```
>> q1 = Entry.objects.filter(headline__startswith="What ")
>> q2 = q1.exclude(pub_date__gte=datetime.now())
>> q3 = q1.filter(pub_date__gte=datetime.now())
```

Este três `QuerySet` são separadas. O primeiro é um `QuerySet` base contendo todas as entradas que contenham um cabeçalho começando com “What”. O segundo é um subconjunto do primeiro, com um critério adicional que exclui dados cujo o `pub_date` é maior que agora. O terceiro é um subconjunto do primeiro, com um critério adicional que seleciona somente os dados cujo `pub_date` é maior que agora. O `QuerySet` inicial (`q1`) não é afetado pelo processo de refinamento.

QuerySets são lazy

Os `QuerySets` are lazy – o ato de criar um `QuerySet` não envolve qualquer atividade de banco de dados. Você pode empilhar filtros juntos ao longo do dia, e o Django não os executa no banco de dados até que o `QuerySet` seja *avaliado*. Dê uma olhada nesse exemplo:

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.now())
>>> q = q.exclude(body_text__icontains="food")
>>> print q
```

Embora isso pareça como três hits no banco de dados, de fato o banco de dados foi consultado somente na última linha (`print q`). Geralmente, os resultados de um `QuerySet` não são buscados do banco de dados até que você “peça” por eles. Quando você o faz, o `QuerySet` é *avaliado* para acessar o banco de dados. Para mais detalhes sobre exatamente quando as avaliações tomam seu lugar, veja [Quando QuerySets são avaliados](#).

Outros métodos do QuerySet

Na maior parte do tempo você usará `all()`, `filter()` e `exclude()` quando você precisar pesquisar por objetos no banco de dados. Entretanto, isso não é tudo que há; veja a [Referência de API do QuerySet](#) para uma lista completa de todos os vários métodos do `QuerySet`.

Limitando QuerySets

Use a sintaxe de array-slicing do Python para limitar seu `QuerySet` com um certo número de resultados. Este é o equivalente as cláusulas SQL `LIMIT` e `OFFSET`.

Por exemplo, isso retorna os primeiros 5 objetos (`LIMIT 5`):

```
>>> Entry.objects.all()[:5]
```

Isso retorna do sexto ao décimo objeto (`OFFSET 5 LIMIT 5`):

```
>>> Entry.objects.all()[5:10]
```

Indexação negativa (i.e. `Entry.objects.all()[-1]`) não é suportada.

Normalmente, dividir um `QuerySet` resulta em um novo `QuerySet` – e não executa um nova consulta. Uma exceção é se você usa o parâmetro “step” do Python na sintaxe do slice. Por exemplo, isso executaria na verdade a consulta para retornar uma lista dos 10 primeiros objetos a cada *segundo*:

```
>>> Entry.objects.all()[0:10:2]
```

Para receber um *único* objeto ao invés de uma lista (e.g. `SELECT foo FROM bar LIMIT 1`), use um index simples ao invés de um slice. Por exemplo, isso retorna a primeira `Entry` do banco de dados, depois de ordenar as entradas alfabeticamente pelos cabeçalhos:

```
>>> Entry.objects.order_by('headline')[0]
```

Isso é o equivalente a:

```
>>> Entry.objects.order_by('headline')[0:1].get()
```

Note, no entanto, que o primeiro desses será lançado `IndexError` enquanto o segundo lançará `DoesNotExist` se nenhum objeto combinar com o critério dado. Veja `get()` para mais detalhes.

Campos de pesquisa

Campos de pesquisa são como você especifica o cerne de uma cláusula `WHERE`. Eles são especificado como argumento nomeados para o método `filter()` do `QuerySet`.

Os argumentos básicos de pesquisa tem a forma `campo__tipodepesquisa=valor`. (Isso é um underscore duplo). Por exemplo:

```
>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

é traduzido para o seguinte SQL:

```
SELECT * FROM blog_entry WHERE pub_date <= '2006-01-01';
```

Como isso é possível

O Python tem a habilidade de definir funções que aceitam argumentos de nome-valor arbitrários cujo os nomes e valores são avaliados em tempo de execução. Para mais informações, veja [Keyword Arguments](#) no tutorial oficial do Python.

Se você passar um argumento inválido, uma função de pesquisa gerará o erro `TypeError`.

A API de banco de dados suporta sobre dois dos tipos de pesquisa; uma referência completa poe ser encontrada em [referencia dos campos de pesquisa](#). Para dar a você um gosto do que está disponível, aqui temos algumas das pesquisas mais comuns que você provavelmente usa:

```
:lookup:`exact`
    Uma combinação "extata". Por exemplo::

    >>> Entry.objects.get(headline__exact="Man bites dog")

    Podria gerar o SQL ao longo dessas linhas:

    .. code-block:: sql

        SELECT ... WHERE headline = 'Man bites dog';

    Se você não fornecer um tipo de pesquisa -- isto é, se seu argumento não
    contém um underscore duplo -- o tipo de pesquisa é assumido como
    ``exact``

    Por exemplo, as duas cláusulas seguintes são equivalentes::

    >>> Blog.objects.get(id__exact=14)  # Forma explícita
    >>> Blog.objects.get(id=14)         # __exact é implícito

    Isso é uma conveniência, pois pesquisas ``exact`` são um caso comum.

:lookup:`iexact`
    Uma combinação não sensível a maiúsculas. Então, a consulta::

    >>> Blog.objects.get(name__iexact="beatles blog")

    Poderia combinar um ``Blog`` entitulado "Beatles Blog", "beatles blog",
    ou mesmo "BeAtLeS bLOg".
```

```
:lookup:`contains`
    Sensível a maiúsculas, testa se contém. Por exemplo::

        Entry.objects.get(headline__contains='Lennon')

    É traduzido para este SQL:

    .. code-block:: sql

        SELECT ... WHERE headline LIKE '%Lennon%';

    Note que combinará com cabeçalhos ``'Today Lennon honored'`` mas não
    ``'today lennon honored'``.

    Há também uma versão não sensível a maiúsculas, :lookup:icontains`.

:lookup:`startswith`, :lookup:`endswith`
    Começa com e termina com, respectivamente. Há também a versão não
    sensível a maiúsculas chamada :lookup:`istartswith` e
    :lookup:`iendswith`.
```

De novo, estes são somente arranhões na superfície. Uma referência completa pode ser encontrada na [referência os campos de pesquisa](#).

Pesquisas que abrangem relacionamentos

O Django oferece um poderoso e intuitivo meio de “seguir” relacionamentos numa pesquisa, preocupando-se com os JOINS SQL por vocês automaticamente, por trás das cenas. Para abranger um relacionamento, é só usar o nome dos campos relacionados através dos models, separados por dois underscores, até que você obtenha os campos que você deseja.

Este exemplo recebe todos os objetos Entry com um Blog cujo name seja 'Beatles Blog':

```
>>> Entry.objects.filter(blog__name__exact='Beatles Blog')
```

Esta abrangência pode ser tão funda quanto você quiser.

Ela funciona com backwards, também. Para referenciar um relacionamento “reverso”, é só colocar em minúsculo o nome do model.

Este exemplo recebe todos os objetos Blog que tenham pelo menos um Entry cujo headline contenha 'Lennon':

```
>>> Blog.objects.filter(entry__headline__contains='Lennon')
```

Se você estiver fazendo filtragem através de vários relacionamentos e um dos models intermediários não tiver um valor que fechar com a condição do filtro, o Django tratará ela como se estivesse vazia (todos os valores são NULL), mas válidos. Tudo isso significa que nenhum erro será gerado. Por exemplo, neste filtro:

```
Blog.objects.filter(entry__author__name='Lennon')
```

(se houvesse um model relacionado Author), caso não houvesse nenhum author associado com uma entrada, ele poderia ser tratado como se também não tivesse nenhum name atachado, ao invés de lançar um erro por não ter o author. Normalmente este é exatamente isso que você espera que aconteça. O único caso onde ele pode ser confuso é se você estiver usando isnull. Deste modo:

```
Blog.objects.filter(entry__author__name__isnull=True)
```

retornará objetos Blog que possuem um name vazio no author e também aqueles que têm um author vazio sobre os entry. Se você não quer esses últimos objetos, você poderia escrever:

```
Blog.objects.filter(entry__author__isnull=False,
                    entry__author__name__isnull=True)
```

Abrangendo relacionamentos de múltiplos valores

Please, see the release notes Quando você está filtrando um objeto baseado no `ManyToManyField` ou numa `ForeignKeyField` reverso, há dois diferentes tipos de filtro que podem ser interessantes. Considere o relacionamento `Blog/Entry` (`Blog` para `Entry` é um relacionamento one-to-many). Nós podemos estar interessados em encontrar blogs que tenham uma entrada que possui ambos, “*Lennon*” no título e que foi publicado em 2008. Ou nós podemos querer encontrar blogs que tenham uma entrada com “*Lennon*” no título bem como uma que fora publicada em 2008. Já que há várias entradas associadas com um único `Blog`, ambas consultas são possíveis e fazem sentido em algumas situações.

O mesmo tipo de situação surge com um `ManyToManyField`. Por exemplo, se uma `Entry` tem um `ManyToManyField` chamado `tags`, nós podemos querer encontrar entradas linkadas às tags chamadas “*music*” e “*bands*” ou nós podemos desejar uma entrada que contenha uma tag com o nome de “*musci*” e um status de “*public*”.

Para lidar com ambas situações, o Django tem um forma consistente de processar chamadas `filter()` e `exclude()`. Tudo dentro de um única chamada `filter()` é aplicado simultaneamente para filtrar itens que combinam com todos esses requisitos. Sucessivas chamadas `filter()` adicionais restringem o conjunto de objetos, mas para relações com múltiplos valores, se aplicam a qualquer objeto linkado a primeira chamada `filter()`.

Eles podem soar um pouco confusa, então espero que um exemplo dê uma clareada. Para selecionar todos os blogs que contêm entradas tanto com “*Lennon*” no título e que foram publicados em 2008 (a mesma entrada satisfazendo ambas condições), nós escreveríamos:

```
Blog.objects.filter(entry__headline__contains='Lennon',
                    entry__pub_date__year=2008)
```

Para selecionar todos os blogs que contêm um entrada com “*Lennon*” no título **bem como** uma entrada que fora publicada em 2008, nós escreveríamos:

```
Blog.objects.filter(entry__headline__contains='Lennon').filter(
    entry__pub_date__year=2008)
```

Neste segundo exemplo, o primeiro filtro restringe o queryset para todos os blogs linkados a esse tipo particular de entrada. O segundo filtro restringe o conjunto de blogs mais as que também foram linkadas ao segundo tipo de entrada. As entradas selecionadas no segundo filtro podem ou não, ser as mesmas entradas do primeiro filtro. Nós estamos filtrando os itens `Blog` com cada declaração de filtro, não itens `Entry`.

Todo esse comportamento também é aplicado ao `exclude()`: todas as condições numa única declaração `exclude()` é aplicada a uma única instância (se estas condições estiverem falando dessa mesma relação de múltiplos valores). Condições em chamadas `filter()` ou `exclude()` subsequentes que referem-se ao mesmo relacionamento podem terminar filtrando diferentes objetos linkados.

Atalho para pk

Por conveniência, o Django fornece um atalho para pesquisas com `pk`, que significa “chave primária”.

No exemplo `Blog` model, a chave primária é o campo `id`, então estas três regras são equivalente:

```
>>> Blog.objects.get(id__exact=14) # Forma explicita
>>> Blog.objects.get(id=14) # __exact é implícito
>>> Blog.objects.get(pk=14) # pk implica em id__exact
```

O uso de `pk` não é limitado a consultas `__exact` – qualquer termo de consulta pode ser combinado com `pk` para executar uma consulta sobre a chave primária de um model:


```
# Pega as entradas dos blogs com id 1, 4 e 7
>>> Blog.objects.filter(pk__in=[1,4,7])

# Pega todas as entradas do blog com id > 14
>>> Blog.objects.filter(pk__gt=14)
```

Pesquisas com `pk` também podem funcionar através de joins. Por exemplo, estas três regras são equivalentes:

```
>>> Entry.objects.filter(blog__id__exact=3) # Froma explicita
>>> Entry.objects.filter(blog__id=3)        # __exact é implícito
>>> Entry.objects.filter(blog__pk=3)         # __pk implica em __id__exact
```

Escapando sinais de porcentagem e underscores em consultas LIKE

Os campos de pesquisa que equacionam consultas SQL LIKE (`iexact`, `contains`, `icontains`, `startswith`, `istartswith`, `endswith` e `iendswith`) terão automaticamente escapados os dois caracteres especiais usados no LIKE – o sinal de porcentagem e o underscore. (Numa regra LIKE, o sinal de porcentagem significa um coringa de multiplos caracteres e o underscore significa um coringa de um único caractere.)

Isso significa que coisas devem funcionar intuitivamente, então a abstração não vaza. Por exemplo, para receber todas as entradas que contenham um sinal de porcentagem, é só usar o sinal de porcentagem como qualquer outro caractere:

```
>>> Entry.objects.filter(headline__contains='%')
```

O Django se encarrega de escapar para você; o SQL resultante parecerá com algo assim:

```
SELECT ... WHERE headline LIKE '%\%%';
```

O mesmo serve para underscores. Ambos, sinais de porcentagem e underscores, são tratados para você transparentemente.

Cacheamento e QuerySets

Cada `QuerySet` contém um cache, para minizar o acesso ao banco de dados. É importante entender como ele funciona, a fim de se escrever um código mais eficiente.

Num `QuerySet` recém criado, o cache está vazio. Na primeira vez que um `QuerySet` for avaliado – e, por isso, uma consulta ao banco de dados acontece – o Django salva o resultado da consulta num cache de `QuerySet` e retorna os resultado que foram explicitamente requisitados (e.g., o próximo elemento, se o `QuerySet` estiver sendo iterado). Avaliações subsequentes de um `QuerySet` reusam os resultados cacheados.

Mantenha esse comportamento de cache em mente, pois ele pode morder você se você não usar seus `QuerySets` correntemente. Por exemplo, a seguir serão criados dois `QuerySets`, eles serão avaliados, e mostrados na tela:

```
>>> print [e.headline for e in Entry.objects.all()]
>>> print [e.pub_date for e in Entry.objects.all()]
```

Isso significa que a mesma consulta ao banco de dados será executada duas vezes, efetivamente dobrando sua carga sobre o banco de dados. Também, há a possibilidade de duas listas que podem não incluir os mesmos dados do banco, pois uma `Entry` pode ter sido adicionada ou deletada na fração de segundo que divide as duas requisições.

Para evitar este problema, simplesmente salve o `QuerySet` e re-use-o:

```
>>> queryset = Poll.objects.all()
>>> print [p.headline for p in queryset] # Avalia o query set.
>>> print [p.pub_date for p in queryset] # Re-usa o que está em cache.
```

Consultas complexas com objetos Q

Consultas com palavras chaves – no `filter()`, etc. – são mescladas com “AND”. Se você precisa executar consultas mais complexas (por exemplo, consultas com OR), você pode usar objetos Q.

Um objeto Q (`django.db.models.Q`) é um objeto usado para encapsular uma coleção de argumentos nomeados. Estes argumentos são especificados assim como nos “Campos de pesquisa” acima.

Por exemplo, este objeto Q encapsula uma única consulta LIKE:

```
Q(question__startswith='What')
```

Objetos Q podem ser combinados usando os operadores & e |. Quando um operador é usado em dois objetos Q, ele produz um novo objeto Q.

Por exemplo, esta regra produz um único objeto Q que representa o “OR” ou duas consultas “question__startswith”:

```
Q(question__startswith='Who') | Q(question__startswith='What')
```

Este é equivalente a seguinte cláusula SQL WHERE:

```
WHERE question LIKE 'Who%' OR question LIKE 'What%'
```

Você pode compor declarações de complexidade arbitrária combinando objetos Q com os operadores & e | e usar o agrupamento paramétrico. Também, objetos Q podem ser negados usando o operador ~, permitindo combinações entre consultas normais e negadas (NOT):

```
Q(question__startswith='Who') | ~Q(pub_date__year=2005)
```

Cada função de pesquisa que recebe argumentos nomeados (e.g. `filter()`, `exclude()`, `get()`) podem também ser passados para um ou mais objetos Q como um argumento posicional. Se você fornece vários argumentos de objeto Q para uma função de pesquisa, os argumentos serão mesclados com “AND”. Por exemplo:

```
Poll.objects.get(
    Q(question__startswith='Who'),
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6))
)
```

... isso se traduz aproximadamente no SQL:

```
SELECT * from polls WHERE question LIKE 'Who%'
AND (pub_date = '2005-05-02' OR pub_date = '2005-05-06')
```

Funções de pesquisa podem combinar o uso de objetos Q e argumentos nomeados. Todos os argumentos fornecidos para uma função de pesquisa (sejam eles argumentos ou objetos Q) são mesclados com “AND”. No entanto, se um objeto Q é fornecido, ele deve preceder a definição de qualquer argumento nomeado. Por exemplo:

```
Poll.objects.get(
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)),
    question__startswith='Who')

```

... poderia ser uma consulta válida, equivalente ao exemplo anterior; mas:

```
# CONSULTA INVÁLIDA
Poll.objects.get(
    question__startswith='Who',
    Q(pub_date=date(2005, 5, 2)) | Q(pub_date=date(2005, 5, 6)))

```

... não seria válida.

See also:

O [exemplos do OR](#) nos unit tests do Django é mostrado alguns possíveis usos do Q.

Comparando objetos

Para comparar duas instâncias de model, é só usar o operador de comparação padrão do Python, dois sinais de igual: `==`. Por trás das cenas, o que é comparado é a chave primária dos dois models.

Usando o exemplo `Entry` acima, as duas declarações a seguir são equivalentes:

```
>>> some_entry == other_entry
>>> some_entry.id == other_entry.id
```

Se uma chave primária de um model não é chamada de `id`, não tem problema. As comparações sempre usaram a chave primária, seja lá como for chamada. Por exemplo, se uma chave primária de um model é chamada `name`, estas duas declarações são equivalentes:

```
>>> some_obj == other_obj
>>> some_obj.name == other_obj.name
```

Deletando objetos

O método `delete()`, convenientemente é chamado `delete()`. Este método deleta imediatamente o objeto e não retorna valor. Exemplo:

```
e.delete()
```

Você pode também deletar objetos em grupos. Todo `QuerySet` tem um método `delete()`, que deleta todos os seus membros.

Por exemplo, isso deleta todos os objetos `Entry` com um `pub_date` do ano 2005:

```
Entry.objects.filter(pub_date__year=2005).delete()
```

Tenha em mente que isso irá, sempre que possível, ser executado puramente em SQL, sendo assim o método `delete()` de instâncias de objeto individuais não precisaram ser chamadas durante o processo. Se você forneceu um método `delete()` a uma classe model e quer se assegurar de que ele será chamado, você precisa deletar as instâncias “manualmente” (e.g., iterando sobre o `QuerySet` e chamando o `delete()` de cada objeto individualmente) ao invés de usar um método `delete()` de grupo do `QuerySet`.

Quando o Django deleta um objeto, ele emula o comportamento de restrições (CONSTRAINT) do SQL `ON DELETE CASCADE` – em outras palavras, quaisquer objetos que possuam uma chave estrangeira apontando ao objeto que está para ser deletado, também será deletado com ele. Por exemplo:

```
b = Blog.objects.get(pk=1)
# Isso irá deletar o Blog e todos os seus objetos Entry.
b.delete()
```

Note que `delete()` é o único método de `QuerySet` que não é exposto num `Manager` em si. Este é um mecanismo de segurança para prevenir que você acidentalmente requisite `Entry.objects.delete()`, e apague todas as entradas. Se você deseja *deletar* todos os objetos, então você deve explicitamente requisitar uma consulta completa:

```
Entry.objects.all().delete()
```

Atualizando vários objetos de uma vez

Please, see the release notes As vezes você quer atribuir um valor em particular a um campo de todos os objetos de um `QuerySet`. Você pode fazer isso com o método `update()`. Por exemplo:

```
# Atualiza todos os headlines com o pub_date em 2007.
Entry.objects.filter(pub_date__year=2007).update(headline='Everything is the same')
```

Você somente pode setar campos não relacionados e campos `ForeignKey` usando este método, e o valor que você configurar o campo deve ser um valor nativo do Python (i.e., você não pode configurar um campo para ser igual a algum outro campo nesse momento).

Para atualizar campos `ForeignKey`, configure o novo valor na nova instância de model que você deseja apontar. Exemplo:

```
>>> b = Blog.objects.get(pk=1)

# Muda todos os Entry que pretendam a este Blog.
>>> Entry.objects.all().update(blog=b)
```

O método `update()` é aplicado instantaneamente e não retorna valores (semelhante ao `delete()`). A única restrição do `QuerySet` que foi atualizado é que ele só pode acessar uma tabela do banco de dados, a tabela principal do model.

Tenha cuidado que o método `update()` é convertido diretamente numa declaração SQL. Ela é uma operação de massa para atualizações direta. Não executa qualquer método `save()` no seus models, ou emite os sinais `pre_save` ou `post_save` (que são consequências da chamada do `save()`). Se você deseja salvar todos os itens de um `QuerySet` assegure-se de que o método `save()` seja chamada em cada instância, você não precisa de nenhuma função especial para lidar com isso. É só iterar sobre eles e chamar o `save()`:

```
for item in my_queryset:
    item.save()
```

Objetos relacionados

Quando você define um relacionamento num model (i.e., um `ForeignKey`, `OneToOneField`, ou `ManyToManyField`), instâncias desse model terão uma API conveniente para acessar os objetos relacionados.

Usando os models do topo dessa página, por exemplo, um objeto `Entry` `e` pode obter seus objetos `Blog` associados acessando um atributo `blog`: `e.blog`.

(Por trás das cenas, esta funcionalidade é implementada por [descriptors](#) Python. Isso realmente não deve importar, mas nós o apontamos aqui só por curiosidade.)

O Django também cria uma API de acessores para o *outro* lado do relacionamento – o link de um model relacionado para outro que define o relacionamento. Por exemplo, um objeto `Blog` `b` tem acesso a lista de todos os objetos `Entry` relacionados via o atributo `entry_set`: `b.entry_set.all()`.

Todos os exemplos nesta seção usam as amostras de model `Blog`, `Author` e `Entry` definidos no topo dessa página.

Relacionamentos Um-para-muitos (One-to-many)

Forward

Se um model tem uma `ForeignKey`, instâncias desse model terão acesso ao objeto relacionado (foreign) via um simples atributo do model.

Exemplo:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog # Retorna o objeto Blog relacionado.
```

Você pode obter um conjunto via um atributo foreign-key. Assim como você pode esperar, mudanças na chave estrangeira não são salvas no banco de dados até que você chame `save()`. Exemplo:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = some_blog
>>> e.save()
```

Se um campo `ForeignKey` tem `null=True` (i.e., ele permite valores `NULL`), você pode atribuir `None` a ele. Exemplo:

```
>>> e = Entry.objects.get(id=2)
>>> e.blog = None
>>> e.save() # "UPDATE blog_entry SET blog_id = NULL ...;"
```

Remeter acesso a relacionamentos one-to-many é chacheado na primeira vez em que o objeto relacionado é acessado. Subsequentes acessos a chave estrangeira do mesmo objeto são cacheados. Exemplo:

```
>>> e = Entry.objects.get(id=2)
>>> print e.blog # Consulta o banco de dados para receber o Blog associado.
>>> print e.blog # Não consulta o banco de dados. usa a versão em cache.
```

Note que o método `select_related()` do `QuerySet` recursivamente prepopula o cache de todos os relacionamentos one-to-many de antemão. Exemplo:

```
>>> e = Entry.objects.select_related().get(id=2)
>>> print e.blog # Não consulta o banco de dados; usa a versão em cache.
>>> print e.blog # Não consulta o banco de dados; usa a versão em cache.
```

Seguindo os relacionamentos “backward”

Se um model tem uma `ForeignKey`, instâncias da chave estrangeira do model terão acesso ao Manager que retorna todas as instâncias do primeiro model. Por padrão, esse Manager é chamado `FOO_set`, onde `FOO` é o nome do model fonte, em minúsculo. Esse Manager retorna `QuerySets`, que podem ser filtrados e manipulados como descrito na seção “Recebendo objetos” acima.

Exemplo:

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.all() # Retorna todas os objetos Entry relacionados ao Blog.

# b.entry_set é um Manager que retorna QuerySets.
>>> b.entry_set.filter(headline__contains='Lennon')
>>> b.entry_set.count()
```

Você pode sobrescrever o nome `FOO_set` através do parâmetro `related_name` na definição do `ForeignKey`. Por exemplo, se o model `Entry` fosse alterado para `blog = ForeignKey(Blog, related_name='entries')`, o código do exemplo acima ficaria:

```
>>> b = Blog.objects.get(id=1)
>>> b.entries.all() # Retorna todos os objetos Entry relacionados ao Blog.

# b.entries é um Manager que retorna QuerySets.
>>> b.entries.filter(headline__contains='Lennon')
>>> b.entries.count()
```

Você não pode acessar um Manager `ForeignKey` de uma classe no sentido contrário; ele deve ser acessado de uma instância:

```
>>> Blog.entry_set
Traceback:
...
AttributeError: "Manager must be accessed via instance".
```

Além dos métodos do `QuerySet` definidos em “Recebendo objetos” acima, o Manager `ForeignKey` possui métodos adicionais utilizados para lidar com conjuntos de objetos relacionados. Uma sinopse de cada um está abaixo, e detalhes completos podem ser encontrados na [referência de objetos relacionados](#).

add(obj1, obj2, ...) Adiciona os objetos do model especificados ao conjunto do objeto relacionado.

create(kwargs)** Cria um novo objeto, salva-o e coloca-o no conjunto do objeto relacionado. Retorna o novo objeto criado.

remove(obj1, obj2, ...) Remove o os objetos model especificados do conjunto do objeto relacionado.

clear() Remove todos os objetos do conjunto do objeto relacionado.

Para atribuir memntros de um conjunto relacionado de uma só vez, é só atribuí-lo de qualquer objeto iterável. O iterável pode conter instâncias de objetos, ou só uma lista de valores de chave primária. Por exemplo:

```
b = Blog.objects.get(id=1)
b.entry_set = [e1, e2]
```

Neste exemplo, e1 e e2 pode ser uma instância completa do Entry, ou valores inteiros de chave primária.

Se o método `clear()` estiver disponível, quaisquer objetos pre-existentes serão removidos do `entry_set` antes de todos os objetos no iterável (neste caso, uma lista) são adicionados ao conjunto. Se o método `clear()` não estiver disponível, todos os objetos no iterável serão adicionados sem remover quaisquer elementos existentes.

Cada operação “reverse” descrita nesta seção tem um efeito imediato sobre o banco de dados. Toda adição, criação e deleção é imediatamente e automaticamente salva no banco de dados.

Relacionamentos muitos-para-muitos (Many-to-many)

Ambas as extremidades de um relacionamento many-to-many obtêm uma API de acesso automática para o outro lado. A API funciona exatamente como o “backward” do relacionamento one-to-many, acima.

A única diferença é na nomeação do atributo: O model que define o `ManyToManyField` usa o nome do atributo do próprio campo, considerando que o model “reverso” usa o nome do model original em minúscula, mais ‘_set’ (assim como reverso de relacionamentos one-to-many).

Com um exemplo fica mais fácil entender:

```
e = Entry.objects.get(id=3)
e.authors.all() # Retorna todos os objetos Author desta Entry.
e.authors.count()
e.authors.filter(name__contains='John')

a = Author.objects.get(id=5)
a.entry_set.all() # Retorna todos os objetos Entry desse Author.
```

Como `ForeignKey`, o `ManyToManyField` pode especificar `related_name`. No exemplo acima, se o `ManyToManyField` em `Entry` tivesse especificado `related_name='entries'`, então cada instância de `Author` teria um atributo `entries` ao invés de `entry_set`.

Relacionamentos Um-para-um (One-to-one)

Relacionamentos One-to-one são muito similares aos relacionamentos many-to-many. Se você define uma `OneToOneField` no seu model, as instâncias desse model terão acesso ao objeto relacionado através de um simples atributo de um model.

Por exemplo:

```
class EntryDetail(models.Model):
    entry = models.OneToOneField(Entry)
    details = models.TextField()

ed = EntryDetail.objects.get(id=2)
ed.entry # Retorna o objeto Entry relacionado.
```

A diferença vem nas consultas “reverse”. O model relacionado num relacionamento one-to-one também tem acesso ao objeto *Manager*, mas esse *Manager* representa um único objeto, ao invés de uma coleção de objetos:

```
e = Entry.objects.get(id=2)
e.entrydetail # retorna o objeto EntryDetail relacionado
```

Se nenhum objeto foi atribuído a este relacionamento, o Django lançará uma exceção `DoesNotExist`.

Instâncias podem ser atribuídas ao relacionamento reverso da mesma forma como você poderia atribuir ao relacionamento forward:

```
e.entrydetail = ed
```

Como é possível relacionamentos backward?

Outro mapeador de objeto relacional requer que você defina relacionamentos em ambos os lados. Os desenvolvedores do Django acreditam que isso é uma violação do princípio DRY (Don't Repeat yourself), então o Django somente requer que você defina o relacionamento de um lado.

Mas como é possível, dado que uma classe model não sabe qual outras classes model estão relacionadas até que essas outras classes sejam carregadas?

A resposta está na configuração `INSTALLED_APPS`. A primeira vez em que qualquer model é carregado, o Django itera sobre todos os models no `INSTALLED_APPS` e cria os relacionamentos backward na memória se necessário. Essencialmente, uma das funções do `INSTALLED_APPS` é dizer ao Django os domínios dos models.

Consultas sobre objetos relacionados

Consultas envolvendo objetos relacionados seguem as mesmas regras de consultas envolvendo campos de valores normais. Quando especificar o valor para combinar numa consulta, você pode usar tanto uma instância de objeto em si, quando o valor da chave primária do objeto.

Por exemplo, se você tem um objeto `Blog` `b` com `id=5`, as três consultas seguintes seriam idênticas:

```
Entry.objects.filter(blog=b) # Consulta usando instância de objeto
Entry.objects.filter(blog=b.id) # Consulta usando id da instância
Entry.objects.filter(blog=5) # Consulta usando id diretamente
```

Voltando ao SQL puro

Se você se encontrou precisando escrever uma consulta SQL que é mais complexa para o mapeador de banco de dados do Django, você pode voltar ao modo de consultas em SQL puro.

A forma preferida para fazer isso é dando ao model, métodos personalizados ou métodos personalizados ao manager, que executam consultas. Embora não há nada no Django que *obrigue* consultas de banco de dados estarem na camada de model, esta abordagem mantém toda a lógica de acesso a dados num único lugar, o que é mais inteligente para organização de código. Para instruções, veja [Performing raw SQL queries](#).

Finalmente, é importante notar que a camada de banco de dados do Django é meramente uma interface para o seu banco de dados. Você pode acessar seu banco por outras ferramentas, linguagens de programação ou frameworks de banco de dados; não há nada específico do Django sobre o seu banco de dados.

Managers

`class Manager`

A `Manager` is the interface through which database query operations are provided to Django models. At least one `Manager` exists for every model in a Django application.

The way `Manager` classes work is documented in [Fazendo consultas](#); this document specifically touches on model options that customize `Manager` behavior.

Manager names

By default, Django adds a `Manager` with the name `objects` to every Django model class. However, if you want to use `objects` as a field name, or if you want to use a name other than `objects` for the `Manager`, you can rename it on a per-model basis. To rename the `Manager` for a given class, define a class attribute of type `models.Manager()` on that model. For example:

```
from django.db import models

class Person(models.Model):
    # ...
    people = models.Manager()
```

Using this example model, `Person.objects` will generate an `AttributeError` exception, but `Person.people.all()` will provide a list of all `Person` objects.

Custom Managers

You can use a custom `Manager` in a particular model by extending the base `Manager` class and instantiating your custom `Manager` in your model.

There are two reasons you might want to customize a `Manager`: to add extra `Manager` methods, and/or to modify the initial `QuerySet` the `Manager` returns.

Adding extra Manager methods

Adding extra `Manager` methods is the preferred way to add “table-level” functionality to your models. (For “row-level” functionality – i.e., functions that act on a single instance of a model object – use *Model methods*, not custom `Manager` methods.)

A custom `Manager` method can return anything you want. It doesn’t have to return a `QuerySet`.

For example, this custom `Manager` offers a method `with_counts()`, which returns a list of all `OpinionPoll` objects, each with an extra `num_responses` attribute that is the result of an aggregate query:

```
class PollManager(models.Manager):
    def with_counts(self):
        from django.db import connection
        cursor = connection.cursor()
        cursor.execute("""
            SELECT p.id, p.question, p.poll_date, COUNT(*)
            FROM polls_opinionpoll p, polls_response r
            WHERE p.id = r.poll_id
            GROUP BY 1, 2, 3
            ORDER BY 3 DESC""")
        result_list = []
        for row in cursor.fetchall():
            p = self.model(id=row[0], question=row[1], poll_date=row[2])
            p.num_responses = row[3]
            result_list.append(p)
        return result_list

class OpinionPoll(models.Model):
    question = models.CharField(max_length=200)
    poll_date = models.DateField()
    objects = PollManager()

class Response(models.Model):
    poll = models.ForeignKey(Poll)
    person_name = models.CharField(max_length=50)
    response = models.TextField()
```


With this example, you'd use `OpinionPoll.objects.with_counts()` to return that list of `OpinionPoll` objects with `num_responses` attributes.

Another thing to note about this example is that `Manager` methods can access `self.model` to get the model class to which they're attached.

Modifying initial Manager QuerySets

A `Manager`'s base `QuerySet` returns all objects in the system. For example, using this model:

```
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)
```

...the statement `Book.objects.all()` will return all books in the database.

You can override a `Manager`'s base `QuerySet` by overriding the `Manager.get_query_set()` method. `get_query_set()` should return a `QuerySet` with the properties you require.

For example, the following model has *two* `Managers` – one that returns all objects, and one that returns only the books by Roald Dahl:

```
# First, define the Manager subclass.
class DahlBookManager(models.Manager):
    def get_query_set(self):
        return super(DahlBookManager, self).get_query_set().filter(author='Roald_
↪Dahl')

# Then hook it into the Book model explicitly.
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    objects = models.Manager() # The default manager.
    dahl_objects = DahlBookManager() # The Dahl-specific manager.
```

With this sample model, `Book.objects.all()` will return all books in the database, but `Book.dahl_objects.all()` will only return the ones written by Roald Dahl.

Of course, because `get_query_set()` returns a `QuerySet` object, you can use `filter()`, `exclude()` and all the other `QuerySet` methods on it. So these statements are all legal:

```
Book.dahl_objects.all()
Book.dahl_objects.filter(title='Matilda')
Book.dahl_objects.count()
```

This example also pointed out another interesting technique: using multiple managers on the same model. You can attach as many `Manager()` instances to a model as you'd like. This is an easy way to define common “filters” for your models.

For example:

```
class MaleManager(models.Manager):
    def get_query_set(self):
        return super(MaleManager, self).get_query_set().filter(sex='M')

class FemaleManager(models.Manager):
    def get_query_set(self):
        return super(FemaleManager, self).get_query_set().filter(sex='F')

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
```

```
sex = models.CharField(max_length=1, choices= (('M', 'Male'), ('F', 'Female')))
people = models.Manager()
men = MaleManager()
women = FemaleManager()
```

This example allows you to request `Person.men.all()`, `Person.women.all()`, and `Person.people.all()`, yielding predictable results.

If you use custom `Manager` objects, take note that the first `Manager` Django encounters (in the order in which they're defined in the model) has a special status. Django interprets this first `Manager` defined in a class as the “default” `Manager`, and several parts of Django (though not the admin application) will use that `Manager` exclusively for that model. As a result, it's often a good idea to be careful in your choice of default manager, in order to avoid a situation where overriding of `get_query_set()` results in an inability to retrieve objects you'd like to work with.

Using managers for related object access

By default, Django uses an instance of a “plain” manager class when accessing related objects (i.e. `choice.poll`), not the default manager on the related object. This is because Django needs to be able to retrieve the related object, even if it would otherwise be filtered out (and hence be inaccessible) by the default manager.

If the normal plain manager class (`django.db.models.Manager`) is not appropriate for your circumstances, you can force Django to use the same class as the default manager for your model by setting the `use_for_related_fields` attribute on the manager class. This is documented fully [below](#).

Custom managers and model inheritance

Class inheritance and model managers aren't quite a perfect match for each other. Managers are often specific to the classes they are defined on and inheriting them in subclasses isn't necessarily a good idea. Also, because the first manager declared is the *default manager*, it is important to allow that to be controlled. So here's how Django handles custom managers and *model inheritance*:

1. Managers defined on non-abstract base classes are *not* inherited by child classes. If you want to reuse a manager from a non-abstract base, redeclare it explicitly on the child class. These sorts of managers are likely to be fairly specific to the class they are defined on, so inheriting them can often lead to unexpected results (particularly as far as the default manager goes). Therefore, they aren't passed onto child classes.
2. Managers from abstract base classes are always inherited by the child class, using Python's normal name resolution order (names on the child class override all others; then come names on the first parent class, and so on). Abstract base classes are designed to capture information and behavior that is common to their child classes. Defining common managers is an appropriate part of this common information.
3. The default manager on a class is either the first manager declared on the class, if that exists, or the default manager of the first abstract base class in the parent hierarchy, if that exists. If no default manager is explicitly declared, Django's normal default manager is used.

These rules provide the necessary flexibility if you want to install a collection of custom managers on a group of models, via an abstract base class, but still customize the default manager. For example, suppose you have this base class:

```
class AbstractBase(models.Model):
    ...
    objects = CustomerManager()

    class Meta:
        abstract = True
```

If you use this directly in a subclass, `objects` will be the default manager if you declare no managers in the base class:

```
class ChildA(AbstractBase):
    ...
    # This class has CustomManager as the default manager.
```

If you want to inherit from `AbstractBase`, but provide a different default manager, you can provide the default manager on the child class:

```
class ChildB(AbstractBase):
    ...
    # An explicit default manager.
    default_manager = OtherManager()
```

Here, `default_manager` is the default. The objects manager is still available, since it's inherited. It just isn't used as the default.

Finally for this example, suppose you want to add extra managers to the child class, but still use the default from `AbstractBase`. You can't add the new manager directly in the child class, as that would override the default and you would have to also explicitly include all the managers from the abstract base class. The solution is to put the extra managers in another base class and introduce it into the inheritance hierarchy *after* the defaults:

```
class ExtraManager(models.Model):
    extra_manager = OtherManager()

    class Meta:
        abstract = True

class ChildC(AbstractBase, ExtraManager):
    ...
    # Default manager is CustomManager, but OtherManager is
    # also available via the "extra_manager" attribute.
```

Controlling Automatic Manager Types

This document has already mentioned a couple of places where Django creates a manager class for you: *default managers* and the “plain” manager used to *access related objects*. There are other places in the implementation of Django where temporary plain managers are needed. Those automatically created managers will normally be instances of the `django.db.models.Manager` class. Throughout this section, we will use the term “automatic manager” to mean a manager that Django creates for you – either as a default manager on a model with no managers, or to use temporarily when accessing related objects.

Sometimes this default class won't be the right choice. One example is in the `django.contrib.gis` application that ships with Django itself. All *gis* models must use a special manager class (`GeoManager`) because they need a special queryset (`GeoQuerySet`) to be used for interacting with the database. It turns out that models which require a special manager like this need to use the same manager class wherever an automatic manager is created.

Django provides a way for custom manager developers to say that their manager class should be used for automatic managers whenever it is the default manager on a model. This is done by setting the `use_for_related_fields` attribute on the manager class:

```
class MyManager(models.Manager):
    use_for_related_fields = True
    ...
```

If this attribute is set on the *default* manager for a model (only the default manager is considered in these situations), Django will use that class whenever it needs to automatically create a manager for the class. Otherwise, it will use `django.db.models.Manager`.

Historical Note

Given the purpose for which it's used, the name of this attribute (`use_for_related_fields`) might seem a little odd. Originally, the attribute only controlled the type of manager used for related field access, which is where the name came from. As it became clear the concept was more broadly useful, the name hasn't been changed. This is primarily so that existing code will *continue to work* in future Django versions.

Writing Correct Managers For Use In Automatic Manager Instances

As already suggested by the `django.contrib.gis` example, above, the `use_for_related_fields` feature is primarily for managers that need to return a custom `QuerySet` subclass. In providing this functionality in your manager, there are a couple of things to be remember and that's the topic of this section.

Do not filter away any results in this type of manager subclass

One reason an automatic manager is used is to access objects that are related to from some other model. In those situations, Django has to be able to see all the objects for the model it is fetching, so that *anything* which is referred to can be retrieved.

If you override the `get_query_set()` method and filter out any rows, Django will return incorrect results. Don't do that. A manager that filters results in `get_query_set()` is not appropriate for use as an automatic manager.

Set `use_for_related_fields` when you define the class

The `use_for_related_fields` attribute must be set on the manager *class*, object not on an *instance* of the class. The earlier example shows the correct way to set it, whereas the following will not work:

```
# BAD: Incorrect code
class MyManager(models.Manager):
    ...

# Sets the attribute on an instance of MyManager. Django will
# ignore this setting.
mgr = MyManager()
mgr.use_for_related_fields = True

class MyModel(models.Model):
    ...
    objects = mgr

# End of incorrect code.
```

You also shouldn't change the attribute on the class object after it has been used in a model, since the attribute's value is processed when the model class is created and not subsequently reread. Set the attribute on the manager class when it is first defined, as in the initial example of this section and everything will work smoothly.

Performing raw SQL queries

Sinta-se livre para escrever consultas SQL nos métodos de seus modelos e métodos a nível de módulo. O objeto `django.db.connection` representa a conexão atual com o banco de dados, e `django.db.transaction` representa a transação atual do banco de dados. Para usá-lo, chame `connection.cursor()` para capturar o objeto cursor. Então, chame `cursor.execute(sql, [params])` para executar o SQL e `cursor.fetchone()` ou `cursor.fetchall()` para retornar o resultado em linhas. Após realizar uma operação que modifica os dados, você deve então chamar `transaction.commit_unless_managed()` para garantir que as mudanças estejam confirmadas no banco de dados. Se sua consulta é puramente uma operação de leitura de dados, nenhum `commit` é requerido. Por exemplo:

```
def my_custom_sql(self):
    from django.db import connection, transaction
    cursor = connection.cursor()

    # Operação de modificação de dado - commit obrigatório
    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
    transaction.commit_unless_managed()

    # Operação de recebimento de dado - não é necessário o commit
    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
    row = cursor.fetchone()

    return row
```

Transações e SQL puro

Se você está usando decoradores de transações (como `commit_on_success`) para envolver seus views e prover controle de transações, você não terá que fazer uma chamada manual para `transaction.commit_unless_managed()` – você pode comitar manualmente se você quiser, mas você não será obrigado, desde que o decorador comita por você. No entanto, se você não comitar manualmente suas mudanças, você terá de marcar, manualmente, a transação como suja, usando `transaction.set_dirty()`:

```
@commit_on_success
def my_custom_sql_view(request, value):
    from django.db import connection, transaction
    cursor = connection.cursor()

    # Operação de modificação de dado
    cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [value])

    # Se nós modificamos dados, marcamos a transação como suja
    transaction.set_dirty()

    # Operação de recebimento de dado. Isso não suja a transação,
    # então não é obrigatório chamar set_dirty()
    cursor.execute("SELECT foo FROM bar WHERE baz = %s", [value])
    row = cursor.fetchone()

    return render_to_response('template.html', {'row': row})
```

A chamada para `set_dirty()` é feita automaticamente quando você usa o ORM do Django para fazer modificações no banco de dados. Entretanto, quando você usa um SQL puro, o Django não tem como saber se seu SQL modifica dados ou não. A chamada manual `set_dirty()` assegura que o Django saiba que estes são dados modificados que devem ser comitados.

Connections e cursors

`connection` e `cursor` maioritariamente implementa a [DB-API padrão do Python](#) (exceto quando se trata de [manipulação de transações](#)). Se você não está familiarizado com a DB-API do Python, note que a consulta SQL em `cursor.execute()` possui marcadores, `"%s"`, ao invés de adicionar parâmetros diretamente dentro do SQL. Se você usa esta técnica, as bibliotecas de banco de dados subjacentes irão automaticamente adicionar aspas e espacar seus parâmetros quando necessário. (Também atente que o Django espera pelo marcador `"%s"`, não pelo marcador `"?"`, que é utilizado pelos bindings para Python do SQLite. Isto é por uma questão de coerência e bom senso.)

Um lembrete final: Se tudo que você quer é fazer uma cláusula `WHERE` customizada, você pode somente usar os argumentos `where`, `tables` e `params` da API padrão.

Gerenciando transações de banco de dados

O Django dá a você poucos caminhos para controlar como as transações de banco de dados são gerenciadas, isso se você estiver usando um banco de dados que suporta transações.

Comportamento padrão de transações com Django

O comportamento padrão do Django é executar com uma transação aberta que ele comita automaticamente quando quaisquer função interna que modifique a base de dados for chamada. Por exemplo, se você chamar `model.save()` ou `model.delete()`, a mudança vai ser realizada imediatamente.

Isso parece muito com a configuração auto-commit para a maioria dos bancos de dados. Tão logo você performe uma ação que precisa escrever no banco de dados, o Django produz um comando `INSERT/UPDATE/DELETE` e então faz um `COMMIT`. Não há `ROLLBACK` implícito.

Amarrando transações às requisições HTTP

A forma recomendada de lidar com transações em requisições Web é amarrá-las às fases requisição/resposta através do `TransactionMiddleware` do Django.

Funciona assim: quando uma requisição é iniciada, o Django começa uma transação. Se a resposta é produzida sem problemas, o Django comita quaisquer transações pendentes. Se a função view produz uma exceção, o Django faz um rollback de quaisquer transações pendentes.

Para ativar esta funcionalidade, apenas adicione o `TransactionMiddleware` middleware nos eu setting `MIDDLEWARE_CLASSES`:

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.CacheMiddleware',
    'django.middleware.transaction.TransactionMiddleware',
)
```

A ordem é bastante importante. O middleware de transação se aplica não somente às funções da view, mas também a todos os módulos middleware que vem após o mesmo. Então, se você usar o session middleware após o transaction middleware, a criação de sessões será parte da transação.

Uma exceção a isso seria o `CacheMiddleware`, que nunca é afetado. O cache middleware usa seu próprio cursor de banco de dados (que é mapeado para a sua própria conexão de banco de dados internamente).

Controlando o gerenciamento de transações direto nas views

Para a maioria das pessoas, transações baseadas em requisições funciona maravilhosamente bem. Contudo, se você precisa de um controle mais fino sobre como suas transações são gerenciadas, você pode usar os decoradores do Python para mudar a forma como as transações são lidadas por uma função view em particular.

Note: Mesmo os exemplos abaixo usando funções de view como exemplo, esses decoradores também podem ser aplicados para funções que não view.

`django.db.transaction.autocommit`

Use o decorador `autocommit` para mudar o comportamento padrão de commit de uma view, independente da configuração global das transações.

Exemplo:

```
from django.db import transaction

@transaction.autocommit
def viewfunc(request):
    ....
```

Dentro de `viewfunc()`, transações serão commitadas quão logo você chame `model.save()`, `model.delete()`, ou qualquer outra função que escreva na base de dados.

`django.db.transaction.commit_on_success`

Utilize o decorador `commit_on_success` para usar transações únicas para todo o trabalho feito em uma função:

```
from django.db import transaction

@transaction.commit_on_success
def viewfunc(request):
    ....
```

Se a função retorna com sucesso, então o Django irá commitar todo o trabalho feito na função até aquele ponto. Se a função levanta uma exceção, todavia, o Django irá fazer um rollback na transação.

`django.db.transaction.commit_manually`

Use o decorador `commit_manually` se você precisa de um controle completo sobre as transações. Ele diz ao Django que você mesmo irá gerenciar a transação.

Se seu view muda o dado e não executa um `commit()` ou `rollback()`, o Django irá lançar uma exceção `TransactionManagementError`.

Gerenciamento manual de transações parece com isso:

```
from django.db import transaction

@transaction.commit_manually
def viewfunc(request):
    ...
    # Você pode comitar/retroceder quando e onde quiser.
    transaction.commit()
    ...

    # Mas terá de lembrar de fazê-lo você mesmo!
    try:
        ...
    except:
        transaction.rollback()
    else:
        transaction.commit()
```

Uma nota importante para usuários de versões anteriores do Django:

Os métodos de banco de dados `connection.commit()` e `connection.rollback()` (chamados `db.commit()` e `db.rollback` na versão 0.91 e anteriores) não existem mais. Eles foram substituídos por `transation.commit()` e `transation.rollback()`.

Como desativar globalmente o gerenciamento de transações

Controladores obsecados, podem desabilitar totalmente todo gerenciamento de transações. Control freaks can totally disable all transaction management by setting `DISABLE_TRANSACTION_MANAGEMENT` to `True` in the Django settings file.

Se você fizer isso, o Django não irá prover qualquer gerenciamento de transação automático qualquer. O middleware não irá implicitamente comitar as transações, e você precisará gerenciar os rollbacks. Isto requer que você mesmo comita as alterações feitas pelo middleware e algo a mais.

Deste modo, é melhor usar isto em situações onde você precisa rodar o seu próprio middleware de controle de transações ou fazer algo realmente estranho. Em quase todas as situações, você se dará melhor usando o comportamento padrão, ou o middleware, e somente modificar as funções selecionadas caso precise.

Savepoints

Um savepoint é um marcador dentro de uma transação que habilita você a retornar parte de uma transação, ao invés de toda ela. Os savepoints estão disponíveis para backends PostgreSQL 8 e Oracle. Outros backends proverão funções savepoint, mas eles são operações vazias - eles na verdade não fazem nada.

Os savepoints não são usuais especialmente se você estiver usando o comportamento padrão autocommit do Django. Entretanto, se você estiver usando `commit_on_success` ou `commit_manually`, cada transação aberta construirá uma série de operações de banco de dados, que aguardarão um commit ou um rollback. Se você emitir um rollback, a transação inteira é retrocedida. Os savepoints fornecem um habilidade de executar rollbacks com granularidade fina, ao invés de um rollback completo que poderia ser executada pelo `transaction.rollback()`.

Os savepoints são controlados por três métodos do objeto transação:

`transaction.savepoint()`

Cria um novo savepoint. Este marca um ponto na transação que é tido como um “bom” estado.

Retorna o ID do savepoint (sid).

`transaction.savepoint_commit(sid)`

Atualiza o savepoint para incluir quaisquer operações que tenham sido executadas desde que o savepoint foi criado, ou desde o último commit.

`transaction.savepoint_rollback(sid)`

Retrocede a transação para o último ponto em que o savepoint foi comitado.

O exemplo a seguir demonstra o uso de savepoints:

```
from django.db import transaction

@transaction.commit_manually
def viewfunc(request):

    a.save()
    # abre a transação agora contendo a.save()
    sid = transaction.savepoint()

    b.save()
    # abre a transação agora contendo a.save() e b.save()

    if want_to_keep_b:
        transaction.savepoint_commit(sid)
        # abre a transação que ainda contém a.save() e b.save()
    else:
        transaction.savepoint_rollback(sid)
        # abre a transação agora contendo somente a.save()

    transaction.commit()
```


Transações no MySQL

Se você estiver usando o MySQL, suas tabelas podem ou não suportar transações; isto depende da versão do MySQL e do tipo de tabelas que você está usando. (Para “tipo de tabela,” significa algo como “InnoDB” ou “MyISAM”.) As peculiaridades de transações no MySQL são fora do escopo do artigo, mas o site do MySQL tem [informações sobre as transações no MySQL](#).

Se sua instalação do MySQL *não* suporta transações, então o Django irá funcionar no modo auto-comit: As consultas serão executadas e comitadas logo que forem chamadas. Se sua instalação do MySQL *suporta* transações, o Django irá manipular as transações como explicado neste documento.

Manipulando exceções em transações do PostgreSQL

Quando uma chamada para um cursor do PostgreSQL lança uma exceção (tipicamente `IntegrityError`), todo o SQL subsequente na mesma transação irá falhar com o erro “current transaction is aborted, queries ignored until end of transaction block”, ou seja, “a transação atual foi abortada, consultas ignoradas até o final do bloco da transação”. Embora seja improvável que o uso de um simples `save()` gere uma exceção no PostgreSQL, há outros usos mais avançados que podem, como o de salvar objetos com campos únicos, salvando usando o flag `force_insert/force_update`, ou invocando uma SQL customizada.

Há várias maneiras de se deparar com este tipo de erro.

Rollback de transações

A primeira opção é retroceder toda a transação. Por exemplo:

```
a.save() # Sucesso, mas pode ser desfeita com rollback de transação
try:
    b.save() # Could throw exception
except IntegrityError:
    transaction.rollback()
c.save() # sucesso, mas a.save() pode ter sido desfeita
```

Chamando `transaction.rollback()` retrocederá toda a transação. Qualquer operação de banco de dados não comitada será perdida. Neste exemplo, as mudanças feitas pelo `a.save()` poderiam ser perdidas, mesmo que a operação não gere um erro por si só.

Rollback de savepoints

Se você está usando o PostgreSQL 8 ou superior, você pode usar [savepoints](#) para controlar a extensão do rollback. Antes de realizar uma operação de banco de dados que possa falhar, você pode setar ou atualizar o savepoint; dessa forma, se a operação falhar, você pode retroceder uma única operação, ao invés de toda transação. Por exemplo:

```
a.save() # Sucesso, e nunca será desfeita pelo rollback do savepoint
try:
    sid = transaction.savepoint()
    b.save() # Poderia lançar uma exceção
    transaction.savepoint_commit(sid)
except IntegrityError:
    transaction.savepoint_rollback(sid)
c.save() # Sucesso, e a.save() não será desfeita
```

Neste exemplo, `a.save()` não será desfeito no caso de `b.save()` lançar uma exceção.

Manipulando requisições HTTP

Informações sobre manipulação de requisições HTTP no Django:

URL dispatcher

Um esquema de URLs limpo e elegante é um detalhe importante numa aplicação Web de qualidade. O Django permite que você defina as URLs da maneira que quiser, sem limitações impostas pelo framework.

Não são necessários `.php`, `.cgi` ou outras coisas sem o menor sentido como `0,2097,1-1-1928,00`.

Veja [Cool URIs don't change](#), pelo criador da World Wide Web Tim Berners-Lee, para ter excelentes argumentos do porquê as URLs devem ser limpas e facilmente utilizáveis.

Visão geral

Para definir URLs para uma aplicação, você cria um módulo Python informalmente chamado de **URLconf** (URL configuration). Este módulo é Python puro e é um mapeamento simples entre padrões de URL (na forma de expressões regulares simples) para funções Python de callback (suas views).

Este mapeamento pode ser tão extenso ou curto quanto necessário. Ele pode referenciar outros mapeamentos. E, por ser código Python puro, pode ser construído dinamicamente.

Como o Django processa uma requisição

Quando um usuário requisita uma página de nosso site em Django, este é o algoritmo que o sistema segue para determinar qual código Python irá executar:

1. O Django determina o módulo URLconf inicial a ser usado. Normalmente, este é o valor de configuração do `ROOT_URLCONF` no arquivo settings, mas se o objeto `HttpRequest` de entrada tem um atributo chamado `urlconf`, esse valor será utilizado no lugar da configuração `ROOT_URLCONF`.
2. O Django carrega o módulo Python em questão e procura a variável `urlpatterns`. Esta deve ser uma lista Python, no formato retornado pela função `django.conf.urls.defaults.patterns()`.
3. O Django percorre cada padrão de URL, ordenadamente, e pára no primeiro padrão em que exista uma correspondência com a URL requisitada.

4. Uma vez que uma das expressões regulares corresponda, o Django importa e chama a view associada, que é uma simples função Python. Um `HttpRequest` é sempre passado como primeiro argumento da view e qualquer outro valor capturado pela expressão regular é passado como argumento adicional.

Exemplo

Aqui está um URLconf de exemplo:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^articles/2003/$', 'news.views.special_case_2003'),
    (r'^articles/(\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(\d{4})/(\d{2})/$', 'news.views.month_archive'),
    (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'news.views.article_detail'),
)
```

Notas:

- `from django.conf.urls.defaults import *` disponibiliza a função `patterns()`.
- Para capturar um valor da URL, simplesmente coloque parênteses em volta dele.
- Não é necessário adicionar uma barra no início, porque toda URL tem isso. Por exemplo, o correto é `^articles`, e não `^/articles`.
- O `'r'` na frente de cada string de expressão regular é opcional, mas recomendado. Ele diz ao Python que o conteúdo de uma string deve levar em conta caracteres como `'\'` literalmente em vez de interpretá-los como um caractere especial. Veja [a explicação do Dive Into Python](#).

Requisições de exemplo:

- Uma requisição para `/articles/2005/03/` corresponde ao terceiro elemento da lista. O Django então chama a função `news.views.month_archive(request, '2005', '03')`.
- `/articles/2005/3/` não corresponde a nenhum padrão, porque o terceiro elemento da lista requer dois dígitos para o mês.
- `/articles/2003/` corresponde ao primeiro elemento da lista, e não ao segundo, porque os padrões são testados seguindo a ordem da lista, portanto o primeiro elemento é o primeiro a ser testado. Esteja à vontade para explorar a ordem das regras para inserir casos especiais como este.
- `/articles/2003` não corresponde a nenhum destes padrões, pois cada padrão exige que a URL termine com uma barra.
- `/articles/2003/03/3/` corresponde ao último padrão. O Django chama a função `news.views.article_detail(request, '2003', '03', '3')`.

Grupos nomeados

O exemplo acima usou grupos de expressões regulares simples e *não-nomeados* (usando parênteses) para capturar pedaços da URL e passá-los como argumentos *posicionais* para a view. Em um uso mais avançado, é possível utilizar grupos *nomeados* de expressões regulares para capturar pedaços de URL e passá-los como argumentos *nomeados* para a view.

Em expressões regulares Python, a sintaxe para grupos nomeados é `(?P<nome>padrão)`, onde `nome` é o nome do grupo e `padrão` é algum padrão a ser correspondido.

Segue o URLconf do exemplo acima, reescrita para utilizar grupos nomeados:

```
urlpatterns = patterns('',
    (r'^articles/2003/$', 'news.views.special_case_2003'),
    (r'^articles/(?P<year>\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', 'news.views.month_archive'),
)
```

```
(r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d+)/$', 'news.views.  
↪article_detail'),  
)
```

Este exemplo faz exatamente a mesma coisa que o anterior, com uma pequena diferença: Os valores capturados são passados para as funções de view como argumentos nomeados em vez de argumentos posicionais. Por exemplo:

- Uma requisição para `/articles/2005/03/` chama a função `news.views.month_archive(request, year='2005', month='03')`, em vez de `news.views.month_archive(request, '2005', '03')`.
- Uma requisição para `/articles/2003/03/3/` chama a função `news.views.article_detail(request, year='2003', month='03', day='3')`.

Na prática, isso significa que seus URLconfs são um pouco mais explícitos e menos propensos a bugs de ordem de argumentos – e você pode reordenar os argumentos nas definições de função de view. Obviamente, estes benefícios são obtidos ao custo da brevidade; alguns desenvolvedores acham a sintaxe de grupos nomeados feia e muito prolixa.

O algoritmo de correspondência/agrupamento

Eis o algoritmo que o parser URLconf segue, no que diz respeito a grupos nomeados contra grupos não-nomeados em uma expressão regular:

Se existe algum argumento nomeado, ele será utilizado, ignorando argumentos não-nomeados. Caso contrário, ele passará todos os argumentos não-nomeados como argumentos posicionais.

Em ambos os casos, serão passados quaisquer argumentos nomeados adicionais como argumentos nomeados. Veja “Passando opções adicionais para funções de view” abaixo.

Onde o URLconf faz sua busca

O URLconf faz a busca de correspondência na URL requisitada, como uma string Python normal. Isso não inclui parâmetros GET ou POST, ou do nome de domínio.

Por exemplo, numa requisição para `http://www.example.com/myapp/`, o URLconf irá procurar por `myapp/`.

Em uma requisição para `http://www.example.com/myapp/?page=3`, o URLconf irá procurar por `myapp/`.

O URLconf não se importa com o método de requisição. Em outras palavras, todos os métodos de requisição – POST, GET, HEAD, etc. – serão roteados para a mesma função da mesma URL.

A sintaxe da variável `urlpatterns`

`urlpatterns` deve ser uma lista Python no formato retornado pela função `django.conf.urls.defaults.patterns()`. Sempre use `patterns()` para criar a variável `urlpatterns`.

A convenção é usar `from django.conf.urls.defaults import *` no topo de seu URLconf. Isso dá ao módulo acesso a estes objetos:

`patterns`

`patterns` (*prefix*, *pattern_description*, ...)

Uma função que recebe um prefixo e um número arbitrário de padrões de URL, e retorna uma lista de padrões de URL no formato de que o Django precisa.

O primeiro argumento para `patterns()` é uma string *prefix*. Veja *O prefixo da view* abaixo.

Os argumentos restantes devem ser tuplas neste formato:

```
(expressão regular, função de callback Python, [, dicionário opcional [, nome_  
↪opcional]])
```

...onde `dicionário opcional` e `nome opcional` são opcionais. (Veja [Passando opções adicionais para funções de view](#) abaixo.)

Note: Por `patterns()` ser uma chamada de função, ela aceita um máximo de 255 argumentos (padrões de URL, neste caso). Isso é um limite para todas as chamadas de função Python. Isso raramente é um problema na prática, porque você normalmente estrutura seus padrões de URL modularmente usando seções `include()`. Entretanto, se mesmo assim você chegar ao limite de 255 argumentos, perceba que `patterns()` retorna uma lista Python, então você pode dividir a construção da lista.

```
urlpatterns = patterns('',  
    ...  
)  
urlpatterns += patterns('',  
    ...  
)
```

As listas Python possuem um tamanho ilimitado, então não há um número máximo para quantos padrões de URL você pode construir. O único limite é que você só pode criar 254 de cada vez (o 255o argumento é o prefixo inicial).

url

Please, see the release notes

url (*regex*, *view*, *kwargs=None*, *name=None*, *prefix=''*)

Você pode utilizar a função `url()` no lugar de uma tupla, como um argumento para `patterns()`. Isso é conveniente se você quer especificar um nome sem utilizar o dicionário de argumentos opcionais adicionais. Por exemplo:

```
urlpatterns = patterns('',  
    url(r'index/$', index_view, name="main-view"),  
    ...  
)
```

Esta função recebe cinco argumentos, onde a maioria é opcional:

```
url(regex, view, kwargs=None, name=None, prefix='')
```

Veja [Nomeando padrões de URL](#) para saber da utilidade do parâmetro `name`.

O parâmetro `prefix` tem o mesmo significado que o primeiro argumento de `patterns()` e somente é relevante quando você está passando uma string como um parâmetro para a `view`.

handler404

handler404

Uma string representando o caminho completo para a importação Python da `view` que deve ser chamada se nenhum dos padrões de URL corresponder.

Por padrão, este valor é `'django.views.defaults.page_not_found'`, que na maioria das vezes deve ser suficiente.

handler500

handler500

Uma string representando o caminho completo para a importação Python da view que deve ser chamada em caso de erros no servidor. Erros no servidor acontecem quando você tem erros em tempo de execução no código da sua view.

Por padrão, este valor é `'django.views.defaults.server_error'`, que na maioria das vezes deve ser suficiente.

include

include (<module or pattern_list>)

Uma função que recebe o caminho completo para a importação Python de outro URLconf que deve ser “incluído” neste local. Veja *Incluindo outros URLconfs* abaixo.

Notas na captura de texto em URLs

Cada argumento capturado é enviado para a view como uma string Python, independente do tipo de correspondência feita pela expressão regular. Por exemplo, nesta linha do URLconf:

```
(r'^articles/(?P<year>\d{4})/$', 'news.views.year_archive'),
```

...o argumento `year` para `news.views.year_archive()` será uma string, não um inteiro, embora o `\d{4}` só vá corresponder a strings que contenham números inteiros.

Um truque conveniente é especificar parâmetros padrão para os argumentos de suas views. Aqui vai um exemplo de URLconf e view:

```
# URLconf
urlpatterns = patterns('',
    (r'^blog/$', 'blog.views.page'),
    (r'^blog/page(?P<num>\d+)/$', 'blog.views.page'),
)

# View (em blog/views.py)
def page(request, num="1"):
    # Mostra a página apropriada das postagens de blog, de acordo com num.
```

No exemplo acima, os dois padrões de URL apontam para a mesma view – `blog.views.page` –, mas o primeiro padrão não captura nada da URL. Se o primeiro padrão corresponder, a função `page()` usará seu argumento padrão para `num`, `"1"`. Se o segundo padrão corresponder, `page()` usará o valor de `num` que foi capturado pela expressão regular.

Performance

Cada expressão regular em um `urlpatterns` é compilada na primeira vez em que é acessada. Isso torna o sistema extremamente rápido.

O prefixo da view

Você pode especificar um prefixo comum em sua chamada `patterns()` para diminuir a duplicação de código.

Segue um exemplo de URLconf de *Visão geral do Django*:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^articles/(?d{4})/$', 'mysite.news.views.year_archive'),
    (r'^articles/(?d{4})/(?d{2})/$', 'mysite.news.views.month_archive'),
    (r'^articles/(?d{4})/(?d{2})/(\d+)/$', 'mysite.news.views.article_detail'),
)
```

Neste exemplo, cada view tem um prefixo comum – 'mysite.news.views'. Em vez de digitar isso para cada entrada em urlpatterns, você pode usar o primeiro argumento da função patterns() para especificar o prefixo que se aplica a cada função de view.

Com isso em mente, o exemplo acima pode ser reescrito de maneira mais concisa assim:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('mysite.news.views',
    (r'^articles/(?d{4})/$', 'year_archive'),
    (r'^articles/(?d{4})/(?d{2})/$', 'month_archive'),
    (r'^articles/(?d{4})/(?d{2})/(\d+)/$', 'article_detail'),
)
```

Note que você não coloca um ponto final (".") no prefixo. O Django coloca isso automaticamente.

Múltiplos prefixos de view

Na prática, você provavelmente acabará misturando views ao ponto em que as views em seu urlpatterns não terão um prefixo comum. Entretanto, você ainda pode tirar vantagem do atalho de prefixo da view para remover código duplicado. Simplesmente concatene múltiplos objetos patterns(), assim:

Antigo:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^$', 'django.views.generic.date_based.archive_index'),
    (r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/$', 'django.views.generic.date_based.
↪archive_month'),
    (r'^tag/(?P<tag>\w+)/$', 'weblog.views.tag'),
)
```

Novo:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('django.views.generic.date_based',
    (r'^$', 'archive_index'),
    (r'^(?P<year>\d{4})/(?P<month>[a-z]{3})/$', 'archive_month'),
)

urlpatterns += patterns('weblog.views',
    (r'^tag/(?P<tag>\w+)/$', 'tag'),
)
```

Incluindo outros URLconfs

Em qualquer ponto, seu urlpatterns pode “incluir” outros módulos URLconf. Isso essencialmente “inclui” um conjunto de URLs abaixo de outras.

Por exemplo, eis o URLconf para o próprio [Web site do Django](#). Ele inclui um número de outros URLconfs:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^weblog/', include('django_website.apps.blog.urls.blog')),
    (r'^documentation/', include('django_website.apps.docs.urls.docs')),
    (r'^comments/', include('django.contrib.comments.urls.comments')),
)
```

Note que as expressões regulares neste exemplo não possuem um \$ (caractere que corresponde ao final de string), mas incluem a barra final. Sempre que o Django encontra `include()`, ele recorta qualquer parte da URL correspondida até aquele ponto e envia a string restante ao URLconf incluído para processamento adicional.

Parâmetros capturados

Um URLconf incluído recebe quaisquer parâmetros capturados do URLconf pai, então o seguinte exemplo é válido:

```
# Em settings/urls/main.py
urlpatterns = patterns('',
    (r'^(?P<username>\w+)/blog/', include('foo.urls.blog')),
)

# Em foo/urls/blog.py
urlpatterns = patterns('foo.views',
    (r'^$', 'blog.index'),
    (r'^archive/$', 'blog.archive'),
)
```

No exemplo acima, a variável "username" capturada é passada para o URLconf, como esperado.

Passando opções adicionais para funções de view

URLconfs possuem um “hook” que permite passar argumentos adicionais para suas funções de view como um dicionário Python.

Qualquer tupla URLconf pode ter um terceiro elemento opcional, que deve ser um dicionário de argumentos nomeados adicionais a serem passados para a função de view.

Por exemplo:

```
urlpatterns = patterns('blog.views',
    (r'^blog/(?P<year>\d{4})/$', 'year_archive', {'foo': 'bar'}),
)
```

Neste exemplo, numa requisição para `/blog/2005/`, o Django chamará a view `blog.views.year_archive()`, passando estes argumentos nomeados:

```
year='2005', foo='bar'
```

Essa técnica é utilizada em *views genéricas* e em *syndication framework* para passar meta-dados e opções para as views.

Lidando com conflitos

É possível ter um padrão de URL que captura argumentos nomeados e também passa argumentos com os mesmos nomes no seu dicionário de argumentos adicionais. Quando isso acontece, os argumentos no dicionário serão usados em vez dos argumentos capturados na URL.

Passando opções adicionais para `include()`

Do mesmo modo, você pode passar opções adicionais para `include()`. Quando você passa opções adicionais para `include()`, elas serão passadas para *cada* linha no URLconf incluído.

Por exemplo, estes dois conjuntos URLconf são funcionalmente idênticos:

Conjunto 1:

```
# main.py
urlpatterns = patterns('',
    (r'^blog/', include('inner'), {'blogid': 3}),
)

# inner.py
urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
)
```

Conjunto 2:

```
# main.py
urlpatterns = patterns('',
    (r'^blog/', include('inner')),
)

# inner.py
urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive', {'blogid': 3}),
    (r'^about/$', 'mysite.views.about', {'blogid': 3}),
)
```

Note que as opções adicionais serão *sempre* passadas para *cada* linha no URLconf incluído, independentemente se a view da linha aceita as opções como válidas. Por este motivo, esta técnica somente é útil se você está seguro de que cada view no URLconf incluído aceita as opções adicionais que você está passando.

Passando objetos que podem ser chamados (callable objects) em vez de strings

Alguns desenvolvedores acham mais natural passar o próprio objeto de função Python do que a string contendo o caminho para seu módulo. Esta alternativa é suportada – você pode passar qualquer objeto que pode ser chamado como a view.

Por exemplo, dado este URLconf na notação de “string”:

```
urlpatterns = patterns('',
    (r'^archive/$', 'mysite.views.archive'),
    (r'^about/$', 'mysite.views.about'),
    (r'^contact/$', 'mysite.views.contact'),
)
```

Você pode obter a mesma coisa passando objetos em vez de strings. Apenas assegure-se de importar os objetos:

```
from mysite.views import archive, about, contact

urlpatterns = patterns('',
    (r'^archive/$', archive),
    (r'^about/$', about),
    (r'^contact/$', contact),
)
```

O exemplo seguinte é funcionalmente idêntico. Somente um pouco mais compacto porque importa o módulo que contém as views, em vez de importar cada view individualmente:

```
from mysite import views

urlpatterns = patterns('',
    (r'^archive/$', views.archive),
    (r'^about/$', views.about),
    (r'^contact/$', views.contact),
)
```

O estilo que você usa fica por sua conta.

Note que se você usa esta técnica – passando objetos em vez de strings –, o prefixo da view (como explicado em “O prefixo da view” acima) não terá efeito.

Nomeando padrões de URL

Please, see the release notes É bastante comum usar a mesma função de view em múltiplos padrões de URL em seu URLconf. Por exemplo, estes dois padrões de URL apontam para a view `archive`:

```
urlpatterns = patterns('',
    (r'^archive/(\d{4})/$', archive),
    (r'^archive-summary/(\d{4})/$', archive, {'summary': True}),
)
```

Isso é complementemente válido, mas ocasiona problemas quando você tenta fazer uma correspondência de URL reversa (através do decorator `permalink()` ou a tag `url`). Continuando este exemplo, se você quisesse recuperar a URL para a view `archive`, o mecanismo de correspondência de URL reverso do Django ficaria confuso, pois *dois* padrões de URL apontam para esta view.

Para resolver este problema, o Django suporta **padrões de URL nomeados**. Ou seja, você pode dar nomes para um padrão de URL afim de distingui-lo de outros padrões usando a mesma view e parâmetros. Então, você pode usar este nome na correspondência reversa de URL.

Veja o exemplo acima, reescrito para utilizar padrões de URL nomeados:

```
urlpatterns = patterns('',
    url(r'^archive/(\d{4})/$', archive, name="full-archive"),
    url(r'^archive-summary/(\d{4})/$', archive, {'summary': True}, "arch-summary"),
)
```

Com estes nomes no lugar (`full-archive` and `arch-summary`), você pode apontar cada padrão individualmente utilizando seu nome:

```
.. code-block:: html+django
```

```
{% url arch-summary 1945 %} {% url full-archive 2007 %}
```

Apesar de ambos padrões de URL referirem-se à view `archive`, ao usar o parâmetro `name` de `url()` é possível distingui-los em templates.

A string usada para o nome da URL pode conter quaisquer caracteres que você queira. Você não está restrito a nomes válidos em Python.

Note: Quando você nomear seus padrões de URL, certifique-se de que está utilizando nomes que não têm chances óbvias de colidir com nomes escolhidos em outras aplicações. Se você chama seu padrão de URL `comment`, e outra aplicação faz a mesma coisa, não há garantias de qual URL será inserida no seu template quando você usa este nome.

Colocar um prefixo nos nomes de suas URLs, talvez derivado do nome da aplicação, diminuirá as chances de colisão. Recomendamos algo como `myapp-comment` em vez de `comment`.

Métodos utilitários

`reverse()`

Se você precisa usar algo similar a tag de template `url` no seu código, o Django provê um método (no módulo `django.core.urlresolvers`):

`reverse` (*viewname*, *urlconf=None*, *args=None*, *kwargs=None*)

viewname é o nome da função (podendo ser uma referência para a função, ou o nome dela como uma string, se foi utilizado desta forma em `urlpatterns`) ou o *Nome do padrão de URL*. Normalmente, você não precisa se preocupar com o parâmetro `urlconf` e irá passar somente os argumentos posicionais e nomeados para serem usados na correspondência de URL. Por exemplo:

```
from django.core.urlresolvers import reverse

def myview(request):
    return HttpResponseRedirect(reverse('arch-summary', args=[1945]))
```

A função `reverse()` pode reverter uma grande variedade de padrões de expressões regulares nas URLs, mas não todas. A principal restrição é quando o padrão não contém escolhas alternativas, como quando se usa o caracter barra vertical (`"|"`). Você pode utilizar esses padrões tranquilamente para corresponderem às requisições recebidas e envia-las para os views, mas você não pode reverter tais padrões.

Esteja certo de que suas views estão todas corretas

Como parte do trabalho no qual os nomes de URL mapeiam os padrões, a função `reverse()` tem de importar todos os seus arquivos `URLconf` e examinar o nome de cada view. Isto envolve importar todas as suas funções view. Se houver *qualquer* erro enquanto importa qualquer uma de suas funções views, irá fazer o `reverse()` gerar um erro, mesmo que para tal função não esteja destinado o `reverse`.

Esteja certo que qualquer view referenciado nos seus arquivos `URLconf` existam e que possam ser importados corretamente. Não inclua linhas que apontem para views que você ainda não criou, por que estas views não serão importáveis.

`resolve()`

A função `django.core.urlresolvers.resolve()` pode ser usada para resolver caminhos de URL que correspondem a um funções view. Ela possui a seguinte assinatura:

`resolve` (*path*, *urlconf=None*)

path é o caminho da URL que você deseja resolver. Como com `reverse()` acima, você não precisa se preocupar com o parâmetro `urlconf`. A função retorna uma tuple tripla (função view, argumentos, argumentos nomeados).

Por exemplo, ela pode ser usada para testar se um view geraria um erro `Http404` antes de redirecioná-lo:

```
from urlparse import urlparse
from django.core.urlresolvers import resolve
from django.http import HttpResponseRedirect, Http404

def myview(request):
    next = request.META.get('HTTP_REFERER', None) or '/'
    response = HttpResponseRedirect(next)
```

```
# modifica a requisição e a resposta quando necessário, e.g. mudar
# localidade e setar o cookie correspondente a localidade

view, args, kwargs = resolve(urlparse(next) [2])
kwargs['request'] = request
try:
    view(*args, **kwargs)
except Http404:
    return HttpResponseRedirect('/')
return response
```

permalink()

O decorator `django.db.models permalink`()` é útil para escrever métodos curtos que retornam o caminho inteiro da URL. Por exemplo, o método `get_absolute_url()` do modelo. Consulte [django.db.models.permalink\(\)](#) para mais informações.

Escrevendo Views

Uma função view, ou somente *view*, é simplesmente uma função Python que recebe uma requisição Web e retorna uma resposta Web. Esta resposta pode ser um conteúdo HTML de uma página, ou um redirecionamento, ou um erro 404, ou um documento XML, ou uma imagem . . . ou qualquer coisa, na verdade. O view em si contém qualquer lógica arbitrária que é necessária para retornar uma resposta. Este código pode ficar em qualquer lugar que você quiser, ao longo do seu Python path. Não há outro requerimento – sem “mágica”, por assim dizer. Por uma questão de colocar o código “em qualquer lugar”, vamos criar um arquivo chamado `views.py` no diretório `mysite`, que você criou no capítulo anterior.

Um view simples

Aqui há um view que retorna a data atual, como um documento HTML:

```
from django.http import HttpResponseRedirect
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponseRedirect(html)
```

Vamos passo-a-passo através deste código, uma linha por vez:

- Primeiro, nós importamos a classe `HttpResponse`, que reside o módulo `django.http`, também a biblioteca `datetime` do Python.
- Depois, nós definimos uma função chamada `current_datetime`. Esta é a função view. Cada função view recebe um objeto `HttpRequest` como seu primeiro parâmetro, que é tipicamente nomeado como `request`.

Note que o nome da função view não importa; ele não tem de ser nomeada de uma certa forma para ser reconhecida pelo Django. Nós estamos chamando-a de `current_datetime` aqui, porque este nome indica claramente o que ela faz.

- O view retorna um objeto `HttpResponse` que contém a resposta gerada. Cada função view é responsável por retornar um objeto `HttpResponse`. (Há exceções, mas serão abordadas mais tarde.)

Django's Time Zone

O Django inclui uma configuração de `TIME_ZONE` que por padrão é `America/Chicago`. Este provavelmente não é onde você mora, então você pode mudá-lo no seu arquivo `settings.py`.

Mapeando URLs para Views

Então, para recapitular, esta função view retorna uma página HTML, que inclui a data e hora atual. Para mostrar este view numa URL em particular, você precisará criar uma no *URLconf*; veja *URL dispatcher* para instruções.

Retornando erros

Retornar códigos de erro HTTP no Django é muito fácil. Existem subclasses do *HttpResponse* para um número de códigos de status comuns do HTTP além do 200 (que significa “OK”). Você pode encontrar a lista completa de subclasses disponíveis na documentação de *requisição/resposta*. Somente retorne um instância de uma destas subclasses ao invés de uma classe normal *HttpResponse* a fim de mostrar um erro. Por exemplo:

```
def my_view(request):
    # ...
    if foo:
        return HttpResponseNotFound('<h1>Page not found</h1>')
    else:
        return HttpResponse('<h1>Page was found</h1>')
```

Não existe uma subclasse especializada para cada código resposta HTTP possível, uma vez que muitos deles não serão comuns. Porém, como documentado na documentação do *HttpResponse*, você também pode passar o código de status HTTP dentro de um construtor do *HttpResponse* para criar uma classe de retorno para qualquer código de status que você quiser. Por exemplo:

```
def my_view(request):
    # ...

    # Retorna um código de resposta "created" (201).
    return HttpResponse(status=201)
```

Como erros 404 são de longe os erros HTTP mais comuns, existe uma forma mais fácil de manipular estes erros.

A exceção Http404

Quando você retorna um erro como *HttpResponseNotFound*, você é responsável por definir o HTML da página de erro resultante:

```
return HttpResponseNotFound('<h1>Page not found</h1>')
```

Por convenção, e porque é uma boa idéia ter uma página de erro 404 consistente para todo site, o Django provê uma exceção *Http404*. Se você lançar um *Http404* em qualquer ponto de sua função view, o Django a pegará e retornará a página de erro padrão para sua aplicação, junto com o código de erro HTTP 404. Example usage:

```
from django.http import Http404

def detail(request, poll_id):
    try:
        p = Poll.objects.get(pk=poll_id)
    except Poll.DoesNotExist:
        raise Http404
    return render_to_response('polls/detail.html', {'poll': p})
```

A fim de usar a exceção *Http404* para sua satisfação, você deve criar um template que será mostrado quando um erro 404 é gerado. Este template deve ser chamado `404.html` e localizado no nível mais alto de sua árvore de templates.

Customizando erros de views

O view 404 (page not found)

Quando você gera uma exceção `Http404`, o Django carrega um view especial devotado a manipular erros 404. Por padrão, ele é o view `django.views.defaults.page_not_found`, que carrega e renderiza o template `404.html`. Isto significa que você precisa definir um template `404.html` na raiz do seu diretório de templates. Este template será usado para todos os erros 404.

Esta view `page_not_found` deve bastar para 99% das aplicações Web, mas se você precisa sobrescrevê-lo, você pode especificar `handler404` no seu URLconf, desta forma:

```
handler404 = 'mysite.views.my_custom_404_view'
```

Por trás das cenas, o Django obriga o view 404 a procurar por `handler404`. Por padrão, os URLconfs contém a seguinte linha:

```
from django.conf.urls.defaults import *
```

Que se preocupa com a configuração do `handler404` no módulo atual. Como você pode ver em `django/conf/urls/defaults.py`, `handler404` é setado para `'django.views.defaults.page_not_found'` por padrão. Three things to note about 404 views:

- O view 404 é também chamado se o Django não encontrar uma combinação depois de checar todas as expressões regulares no URLconf.
- Se você não definir seu próprio view 404 – e simplesmente usar o padrão, o que é recomendado – você ainda tem uma obrigação: você deve criar um template `404.html` na raiz do seu diretório de templates. Por padrão, o view 404 usará este template para todos os erros 404. Por padrão, o view 404 passará uma variável para o template: `request_path`, que é a URL que resultou no erro 404.
- Ao view 404 é passado um `RequestContext` e terá acesso a variáveis fornecidas por sua configuração `TEMPLATE_CONTEXT_PROCESSORS` (e.g., `MEDIA_URL`).
- Se `DEBUG` é setado como `True` (no seu módulo settings), então seu view 404 nunca será usado, e o trace-back será mostrado no lugar dele.

O view 500 (server error)

Similarmente, o Django executa um comportamento especial no caso de um erro de execução no código do view. Se um view resulta num exceção, o Django irá, por padrão, chamar o view `django.view.defaults.server_error`, que carrega e renderiza o template `500.html`.

Isto significa que você precisa definir um template `500.html` na raiz do seu diretório de templates. Este template será usado para todos os erros de servidor. O view 500 padrão não passa variáveis para este template e é renderizado com um `Context` vazio para reduzir as chances de erros adicionais.

Este view `server_error` deve bastar para 99% das aplicações Web, mas se você quiser sobrescrever o view, você pode especificar `handler500` no seu URLconf, desta forma:

```
handler500 = 'mysite.views.my_custom_error_view'
```

Por trás das cenas, o Django obriga o view de erro a procurar pelo `handler500`. Por padrão, os URLconfs contém a seguinte linha:

```
from django.conf.urls.defaults import *
```

Que se preocupa de setar `handler500` no módulo atual. Como você pode ver em `django/conf/urls/defaults.py`, `handler500` é setado para `'django.views.defaults.server_error'` por padrão.

Upload de arquivos

Please, see the release notes Quando o Django manipula um upload de arquivo, os dados do arquivo são acessíveis pelo `request.FILES` (para saber mais sobre o objeto `request`, veja a documentação dos *objetos request e response*). Este documento explica como os arquivos são armazenados no disco e na memória, e como personalizar o comportamento padrão.

Upload de arquivos básico

Considere um simples formulário contendo um `FileField`:

```
from django import forms

class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

Um view que manipula este form receberá os dados do arquivo no `request.FILES`, é um dicionário contendo uma chave para cada `FileField` (ou `ImageField`, outra subclasse de `FileField`) no formulário. De modo que os dados do formulário acima seriam acessíveis como `request.FILES['file']`.

Note que `request.FILES` somente conterá dados se o método `request` for `POST` e o `<form>` postado tiver um atributo `enctype="multipart/form-data"`. Caso contrário, `request.FILES` será vazio.

Na maior parte do tempo, você simplesmente passará os dados do arquivo do `request` como descrito em *Vinculando arquivos enviados a um formulário*. Isso deveria parecer com algo tipo:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response

# Função imaginária para manipular um upload de arquivo.
from somewhere import handle_uploaded_file

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('/success/url/')
    else:
        form = UploadFileForm()
    return render_to_response('upload.html', {'form': form})
```

Observe que nós temos de passar `request.FILES` para o construtor do formulário, é assim que o dos dados do arquivo são incorporados pelo form.

Manipulando arquivos enviados

A peça final do quebra-cabeça é manipular os dados do arquivo do `request.FILES`. Cada entrada neste dicionário é um objeto `UploadedFile` – uma simples classe que envolve o arquivo enviado. Você pode habitualmente usar um destes métodos para acessar o conteúdo submetido:

`UploadedFile.read()` Ler todos os dados enviados do arquivo. Seja cuidadoso com este método: se o arquivo enviado for muito grande ele pode sobrecarregar o sistema se você tentar carregá-lo na memória. Você provavelmente vai querer usar `chunks()` ao invés, veja abaixo.

`UploadedFile.multiple_chunks()` Retorna `True` se o arquivo enviado é grande o suficiente para ser lido em vários pedaços. Por padrão este será qualquer arquivo maior que 2.5 megabytes, mas isto é configurável; veja abaixo.

UploadedFile.chunks() Um gerador retornando pedaços do arquivo. Se `multiple_chunks()` for `True`, você deve usar este método num loop ao invés de `read()`.

Na prática, é frequentemente muito mais fácil usar `chunks()` o tempo todo; veja o exemplo abaixo.

UploadedFile.name O nome do arquivo enviado (e.g. `my_file.txt`).

UploadedFile.size O tamanho, em bytes, do arquivo enviado.

Existem uns outros métodos e atributos disponíveis nos objetos `UploadedFile`; veja [Objeto UploadedFile](#) para uma referência completa.

Colando tudo isso junto, temos uma forma comum de manipular o upload de um arquivo:

```
def handle_uploaded_file(f):
    destination = open('some/file/name.txt', 'wb+')
    for chunk in f.chunks():
        destination.write(chunk)
    destination.close()
```

Iterando sobre `UploadedFile.chunks()` no lugar de usar `read()` assegura que arquivos grandes não sobrecarreguem a memória do seu sistema.

Onde os dados enviados são armazenados

Antes de salvar os arquivos enviados, os dados precisam ser armazenados em algum lugar.

Por padrão, se um upload é menor que 2.5 megabytes, o Django irá tratar o conteúdo inteiramente na memória. Isto significa que salvar o arquivo envolve somente ler da memória e escrever no disco e isto é muito rápido.

No entanto, se um upload é muito grande, o Django armazenará o arquivo enviado num arquivo temporário, dentro do diretório temporário do seu sistema. Numa plataforma baseada no Unix, isto significa que o Django gerará um arquivo com um nome tipo `/tmp/tmpzfp6I6.upload`. Se um arquivo é grande o suficiente, você pode assistir o crescimento deste arquivo a medida que o Django realiza a transferência para o disco.

Estas especificidades – 2.5 megabytes; `/tmp`; etc. – são simplesmente “padrões razoáveis”. Leia sobre, para saber detalhes de como você pode personalizar ou substituir completamente o comportamento do upload.

Mudando o comportamento do manipulador de upload

Três configurações controlam o comportamento do upload de arquivo do Django:

FILE_UPLOAD_MAX_MEMORY_SIZE O tamanho máximo, em bytes, para arquivos que serão armazenados em memória. Arquivos maiores que `FILE_UPLOAD_MAX_MEMORY_SIZE` serão manipulados por disco.

Padrão é 2.5 megabytes.

FILE_UPLOAD_TEMP_DIR O diretório onde os arquivos enviados maiores que o `FILE_UPLOAD_MAX_MEMORY_SIZE` serão armazenados.

Ele segue o padrão do diretório temporário do seu sistema (i.e. `/tmp` nos sistemas baseados no Unix).

FILE_UPLOAD_PERMISSIONS O modo número (i.e. `0644`) para setar o novo arquivo enviado.

Para mais informações sobre o que estes modos significam, veja a [documentação do os.chmod](#)

Se este não for dado ou for `None`, você terá o comportamento inerente do sistema operacional. Na maioria das plataformas, os arquivos temporários são salvos usando o padrão do sistema `umask`.

Warning: Se você não está familiarizado com os modos de arquivos, por favor note que o 0 que fica na frente é muito importante: ele indica um número octal, que é a forma como os modos são especificados. Se você tentar usar 644, você receberá um comportamento totalmente incorreto.

Sempre prefixe o modo com um 0.

FILE_UPLOAD_HANDLERS

O manipulador atual para os arquivos enviados. Mudando esta configuração permite uma completa personalização – ou mesmo substituição – do processo de upload do Django. Veja *manipuladores de upload* abaixo, para detalhes.

O padrão é:

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",  
 "django.core.files.uploadhandler.TemporaryFileUploadHandler", )
```

O que significa “tente fazer o upload em memória primeiro, e então faça com arquivo temporário.”

Objeto UploadedFile

class UploadedFile

Além de estender a classe *File*, todo objeto *UploadedFile* define os seguintes métodos/atributos:

UploadedFile.content_type O cabeçalho content-type enviado com o arquivo (e.g. `text/plain` ou `application/pdf`). Como qualquer dado fornecido pelo usuário, você não deve confiar que o arquivos enviado seja realmente deste tipo. Você ainda precisará validar este arquivo, verificando se ele possui o conteúdo que seu cabeçalho content-type informa – “confiar mas verificar.”

UploadedFile.charset Para content-types `text/*`, o conjunto de caracteres (i.e. `utf8`) fornecido pelo navegador. Novamente, “confiar mas verificar” é a melhor política aqui.

UploadedFile.temporary_file_path() Somente arquivos que tiveram o upload realizado por disco terão este método; ele retorna o caminho completo para o arquivo temporário.

Note: Como arquivos regulares do Python, você pode ler o arquivos linha-por-linha simplesmente iterando sobre o arquivos enviado:

```
for line in uploadedfile:  
    do_something_with(line)
```

Entretanto, *ao contrário* dos arquivos padrões do Python, o *UploadedFile* somente entende `\n` (também conhecido como “Unix-style”) como final de linha. Se você sabe que precisa manipular arquivos diferentes tipos de finalização de linha, então você precisará fazê-lo no seu view.

Manipuladores de upload

When a user uploads a file, Django passes off the file data to an *upload handler* – a small class that handles file data as it gets uploaded. Upload handlers are initially defined in the `FILE_UPLOAD_HANDLERS` setting, which defaults to:

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",  
 "django.core.files.uploadhandler.TemporaryFileUploadHandler", )
```

Together the `MemoryFileUploadHandler` and `TemporaryFileUploadHandler` provide Django's default file upload behavior of reading small files into memory and large ones onto disk.

You can write custom handlers that customize how Django handles files. You could, for example, use custom handlers to enforce user-level quotas, compress data on the fly, render progress bars, and even send data to another storage location directly without storing it locally.

Modifying upload handlers on the fly

Algumas vezes views particulares requerem um comportamento de upload diferente. Nestes casos, você pode sobrescrever os manipuladores de upload, basicamente modificando o `request.upload_handlers`. Por padrão, esta lista conterá os manipuladores de upload dados pelo `FILE_UPLOAD_HANDLERS`, mas você pode modificar a lista como se você qualquer outra lista.

Para instâncias, suponhamos que você tenha estrito um `ProgressBarUploadHandler` que provê uma resposta sobre o progresso do upload para algum tipo de widget AJAX. Você poderia adicionar este handler ao seu manipulador de upload desta forma:

```
request.upload_handlers.insert(0, ProgressBarUploadHandler())
```

Você poderia provavelmente querer usar o `list.insert()` neste caso (ao invés de `append()`) porque um manipulador de barra de progresso precisaria rodar *antes* de qualquer outro manipulador. Lembre-se, os manipuladores de upload são processados em ordem.

Se você deseja substituir o manipulador de upload completamente, você pode somente atribuir uma nova lista:

```
request.upload_handlers = [ProgressBarUploadHandler()]
```

Note: Você pode somente modificar manipuladores de upload *antes* deles acessarem o `request.POST` ou `request.FILES` – pois não faz sentido mudar o manipulador depois que ele já tiver iniciado. Se você tentar modificar `request.upload_handlers` depois que estiver lendo o `request.POST` ou `request.FILES` o Django gerará um erro.

Sendo assim, você deve sempre modificar os manipuladores de upload o mais cedo possível dentro do seu view.

Escrevendo um manipulador de upload personalizado

Todo manipulador de upload de arquivo deve ser uma subclasse de `django.core.files.uploadhandler.FileUploadHandler`. Você pode definir o manipulador de upload aonde você desejar.

Métodos obrigatórios

Manipuladores de arquivos personalizados **devem** definir os seguintes métodos:

```
``FileUploadHandler.receive_data_chunk(self, raw_data, start)``
    Recebe um "pedaço" de dado do arquivos enviado.

    ``raw_data`` é uma byte string contendo o dado enviado.

    ``start`` é a posição no arquivo onde este pedaço ``raw_data`` começa.

    Os dados que você retornar serão alimentados num método subsequente do
    manipulador de upload ``receive_data_chunk``. Desta forma, um
    manipulador pode ser um "filter" para outros manipuladores.

    Retorna ``None`` do ``receive_data_chunk`` lembrando o manipulador de
```

upload para começar este pedaço. Isso é usual se você estiver armazenando o arquivo enviado você mesmo, e não quer que os próximos manipuladores façam uma cópia dos dados.

Se você lançar uma exceção `StopUpload` ou `SkipFile`, o upload será abortado ou o arquivo pulado.

```
FileUploadHandler.file_complete(self, file_size)
```

Chamado quando o arquivos finalizou o upload.

O manipulador deve retornar um objeto `UploadedFile` que será armazenado em `request.FILES`. Manipuladores podem também retornar `None` para indicar que o objeto `UploadedFile` deveria vir de uma manipulador subsequente.

Métodos opcionais

Um manipulador de upload personalizado também define qualquer um dos seguintes métodos ou atributos opcionais:

`FileUploadHandler.chunk_size` Tamanho, em bytes, dos “pedaços” que o Django deve armazenar em memória dentro do manipulador. Isto é, este atributo controla o tamanho dos pedaços dentro do `FileUploadHandler.receive_data_chunk`.

Para uma performance máxima o tamanho dos arquivos devem ser divisíveis por 4 e não devem exceder 2GB (2^{31} bytes). Quando existem vários tamanhos de pedaços fornecidos por vários manipuladores, o Django usará o menor tamanho definido para qualquer manipulador.

O tamanho padrão é 64×2^{10} bytes, ou 64KB.

`FileUploadHandler.new_file(self, field_name, file_name, content_type, content_length, charset)`

Um callback sinalizando que um novo upload de arquivo começou. Este é chamado antes de qualquer dado ser alimentado qualquer manipulador de upload.

`field_name` é uma string com o nome do campo de arquivo `<input>`.

`file_name` é o nome do arquivo unicode que foi fornecido pelo navegador.

`content_type` é o tipo MIME fornecido pelo navegador – E.g 'image/jpeg'.

`content_length` é o comprimento da imagem dado pelo navegador. As vezes ele não será fornecido e será `None`, None caso contrário.

`charset` é o conjunto de caracteres (i.e. utf8) dado pelo navegador. Como o `content_length`, algumas vezes ele não será fornecido.

Este método pode lançar uma exceção `StopFutureHandlers` para prevenir que os próximos handlers de manipular este arquivo.

`FileUploadHandler.upload_complete(self)` Uma callback sinalizando que o upload inteiro (todos os arquivos) foi completado.

`FileUploadHandler.handle_raw_input(self, input_data, META, content_length, boundary, encoding)`

Permite o manipulador sobrescrever completamente o parseamento da entrada pura do HTTP.

`input_data` é um objeto, tipo arquivo, que suporta leitura pelo `read()`.

`META` é o mesmo objeto como `request.META`.

`content_length` é o comprimento de dados no `input_data`. Não leia mais bytes do `content_length` do que tem no `input_data`.

`boundary` é o limite do MIME para esta requisição.

`encoding` é a codificação da requisição.

Retorna `None` se você quer que o manipulador de upload continue, ou uma tupla com `(POST, FILES)` se você quiser retornar a nova estrutura de dados adequada para a requisição diretamente.

Funções de atalho do Django

O pacote `django.shortcuts` agrega funções auxiliares e classes que “surtem” em múltiplos níveis do MVC. Em outras palavras, essas funções/classes introduzem uma união controlada em prol da conveniência.

`render_to_response()`

`django.shortcuts.render_to_response` renderiza um dado template com um determinado dicionário de contexto e retorna um objeto `HttpResponse` com o texto renderizado.

Argumentos obrigatórios

template O nome completo do template utilizado ou uma sequência de nomes de template.

Argumentos opcionais

dictionary Um dicionário com os valores a serem adicionados ao contexto do template. Por via de regra, é informado um dicionário vazio. Se o valor no dicionário é chamável, a view irá chamá-lo antes de renderizar o template.

context_instance Instância de contexto a ser renderizada com o template. Por padrão, o o template vai renderizar com uma instância do `Context` (preenchida com valores de dicionário). Se você precisa usar o *context processors*, renderize o template com uma instância de `RequestContext`. Seu código pode ficar parecido com o seguinte:

```
return render_to_response('my_template.html',
                          my_data_dictionary,
                          context_instance=RequestContext(request))
```

mimetype

Please, see the release notes O tipo MIME a ser usado para o documento resultante. Caso nenhum valor seja fornecido, será utilizado o valor de `DEFAULT_CONTENT_TYPE` do arquivo de settings.py.

Exemplo

O exemplo a seguir renderiza o template `myapp/index.html` com o tipo MIME `application/xhtml+xml`:

```
from django.shortcuts import render_to_response

def my_view(request):
    # Código da View aqui...
    return render_to_response('myapp/index.html', {"foo": "bar"},
                              mimetype="application/xhtml+xml")
```

Esse exemplo é equivalente a:

```
from django.http import HttpResponse
from django.template import Context, loader

def my_view(request):
    # Código da View aqui...
```

```
t = loader.get_template('myapp/template.html')
c = Context({'foo': 'bar'})
return HttpResponse(t.render(c),
                    mimetype="application/xhtml+xml")
```

get_object_or_404

`django.shortcuts.get_object_or_404` chama `get()` de um dado gerenciador de modelos, mas lança um `django.http.Http404` no lugar da exceção `DoesNotExist`.

Argumentos obrigatórios

klass Uma instância de `Model`, `Manager` ou `QuerySet` da qual se pega o objeto.

****kwargs** Parâmetros de busca que devem estar em um formato aceito por `get()` e `filter()`.

Exemplo

O seguinte exemplo pega o objeto com a chave primária 1 de `MyModel`:

```
from django.shortcuts import get_object_or_404

def my_view(request):
    my_object = get_object_or_404(MyModel, pk=1)
```

Esse exemplo é equivalente a:

```
from django.http import Http404

def my_view(request):
    try:
        my_object = MyModel.objects.get(pk=1)
    except MyModel.DoesNotExist:
        raise Http404
```

Nota: Como com `get()`, uma exceção `MultipleObjectsReturned` será lançada caso mais de um objeto seja encontrado.

get_list_or_404

`django.shortcuts.get_list_or_404` devolve o resultado de `filter()` em um dado gerenciador de modelo, lançando um `django.http.Http404` se o resultado da lista for vazio.

Argumentos obrigatórios

klass Uma instância de `Model`, `Manager` ou `QuerySet` da qual se pega o objeto.

****kwargs** Parâmetros de busca que devem estar em um formato aceito por `get()` e `filter()`.

Exemplo

O seguinte exemplo pega todos os objetos publicados de `MyModel`:

```
from django.shortcuts import get_list_or_404

def my_view(request):
    my_objects = get_list_or_404(MyModel, published=True)
```

Esse exemplo é equivalente a:

```
from django.http import Http404

def my_view(request):
    my_objects = list(MyModel.objects.filter(published=True))
    if not my_objects:
        raise Http404
```

Visões genéricas

Veja *Generic views (Visões genéricas)*.

Middleware

O Middleware é um framework de hook dentro do processamento de requisição/resposta do Django. Ele é um sistema de “plugins” leve, e de baixo nível para alterar globalmente a entrada ou saída do Django.

Cada componente middleware é responsável por fazer alguma função específica. Por exemplo, o Django inclui um componente middleware, `XViewMiddleware`, que adiciona um cabeçalho HTTP "X-View" a toda resposta para uma requisição HEAD.

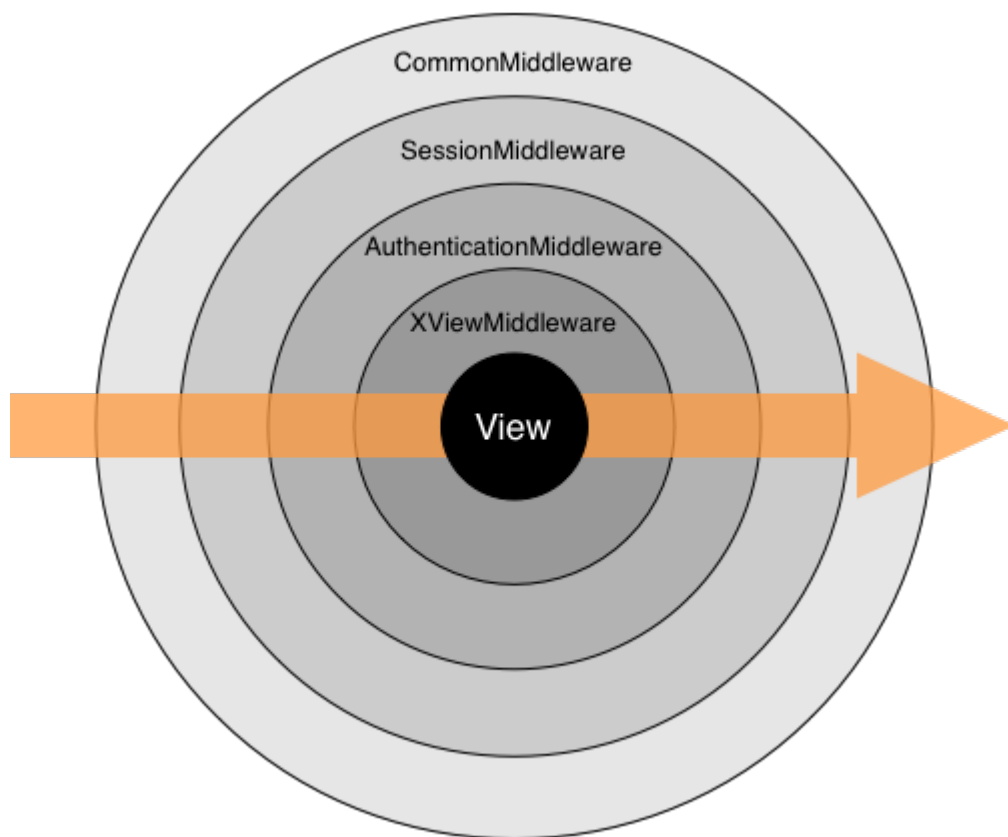
Este documento explica como o middleware funciona, como você ativa o middleware, e como escrever seu próprio middleware. O Django vem com alguns middleware embutidos você pode user; eles são documentados na *referência de middlewares embutidos*.

Ativando um middleware

Para ativar um componente middleware, adicione-o a lista `MIDDLEWARE_CLASSES` no suas configurações do Django. No `MIDDLEWARE_CLASSES`, cada componente middleware é representado por uma string: o caminho Python completo com o nome da classe middleware. Por exemplo, o `MIDDLEWARE_CLASSES` padrão criado pelo `django-admin.py startproject`:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.middleware.doc.XViewMiddleware',
)
```

Durante as fases da requisição (nos middlewares `process_request()` e `process_view()`), o Django aplica o middleware na ordem que está definida no `MIDDLEWARE_CLASSES`, de cima pra baixo. Durante as fases de repostas (nos middlewares `process_response()` e `process_exception()`), as classes são aplicada na ordem inversa, de baixo pra cima. Você pode imaginar como se fosse uma cebola: cada classe middleware é uma “camada” que envolve a view:



A instalação do Django não requer qualquer middleware – e.g., `MIDDLEWARE_CLASSES` pode ser vazio, se você quiser – mas é fortemente recomendado que você use pelo menos o `CommonMiddleware`.

Escrevendo seu próprio middleware

Escrever seu próprio middleware é fácil. Cada componente middleware é uma única classe Python que define um ou mais dos seguintes métodos:

`process_request`

`process_request` (*self*, *request*)

O *request* é um objeto `HttpRequest`. Este método é chamado em cada request, antes do Django decidir qual view executar.

O `process_request()` deve retornar um `None` ou um objeto `HttpResponse`. Se ele retornar `None`, o Django continuará processando a requisição, executando qualquer outro middleware e, então, a view apropriada. Se ele retorna um objeto `HttpResponse`, o Django não se incomodará em chamar QUALQUER outra requisição, view ou exceção de middleware, ou a view apropriada; ele retornará o `HttpResponse`. O middleware de resposta é sempre chamado para todas as respostas.

`process_view`

`process_view` (*self*, *request*, *view_func*, *view_args*, *view_kwargs*)

O *request* é um objeto `HttpRequest`. O *view_func* é uma função Python que o Django usa. (É o objeto da função atual, não uma string com o nome da função.) O *view_args* é uma lista de argumentos posicionais que serão passados para o view, e *view_kwargs* é um dicionário com argumentos nomeados que serão passados para o view. Nenhum, *view_args* nem *view_kwargs* incluem o primeiro argumento do view (*request*).

O `process_view()` é chamado logo após o Django chamar o view. Ele deve retornar um `None` ou um objeto `HttpResponse`. Se ele retorna `None`, o Django continuará processando esta requisição, executando qualquer outro middleware `process_view()` e, então, a view apropriada. Se ele retornar um objeto `HttpResponse`, o Django não se incomodará em chamar QUALQUER outra requisição, view ou exceção de middleware, ou a view apropriada; Ele retornará o `HttpResponse`. O middleware é sempre chamado para todas as repostas.

`process_response`

`process_response` (*self, request, response*)

`request` is an `HttpRequest` object. `response` is the `HttpResponse` object returned by a Django view.

`process_response()` must return an `HttpResponse` object. It could alter the given response, or it could create and return a brand-new `HttpResponse`.

Diferentemente dos métodos `process_request()` é `process_view()`, o método `process_response()` é sempre chamado, mesmo que os métodos `process_request()` e `process_view()` da mesma classe middleware foram pulados por causa de um método de um middleware anterior que retornou um `HttpResponse` (isto significa que seu método `process_response()` não pode invocar a configuração feita em `process_request`, por exemplo). Além do mais, durante a fase de resposta as classes são aplicadas na ordem reversa, de baixo pra cima. Isto quer dizer que classes definidas no final da lista `MIDDLEWARE_CLASSES` serão executadas primeiro.

`process_exception`

`process_exception` (*self, request, exception*)

O `request` é um objeto `HttpRequest`. O `exception` é um objeto `Exception` lançado por uma função view.

O Django chama `process_exception()` quando uma view lança um exceção. O `process_exception()` deve retornar um `None` ou um objeto `HttpResponse`. Se ele retorna um objeto `HttpResponse`, a resposta será retornada para o navegador. Por outro lado, o um manipulador de exceção padrão entra em ação.

Novamente, os middlewares são rodados na ordem inversa durante o processo de resposta, que inclui `process_exception`. Se uma exceção de middleware retorna uma resposta, nenhuma das classes middlewares acima deste middleware serão chamadas.

`__init__`

A maioria das classes middleware não precisam de um inicializador desde que classes middleware são essencialmente marcadores para os métodos `process_*`. Se você precisar de algum estado global, você pode usar o `__ini__` para configurá-lo. Entretanto, mantenha em mente algumas ressalvas:

- O Django inicializa seus métodos sem qualquer argumento, então você não pode definir o `__init__` requerendo quaisquer argumentos.
- Diferentemente dos métodos `process_*` que são chamados uma vez por requisição, `__ini__` é chamado uma somente, quando o servidor Web é iniciado.

Marcando um middleware como inutilizado

Algumas vezes é útil determinar, em tempo de execução, se um middleware deve ser usado. Nestes casos, o método `__ini__` de seu middleware pode lançar uma exceção `django.core.exceptions.MiddlewareNotUsed`. O Django irá então remover este middleware de seu processo de middlewares.

Guidelines

- Classes middleware não tem de estender ninguém.
- A classe middleware pode estar em qualquer lugar de seu Python path. Todo Django se preocupa é com o fato do `MIDDLEWARE_CLASSES` ter o caminho para ele.
- Sintá-se livre para olhar em *middlewares disponíveis do Django* por exemplos.
- Se você escreve um componente middleware que você pensa ser útil para outras pessoas, contribua com a comunidade! *Deixe-nos saber*, e nós consideraremos adicioná-lo ao Django.

Como utilizar sessões

O Django provê suporte integral para sessões anônimas. O framework de sessão deixa você armazenar e recuperar dados arbitrários para cada visitante do site. Ele armazena dados no lado do servidor e abstrai o envio e recebimento de cookies. Os cookies contêm um ID da sessão – não os dados em si.

Habilitando as sessões

As sessões são implementadas por meio de um *middleware*.

Para habilitar a funcionalidade de sessões, faça o seguinte:

- Edite o parâmetro de configuração `MIDDLEWARE_CLASSES` e assegure-se de que ele contenha `'django.contrib.sessions.middleware.SessionMiddleware'`. O arquivo padrão `settings.py` criado pelo `django-admin.py startproject` tem o `SessionMiddleware` ativado.
- Adicione `'django.contrib.sessions'` à sua configuração `INSTALLED_APPS` e rode `manage.py syncdb` para instalar a única tabela de banco de dados que armazena os dados de sessão.

Novo na versão de desenvolvimento do Django: esse passo é opcional se você não estiver usando um backend de sessão de banco de dados; veja *configurando o mecanismo de sessão*.

Este passo é opcional se você não está usando o banco de dados sessão backend; veja *configurando o mecanismo de sessão*. Se você não deseja utilizar sessões, você pode remover a linha `SessionMiddleware` do `MIDDLEWARE_CLASSES` e `'django.contrib.sessions'` de seu `INSTALLED_APPS`. Isso poupará um pouco o processamento adicional.

Configurando o mecanismo de sessão

Please, see the release notes Por padrão, o Django armazena as sessões no seu banco de dados (usando o modelo `django.contrib.sessions.models.Session`). Apesar de ser conveniente, em algumas configurações é mais rápido armazenar esses dados em algum outro lugar, então o Django pode ser configurado para armazenar dados de sessão no seu sistema de arquivos ou no cache.

Usando sessões baseadas em arquivos

Para usar sessões em arquivos, configure o parâmetro `SESSION_ENGINE` para `"django.contrib.sessions.backends.file"`.

Você pode querer configurar o parâmetro `SESSION_FILE_PATH` (que obtém o valor padrão de saída de `tempfile.gettempdir()`, comumente `/tmp`) para controlar onde o Django deve armazenar os arquivos de sessão. Certifique-se de que seu servidor Web tem permissão para ler e escrever neste local.

Usando sessões baseadas em cache

Para armazenar dados de sessão utilizando o sistema de cache do Django, configure `SESSION_ENGINE` para `"django.contrib.sessions.backends.cache"`. É importante confirmar que o seu cache esteja configurado; veja a [documentação de cache](#) para mais detalhes.

Note: Você provavelmente só deve utilizar sessões baseadas em cache se estiver usando o backend de cache Memcached. O backend de cache de memória local não guarda os dados por um período longo o suficiente para ser considerado uma boa escolha, e será mais rápido usar sessões de arquivos ou banco de dados diretamente em vez de enviar tudo por meio do backend de cache baseado em arquivos ou banco de dados.

Utilizando sessões nas views

Quando `SessionMiddleware` está ativo, cada objeto `HttpRequest` – o primeiro argumento para qualquer view em Django – terá um atributo `session`, que é um objeto que se comporta como dicionário. Você pode ler ou escrever nele.

Ele implementa os seguintes métodos padrões de dicionários:

- `__getitem__(key)`

Exemplo: `fav_color = request.session['fav_color']`

- `__setitem__(key, value)`

Exemplo: `request.session['fav_color'] = 'blue'`

- `__delitem__(key)`

Exemplo: `del request.session['fav_color']`. Se a chave `key` ainda não está na sessão, uma exceção `KeyError` será lançada.

- `__contains__(key)`

Exemplo: `'fav_color' in request.session`

- `get(key, default=None)`

Exemplo: `fav_color = request.session.get('fav_color', 'red')`

- `keys()`

- `items()`

- `setdefault()`

- `clear()`

`setdefault()` and `clear()` são novos nesta versão. Existem também estes métodos:

- `flush()` *Please, see the release notes* Deleta os dados da sessão corrente da sessão, e re-gera a chave o valor da chave da sessão que é enviado de volta para o cookie do usuário. Isto é usado se você quer assegurar que os dados de uma sessão prévia não possa ser acessados novamente por usuário do navegador (por exemplo, a função `:func:django.contrib.auth.logout()` o chama).

- `set_test_cookie()`

Cria um cookie de teste para determinar se o navegador do usuário suporta cookies. Devido à maneira como os cookies funcionam, você não é capaz de testá-lo até que o usuário requisiite a próxima página. Veja [Criando cookies de teste](#) abaixo para mais informações.

- `test_cookie_worked()`

Devolve `True` ou `False`, a depender de o navegador do usuário ter aceito o cookie de teste ou não. Devido à maneira como os cookies funcionam, você deverá chamar `set_test_cookie()` em uma requisição anterior de uma página separada. Veja [Criando cookies de teste](#) abaixo para mais informações.

- `delete_test_cookie()`

Apaga o cookie de test. Use isto você mesmo, para limpar tudo.

- `set_expiry(value)` *Please, see the release notes* Configura o tempo de expiração para a sessão. Você pode passar vários valores diferentes:
 - Se `value` é um inteiro, a sessão irá expirar depois dessa quantidade de segundos de inatividade. Por exemplo, chamar `request.session.set_expiry(300)` fará com que a sessão expire em 5 minutos.
 - Se `value` é um objeto `datetime` ou `timedelta`, a sessão irá expirar nessa data/hora específica.
 - Se `value` é 0, o cookie de sessão do usuário expirará quando o navegador Web do usuário for fechado.
 - Se `value` é `None`, a sessão volta a utilizar a política global de expiração.
- `get_expiry_age()` *Please, see the release notes* Devolve o número de segundos até que a sessão expire. Para sessões sem uma expiração customizada (ou aquelas que são configuradas para expirar quando o navegador fecha), esse valor será igual a `settings.SESSION_COOKIE_AGE`.
- `get_expiry_date()` *Please, see the release notes* Devolve a data em que a sessão expirará. Para sessões sem expiração customizada (ou aquelas que são configuradas para expirar quando o navegador fechar), esse valor é igual à data `settings.SESSION_COOKIE_AGE` segundos de agora.
- `get_expire_at_browser_close()` *Please, see the release notes* Devolve `True` ou `False`, dependendo se o cookie de sessão do usuário expirar quando o navegador Web do usuário for fechado.

Você pode editar `request.session` em qualquer ponto de sua view, podendo ser editado múltiplas vezes.

Guia de uso do objeto de sessão

- Use strings normais Python como chaves de dicionário em `request.session`. Isso é mais uma convenção do que uma regra.
- Chaves de dicionário de sessão que começam com “underscore” (“_”) são reservadas para uso interno do Django.
- Não sobrescreva `request.session` com um novo objeto e não acesse ou mude o valor de seus atributos. Use-o como um dicionário Python.

Exemplos

Esta view simplista muda o valor da variável `has_commented` para `True` depois que um usuário posta um comentário. Ela não deixa que um usuário poste um comentário mais de uma vez:

```
def post_comment(request, new_comment):
    if request.session.get('has_commented', False):
        return HttpResponse(u"Você já comentou.")
    c = comments.Comment(comment=new_comment)
    c.save()
    request.session['has_commented'] = True
    return HttpResponse(u'Obrigado pelo seu comentário!')
```

Esta outra view simples autentica um “membro” do site:

```
def login(request):
    m = Member.objects.get(username=request.POST['username'])
    if m.password == request.POST['password']:
        request.session['member_id'] = m.id
        return HttpResponse(u"Você está autenticado.")
    else:
        return HttpResponse(u"Seu nome de usuário e senha não conferem.")
```

...E esta encerra a sessão do membro, de acordo com o `login()` acima:

```
def logout(request):
    try:
        del request.session['member_id']
    except KeyError:
        pass
    return HttpResponse(u"Você saiu.")
```

A função padrão `django.contrib.auth.logout()` na verdade faz um pouco mais do que isso para evitar fugas de dados inadvertida. Ele chama `request.session.flush()`. Nós estamos usando este exemplo como uma demonstração de como trabalhar com objetos de sessão, não uma implementação completa de `logout()`.

Criando cookies de teste

Como uma conveniência, o Django provê uma maneira fácil de testar se o navegador do usuário aceita cookies. Simplesmente chame `request.session.set_test_cookie()` em uma view e chame `request.session.test_cookie_worked()` em uma view subsequente – não na mesma chamada de view.

Essa estranha separação entre `set_test_cookie()` e `test_cookie_worked()` é necessária devido à maneira como os cookies funcionam. Quando você cria um cookie, você não pode dizer se o navegador o aceitou até a próxima requisição do navegador.

É uma boa prática usar `delete_test_cookie()` para limpar o cookie de teste. Faça isso depois que você verificou que o cookie funcionou.

Aqui vai um exemplo típico de uso:

```
def login(request):
    if request.method == 'POST':
        if request.session.test_cookie_worked():
            request.session.delete_test_cookie()
            return HttpResponse(u"Você está autenticado.")
        else:
            return HttpResponse(u"Por favor habilite os cookies e tente novamente. ↩")
    request.session.set_test_cookie()
    return render_to_response('foo/login_form.html')
```

Usando sessões fora das views

Please, see the release notes Uma API está disponível para manipular os dados da sessão fora de uma view:

```
>>> from django.contrib.sessions.backends.db import SessionStore
>>> s = SessionStore(session_key='2b1189a188b44ad18c35e113ac6ceead')
>>> s['last_login'] = datetime.datetime(2005, 8, 20, 13, 35, 10)
>>> s['last_login']
datetime.datetime(2005, 8, 20, 13, 35, 0)
>>> s.save()
```

Se você está usando o backend `django.contrib.sessions.backends.db`, cada sessão é simplesmente um modelo normal do Django. O modelo `Session` é definido em `django/contrib/sessions/models.py`. Por ser um modelo normal, você pode acessar as sessões usando a API normal de banco de dados do Django:

```
>>> from django.contrib.sessions.models import Session
>>> s = Session.objects.get(pk='2b1189a188b44ad18c35e113ac6ceead')
>>> s.expire_date
datetime.datetime(2005, 8, 20, 13, 35, 12)
```

Note que você precisa chamar `get_decoded()` para obter o dicionário da sessão. Isso é necessário porque o dicionário é armazenado em um formato codificado:

```
>>> s.session_data
'KGRwMQpTJl9hdXRoX3VzZXJfaWQnCnAyCkxkxNmuMTEsY2ZjODI2Yj...'
>>> s.get_decoded()
{'user_id': 42}
```

Quando as sessões são gravadas

Por padrão, o Django somente grava no banco de dados da sessão quando ela foi modificada – ou seja, se algum de seus valores de dicionário foram atribuídos ou apagados:

```
# A sessão é modificada.
request.session['foo'] = 'bar'

# A sessão é modificada.
del request.session['foo']

# A sessão é modificada.
request.session['foo'] = {}

# Pegadinha: A sessão NÃO é modificada, porque isso altera
# request.session['foo'] em vez de request.session.
request.session['foo']['bar'] = 'baz'
```

No último caso do exemplo acima, podemos dizer explicitamente ao objeto de sessão que ele foi modificado por meio do atributo `modified` do objeto de sessão:

```
request.session.modified = True
```

Para mudar esse comportamento padrão, coloque o valor de configuração `SESSION_SAVE_EVERY_REQUEST` para `True`. Se `SESSION_SAVE_EVERY_REQUEST` for `True`, o Django gravará a sessão no banco de dados a cada requisição.

Note que o cookie de sessão somente é enviado quando uma sessão foi criada ou modificada. Se `SESSION_SAVE_EVERY_REQUEST` for `True`, o cookie de sessão será enviado em todos os requests.

Similarmente, a parte de `expires` de um cookie de sessão é atualizado a cada vez que o cookie é enviado.

Sessões que duram até o navegador fechar contra sessões persistentes

Você pode controlar se o framework de sessão usa sessões que duram enquanto o navegador está aberto ou sessões persistentes com o parâmetro de configuração `SESSION_EXPIRE_AT_BROWSER_CLOSE`.

Por padrão, `SESSION_EXPIRE_AT_BROWSER_CLOSE` é `False`, o que significa que os cookies de sessão serão armazenados nos navegadores dos usuários pelo tempo determinado no parâmetro `SESSION_COOKIE_AGE`. Use isso se você não quer que as pessoas tenham que se autenticar toda vez que abrem o navegador.

Se `SESSION_EXPIRE_AT_BROWSER_CLOSE` for `True`, o Django usará cookies que duram enquanto o navegador estiver aberto – ou seja, eles expirarão assim que o usuário fechar o seu navegador. Use isso se você deseja que as pessoas tenham de se autenticar toda vez que abrem o seu navegador. *Please, see the release notes* Esse parâmetro é um padrão global que pode ser redefinido para cada sessão chamando explicitamente `request.session.set_expiry()` como descrito acima, em *utilizando sessões nas views*.

Limpendo a tabela de sessão

Se você está usando um banco de dados como backend para as sessões, perceba que os dados de sessão podem acumular na tabela de banco de dados `django_session` e o Django *não* dispõe de uma limpeza automática. Portanto, é seu trabalho limpar as sessões expiradas regularmente.

Para entender esse problema, considere o que acontece quando um usuário usa uma sessão. Quando um usuário se autentica, o Django adiciona uma linha na tabela `django_session`. O Django atualiza essa linha cada vez que os dados da sessão mudam. Se o usuário encerra sua sessão manualmente (logout), o Django apaga essa linha. Mas se o usuário não *sai* (não dá logout), a linha nunca é apagada.

O Django tem um script de exemplo para limpeza em `django-admin.py cleanup`. Esse script apaga qualquer sessão na tabela em que `expire_date` está em uma data no passado –, mas sua aplicação pode ter requisitos diferentes.

Configurações

Algumas *configurações do Django* te dão controle sobre o comportamento de sessões:

SESSION_ENGINE

Please, see the release notes Padrão: `django.contrib.sessions.backends.db`

Controla onde o Django armazena os dados de sessão. Os valores válidos são:

- `'django.contrib.sessions.backends.db'`
- `'django.contrib.sessions.backends.file'`
- `'django.contrib.sessions.backends.cache'`

Veja *configurando o mecanismo de sessão* para mais detalhes.

SESSION_FILE_PATH

Please, see the release notes Padrão: `/tmp/`

Se você está usando um armazenamento baseado em arquivos, esse parâmetro configura o diretório no qual o Django irá armazenar os dados de sessão.

SESSION_COOKIE_AGE

Padrão: `1209600` (2 semanas, em segundos)

O tempo de duração de um cookie de sessão, em segundos.

SESSION_COOKIE_DOMAIN

Padrão: `None`

O domínio para seu uso nos cookies de sessão. Configure-o para uma string como `".lawrence.com"` (repare no ponto inicial) para cookies entre domínios diferentes (cross-domain cookies), ou use `None` para um cookie de domínio padrão.

SESSION_COOKIE_NAME

Padrão: `'sessionid'`

O nome do cookie a usar para as sessões. Esse valor pode ser o que você quiser.

SESSION_COOKIE_SECURE

Padrão: `False`

Se deseja usar cookie seguro para os cookies de sessão ou não. Se este parâmetro for `True`, o cookie será marcado como “seguro”, o que significa que os navegadores devem assegurar-se de que o cookie somente será enviado por meio de uma conexão HTTPS.

SESSION_EXPIRE_AT_BROWSER_CLOSE

Padrão: `False`

Se a sessão deve expirar quando o usuário fecha seu navegador. Veja “Sessões que duram até o navegador fechar contra sessões persistentes” acima.

SESSION_SAVE_EVERY_REQUEST

Padrão: `False`

Se os dados da sessão devem ser gravados a cada request. Se for `False` (padrão), então os dados da sessão somente serão gravados quando forem modificados – ou seja, se algum dos valores de seu dicionário forem alterados ou excluídos.

Detalhes técnicos

- O dicionário de sessão deve aceitar qualquer objeto Python que possa ser serializado com “pickle”. Veja o [módulo pickle](#) para mais informações.
- Os dados de sessão são armazenados em uma tabela de banco de dados chamada `django_session`.
- O Django somente envia o cookie se for necessário. Se você não coloca nenhum dado na sessão, ele não enviará o cookie de sessão.

IDs de sessão em URLs

O framework de sessão do Django é única e inteiramente baseado em cookies. Ele não tenta colocar IDs de sessão em URLs como um último recurso se o navegador não aceita cookies, como o PHP faz. Isso é uma decisão de projeto intencional. Esse comportamento não só cria URLs horríveis, como deixa seu site vulnerável a roubo de ID de sessão por meio do cabeçalho “Referer”.

Trabalhando com formulários

Sobre este documento

Este documento fornece uma introdução as funcionalidades de manipulação de formulários do Django. Para uma visão mais detalhada da API de formulários, veja [A API de formulários](#). Para documentação de tipos de campos disponíveis, veja [Form fields](#).

`django.forms` é a biblioteca de manipulação de formulários.

Embora seja possível processar submissões de formulário somente usando a classe `HttpRequest` do Django, usando a biblioteca de formulários fica melhor para a realização de uma série de tarefas comuns relacionadas a formulários. Usando-o, você pode:

1. Mostrar um formulário HTML com widgets gerados automaticamente.
2. Verificar os dados submetidos conforme um conjunto de regras de validação.
3. Re-exibir um formulário no caso de haver erros de validação.
4. Converter dados de formulários submetidos a tipos relevantes do Python.

Visão geral

A biblioteca trabalha com os seguintes conceitos:

Widget Uma classe que corresponde a um widget de formulário HTML, e.g. `<input type="text">` ou `<textarea>`. Este manipula a renderização do widget como HTML.

Field Uma classe que é responsável por fazer validação, e.g. um `EmailField` que assegura-se de que seu dado é um endereço de e-mail válido.

Form Uma coleção de campos que sabem como validar e mostrar a si mesmo como HTML.

Form Media Os recursos CSS e Javascript que são requeridos para renderizar um formulário.

A biblioteca é dissociada de outros componentes do Django, como a camada de banco de dados., views e templates. Ele invoca somente do Django settings, algumas funções helpers `django.utils` e hooks de internacionalização do Django (mas você não é obrigado a usar internacionalização para usar esta biblioteca).

Objetos Form

Um objeto Form encapsula uma sequência de campos de formulário e uma coleção de regras de validação que devem ser preenchidas para que o formulário seja aceito. Classes Form são criadas como subclasses de `django.forms.Form` e fazem uso de um estilo declarativo, que se você irá familiarizar-se, pois ele é usado nos modelos de banco de dados do Django.

Por exemplo, considere um formulário usado para implementar a funcionalidade “contact me” num Web site pessoal:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

Um formulário é composto de objetos Field. Neste caso, nosso formulário tem quatro campos: `subject`, `message`, `sender` e `cc_myself`. `CharField`, `EmailField` e `BooleanField` são somente três dos tipos tipos de campos disponíveis; uma lista completa pode ser encontrada em [Form fields](#).

Se seu formulário será usado para adicionar ou editar um modelo do Django diretamente, você pode usar um [ModelForm](#) para evitar duplicações de suas descrições de modelo.

Usando um formulário num view

O padrão para processar um formulário num view parece com isso:

```
def contact(request):
    if request.method == 'POST': # If the form has been submitted...
        form = ContactForm(request.POST) # Um form com os dados de POST
        if form.is_valid(): # All validation rules pass
            # Processa os dados no form.cleaned_data
            # ...
            return HttpResponseRedirect('/thanks/') # Redireciona depois do POST
    else:
        form = ContactForm() # Um formulário vazio

    return render_to_response('contact.html', {
        'form': form,
    })
```

Existem três caminhos de código aqui:

1. Se o form não foi submetido, uma instância **unbound** do `ContactForm` é criada e passada para o template.
2. Se o form foi submetido, uma instância **bound** do form é criada usando o `request.POST`. Se os dados submetidos forem válidos, ele é processado e o usuário é redirecionado para uma página de “agradecimento”.
3. Se o formulário foi submetido mas é inválido, uma instância **bound** do form é passada ao template.

O atributo `cleaned_data` foi chamado de `clean_data` em versões anteriores. A distinção entre formulários **bound** e **unbound** é importante. Um formulário **unbound** não tem qualquer dado associado a ele; quando o formulário for renderizado para o usuário, ele estará vazio ou conterá valores padrão. Um formulário **bound** possui dados submetidos, e por isso pode ser usado para dizer se os dados são válidos. Se um formulário **bound** é inválido for renderizado ele pode incluir mensagens de erros, dizendo ao usuário onde ele errou.

Veja [Bound e unbound formulários](#) para mais informações sobre diferenças entre formulários **bound** e **unbound**.

Manipulando upload de arquivos com um form

Para ver como manipular upload de arquivos com seu formulário veja [Vinculando arquivos enviados a um formulário](#) para mais informações.

Processando os dados de um formulário

Uma vez que `is_valid()` retorne `True`, você pode processar a submissão do formulário sabendo que ela está em conformidade com as regras de validação definidas por seu formulário. Embora você possa acessar `request.POST` diretamente deste ponto, é melhor acessar `form.cleaned_data`. Estes dados não serão somente válidos, mas estarão convertidos em tipos relevantes do Python. No exemplo acima, `cc_myself` será um valor booleano. Da mesma forma, campos como `IntegerField` e `FloatField` serão convertidos para `int` e `float` do Python respectivamente.

Extendendo o exemplo acima, aqui temos como os dados do formulário podem ser processados:

```
if form.is_valid():
    subject = form.cleaned_data['subject']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    from django.core.mail import send_mail
    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/') # Redirect after POST
```

Para saber mais sobre como enviar e-mail do Django, veja [Sending e-mail](#).

Mostrando um formulário usando um template

O Forms é projetados para trabalhar com a linguagem de templates do Django. No exemplo acima, nós passamos nossa instância do `ContactForm` para o template usando a variável de contexto `form`. Aqui temos um exemplo simples de template:

```
<form action="/contact/" method="POST"> {{ form.as_p }} <input type="submit" value="Submit" /> </form>
```

O form somente mostra seus próprios campos; ele não fornece tags `<form>` e o botão submit.

O `forms.as_p` mostrará o formulário com cada campo e sua respectiva label envolvidos por um parágrafo. Aqui temos uma saída para nosso template de exemplo:

```
<form action="/contact/" method="POST">
<p><label for="id_subject">Subject:</label>
  <input id="id_subject" type="text" name="subject" maxlength="100" /></p>
<p><label for="id_message">Message:</label>
  <input type="text" name="message" id="id_message" /></p>
<p><label for="id_sender">Sender:</label>
  <input type="text" name="sender" id="id_sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label>
  <input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
<input type="submit" value="Submit" />
</form>
```

Note que cada campo de formulário tem um atributo ID setado para `id_<field-name>`, que é referenciado por uma tag label. Isto é importante para assegurar que os formulários sejam acessíveis para tecnologias assistivas como leitores de tela. Você também pode *customizar a forma em que as labels e ids são gerados*.

Você pode também usar `forms.as_table` para gerar saídas como linhas de tabela. (você precisará fornecer suas próprias tags `<table>`) e `forms.as_ul` para mostrar em itens de lista.

Personalizando o templates de formulário

Se o HTML padrão gerado não é de seu gosto, você pode personalizar completamente a forma como o formulário é apresentado usando a linguagem de template do Django. Extendendo o exemplo acima:

```
<form action="/contact/" method="POST">
  <div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="id_subject">E-mail subject:</label>
    {{ form.subject }}
  </div>
  <div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="id_message">Your message:</label>
    {{ form.message }}
  </div>
  <div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="id_sender">Your email address:</label>
    {{ form.sender }}
  </div>
  <div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="id_cc_myself">CC yourself?</label>
    {{ form.cc_myself }}
  </div>
  <p><input type="submit" value="Send message" /></p>
</form>
```

Cada campo de formulário nomeado pode ser mostrado ao template usando `{{ form.nome_do_campo }}`, que produzirá o HTML necessário para mostrar o widget de formulário. Usando `{{ form.nome_do_campo.errors }}` mostra uma lista de erros de formulário, renderizado como uma lista não ordenada. Ele pode parecer com isso:

```
<ul class="errorlist">
  <li>Sender is required.</li>
</ul>
```

The list has a CSS class of `errorlist` to allow you to style its appearance. If you wish to further customize the display of errors you can do so by looping over them:: A lista tem uma classe CSS `errorlist` para permitir que você mude sua aparência. Se você desejar personalizar a exibição dos erros, você pode fazê-lo iterando sobre eles:

```
{% if form.subject.errors %}
  <ol>
    {% for error in form.subject.errors %}
      <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
  </ol>
{% endif %}
```

Iterando sobre os campos de formulário

Se você estiver usando o mesmo HTML para cada um dos seus campos de formulário, você pode reduzir a duplicidade de código iterando sobre os campos usando um loop `{% for %}`:

```
<form action="/contact/" method="POST">
    {% for field in form %}
        <div class="fieldWrapper">
            {{ field.errors }}
            {{ field.label_tag }}: {{ field }}
        </div>
    {% endfor %}
    <p><input type="submit" value="Send message" /></p>
</form>
```

Dentro deste loop, `{{ field }}` é uma instância de `:class`BoundField``. O `BoundField` também tem os seguintes atributos, que podem ser úteis nos seus templates:

- `{{ field.label }}` A label do campo, e.g. E-mail address
- `{{ field.label_tag }}` A label do campo envolvida por uma tag HTML `<label>`, e.g. `<label for="id_email">E-mail address</label>`
- `{{ field.html_name }}` O nome do campo que será usado no nome do elemento campo input. Este recebe o prefixo do formulário, se ele tiver sido definido
- `{{ field.help_text }}` Qualquer texto de ajuda que deva ser associado com o campo.
- `{{ field.errors }}` Gera um `<ul class="errorlist">` contendo qualquer erro de validação correspondendo a este campo. Você pode personalizar a apresentação dos erros com um loop `{% for error in field.errors %}`. Neste caso, cada objeto no loop é uma string simples contendo a mensagem de erro.

Templates de formulário reusáveis

Se seu site usa a mesma lógica de renderização para formulários em vários lugares, você pode reduzir a duplicidade utilizando um loop de formulário num template autônomo e usando a tag `include` para reusá-lo noutros templates:

```
<form action="/contact/" method="POST">
    {% include "form_snippet.html" %}
    <p><input type="submit" value="Send message" /></p>
</form>

# No form_snippet.html:

{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }}: {{ field }}
    </div>
{% endfor %}
```

Se o objeto `form` passado a um template tem um nome diferente dentro do contexto, você pode criar um alias usando a tag `with`:

```
<form action="/comments/add/" method="POST">
    {% with comment_form as form %}
        {% include "form_snippet.html" %}
    {% endwith %}
    <p><input type="submit" value="Submit comment" /></p>
</form>
```

Se você se pegar fazendo isso frequentemente, você pode considerar criar uma *inclusion tag* personalizada.

Tópicos adicionais

Isto cobre o básico, mas os forms podem fazer muito mais:

Criando formulários a partir de models

ModelForm

Se você está construindo uma aplicação baseada em banco de dados, existe uma grande chance de que seus formulários corresponderão com os seus modelos Django. Por exemplo, você tem um modelo `BlogComment`, e quer criar um formulário que possibilite que as pessoas enviem comentários. Neste caso, seria redundante definir os tipos de campo no seu formulário, porque isso já foi feito no seu modelo.

Por este motivo, o Django disponibiliza uma classe de ajuda que possibilita a criação de uma classe `Form` a partir de um modelo Django.

Por exemplo:

```
>>> from django.forms import ModelForm

# Cria a classe de formulário
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article

# Criando um formulário para adicionar um artigo.
>>> form = ArticleForm()

# Criando um formulário para atualizar dados de um artigo.
>>> article = Article.objects.get(pk=1)
>>> form = ArticleForm(instance=article)
```

Tipos de campos

A classe `Form` gerada terá um campo de formulário para cada campo de modelo. Cada campo de modelo tem um campo de formulário correspondente padrão. Por exemplo, um `CharField` num modelo é representado como um `CharField` num formulário. Um `ManyToManyField` no modelo é representado como um `MultipleChoiceField`. Segue uma lista completa de conversões:

Campo de modelo	Campo de formulário
AutoField	Não é representado no formulário
BooleanField	BooleanField
CharField	CharField com <code>max_length</code> igual ao valor de <code>max_length</code> do campo do modelo
CommaSeparatedIntegerField	CharField
DateField	DateField
DateTimeField	DateTimeField
DecimalField	DecimalField
EmailField	EmailField
FileField	FileField
FilePathField	CharField
FloatField	FloatField
ForeignKey	ModelChoiceField (veja abaixo)
ImageField	ImageField
IntegerField	IntegerField
IPAddressField	IPAddressField
ManyToManyField	ModelMultipleChoiceField (veja abaixo)
NullBooleanField	CharField
PhoneNumberField	USPhoneNumberField (de <code>django.contrib.localflavor.us</code>)
PositiveIntegerField	IntegerField
PositiveSmallIntegerField	IntegerField
SlugField	CharField
SmallIntegerField	IntegerField
TextField	CharField com <code>widget=forms.Textarea</code>
TimeField	TimeField
URLField	URLField com <code>verify_exists</code> igual ao valor de <code>verify_exists</code> do campo do modelo
XMLField	CharField com <code>widget=Textarea</code>

O campo de formulário `FloatField` juntamente com os campos de formulário e de modelo `DecimalField` são novos no Django 1.0. Como esperado, os campos de modelo do tipo `ForeignKey` e `ManyToManyField` são casos especiais:

- `ForeignKey` é representado por `django.forms.ModelChoiceField`, que é um `ChoiceField` em que `choices` é um `QuerySet` do modelo.
- `ManyToManyField` é representado por `django.forms.ModelMultipleChoiceField`, que é um `MultipleChoiceField` em que `choices` é um `QuerySet` do modelo.

Além disso, cada campo de formulário gerado tem os valores de atributos definidos como asseguir:

- Se um campo de modelo tem `blank=True`, então o valor de `required` será `False` no campo de formulário. Caso contrário, `required=True`.
- O atributo `label` do campo de formulário será igual ao `verbose_name` do campo de modelo, com o primeiro caractere em maiúsculo.
- O `help_text` do campo de formulário é igual ao `help_text` do campo de modelo.
- Se o campo de modelo tem o atributo `choices` definido, então o `widget` do campo de formulário será o `Select`, com a lista de opções vindas do atributo `choices` do campo de modelo. As opções normalmente incluirão o valor em branco, que é selecionado por padrão. Se o campo é requerido, isso força o usuário a fazer uma escolha. O valor em branco não será incluído se o campo de modelo tem atributo `blank=False` e um valor de `default` explícito (em vez disso o valor de `default` será selecionado inicialmente).

Finalmente, note que você pode redefinir o campo de formulário utilizado por um determinado modelo. Veja *Redefinindo os tipos de campo padrão* abaixo.

Um exemplo completo

Considere este conjunto de modelos:

```
from django.db import models
from django.forms import ModelForm

TITLE_CHOICES = (
    ('MR', 'Mr.'),
    ('MRS', 'Mrs.'),
    ('MS', 'Ms.'),
)

class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)
    birth_date = models.DateField(blank=True, null=True)

    def __unicode__(self):
        return self.name

class Book(models.Model):
    name = models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)

class AuthorForm(ModelForm):
    class Meta:
        model = Author

class BookForm(ModelForm):
    class Meta:
        model = Book
```

Com estes modelos, as subclasses de `ModelForm` acima seriam equivalentes a isto (a única diferença sendo o método `save()`, que discutiremos daqui a pouco.):

```
class AuthorForm(forms.Form):
    name = forms.CharField(max_length=100)
    title = forms.CharField(max_length=3,
                           widget=forms.Select(choices=TITLE_CHOICES))
    birth_date = forms.DateField(required=False)

class BookForm(forms.Form):
    name = forms.CharField(max_length=100)
    authors = forms.ModelMultipleChoiceField(queryset=Author.objects.all())
```

O método `save()`

Todo formulário produzido por `ModelForm` tem também um método `save()`. Este método cria e grava um objeto no banco de dados a partir dos dados atrelados ao formulário. Uma subclasse de `ModelForm` pode aceitar uma instância de modelo existente como uma argumento nomeado `instance`; se este argumento é fornecido, o `save()` irá atualizar a instância. Se não é fornecido, o `save()` criará uma nova instância do modelo especificado:

```
# Cria uma instância de formulário com dados do POST.
>>> f = ArticleForm(request.POST)

# Grava um novo objeto Article com os dados do formulário.
>>> new_article = f.save()
```

```
# Cria um formulário para editar um Article existente.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(instance=a)
>>> f.save()

# Cria um formulário para editar um Article existente, mas
# usa os dados do POST para popular o formulário.
>>> a = Article.objects.get(pk=1)
>>> f = ArticleForm(request.POST, instance=a)
>>> f.save()
```

Note que `save()` irá levantar uma exceção `ValueError` se os dados no formulário não validarem – ou seja `if form.errors`.

Este método `save()` aceita um argumento nomeado opcional `commit`, que aceita ou `True` ou `False`. Se você chamar `save()` com `commit=False`, então ele devolverá um objeto que ainda não foi gravado no banco de dados. Neste caso, é sua responsabilidade chamar `save()` na instância de modelo. Isso é útil se você quer fazer algum custom processamento customizado no objeto antes de gravá-lo, ou se você quer usar um das *opções de gravação de modelo* especializadas. `commit` é `True` por padrão.

Um efeito colateral no uso de `commit=False` é notado quando o seu modelo tem um relacionamento de muitos para muitos com outro modelo. Se seu modelo tem um relacionamento de muitos para muitos e você especifica `commit=False` quando vai gravar um formulário, o Django não pode gravar os dados para o relacionamento de muitos para muitos imediatamente. Isso se deve ao fato de não ser possível gravar os dados de muitos para muitos para uma instância que ainda não existe no banco de dados.

Para contornar este problema, cada vez que você grava um formulário usando `commit=False`, o Django adiciona um método `save_m2m()` para a sua subclasse de `ModelForm`. Depois de gravar manualmente a instância produzida pelo formulário, você pode chamar `save_m2m()` para gravar os dados de muitos para muitos do formulário. Por exemplo:

```
# Cria uma instância de formulário com os dados de POST.
>>> f = AuthorForm(request.POST)

# Cria, mas não grava a nova instância de Author.
>>> new_author = f.save(commit=False)

# Modifica o Author de alguma maneira.
>>> new_author.some_field = 'some_value'

# Grava a nova instância.
>>> new_author.save()

# Agora, grava os dados de muitos para muitos para o formulário.
>>> f.save_m2m()
```

Só é necessário chamar o `save_m2m()` se você usar `save(commit=False)`. Quando você simplesmente usa o `save()` num formulário, todos os dados – incluindo os dados de muitos para muitos – são gravados sem a necessidade de chamadas a nenhum método adicional. Por exemplo:

```
# Cria uma instância de formulário com os dados de POST.
>>> a = Author()
>>> f = AuthorForm(request.POST, instance=a)

# Cria e grava a nova instância de Author. Não há necessidade de fazer
# nada mais.
>>> new_author = f.save()
```

A não ser pelos métodos `save()` e `save_m2m()`, um `ModelForm` funciona exatamente igual a qualquer outro formulário de forms. Por exemplo, o método `is_valid()` é utilizado para validar os dados, o método `is_multipart()` para determinar se um formulário requer upload de arquivo multipart (e se `request.FILES` deve ser passado ao formulário), etc. Veja *Vinculando arquivos enviados a um formulário* para mais

informação.

Usando somente alguns campos no formulário

Em alguns casos, você não quer que todos os campos do modelo apareçam no formulário gerado. Existem três maneiras de dizer ao `ModelForm` para usar somente alguns campos do modelo:

1. Coloque `editable=False` no campo do modelo. Como resultado, *qualquer* formulário criado via `ModelForm` não incluirá este campo.
2. Use o atributo `fields` da classe interna `Meta` do `ModelForm`. Esse atributo, se especificado, deve ser uma lista de nomes de campos a serem incluídos no formulário.
3. Use o atributo `exclude` da classe interna `Meta` do `ModelForm`. Esse atributo, se especificado, deve ser uma lista de nomes de campos a serem excluídos do formulário.

Por exemplo, se você quer que um formulário para o modelo `Author` (definido acima) tenha somente os campos `name` e `title`, você especificaria `fields` ou `exclude` assim:

```
class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        fields = ('name', 'title')

class PartialAuthorForm(ModelForm):
    class Meta:
        model = Author
        exclude = ('birth_date',)
```

Já que o modelo `Author` tem somente três campos, `'name'`, `'title'`, e `'birth_date'`, os formulários acima conterão exatamente os mesmos campos.

Note: Se você especifica `fields` ou `exclude` na criação de um formulário com `ModelForm`, então na chamada do método `save()`, não serão atribuídos valores aos campos que não constam do formulário resultante. O Django impedirá qualquer tentativa de gravar um modelo incompleto, então se o modelo não permite que os campos faltantes sejam vazios, e não existe um valor padrão definido para eles, qualquer tentativa de chamar `save()` num `ModelForm` com campos faltantes não funcionará. Para evitar esse erro, você deve instanciar seu modelo com valores iniciais para os campos vazios, porém obrigatórios:

```
author = Author(title='Mr')
form = PartialAuthorForm(request.POST, instance=author)
form.save()
```

Alternatively, you can use `save(commit=False)` and manually set any extra required fields:

```
form = PartialAuthorForm(request.POST)
author = form.save(commit=False)
author.title = 'Mr'
author.save()
```

Veja a [seção sobre gravação de formulários](#) para mais detalhes no uso de `save(commit=False)`.

Redefinindo os tipos de campo padrão

Os tipos de campos padrão, como descritos na tabela *Tipos de campos* acima, são padrões sensatos. Se você tem um `DateField` no seu modelo, existem grandes chances de que você queira que ele seja representado como um `DateField` no seu formulário. Mas o `ModelForm` te dá a flexibilidade de mudar o campo de formulário para um determinado campo de modelo. Isso é feito de maneira declarativa especificando os campos como faria num `Form` normal. Campos declarados redefinirão os campos padrões gerados pelo uso do atributo `model`.

Por exemplo, se você quiser usar `MyDateFormField` para o campo `pub_date`, faça o seguinte:

```
>>> class ArticleForm(ModelForm):
...     pub_date = MyDateFormField()
...
...     class Meta:
...         model = Article
```

Se quiser redefinir um widget padrão de um campo, então especifique o parâmetro `widget` quando declarar o campo de formulário:

```
>>> class ArticleForm(ModelForm):
...     pub_date = DateField(widget=MyDateWidget())
...
...     class Meta:
...         model = Article
```

Sobrescrevendo o método `clean()`

Você pode sobrescrever o método `clean()` em um formulário de modelo para fornecer informações adicionais como uma validação, da mesma forma que faria num formulário normal.

In this regard, model forms have two specific characteristics when compared to forms:

Por padrão o método `clean()` valida a unicidade dos campos que são marcados como `unique`, `unique_together` ou `unique_for_date|month|year` no modelo. Portanto, se você gostaria de sobrescrever o método `clean()` e manter a validação padrão, você deve chamar o método da `clean()` da classe pai.

Also, a model form instance bound to a model object will contain a `self.instance` attribute that gives model form methods access to that specific model instance.

Herança de formulário

Como nos formulários básicos, você pode estender e reutilizar `ModelForms` através da herança. Isso é útil se você precisa declarar campos ou métodos adicionais em uma classe pai para uso em alguns formulários derivados de modelos. Por exemplo, usando a classe `ArticleForm` anterior:

```
>>> class EnhancedArticleForm(ArticleForm):
...     def clean_pub_date(self):
...         ...
```

Isso cria um formulário que se comporta da mesma maneira que `ArticleForm`, exceto por alguma validação e limpeza de dados adicional para o campo `pub_date`.

Você pode criar uma subclasse da classe interna `Meta` da classe pai se você quer mudar as listas `Meta.fields` ou `Meta.excludes`:

```
>>> class RestrictedArticleForm(EnhancedArticleForm):
...     class Meta(ArticleForm.Meta):
...         exclude = ['body']
```

Isso adiciona o método de `EnhancedArticleForm` e modifica o `ArticleForm.Meta` original para remover um campo.

Entretanto, existem algumas coisas a serem notadas.

- As regras normais de resolução de nomes Python se aplicam. Se você tem múltiplas classes base que declaram uma classe interna `Meta`, somente a primeira será utilizada. Isso significa o `Meta` do filho, se existir, caso contrário o `Meta` do primeiro pai, etc.

- Pela mesma razão, uma subclasse não pode herdar de `ModelForm` e `Form` simultaneamente.

É bem provável que estas notas não te afetarão, a não ser que esteja tentando fazer algo complexo utilizando subclasses.

Model Formsets

Como nos *formsets comuns*, o Django fornece um par de classes avançadas de formset que tornam fácil trabalhar com modelos do Django. Vamos reutilizar o modelo `Author` do exemplo acima:

```
>>> from django.forms.models import modelformset_factory
>>> AuthorFormSet = modelformset_factory(Author)
```

Isso criará um formset que é capaz de funcionar com os dados associados ao modelo `Author` que funciona exatamente igual a um formset comum:

```
>>> formset = AuthorFormSet()
>>> print formset
<input type="hidden" name="form-TOTAL_FORMS" value="1" id="id_form-TOTAL_FORMS" />
↪<input type="hidden" name="form-INITIAL_FORMS" value="0" id="id_form-INITIAL_
↪FORMS" />
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-name
↪" type="text" name="form-0-name" maxlength="100" /></td></tr>
<tr><th><label for="id_form-0-title">Title:</label></th><td><select name="form-0-
↪title" id="id_form-0-title">
<option value="" selected="selected">-----</option>
<option value="MR">Mr.</option>
<option value="MRS">Mrs.</option>
<option value="MS">Ms.</option>
</select></td></tr>
<tr><th><label for="id_form-0-birth_date">Birth date:</label></th><td><input type=
↪"text" name="form-0-birth_date" id="id_form-0-birth_date" /><input type="hidden"
↪name="form-0-id" id="id_form-0-id" /></td></tr>
```

Note: Uma coisa para se notar é que `modelformset_factory` usa `formset_factory` para gerar formsets. Isto significa que um `formset` de `model` é somente uma extensão de um `formset` básico que sabe como interagir com um `model` em particular.

Mudando o queryset

Por padrão quando você cria um formset de um modelo o formset irá usar um `queryset` que inclui todos os objetos no modelo (ex: `Author.objects.all()`). Você pode sobrescrever esse comportamento usando o argumento `queryset`:

```
>>> formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith='O'))
```

Alternativamente, você pode criar uma subclasse que defina o `self.queryset` no `__init__`:

```
from django.forms.models import BaseModelFormSet

class BaseAuthorFormSet(BaseModelFormSet):
    def __init__(self, *args, **kwargs):
        self.queryset = Author.objects.filter(name__startswith='O')
        super(BaseAuthorFormSet, self).__init__(*args, **kwargs)
```

Então, passe sua classe `BaseAuthorFormSet` para a função `factory`:

```
>>> AuthorFormSet = modelformset_factory(Author, formset=BaseAuthorFormSet)
```

Controlando quais campos são usados com `fields` e `exclude`

Por padrão um model formset usará todos os campos do model, que não estão marcados com `editable=False`. Entretanto, isto pode ser sobrescrito no nível do formset:

```
>>> AuthorFormSet = modelformset_factory(Author, fields=('name', 'title'))
```

Usando `fields` você irá restringir o formset para usar somente os campos fornecidos. Alternativamente, você pode ter uma abordagem de “out-put”, especificando quais campos serão excluídos:

```
>>> AuthorFormSet = modelformset_factory(Author, exclude=('birth_date',))
```

Usando `exclude` você irá evistar que dados campos sejam usados em um formset.

Gravando objetos num formset

Como num `ModelForm` você pode gravar os dados no objeto do modelo. Isso é feito com o método `save()` do formset:

```
# Cria uma instância de formset com os dados do POST.
>>> formset = AuthorFormSet(request.POST)

# Supondo que tudo está válido, grave os dados
>>> instances = formset.save()
```

O método `save()` devolverá as instâncias que foram gravadas no banco de dados. Se uma instância não mudar seus nos dados atrelados ela não será gravada no banco e não será encontrada no valor de return (`instances` no exemplo acima).

Pass `commit=False` to return the unsaved model instances:

```
# não grava no banco de dados
>>> instances = formset.save(commit=False)
>>> for instance in instances:
...     # faz alguma coisa com a instância
...     instance.save()
```

Isso dá a habilidade de anexar dados às instâncias antes de gravá-las no banco de dados. Se seu formset contém um `ManyToManyField` você precisará também chamar `formset.save_m2m()` para assegurar que os relacionamentos de muitos para muitos serão gravados apropriadamente.

Limitando o número de objetos editáveis

Como em formsets comuns você pode usar um parâmetro `max_num` no `modelformset_factory` para limitar o número de formulários mostrados. Com formsets de modelo, isso limitará apropriadamente a query para selecionar somente o máximo de objetos necessários:

```
>>> Author.objects.order_by('name')
[<Author: Charles Baudelaire>, <Author: Paul Verlaine>, <Author: Walt Whitman>]

>>> AuthorFormSet = modelformset_factory(Author, max_num=2, extra=1)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> formset.initial
[{'id': 1, 'name': u'Charles Baudelaire'}, {'id': 3, 'name': u'Paul Verlaine'}]
```

Se o valor de `max_num` é maior que o número de objetos retornados, formulários em branco extra serão adicionados ao `formset`, tão logo o total de formulários não exceda `max_num`:

```
>>> AuthorFormSet = modelformset_factory(Author, max_num=4, extra=2)
>>> formset = AuthorFormSet(queryset=Author.objects.order_by('name'))
>>> for form in formset.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-name">Name:</label></th><td><input id="id_form-0-name"
↪ type="text" name="form-0-name" value="Charles Baudelaire" maxlength="100" />
↪<input type="hidden" name="form-0-id" value="1" id="id_form-0-id" /></td></tr>
<tr><th><label for="id_form-1-name">Name:</label></th><td><input id="id_form-1-name"
↪ type="text" name="form-1-name" value="Paul Verlaine" maxlength="100" /><input
↪ type="hidden" name="form-1-id" value="3" id="id_form-1-id" /></td></tr>
<tr><th><label for="id_form-2-name">Name:</label></th><td><input id="id_form-2-name"
↪ type="text" name="form-2-name" value="Walt Whitman" maxlength="100" /><input
↪ type="hidden" name="form-2-id" value="2" id="id_form-2-id" /></td></tr>
<tr><th><label for="id_form-3-name">Name:</label></th><td><input id="id_form-3-name"
↪ type="text" name="form-3-name" maxlength="100" /><input type="hidden" name=
↪ "form-3-id" id="id_form-3-id" /></td></tr>
```

Usando um model formset em uma visão

Models formsets são muito similares a formsets. Digamos que queiramos apresentar um formset para editar instâncias do modelo `Author`:

```
def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author)
    if request.method == 'POST':
        formset = AuthorFormSet(request.POST, request.FILES)
        if formset.is_valid():
            formset.save()
            # Faça algo.
        else:
            formset = AuthorFormSet()
    return render_to_response("manage_authors.html", {
        "formset": formset,
    })
```

Como você pode ver a lógica da visão de um formset de modelo não é drasticamente diferente de como usar um formset “normal”. A única diferença é que nos chamamos `formset.save()` para salvar os dados no banco de dados. (Isto foi descrito acima em *Gravando objetos num formset*.)

Sobrescrevendo o `clean()` sobre um `model_formset`

Assim como com `ModelForms`, por padrão o método `clean()` de um `model_formset` irá validar se nenhum dos itens do formset viola as restrições de unicidade do seu model (qualquer uma `unique`, `unique_together` ou `unique_for_date|month|year`). Se você quiser sobrescrever o método `clean()` de um `model_formset` e manter esta validação, você deve chamar o método `clean` da classe pai:

```
class MyModelFormSet(BaseModelFormSet):
    def clean(self):
        super(MyModelFormSet, self).clean()
        # exemplo de validação customizada de forms do formset:
        for form in self.forms:
            # your custom formset validation
            # sua validação personalizada de formset
```

Usando um queryset customizado

Como já foi dito, você pode sobrescrever o queryset padrão usado pelo model formset:

```
def manage_authors(request):
    AuthorFormSet = modelformset_factory(Author)
    if request.method == "POST":
        formset = AuthorFormSet(request.POST, request.FILES,
                                queryset=Author.objects.filter(name__startswith='O
↪'))
        if formset.is_valid():
            formset.save()
            # Faça algo.
        else:
            formset = AuthorFormSet(queryset=Author.objects.filter(name__startswith='O
↪'))
    return render_to_response("manage_authors.html", {
        "formset": formset,
    })
```

Note que nós passamos o argumento `queryset` em ambos os casos POST e GET neste exemplo.

Usando o formset no template

Já três formas de renderizar um formset num template Django.

Primeiro, você pode deixar o formset fazer a maior parte do trabalho:

```
<form method="POST" action="">
    {{ formset }}
</form>
```

Segundo, você pode manualmente renderizar o formset, mas deixe o form trabalhar por conta própria:

```
<form method="POST" action="">
    {{ formset.management_form }}
    {% for form in formset.forms %}
        {{ form }}
    {% endfor %}
</form>
```

Quando você renderizar formulários manualmente, certifique-se de renderizar o `management_form` como mostrado acima. Veja a [documentação do management form](#).

Terceiro, você pode renderizar manualmente cada campo:

```
<form method="POST" action="">
    {{ formset.management_form }}
    {% for form in formset.forms %}
        {% for field in form %}
            {{ field.label_tag }}: {{ field }}
        {% endfor %}
    {% endfor %}
</form>
```

Se você optar por usar este terceiro método e você não iterar sobre os campos com um loop `{% for %}`, você precisará renderizar a chave primária. Por exemplo, se você renderizou os campos `name` e `age` de um model:

```
<form method="POST" action="">
    {{ formset.management_form }}
    {% for form in formset.forms %}
        {{ form.id }}
```

```
<ul>
    <li>{{ form.name }}</li>
    <li>{{ form.age }}</li>
</ul>
{% endfor %}
</form>
```

Observe como é preciso, explicitamente, renderizar `{{ form.id }}`. Isto garante que o formset do model, no caso do POST, irá funcionar perfeitamente. (Este exemplo assume uma chave primária chamada `id`, certifique-se de renderizá-la.)

Formsets em linha (*inline*)

Formsets em linha é uma pequena camada de abstração sobre os formsets de model. Este simplificam o caso de trabalhar com objetos relacionados através de uma chave estrangeira. Suponhamos que você tenha dois models:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author)
    title = models.CharField(max_length=100)
```

Se você quiser criar um formset que permita editar books pertencentes a um autor em particular, você pode fazer isto:

```
>>> from django.forms.models import inlineformset_factory
>>> BookFormSet = inlineformset_factory(Author, Book)
>>> author = Author.objects.get(name=u'Mike Royko')
>>> formset = BookFormSet(instance=author)
```

Note: `inlineformset_factory` usa `modelformset_factory` e marca `can_delete=True`.

Mais de uma chave estrangeira para o mesmo model

Se o seu model contém mais de uma chave estrangeira para o mesmo model, você precisará resolver a ambiguidade manualmente utilizando `fk_name`. Por exemplo, considere o seguinte model:

```
class Friendship(models.Model):
    from_friend = models.ForeignKey(Friend)
    to_friend = models.ForeignKey(Friend)
    length_in_months = models.IntegerField()
```

Para resolver isto, você pode usar `fk_name` para `inlineformset_factory`:

```
>>> FriendshipFormSet = inlineformset_factory(Friend, Friendship, fk_name="from_
↪friend")
```

Usando um formset em linha em um view

Você pode querer prover um view que permite um usuário editar os objetos relacionados de um model. Aqui tem uma maneira de se fazer isso:

```
def manage_books(request, author_id):
    author = Author.objects.get(pk=author_id)
    BookInlineFormSet = inlineformset_factory(Author, Book)
    if request.method == "POST":
        formset = BookInlineFormSet(request.POST, request.FILES, instance=author)
        if formset.is_valid():
            formset.save()
            # Do something.
    else:
        formset = BookInlineFormSet(instance=author)
    return render_to_response("manage_books.html", {
        "formset": formset,
    })
```

Observe como passamos `instance` em ambos os casos POST e GET.

Formsets

A formset is a layer of abstraction to working with multiple forms on the same page. It can be best compared to a data grid. Let's say you have the following form:

```
>>> from django import forms
>>> class ArticleForm(forms.Form):
...     title = forms.CharField()
...     pub_date = forms.DateField()
```

You might want to allow the user to create several articles at once. To create a formset out of an `ArticleForm` you would do:

```
>>> from django.forms.formsets import formset_factory
>>> ArticleFormSet = formset_factory(ArticleForm)
```

You now have created a formset named `ArticleFormSet`. The formset gives you the ability to iterate over the forms in the formset and display them as you would with a regular form:

```
>>> formset = ArticleFormSet()
>>> for form in formset.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
↪name="form-0-title" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text
↪" name="form-0-pub_date" id="id_form-0-pub_date" /></td></tr>
```

As you can see it only displayed one form. This is because by default the `formset_factory` defines one extra form. This can be controlled with the `extra` parameter:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
```

Using initial data with a formset

Initial data is what drives the main usability of a formset. As shown above you can define the number of extra forms. What this means is that you are telling the formset how many additional forms to show in addition to the number of forms it generates from the initial data. Lets take a look at an example:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2)
>>> formset = ArticleFormSet(initial=[
...     {'title': u'Django is now open source',
...     'pub_date': datetime.date.today()},
... ])
```



```
>>> for form in formset.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
↪name="form-0-title" value="Django is now open source" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text"
↪" name="form-0-pub_date" value="2008-05-12" id="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text"
↪name="form-1-title" id="id_form-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text"
↪" name="form-1-pub_date" id="id_form-1-pub_date" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text"
↪name="form-2-title" id="id_form-2-title" /></td></tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text"
↪" name="form-2-pub_date" id="id_form-2-pub_date" /></td></tr>
```

There are now a total of three forms showing above. One for the initial data that was passed in and two extra forms. Also note that we are passing in a list of dictionaries as the initial data.

See also:

Creating formsets from models with model formsets.

Limiting the maximum number of forms

The `max_num` parameter to `formset_factory` gives you the ability to force the maximum number of forms the formset will display:

```
>>> ArticleFormSet = formset_factory(ArticleForm, extra=2, max_num=1)
>>> formset = ArticleFormSet()
>>> for form in formset.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
↪name="form-0-title" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text"
↪" name="form-0-pub_date" id="id_form-0-pub_date" /></td></tr>
```

A `max_num` value of 0 (the default) puts no limit on the number forms displayed.

Formset validation

Validation with a formset is almost identical to a regular `Form`. There is an `is_valid` method on the formset to provide a convenient way to validate all forms in the formset:

```
>>> ArticleFormSet = formset_factory(ArticleForm)
>>> formset = ArticleFormSet({})
>>> formset.is_valid()
True
```

We passed in no data to the formset which is resulting in a valid form. The formset is smart enough to ignore extra forms that were not changed. If we provide an invalid article:

```
>>> data = {
...     'form-TOTAL_FORMS': u'2',
...     'form-INITIAL_FORMS': u'0',
...     'form-0-title': u'Test',
...     'form-0-pub_date': u'16 June 1904',
...     'form-1-title': u'Test',
...     'form-1-pub_date': u'', # <-- this date is missing but required
```

```
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {'pub_date': [u'This field is required.']]]
```

As we can see, `formset.errors` is a list whose entries correspond to the forms in the formset. Validation was performed for each of the two forms, and the expected error message appears for the second item.

Understanding the ManagementForm

You may have noticed the additional data that was required in the formset's data above. This data is coming from the `ManagementForm`. This form is dealt with internally to the formset. If you don't use it, it will result in an exception:

```
>>> data = {
...     'form-0-title': u'Test',
...     'form-0-pub_date': u'',
... }
>>> formset = ArticleFormSet(data)
Traceback (most recent call last):
...
django.forms.util.ValidationError: [u'ManagementForm data is missing or has been_
↳tampered with']
```

It is used to keep track of how many form instances are being displayed. If you are adding new forms via JavaScript, you should increment the count fields in this form as well.

Custom formset validation

A formset has a `clean` method similar to the one on a `Form` class. This is where you define your own validation that works at the formset level:

```
>>> from django.forms.formsets import BaseFormSet

>>> class BaseArticleFormSet(BaseFormSet):
...     def clean(self):
...         """Checks that no two articles have the same title."""
...         if any(self.errors):
...             # Don't bother validating the formset unless each form is valid on_
↳its own
...             return
...         titles = []
...         for i in range(0, self.total_form_count()):
...             form = self.forms[i]
...             title = form.cleaned_data['title']
...             if title in titles:
...                 raise forms.ValidationError, "Articles in a set must have_
↳distinct titles."
...                 titles.append(title)

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> data = {
...     'form-TOTAL_FORMS': u'2',
...     'form-INITIAL_FORMS': u'0',
...     'form-0-title': u'Test',
...     'form-0-pub_date': u'16 June 1904',
...     'form-1-title': u'Test',
```

```
...     'form-1-pub_date': u'23 June 1912',
... }
>>> formset = ArticleFormSet(data)
>>> formset.is_valid()
False
>>> formset.errors
[{}, {}]
>>> formset.non_form_errors()
[u'Articles in a set must have distinct titles.']
```

The `formset` `clean` method is called after all the `Form.clean` methods have been called. The errors will be found using the `non_form_errors()` method on the `formset`.

Dealing with ordering and deletion of forms

Common use cases with a `formset` is dealing with ordering and deletion of the form instances. This has been dealt with for you. The `formset_factory` provides two optional parameters `can_order` and `can_delete` that will do the extra work of adding the extra fields and providing simpler ways of getting to that data.

`can_order`

Default: `False`

Lets create a `formset` with the ability to order:

```
>>> ArticleFormSet = formset_factory(ArticleForm, can_order=True)
>>> formset = ArticleFormSet(initial=[
...     {'title': u'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': u'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> for form in formset.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
↪name="form-0-title" value="Article #1" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text"
↪ name="form-0-pub_date" value="2008-05-10" id="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-0-ORDER">Order:</label></th><td><input type="text"
↪name="form-0-ORDER" value="1" id="id_form-0-ORDER" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text"
↪name="form-1-title" value="Article #2" id="id_form-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text"
↪ name="form-1-pub_date" value="2008-05-11" id="id_form-1-pub_date" /></td></tr>
<tr><th><label for="id_form-1-ORDER">Order:</label></th><td><input type="text"
↪name="form-1-ORDER" value="2" id="id_form-1-ORDER" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text"
↪name="form-2-title" id="id_form-2-title" /></td></tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text"
↪ name="form-2-pub_date" id="id_form-2-pub_date" /></td></tr>
<tr><th><label for="id_form-2-ORDER">Order:</label></th><td><input type="text"
↪name="form-2-ORDER" id="id_form-2-ORDER" /></td></tr>
```

This adds an additional field to each form. This new field is named `ORDER` and is an `forms.IntegerField`. For the forms that came from the initial data it automatically assigned them a numeric value. Lets look at what will happen when the user changes these values:

```
>>> data = {
...     'form-TOTAL_FORMS': u'3',
...     'form-INITIAL_FORMS': u'2',
...     'form-0-title': u'Article #1',
...     'form-0-pub_date': u'2008-05-10',
```

```
...     'form-0-ORDER': u'2',
...     'form-1-title': u'Article #2',
...     'form-1-pub_date': u'2008-05-11',
...     'form-1-ORDER': u'1',
...     'form-2-title': u'Article #3',
...     'form-2-pub_date': u'2008-05-01',
...     'form-2-ORDER': u'0',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': u'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': u'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> formset.is_valid()
True
>>> for form in formset.ordered_forms:
...     print form.cleaned_data
{'pub_date': datetime.date(2008, 5, 1), 'ORDER': 0, 'title': u'Article #3'}
{'pub_date': datetime.date(2008, 5, 11), 'ORDER': 1, 'title': u'Article #2'}
{'pub_date': datetime.date(2008, 5, 10), 'ORDER': 2, 'title': u'Article #1'}
```

can_delete

Default: False

Lets create a formset with the ability to delete:

```
>>> ArticleFormSet = formset_factory(ArticleForm, can_delete=True)
>>> formset = ArticleFormSet(initial=[
...     {'title': u'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': u'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> for form in formset.forms:
...     print form.as_table()
<input type="hidden" name="form-TOTAL_FORMS" value="3" id="id_form-TOTAL_FORMS" />
↪<input type="hidden" name="form-INITIAL_FORMS" value="2" id="id_form-INITIAL_
↪FORMS" />
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
↪name="form-0-title" value="Article #1" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text"
↪name="form-0-pub_date" value="2008-05-10" id="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-0-DELETE">Delete:</label></th><td><input type="checkbox"
↪name="form-0-DELETE" id="id_form-0-DELETE" /></td></tr>
<tr><th><label for="id_form-1-title">Title:</label></th><td><input type="text"
↪name="form-1-title" value="Article #2" id="id_form-1-title" /></td></tr>
<tr><th><label for="id_form-1-pub_date">Pub date:</label></th><td><input type="text"
↪name="form-1-pub_date" value="2008-05-11" id="id_form-1-pub_date" /></td></tr>
<tr><th><label for="id_form-1-DELETE">Delete:</label></th><td><input type="checkbox"
↪name="form-1-DELETE" id="id_form-1-DELETE" /></td></tr>
<tr><th><label for="id_form-2-title">Title:</label></th><td><input type="text"
↪name="form-2-title" id="id_form-2-title" /></td></tr>
<tr><th><label for="id_form-2-pub_date">Pub date:</label></th><td><input type="text"
↪name="form-2-pub_date" id="id_form-2-pub_date" /></td></tr>
<tr><th><label for="id_form-2-DELETE">Delete:</label></th><td><input type="checkbox"
↪name="form-2-DELETE" id="id_form-2-DELETE" /></td></tr>
```

Similar to `can_order` this adds a new field to each form named `DELETE` and is a `forms.BooleanField`. When data comes through marking any of the delete fields you can access them with `deleted_forms`:

```
>>> data = {
...     'form-TOTAL_FORMS': u'3',
```

```
...     'form-INITIAL_FORMS': u'2',
...     'form-0-title': u'Article #1',
...     'form-0-pub_date': u'2008-05-10',
...     'form-0-DELETE': u'on',
...     'form-1-title': u'Article #2',
...     'form-1-pub_date': u'2008-05-11',
...     'form-1-DELETE': u'',
...     'form-2-title': u'',
...     'form-2-pub_date': u'',
...     'form-2-DELETE': u'',
... }

>>> formset = ArticleFormSet(data, initial=[
...     {'title': u'Article #1', 'pub_date': datetime.date(2008, 5, 10)},
...     {'title': u'Article #2', 'pub_date': datetime.date(2008, 5, 11)},
... ])
>>> [form.cleaned_data for form in formset.deleted_forms]
[{'DELETE': True, 'pub_date': datetime.date(2008, 5, 10), 'title': u'Article #1'}]
```

Adding additional fields to a formset

If you need to add additional fields to the formset this can be easily accomplished. The formset base class provides an `add_fields` method. You can simply override this method to add your own fields or even redefine the default fields/attributes of the order and deletion fields:

```
>>> class BaseArticleFormSet(BaseFormSet):
...     def add_fields(self, form, index):
...         super(BaseArticleFormSet, self).add_fields(form, index)
...         form.fields["my_field"] = forms.CharField()

>>> ArticleFormSet = formset_factory(ArticleForm, formset=BaseArticleFormSet)
>>> formset = ArticleFormSet()
>>> for form in formset.forms:
...     print form.as_table()
<tr><th><label for="id_form-0-title">Title:</label></th><td><input type="text"
↪ name="form-0-title" id="id_form-0-title" /></td></tr>
<tr><th><label for="id_form-0-pub_date">Pub date:</label></th><td><input type="text"
↪ name="form-0-pub_date" id="id_form-0-pub_date" /></td></tr>
<tr><th><label for="id_form-0-my_field">My field:</label></th><td><input type="text"
↪ name="form-0-my_field" id="id_form-0-my_field" /></td></tr>
```

Using a formset in views and templates

Using a formset inside a view is as easy as using a regular `Form` class. The only thing you will want to be aware of is making sure to use the management form inside the template. Let's look at a sample view:

```
def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    if request.method == 'POST':
        formset = ArticleFormSet(request.POST, request.FILES)
        if formset.is_valid():
            # do something with the formset.cleaned_data
        else:
            formset = ArticleFormSet()
    return render_to_response('manage_articles.html', {'formset': formset})
```

The `manage_articles.html` template might look like this:

```
<form method="POST" action="">
    {{ formset.management_form }}
    <table>
        {% for form in formset.forms %}
            {{ form }}
        {% endfor %}
    </table>
</form>
```

However the above can be slightly shortcutted and let the formset itself deal with the management form:

```
<form method="POST" action="">
    <table>
        {{ formset }}
    </table>
</form>
```

The above ends up calling the `as_table` method on the formset class.

Using more than one formset in a view

You are able to use more than one formset in a view if you like. Formsets borrow much of its behavior from forms. With that said you are able to use `prefix` to prefix formset form field names with a given value to allow more than one formset to be sent to a view without name clashing. Lets take a look at how this might be accomplished:

```
def manage_articles(request):
    ArticleFormSet = formset_factory(ArticleForm)
    BookFormSet = formset_factory(BookForm)
    if request.method == 'POST':
        article_formset = ArticleFormSet(request.POST, request.FILES, prefix=
        ↪'articles')
        book_formset = BookFormSet(request.POST, request.FILES, prefix='books')
        if article_formset.is_valid() and book_formset.is_valid():
            # do something with the cleaned_data on the formsets.
    else:
        article_formset = ArticleFormSet(prefix='articles')
        book_formset = BookFormSet(prefix='books')
    return render_to_response('manage_articles.html', {
        'article_formset': article_formset,
        'book_formset': book_formset,
    })
```

You would then render the formsets as normal. It is important to point out that you need to pass `prefix` on both the POST and non-POST cases so that it is rendered and processed correctly.

Form Media

Renderizar um formulário atrativo e fácil de usar requer mais do que só HTML - ele requer CSS, e se você quiser usar widgets de “Web2.0”, você poderá também precisar de algum JavaScript em cada página. A combinação exata de CSS e JavaScript requerida por qualquer página depende dos widgets que você estiver usando.

É aí que as definições de media do Django entram. O Django permite você associar diferentes arquivos de media com os formulários e widgets que necessitam delas. Por exemplo, se você quiser usar um renderizador de calendários para `DateFields`, você pode definir um widget `Calendar` customizado. Este widget pode então ser associado ao CSS e JavaScript que é requerido para renderizar o calendário. Quando o widget `Calendar` for usado num formulário, o Django estará pronto para identificar os arquivos CSS e JavaScript necessários, e fornecer a lista de nomes de arquivos num formato adequado para fácil inclusão em sua página Web.

Media e o Django Admin

A aplicação Django Admin define vários widgets customizados para calendários, seleções filtradas, e assim por diante. Estes widgets definem requerimentos de media, e o Django Admin usa os widgets customizados no lugar dos padrão do Django. Os templates do Admin somente incluirão estes arquivos de media que são necessários para renderizar os widgets em qualquer página.

Se você gosta dos widgets que a aplicação Django Admin utiliza, sintá-se livre para usá-los em suas aplicações! Elas estão todas armazenadas em `django.contrib.admin.widgets`.

Qual toolkit JavaScript?

Existem muitos toolkits JavaScript, e muitos deles incluem widgets (como os widgets de calendário) que podem ser utilizado para melhorar sua aplicação. O Django deliberadamente evitar apologias a qualquer toolkit Javascript. Cada toolkit tem seus próprios pontos fortes e fracos - use qualquer toolkit que atenda suas necessidades. O Django é hábil em integrar-se com qualquer toolkit JavaScript.

Media como uma definição estática

A forma mais fácil de definir media é como uma definição estática. Usando este método, a declaração media fica numa classe interna. As propriedades dessa classe interna definem os requerimentos de media.

Aqui tem um exemplo simples:

```
class CalendarWidget (forms.TextInput):
    class Media:
        css = {
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')
```

Este código define um `CalendarWidget`, que será baseado no `TextInput`. Toda vez que o `CalendarWidget` for usado num formulário, este formulário incluirá diretamente o arquivo de CSS `pretty.css`, e os arquivos de JavaScript `animations.js` e `actions.js`.

Esta definição estática de media é convertida, em tempo de execução, numa propriedade chamada `media`. As medias para uma instância do `CalendarWidget` podem ser acessadas através desta propriedade:

```
>>> w = CalendarWidget()
>>> print w.media
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel=
↪"stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></
↪script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
```

Aqui tem uma lista de todas as opções possíveis de `Media`. Não há opções obrigatórias.

CSS

Um dicionário descrevendo os arquivos CSS requeridos para as diversas formas de saídas de media.

Os valores no dicionário devem ser uma tupla/lista de nomes de arquivos. Veja a [seção sobre caminhos de media](#) para mais detalhes de como especificar os caminhos dos arquivos de media. As chaves no dicionário são os tipos de saídas de media. Este são os mesmos tipos aceitos por arquivos CSS em declarações de media: 'all', 'aural', 'braille', 'embossed', 'handheld', 'print', 'projection', 'screen', 'tty' e 'tv'. Se você precisa ter diferentes folhas de estilo para tipos diferentes de media, forneça a lista de arquivos CSS para cada ambiente de saída. o exemplo a seguir provê duas opções de CSS – um para screen, e um para print:

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'print': ('newspaper.css',)
    }
```

Se um grupo de arquivos CSS são adequados para múltiplos tipos de saídas, a chave do dicionário pode ser uma lista separada por vírgula dos tipos de media. No exemplo a seguir, TV e projectors terão o mesmo requerimento de media:

```
class Media:
    css = {
        'screen': ('pretty.css',),
        'tv,projector': ('lo_res.css',),
        'print': ('newspaper.css',)
    }
```

Se esta última definição de CSS fosse renderizada, ela poderia tornar-se o seguinte HTML:

```
<link href="http://media.example.com/pretty.css" type="text/css" media="screen"
↪rel="stylesheet" />
<link href="http://media.example.com/lo_res.css" type="text/css" media="tv,
↪projector" rel="stylesheet" />
<link href="http://media.example.com/newspaper.css" type="text/css" media="print"
↪rel="stylesheet" />
```

js

Uma tupla descrevendo os arquivos JavaScript requeridos. Veja a [seção sobre caminhos de media](#) para mais detalhes de como especificar os caminhos para os arquivos de media.

extend

Um booleano definindo comportamento de herança para declarações de media.

Por padrão, qualquer objeto usando uma definição estática de media herdará todas as medias associadas do widget pai. Isto ocorre indiferente de como o widget pai define seus requerimentos de media. Por exemplo, se nós formos estender nosso widget básico Calendar, do exemplo acima:

```
class FancyCalendarWidget(CalendarWidget):
    class Media:
        css = {
            'all': ('fancy.css',)
        }
        js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print w.media
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel=
↪"stylesheet" />
<link href="http://media.example.com/fancy.css" type="text/css" media="all" rel=
↪"stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></
↪script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

O widget FancyCalendarWidget herda todas as medias de seu widget pai. Se você não quer que o media seja herdado desta forma, adicione uma declaração `extend=False` para a declaração media:


```
class FancyCalendarWidget(CalendarWidget):
    class Media:
        extend = False
        css = {
            'all': ('fancy.css',)
        }
        js = ('whizbang.js',)

>>> w = FancyCalendarWidget()
>>> print w.media
<link href="http://media.example.com/fancy.css" type="text/css" media="all" rel=
↪"stylesheet" />
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

Se você necessita de ainda mais controle sobre a herança de media, defina sua media utilizando uma *propriedade dinâmica*. Propriedades dinâmicas dão a você controle completo sobre quais arquivos de media são herdados, e quais não são.

Media como uma propriedade dinâmica

Se você necessita realizar alguma manipulação mais sofisticada nos requerimentos de media, você pode definir a propriedade media diretamente. Isto é feito definindo uma propriedade do model que retorna uma instância de `forms.Media`. O construtor de `forms.Media` aceita os argumentos nomeados `css` e `js` no mesmo formato que é usado na definição estática do media.

Por exemplo, a definição estática de media para nosso Widget Calendar poderia também ser definido de forma dinâmica:

```
class CalendarWidget(forms.TextInput):
    def _media(self):
        return forms.Media(css={'all': ('pretty.css',)},
                           js=('animations.js', 'actions.js'))
    media = property(_media)
```

Veja a seção sobre *Objetos Media* para mais detalhes sobre como construir retornos de valores para propriedades dinâmicas de media.

Caminhos em definições de media

Caminhos usados para especificar media podem ser ambos relativos e absolutos. Se um caminho começa com `'/'`, `'http://'` ou `'https://'`, ele será interpretado como um caminho absoluto, e deixado como está. Todos os outros caminhos serão prefixados com valor de `settings.MEDIA_URL`. Por exemplo, se o `MEDIA_URL` para seu site for `http://media.example.com/`:

```
class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('/css/pretty.css',),
        }
        js = ('animations.js', 'http://othersite.com/actions.js')

>>> w = CalendarWidget()
>>> print w.media
<link href="/css/pretty.css" type="text/css" media="all" rel="stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></
↪script>
<script type="text/javascript" src="http://othersite.com/actions.js"></script>
```

Objetos Media

Quando você interroga o atributo `media` de um widget ou formulário, o valor que é retornado é um objeto `forms.Media`. Como nós já temos visto, a representação em string de um objeto `Media` é um HTML necessário para incluir a media no bloco `<head>` de sua página HTML.

No entanto, objetos `Media` tem algumas outras propriedades interessantes.

Media subsets

Se você somente quer media de um tipo particular, você pode usar o operador subscript para filtrar a media que interessa. Por exemplo:

```
>>> w = CalendarWidget()
>>> print w.media
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel=
↪"stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></
↪script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>

>>> print w.media['css']
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel=
↪"stylesheet" />
```

Quando você usa um operador subscript, o valor que é retornado é um novo objeto `Media` – mas um que contém somente as medias que interessa.

Combinando objetos media

Objetos `Media` pode também ser adicionados juntos. Quando dois objetos media são adicionados o objeto resultante contém a união de media de ambos os arquivos:

```
class CalendarWidget(forms.TextInput):
    class Media:
        css = {
            'all': ('pretty.css',)
        }
        js = ('animations.js', 'actions.js')

class OtherWidget(forms.TextInput):
    class Media:
        js = ('whizbang.js',)

>>> w1 = CalendarWidget()
>>> w2 = OtherWidget()
>>> print w1.media + w2.media
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel=
↪"stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></
↪script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

Media nos Forms

Regardless of whether you define a media declaration, *all* Form objects have a `media` property. The default value for this property is the result of adding the media definitions for all widgets that are part of the form:: Indiferente

de você definir uma declaração `media`, *todos* os objetos `Form` possuem uma propriedade `media`. O valor padrão para esta propriedade é o resultado da soma das definições de `media` de todos os widgets que fazem parte do formulário:

```
class ContactForm(forms.Form):
    date = DateField(widget=CalendarWidget)
    name = CharField(max_length=40, widget=OtherWidget)

>>> f = ContactForm()
>>> f.media
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel=
↪"stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></
↪script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

Se você quiser associar `media` adicional ao formulário – por exemplo, CSS para o layout do formulário – simplesmente adicione uma declaração `media` para no formulário:

```
class ContactForm(forms.Form):
    date = DateField(widget=CalendarWidget)
    name = CharField(max_length=40, widget=OtherWidget)

    class Media:
        css = {
            'all': ('layout.css',)
        }

>>> f = ContactForm()
>>> f.media
<link href="http://media.example.com/pretty.css" type="text/css" media="all" rel=
↪"stylesheet" />
<link href="http://media.example.com/layout.css" type="text/css" media="all" rel=
↪"stylesheet" />
<script type="text/javascript" src="http://media.example.com/animations.js"></
↪script>
<script type="text/javascript" src="http://media.example.com/actions.js"></script>
<script type="text/javascript" src="http://media.example.com/whizbang.js"></script>
```

See also:

A referência de API do form.

CHAPTER 12

The Django template language

About this document

This document explains the language syntax of the Django template system. If you're looking for a more technical perspective on how it works and how to extend it, see *The Django template language: For Python programmers*.

Django's template language is designed to strike a balance between power and ease. It's designed to feel comfortable to those used to working with HTML. If you have any exposure to other text-based template languages, such as *Smarty* or *CheetahTemplate*, you should feel right at home with Django's templates.

Philosophy

If you have a background in programming, or if you're used to languages like PHP which mix programming code directly into HTML, you'll want to bear in mind that the Django template system is not simply Python embedded into HTML. This is by design: the template system is meant to express presentation, not program logic.

The Django template system provides tags which function similarly to some programming constructs – an *if* tag for boolean tests, a *for* tag for looping, etc. – but these are not simply executed as the corresponding Python code, and the template system will not execute arbitrary Python expressions. Only the tags, filters and syntax listed below are supported by default (although you can add *your own extensions* to the template language as needed).

Templates

A template is simply a text file. It can generate any text-based format (HTML, XML, CSV, etc.).

A template contains **variables**, which get replaced with values when the template is evaluated, and **tags**, which control the logic of the template.

Below is a minimal template that illustrates a few basics. Each element will be explained later in this document.:

```
{% extends "base_generic.html" %}

{% block title %}{{ section.title }}{% endblock %}

{% block content %}
<h1>{{ section.title }}</h1>
```

```
{% for story in story_list %}
<h2>
  <a href="{{ story.get_absolute_url }}">
    {{ story.headline|upper }}
  </a>
</h2>
<p>{{ story.tease|truncatewords:"100" }}</p>
{% endfor %}
{% endblock %}
```

Philosophy

Why use a text-based template instead of an XML-based one (like Zope’s TAL)? We wanted Django’s template language to be usable for more than just XML/HTML templates. At World Online, we use it for e-mails, JavaScript and CSV. You can use the template language for any text-based format.

Oh, and one more thing: Making humans edit XML is sadistic!

Variables

Variables look like this: `{{ variable }}`. When the template engine encounters a variable, it evaluates that variable and replaces it with the result.

Use a dot (.) to access attributes of a variable.

Behind the scenes

Technically, when the template system encounters a dot, it tries the following lookups, in this order:

- Dictionary lookup
 - Attribute lookup
 - Method call
 - List-index lookup
-

In the above example, `{{ section.title }}` will be replaced with the `title` attribute of the `section` object.

If you use a variable that doesn’t exist, the template system will insert the value of the `TEMPLATE_STRING_IF_INVALID` setting, which is set to `' '` (the empty string) by default.

See *Using the built-in reference*, below, for help on finding what variables are available in a given template.

Filters

You can modify variables for display by using **filters**.

Filters look like this: `{{ name|lower }}`. This displays the value of the `{{ name }}` variable after being filtered through the `lower` filter, which converts text to lowercase. Use a pipe (|) to apply a filter.

Filters can be “chained.” The output of one filter is applied to the next. `{{ text|escape|linebreaks }}` is a common idiom for escaping text contents, then converting line breaks to `<p>` tags.

Some filters take arguments. A filter argument looks like this: `{{ bio|truncatewords:30 }}`. This will display the first 30 words of the `bio` variable.

Filter arguments that contain spaces must be quoted; for example, to join a list with commas and spaced you'd use `{{ list|join:", " }}`.

Django provides about thirty built-in template filters. You can read all about them in the [built-in filter reference](#). To give you a taste of what's available, here are some of the more commonly used template filters:

default If a variable is false or empty, use given default. Otherwise, use the value of the variable

For example:

```
{{ value|default:"nothing" }}
```

If value isn't provided or is empty, the above will display "nothing".

length Returns the length of the value. This works for both strings and lists; for example:

```
{{ value|length }}
```

If value is `['a', 'b', 'c', 'd']`, the output will be 4.

striptags Strips all [X]HTML tags. For example:

```
{{ value|striptags }}
```

If value is `"Joel <button>is</button> a slug"`, the output will be `"Joel is a slug"`.

Again, these are just a few examples; see the [built-in filter reference](#) for the complete list.

You can also create your own custom template filters; see [Tags e filtros de template personalizados](#).

Tags

Tags look like this: `{% tag %}`. Tags are more complex than variables: Some create text in the output, some control flow by performing loops or logic, and some load external information into the template to be used by later variables.

Some tags require beginning and ending tags (i.e. `{% tag %} ... tag contents ... {% endtag %}`).

Django ships with about two dozen built-in template tags. You can read all about them in the [built-in tag reference](#). To give you a taste of what's available, here are some of the more commonly used tags:

for Loop over each item in an array. For example, to display a list of athletes provided in `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

if and else Evaluates a variable, and if that variable is "true" the contents of the block are displayed:

```
{% if athlete_list %}
    Number of athletes: {{ athlete_list|length }}
{% else %}
    No athletes.
{% endif %}
```

In the above, if `athlete_list` is not empty, the number of athletes will be displayed by the `{{ athlete_list|length }}` variable.

ifequal and ifnotequal Display some contents if two arguments are or are not equal. For example:

```
{% ifequal athlete.name coach.name %}
...
{% endifequal %}
```

Or:

```
{% ifnotequal athlete.name "Joe" %}
...
{% endifnotequal %}
```

block and **extends** Set up *template inheritance* (see below), a powerful way of cutting down on “boilerplate” in templates.

Again, the above is only a selection of the whole list; see the *built-in tag reference* for the complete list.

You can also create your own custom template tags; see *Tags e filtros de template personalizados*.

Comments

To comment-out part of a line in a template, use the comment syntax: `{# #}`.

For example, this template would render as 'hello':

```
{# greeting #}hello
```

A comment can contain any template code, invalid or not. For example:

```
{# {% if foo %}bar{% else %} #}
```

This syntax can only be used for single-line comments (no newlines are permitted between the `{#` and `#}` delimiters). If you need to comment out a multiline portion of the template, see the *comment* tag.

Template inheritance

The most powerful – and thus the most complex – part of Django’s template engine is template inheritance. Template inheritance allows you to build a base “skeleton” template that contains all the common elements of your site and defines **blocks** that child templates can override.

It’s easiest to understand template inheritance by starting with an example:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
            <ul>
                <li><a href="/">Home</a></li>
                <li><a href="/blog/">Blog</a></li>
            </ul>
        {% endblock %}
    </div>

    <div id="content">
```

```

        {% block content %}{% endblock %}
    </div>
</body>
</html>

```

This template, which we’ll call `base.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It’s the job of “child” templates to fill the empty blocks with content.

In this example, the `{% block %}` tag defines three blocks that child templates can fill in. All the `block` tag does is to tell the template engine that a child template may override those portions of the template.

A child template might look like this:

```

{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}

```

The `{% extends %}` tag is the key here. It tells the template engine that this template “extends” another template. When the template system evaluates this template, first it locates the parent – in this case, “`base.html`”.

At that point, the template engine will notice the three `{% block %}` tags in `base.html` and replace those blocks with the contents of the child template. Depending on the value of `blog_entries`, the output might look like:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>My amazing blog</title>
</head>

<body>
    <div id="sidebar">
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
    </div>

    <div id="content">
        <h2>Entry one</h2>
        <p>This is my first entry.</p>

        <h2>Entry two</h2>
        <p>This is my second entry.</p>
    </div>
</body>
</html>

```

Note that since the child template didn’t define the `sidebar` block, the value from the parent template is used instead. Content within a `{% block %}` tag in a parent template is always used as a fallback.

You can use as many levels of inheritance as needed. One common way of using inheritance is the following three-level approach:

- Create a `base.html` template that holds the main look-and-feel of your site.

- Create a `base_SECTIONNAME.html` template for each “section” of your site. For example, `base_news.html`, `base_sports.html`. These templates all extend `base.html` and include section-specific styles/design.
- Create individual templates for each type of page, such as a news article or blog entry. These templates extend the appropriate section template.

This approach maximizes code reuse and makes it easy to add items to shared content areas, such as section-wide navigation.

Here are some tips for working with inheritance:

- If you use `{% extends %}` in a template, it must be the first template tag in that template. Template inheritance won’t work, otherwise.
- More `{% block %}` tags in your base templates are better. Remember, child templates don’t have to define all parent blocks, so you can fill in reasonable defaults in a number of blocks, then only define the ones you need later. It’s better to have more hooks than fewer hooks.
- If you find yourself duplicating content in a number of templates, it probably means you should move that content to a `{% block %}` in a parent template.
- If you need to get the content of the block from the parent template, the `{{ block.super }}` variable will do the trick. This is useful if you want to add to the contents of a parent block instead of completely overriding it. Data inserted using `{{ block.super }}` will not be automatically escaped (see the [next section](#)), since it was already escaped, if necessary, in the parent template.
- For extra readability, you can optionally give a *name* to your `{% endblock %}` tag. For example:

```
{% block content %}
...
{% endblock content %}
```

In larger templates, this technique helps you see which `{% block %}` tags are being closed.

Finally, note that you can’t define multiple `{% block %}` tags with the same name in the same template. This limitation exists because a block tag works in “both” directions. That is, a block tag doesn’t just provide a hole to fill – it also defines the content that fills the hole in the *parent*. If there were two similarly-named `{% block %}` tags in a template, that template’s parent wouldn’t know which one of the blocks’ content to use.

Automatic HTML escaping

Please, see the release notes When generating HTML from templates, there’s always a risk that a variable will include characters that affect the resulting HTML. For example, consider this template fragment:

```
Hello, {{ name }}.
```

At first, this seems like a harmless way to display a user’s name, but consider what would happen if the user entered his name as this:

```
<script>alert('hello')</script>
```

With this name value, the template would be rendered as:

```
Hello, <script>alert('hello')</script>
```

...which means the browser would pop-up a JavaScript alert box!

Similarly, what if the name contained a `'<'` symbol, like this?

```
<b>username
```

That would result in a rendered template like this:

```
Hello, <b>username
```

...which, in turn, would result in the remainder of the Web page being bolded!

Clearly, user-submitted data shouldn't be trusted blindly and inserted directly into your Web pages, because a malicious user could use this kind of hole to do potentially bad things. This type of security exploit is called a [Cross Site Scripting \(XSS\)](#) attack.

To avoid this problem, you have two options:

- One, you can make sure to run each untrusted variable through the `escape` filter (documented below), which converts potentially harmful HTML characters to unharmed ones. This was the default solution in Django for its first few years, but the problem is that it puts the onus on *you*, the developer / template author, to ensure you're escaping everything. It's easy to forget to escape data.
- Two, you can take advantage of Django's automatic HTML escaping. The remainder of this section describes how auto-escaping works.

By default in Django, every template automatically escapes the output of every variable tag. Specifically, these five characters are escaped:

- `<` is converted to `<`;
- `>` is converted to `>`;
- `'` (single quote) is converted to `'`;
- `"` (double quote) is converted to `"`;
- `&` is converted to `&`;

Again, we stress that this behavior is on by default. If you're using Django's template system, you're protected.

How to turn it off

If you don't want data to be auto-escaped, on a per-site, per-template level or per-variable level, you can turn it off in several ways.

Why would you want to turn it off? Because sometimes, template variables contain data that you *intend* to be rendered as raw HTML, in which case you don't want their contents to be escaped. For example, you might store a blob of HTML in your database and want to embed that directly into your template. Or, you might be using Django's template system to produce text that is *not* HTML – like an e-mail message, for instance.

For individual variables

To disable auto-escaping for an individual variable, use the `safe` filter:

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

Think of *safe* as shorthand for *safe from further escaping* or *can be safely interpreted as HTML*. In this example, if `data` contains `''`, the output will be:

```
This will be escaped: &lt;b&gt;
This will not be escaped: <b>
```

For template blocks

To control auto-escaping for a template, wrap the template (or just a particular section of the template) in the `autoescape` tag, like so:

```
{% autoescape off %}
    Hello {{ name }}
{% endautoescape %}
```

The `autoescape` tag takes either `on` or `off` as its argument. At times, you might want to force auto-escaping when it would otherwise be disabled. Here is an example template:

```
Auto-escaping is on by default. Hello {{ name }}

{% autoescape off %}
    This will not be auto-escaped: {{ data }}.

    Nor this: {{ other_data }}
{% autoescape on %}
    Auto-escaping applies again: {{ name }}
{% endautoescape %}
```

The auto-escaping tag passes its effect onto templates that extend the current one as well as templates included via the `include` tag, just like all block tags. For example:

```
# base.html

{% autoescape off %}
<h1>{% block title %}{% endblock %}</h1>
{% block content %}
{% endblock %}
{% endautoescape %}

# child.html

{% extends "base.html" %}
{% block title %}This & that{% endblock %}
{% block content %}{{ greeting }}{% endblock %}
```

Because auto-escaping is turned off in the base template, it will also be turned off in the child template, resulting in the following rendered HTML when the `greeting` variable contains the string `Hello!`:

```
<h1>This & that</h1>
<b>Hello!</b>
```

Notes

Generally, template authors don't need to worry about auto-escaping very much. Developers on the Python side (people writing views and custom filters) need to think about the cases in which data shouldn't be escaped, and mark data appropriately, so things Just Work in the template.

If you're creating a template that might be used in situations where you're not sure whether auto-escaping is enabled, then add an escape filter to any variable that needs escaping. When auto-escaping is on, there's no danger of the escape filter *double-escaping* data – the escape filter does not affect auto-escaped variables.

String literals and automatic escaping

As we mentioned earlier, filter arguments can be strings:

```
{{ data|default:"This is a string literal." }}
```

All string literals are inserted **without** any automatic escaping into the template – they act as if they were all passed through the `safe` filter. The reasoning behind this is that the template author is in control of what goes into the string literal, so they can make sure the text is correctly escaped when the template is written.

This means you would write

```
{{ data|default:"3 &lt; 2" }}
```

...rather than

```
{{ data|default:"3 < 2" }} <-- Bad! Don't do this.
```

This doesn't affect what happens to data coming from the variable itself. The variable's contents are still automatically escaped, if necessary, because they're beyond the control of the template author.

Using the built-in reference

Django's admin interface includes a complete reference of all template tags and filters available for a given site. To see it, go to your admin interface and click the "Documentation" link in the upper right of the page.

The reference is divided into 4 sections: tags, filters, models, and views.

The **tags** and **filters** sections describe all the built-in tags (in fact, the tag and filter references below come directly from those pages) as well as any custom tag or filter libraries available.

The **views** page is the most valuable. Each URL in your site has a separate entry here, and clicking on a URL will show you:

- The name of the view function that generates that view.
- A short description of what the view does.
- The **context**, or a list of variables available in the view's template.
- The name of the template or templates that are used for that view.

Each view documentation page also has a bookmarklet that you can use to jump from any page to the documentation page for that view.

Because Django-powered sites usually use database objects, the **models** section of the documentation page describes each type of object in the system along with all the fields available on that object.

Taken together, the documentation pages should tell you every tag, filter, variable and object available to you in a given template.

Custom tag and filter libraries

Certain applications provide custom tag and filter libraries. To access them in a template, use the `{% load %}` tag:

```
{% load comments %}

{% comment_form for blogs.entries entry.id with is_public yes %}
```

In the above, the `load` tag loads the `comments` tag library, which then makes the `comment_form` tag available for use. Consult the documentation area in your admin to find the list of custom libraries in your installation.

The `{% load %}` tag can take multiple library names, separated by spaces. Example:

```
{% load comments i18n %}
```

See *Tags e filtros de template personalizados* for information on writing your own custom template libraries.

Custom libraries and template inheritance

When you load a custom tag or filter library, the tags/filters are only made available to the current template – not any parent or child templates along the template-inheritance path.

For example, if a template `foo.html` has `{% load comments %}`, a child template (e.g., one that has `{% extends "foo.html" %}`) will *not* have access to the `comments` template tags and filters. The child template is responsible for its own `{% load comments %}`.

This is a feature for the sake of maintainability and sanity.

Views genéricas

Escrever aplicações web pode ser monótono, porque nós repetimos certos padrões várias e várias vezes. O Django tenta tirar um pouco dessa monotonia nas camadas de model e template, mas os desenvolvedores web também experimentam esse tédio na camada da view.

As *views genéricas* do Django foram criadas para diminuir esse sofrimento. Elas pegam padrões comuns encontrados no desenvolvimento web e abstraem eles, assim você pode escrever rapidamente views comuns sem ter que escrever muito código.

Podemos identificar algumas tarefas comuns, como mostrar uma lista de objetos, e escrever código que mostre um lista de *qualquer* objeto. Então, o model em questão pode ser passado como um argumento extra ao URLconf.

O Django vem com views genéricas que fazem o seguinte:

- Efetuam “simples” tarefas comuns: redirecionar para uma página diferente e renderizar determinado template.
- Mostrar uma lista e página com detalhes de um único objeto. Se estamos criando uma aplicação para gerenciar conferências, então uma view `talk_list` e uma view `registered_user_list` seriam exemplo de views de listas. Uma única página de conversa seria o que chamamos de view de “detalhe”.
- Apresentar objetos baseados em data em páginas categorizadas por ano/mês/dia, os detalhes associados, e as “últimas” páginas. Os arquivos por ano, mês e dia do Blog da Django Brasil (<http://www.djangobrasil.org/weblog/>) foram construídos com nessas views, como exemplo de um típico arquivo de jornal.
- Permitir usuários a criar, atualizar e deletar objetos – com ou sem autorização.

Juntas, essas views oferecem uma interface fácil para realizar as tarefas mais comuns que os desenvolvedores encontram.

Usando views genéricas

Todas essas views são usadas através da criação de dicionários de configurações nos seus arquivos URLconf e passando esses dicionários como o terceiro membro da tupla URLconf para um determinado padrão.

Por exemplo, aqui está uma simples URLconf que poderíamos usar para apresentar uma página estática “sobre”:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template
```

```
urlpatterns = patterns('',
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    })
)
```

Isso pode parecer um pouco “mágico” à primeira vista – olhe, uma view sem código! –, atualmente a view `direct_to_template` simplesmente pega a informação do dicionário de parâmetros extras e usa essa informação ao renderizar a view.

Como essa view genérica – e todas as outras – é uma view normal como qualquer outra, podemos reusá-la dentro de nossas próprias views. Como exemplo, vamos estender nossa página “sobre” para mapear as URLs na forma `/about/<qualquer coisa>/` para arquivos `about/<qualquer coisa>.html`. Nós faremos isso modificando primeiramente a URLconf, para apontar para uma função de view:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template
from mysite.books.views import about_pages

urlpatterns = patterns('',
    ('^about/$', direct_to_template, {
        'template': 'about.html'
    }),
    ('^about/(w+)/$', about_pages),
)
```

Depois, vamos escrever a view `about_pages`:

```
from django.http import Http404
from django.template import TemplateDoesNotExist
from django.views.generic.simple import direct_to_template

def about_pages(request, page):
    try:
        return direct_to_template(request, template="about/%s.html" % page)
    except TemplateDoesNotExist:
        raise Http404()
```

Aqui tratamos o `direct_to_template` como qualquer outra função. Uma vez que ela retorna um `HttpResponse`, podemos fazer como o exemplo. O único pequeno truque aqui é lidar com a falta de templates. Não queremos que um template inexistente cause um erro do servidor, então, capturamos as exceções `TemplateDoesNotExist` e retornamos um erro 404.

Existe uma falha de segurança aqui?

Os leitores mais atentos podem ter percebido uma possível falha de segurança: estamos contruindo o nome do template usando conteúdo do navegador (`template="about/%s.html" % page`). A primeira vista, isso parece uma clássica vulnerabilidade *directory traversal*. Mas realmente é?

Não exatamente. Sim, um valor malicioso para `page` poderia causar *directory traversal*, porém, a variável `page` é advinda da URL, e não é qualquer valor que será aceito. A chave para isso está na URLconf: estamos usando a expressão regular `\w+` para obter parte da URL, e `\w` aceita somente letras e números. Portanto, qualquer caracter malicioso (pontos e barras, aqui) serão rejeitados pelo resolvidor de URL antes de chegar à view.

Views genéricas de objetos

The `direct_to_template` certainly is useful, but Django’s generic views really shine when it comes to presenting views on your database content. Because it’s such a common task, Django comes with a handful of

built-in generic views that make generating list and detail views of objects incredibly easy.

Let’s take a look at one of these generic views: the “object list” view. We’ll be using these models:

```
# models.py
from django.db import models

class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    city = models.CharField(max_length=60)
    state_province = models.CharField(max_length=30)
    country = models.CharField(max_length=50)
    website = models.URLField()

    def __unicode__(self):
        return self.name

    class Meta:
        ordering = ["-name"]

class Book(models.Model):
    title = models.CharField(max_length=100)
    authors = models.ManyToManyField('Author')
    publisher = models.ForeignKey(Publisher)
    publication_date = models.DateField()
```

To build a list page of all books, we’d use a URLconf along these lines:

```
from django.conf.urls.defaults import *
from django.views.generic import list_detail
from mysite.books.models import Publisher

publisher_info = {
    "queryset" : Publisher.objects.all(),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

That’s all the Python code we need to write. We still need to write a template, however. We could explicitly tell the `object_list` view which template to use by including a `template_name` key in the extra arguments dictionary, but in the absence of an explicit template Django will infer one from the object’s name. In this case, the inferred template will be `books/publisher_list.html` – the “books” part comes from the name of the app that defines the model, while the “publisher” bit is just the lowercased version of the model’s name.

This template will be rendered against a context containing a variable called `object_list` that contains all the book objects. A very simple template might look like the following:

```
{% extends "base.html" %}

{% block content %}
<h2>Publishers</h2>
<ul>
    {% for publisher in object_list %}
        <li>{{ publisher.name }}</li>
    {% endfor %}
</ul>
{% endblock %}
```

That’s really all there is to it. All the cool features of generic views come from changing the “info” dictionary passed to the generic view. The [generic views reference](#) documents all the generic views and all their options in

detail; the rest of this document will consider some of the common ways you might customize and extend generic views.

Extending generic views

There's no question that using generic views can speed up development substantially. In most projects, however, there comes a moment when the generic views no longer suffice. Indeed, the most common question asked by new Django developers is how to make generic views handle a wider array of situations.

Luckily, in nearly every one of these cases, there are ways to simply extend generic views to handle a larger array of use cases. These situations usually fall into a handful of patterns dealt with in the sections that follow.

Making “friendly” template contexts

You might have noticed that our sample publisher list template stores all the books in a variable named `object_list`. While this works just fine, it isn't all that “friendly” to template authors: they have to “just know” that they're dealing with books here. A better name for that variable would be `publisher_list`; that variable's content is pretty obvious.

We can change the name of that variable easily with the `template_object_name` argument:

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info)
)
```

Providing a useful `template_object_name` is always a good idea. Your coworkers who design templates will thank you.

Adding extra context

Often you simply need to present some extra information beyond that provided by the generic view. For example, think of showing a list of all the other publishers on each publisher detail page. The `object_detail` generic view provides the publisher to the context, but it seems there's no way to get a list of *all* publishers in that template.

But there is: all generic views take an extra optional parameter, `extra_context`. This is a dictionary of extra objects that will be added to the template's context. So, to provide the list of all publishers on the detail view, we'd use an info dict like this:

```
from mysite.books.models import Publisher, Book

publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : {"book_list" : Book.objects.all()}
}
```

This would populate a `{{ book_list }}` variable in the template context. This pattern can be used to pass any information down into the template for the generic view. It's very handy.

However, there's actually a subtle bug here – can you spot it?

The problem has to do with when the queries in `extra_context` are evaluated. Because this example puts `Publisher.objects.all()` in the `URLconf`, it will be evaluated only once (when the `URLconf` is first loaded). Once you add or remove publishers, you'll notice that the generic view doesn't reflect those changes until

you reload the Web server (see [Cacheamento e QuerySets](#) for more information about when QuerySets are cached and evaluated).

Note: This problem doesn't apply to the `queryset` generic view argument. Since Django knows that particular QuerySet should *never* be cached, the generic view takes care of clearing the cache when each view is rendered.

The solution is to use a callback in `extra_context` instead of a value. Any callable (i.e., a function) that's passed to `extra_context` will be evaluated when the view is rendered (instead of only once). You could do this with an explicitly defined function:

```
def get_books():
    return Book.objects.all()

publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : {"book_list" : get_books}
}
```

or you could use a less obvious but shorter version that relies on the fact that `Book.objects.all` is itself a callable:

```
publisher_info = {
    "queryset" : Publisher.objects.all(),
    "template_object_name" : "publisher",
    "extra_context" : {"book_list" : Book.objects.all}
}
```

Notice the lack of parentheses after `Book.objects.all`; this references the function without actually calling it (which the generic view will do later).

Viewing subsets of objects

Now let's take a closer look at this `queryset` key we've been using all along. Most generic views take one of these `queryset` arguments – it's how the view knows which set of objects to display (see [Fazendo consultas](#) for more information about QuerySet objects, and see the [generic views reference](#) for the complete details).

To pick a simple example, we might want to order a list of books by publication date, with the most recent first:

```
book_info = {
    "queryset" : Book.objects.all().order_by("-publication_date"),
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/$', list_detail.object_list, book_info),
)
```

That's a pretty simple example, but it illustrates the idea nicely. Of course, you'll usually want to do more than just reorder objects. If you want to present a list of books by a particular publisher, you can use the same technique:

```
acme_books = {
    "queryset": Book.objects.filter(publisher__name="Acme Publishing"),
    "template_name" : "books/acme_list.html"
}

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/acme/$', list_detail.object_list, acme_books),
)
```

Notice that along with a filtered `queryset`, we’re also using a custom template name. If we didn’t, the generic view would use the same template as the “vanilla” object list, which might not be what we want.

Also notice that this isn’t a very elegant way of doing publisher-specific books. If we want to add another publisher page, we’d need another handful of lines in the `URLconf`, and more than a few publishers would get unreasonable. We’ll deal with this problem in the next section.

Note: If you get a 404 when requesting `/books/acme/`, check to ensure you actually have a `Publisher` with the name ‘ACME Publishing’. Generic views have an `allow_empty` parameter for this case. See the [generic views reference](#) for more details.

Complex filtering with wrapper functions

Another common need is to filter down the objects given in a list page by some key in the URL. Earlier we hard-coded the publisher’s name in the `URLconf`, but what if we wanted to write a view that displayed all the books by some arbitrary publisher? We can “wrap” the `object_list` generic view to avoid writing a lot of code by hand. As usual, we’ll start by writing a `URLconf`:

```
from mysite.books.views import books_by_publisher

urlpatterns = patterns('',
    (r'^publishers/$', list_detail.object_list, publisher_info),
    (r'^books/(w+)/$', books_by_publisher),
)
```

Next, we’ll write the `books_by_publisher` view itself:

```
from django.http import Http404
from django.views.generic import list_detail
from mysite.books.models import Book, Publisher

def books_by_publisher(request, name):

    # Look up the publisher (and raise a 404 if it can't be found).
    try:
        publisher = Publisher.objects.get(name__iexact=name)
    except Publisher.DoesNotExist:
        raise Http404

    # Use the object_list view for the heavy lifting.
    return list_detail.object_list(
        request,
        queryset = Book.objects.filter(publisher=publisher),
        template_name = "books/books_by_publisher.html",
        template_object_name = "books",
        extra_context = {"publisher" : publisher}
    )
```

This works because there’s really nothing special about generic views – they’re just Python functions. Like any view function, generic views expect a certain set of arguments and return `HttpResponse` objects. Thus, it’s incredibly easy to wrap a small function around a generic view that does additional work before (or after; see the next section) handing things off to the generic view.

Note: Notice that in the preceding example we passed the current publisher being displayed in the `extra_context`. This is usually a good idea in wrappers of this nature; it lets the template know which “parent” object is currently being browsed.

Performing extra work

The last common pattern we'll look at involves doing some extra work before or after calling the generic view.

Imagine we had a `last_accessed` field on our `Author` object that we were using to keep track of the last time anybody looked at that author:

```
# models.py

class Author(models.Model):
    salutation = models.CharField(max_length=10)
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)
    email = models.EmailField()
    headshot = models.ImageField(upload_to='/tmp')
    last_accessed = models.DateTimeField()
```

The generic `object_detail` view, of course, wouldn't know anything about this field, but once again we could easily write a custom view to keep that field updated.

First, we'd need to add an author detail bit in the `URLconf` to point to a custom view:

```
from mysite.books.views import author_detail

urlpatterns = patterns('',
    #...
    (r'^authors/(?P<author_id>d+)/$', author_detail),
)
```

Then we'd write our wrapper function:

```
import datetime
from mysite.books.models import Author
from django.views.generic import list_detail
from django.shortcuts import get_object_or_404

def author_detail(request, author_id):
    # Look up the Author (and raise a 404 if she's not found)
    author = get_object_or_404(Author, pk=author_id)

    # Record the last accessed date
    author.last_accessed = datetime.datetime.now()
    author.save()

    # Show the detail page
    return list_detail.object_detail(
        request,
        queryset = Author.objects.all(),
        object_id = author_id,
    )
```

Note: This code won't actually work unless you create a `books/author_detail.html` template.

We can use a similar idiom to alter the response returned by the generic view. If we wanted to provide a downloadable plain-text version of the list of authors, we could use a view like this:

```
def author_list_plaintext(request):
    response = list_detail.object_list(
        request,
        queryset = Author.objects.all(),
        mimetype = "text/plain",
        template_name = "books/author_list.txt"
```

```
)  
response["Content-Disposition"] = "attachment; filename=authors.txt"  
return response
```

This works because the generic views return simple `HttpResponse` objects that can be treated like dictionaries to set HTTP headers. This `Content-Disposition` business, by the way, instructs the browser to download and save the page instead of displaying it in the browser.

Gerenciando arquivos

Please, see the release notes Este documento descreve a API de acesso a arquivos do Django.

Por padrão, o Django armazena arquivos localmente, usando as configurações `MEDIA_ROOT` e `MEDIA_URL`. O exemplos abaixo assumem que você está usando estes padrões.

No entanto, o Django provê formas de se criar um *sistema de armazenamento* customizado, que permite você personalizar completamente, onde e como o Django armazena arquivos. A segunda metade deste documento descreve como esses sistemas de armazenamento funcionam.

Usando arquivos em models

Quando você usa um `FileField` ou `ImageField`, o Django provê um conjunto de APIs que você pode usar para negociar com este arquivo.

Considere o seguinte model, usando um `ImageField` para armazenar uma foto:

```
class Car(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField(max_digits=5, decimal_places=2)
    photo = models.ImageField(upload_to='cars')
```

Qualquer instância de `Car` terá um atributo `photo` que você pode usar para obter os detalhes da foto anexa:

```
>>> car = Car.objects.get(name="57 Chevy")
>>> car.photo
<ImageFieldFile: chevy.jpg>
>>> car.photo.name
u'cars/chevy.jpg'
>>> car.photo.path
u'/media/cars/chevy.jpg'
>>> car.photo.url
u'http://media.example.com/cars/chevy.jpg'
```

Este objeto – `car.photo` no exemplo – é um objeto `File`, o que significa que ele tem todos os métodos e atributos descritos abaixo.

O objeto File

Internamente, o Django usa um `django.core.files.File` toda vez que ele precisa representar um arquivo. Este objeto é um pequeno contorno ao [objeto file nativo](#) do Python com algumas adições específicas do Django.

Na maior parte do tempo você simplesmente irá usar um `File` que o Django fornece a você (i.e. um arquivo atachado em um model, como mostrado acima, ou talvez um arquivo de upload).

Se você precisa construir um `File` você mesmo, o caminho mais fácil é criar um usando o objeto nativo `file` do Python:

```
>>> from django.core.files import File

# Cria um objeto file do Python usando open()
>>> f = open('/tmp/hello.world', 'w')
>>> myfile = File(f)
```

Agora você pode usar qualquer atributo ou método do `File` documentado em [O objeto File](#).

Armazenamento de arquivo

Por trás das cenas, o Django se encarrega das decisões sobre como e onde os arquivos são armazenados no sistema de arquivos. Este é um objeto que na verdade entende coisas como sistema de arquivos, abertura e leitura de arquivos, etc.

O sistema de armazenamento padrão do Django é dado pela configuração `DEFAULT_FILE_STORAGE`; se você não provê explicitamente um sistema de armazenamento, é este que será utilizado.

Veja abaixo detalhes do sistema de armazenamento nativo, e veja [Escrevendo um sistema de armazenamento customizado](#) para mais informações sobre criar seu próprio sistema de armazenamento.

Objetos de armazenamento

Embora a maior parte do tempo você pretenda utilizar um objeto `File` (que se encarrega de armazenar os arquivos apropriadamente), você pode usar sistemas de armazenamento diretamente. Você pode criar uma instância de alguma classe customizada de armazenamento de arquivos, ou – o que é mais frequente – você pode usar o sistema padrão de armazenamento global:

```
>>> from django.core.files.storage import default_storage
>>> from django.core.files.base import ContentFile

>>> path = default_storage.save('/path/to/file', ContentFile('new content'))
>>> path
u'/path/to/file'

>>> default_storage.size(path)
11
>>> default_storage.open(path).read()
'new content'

>>> default_storage.delete(path)
>>> default_storage.exists(path)
False
```

Consulte [API de armazenamento de arquivos](#), para saber mais sobre a API de armazenamento de arquivos.

A classe nativa de armazenamento filesystem

O Django vem com uma classe nativa `FileSystemStorage` (definida em `django.core.files.storage`) que implementa o sistema de arquivos básico local. Seu construtor recebe dois argumentos:

Argumento	Descrição
<code>location</code>	Opicional. O caminho absoluto para o diretório que receberá os arquivos. Se omitido, ele será setado com o valor do <code>MEDIA_ROOT</code> .
<code>base_url</code>	Opicional. URL que serve o arquivo armazenado em seu local. Se omitido, ele será o padrão definido no seu <code>MEDIA_URL</code> .

Por exemplo, o seguinte código irá armazenar os arquivos enviados em `/media/photos` independentemente do que seu `MEDIA_ROOT` é:

```
from django.db import models
from django.core.files.storage import FileSystemStorage

fs = FileSystemStorage(location='/media/photos')

class Car(models.Model):
    ...
    photo = models.ImageField(storage=fs)
```

Os *sistemas de armazenamento customizados* funcionam da mesma forma: você pode passá-los como o argumento `storage` para um `FileField`.

Testando aplicações Django

O teste automatizado é uma ferramenta extremamente útil para eliminar bugs utilizada pelo desenvolvedor Web moderno. Você pode usar uma coleção de testes – uma **test suite** – para resolver, ou evitar, vários problemas:

- Quando você está escrevendo um código novo, pode usar os testes para verificar se seu código funciona como esperado.
- Quando está refatorando ou modificando um código antigo, você pode usar os testes para garantir que suas mudanças não afetaram inesperadamente o comportamento de sua aplicação.

Testar uma aplicação Web é uma tarefa complexa, porque uma aplicação Web é feita de várias camadas de lógica – da manipulação de uma requisição em nível HTTP, para a validação e processamento de formulário, para a renderização de template. Com o framework de teste-execução do Django e outros utilitários, você pode simular requisições, inserir dados de teste, inspecionar a saída de sua aplicação e frequentemente verificar se seu código está fazendo o que deveria.

A melhor parte é que isso tudo é muito fácil.

Este documento é dividido em duas duas seções. Na primeira, explicamos como escrever testes com Django e, posteriormente, explicamos como rodá-los.

Escrevendo testes

Existem duas maneiras de se escrever testes com Django, correspondendo com os dois frameworks de teste que estão na biblioteca padrão do Python, que são:

- **Doctests** – os testes estão embutidos nas docstrings (strings de documentação) de suas funções e são escritas de tal maneira a emular uma sessão do interpretador interativo do Python. Por exemplo:

```
def my_func(a_list, idx):  
    """  
    >>> a = ['larry', 'curly', 'moe']  
    >>> my_func(a, 0)  
    'larry'  
    >>> my_func(a, 1)  
    'curly'  
    """  
    return a_list[idx]
```

- **Unit tests** – (Testes unitários) são testes que são expressados como métodos em uma classe Python que é uma subclasse de `unittest.TestCase`. Por exemplo:

```
import unittest

class MyFuncTestCase(unittest.TestCase):
    def testBasic(self):
        a = ['larry', 'curly', 'moe']
        self.assertEqual(my_func(a, 0), 'larry')
        self.assertEqual(my_func(a, 1), 'curly')
```

Você pode escolher o framework de teste que te agrada, dependendo da sintaxe que preferir, ou você pode misturar os dois, utilizando um framework para parte de seu código e outro framework para outra parte. Você também pode usar qualquer *outro* framework de testes Python, como explicaremos adiante.

Escrevendo doctests

Os doctests usam o módulo `doctest` padrão do Python, que procura em suas docstrings por instruções que se pareçam com uma sessão do interpretador interativo do Python. Uma explicação completa de como funciona o doctest está fora do escopo deste documento; leia a documentação oficial do Python para maiores detalhes.

O que é uma docstring?

Uma boa explicação de docstrings (e algumas diretrizes para utilizá-la de maneira completa) pode ser encontrada em [PEP 257](#):

Uma docstring é uma literal de string que ocorre como a primeira instrução em um módulo, função, classe, ou definição de método. Esta docstring torna-se o atributo especial `__doc__` do objeto em questão.

Por exemplo, esta função possui uma docstring que descreve o que ela faz:

```
def add_two(num):
    "Retorna o resultado da adição de dois ao número informado."
    return num + 2
```

Pelo motivo de que testes freqüentemente geram uma boa documentação, colocar testes diretamente nas suas docstrings é uma maneira eficiente de documentar *e* testar o seu código.

Para uma dada aplicação Django, o executor de testes procura por doctests em dois lugares:

- No arquivo `models.py`. Você pode definir doctests para o módulo todo e/ou um doctest para cada modelo. É uma pratica comum colocar doctests de nível de aplicação numa docstring do módulo e doctests de nível de modelo em docstrings de modelos.
- Um arquivo chamado `tests.py` no diretório da aplicação – ou seja, o diretório que contém `models.py`. Esse arquivo é um `hook` para todos e quaisquer doctests que você queira escrever que não sejam necessariamente relacionados a modelos.

Aqui vai um exemplo de doctest de modelo:

```
# models.py

from django.db import models

class Animal(models.Model):
    """
    An animal that knows how to make noise

    # Create some animals
    >>> lion = Animal.objects.create(name="lion", sound="roar")
```

```
>>> cat = Animal.objects.create(name="cat", sound="meow")

# Make 'em speak
>>> lion.speak()
'The lion says "roar"'
>>> cat.speak()
'The cat says "meow"'
"""

name = models.CharField(max_length=20)
sound = models.CharField(max_length=20)

def speak(self):
    return 'The %s says "%s"' % (self.name, self.sound)
```

Quando você *roda seus testes*, o executor de testes irá encontrar essa docstring – note que pedaços dela parecem uma sessão interativa de Python – e executar suas linhas verificando se os resultados batem.

No caso de testes de modelo, note que o executor de testes cuida de criar seu próprio banco de dados. Ou seja, qualquer teste que acesse banco de dados – criando e gravando instâncias de modelos, por exemplo – não afetará seu banco de dados em produção. Cada doctest inicia com um banco de dados novo contendo uma tabela vazia para cada modelo. (Veja a seção de fixtures, abaixo, para mais detalhes.) Note que para usar esse recurso, o usuário que o Django utiliza para se conectar ao banco de dados deve ter direito de criar novos bancos CREATE DATABASE.

Para mais detalhes sobre como funciona o doctest, veja a [documentação da biblioteca padrão para doctest](#)

Escrevendo testes unitários

Como doctests, os testes unitários do Django usam um módulo da biblioteca padrão: `unittest`. Esse módulo usa uma maneira diferente de definir testes, utilizando um método baseado em classes.

Como nos doctests, para uma dada aplicação Django, o executor de testes procura por testes unitários em dois lugares:

- O arquivo `models.py`. O executor de testes procura por qualquer subclasse de `unittest.TestCase` nesse módulo.
- Um arquivo chamado `tests.py` no diretório da aplicação – o mesmo que contém `models.py`. Novamente, o executor de testes procura por qualquer subclasse de `unittest.TestCase` nesse módulo.

Este exemplo de subclasse de `unittest.TestCase` é equivalente ao exemplo dado na seção de doctest acima:

```
import unittest
from myapp.models import Animal

class AnimalTestCase(unittest.TestCase):
    def setUp(self):
        self.lion = Animal.objects.create(name="lion", sound="roar")
        self.cat = Animal.objects.create(name="cat", sound="meow")

    def testSpeaking(self):
        self.assertEqual(self.lion.speak(), 'The lion says "roar"')
        self.assertEqual(self.cat.speak(), 'The cat says "meow"')
```

Quando você *roda seus testes*, o comportamento padrão do utilitário de teste é encontrar todos os test cases (ou seja, subclasses de `unittest.TestCase`) nos arquivos `models.py` e `tests.py`, automaticamente montar uma test suite (conjunto de testes) destes test cases, e rodá-la.

Há uma segunda maneira de se definir um test suite para um módulo: se você define uma função chamada `suite()` seja em `models.py` ou `tests.py`, o executor de testes do Django usará essa função para construir a test suite para o módulo. Isso segue a [organização sugerida](#) para testes unitários. Veja a documentação do Python para mais detalhes de como construir uma test suite complexa.

Para mais detalhes sobre `unittest`, veja a [documentação de unittest da biblioteca padrão](#).

Qual devo usar?

Pelo motivo de o Django suportar ambos os frameworks de testes padrão do Python, cabe a você, de acordo com seu gosto, decidir qual utilizar. Você pode até mesmo decidir usar *ambos*.

Para desenvolvedores novatos em testes, essa escolha pode parecer confusa. Aqui estão algumas diferenças chave para ajudá-lo a decidir qual método é melhor:

- Se você já utiliza Python por algum tempo, o `doctest` provavelmente parecerá mais “pythônico”. Ele foi projetado para tornar a escrita de testes o mais fácil possível, então não requer nenhum esforço extra de escrever classes ou métodos. Você simplesmente coloca seus testes em docstrings. Isso tem a vantagem adicional de servir como documentação (e documentação correta!).

Se você está iniciando com testes, utilizar doctests farão com que você obtenha resultados mais rapidamente.

- O framework `unittest` provavelmente será bem familiar para desenvolvedores vindos do Java. O `unittest` é inspirado pelo JUnit do Java, então você se sentirá em casa com esse método se você já usou o JUnit ou qualquer outro framework de testes inspirado no JUnit.
- Se você precisa escrever muitos testes que compartilham código parecido, então você vai gostar da organização baseada em classes e métodos do framework `unittest`. Ele facilita abstrair tarefas comuns em métodos comuns. O framework também suporta configuração explícita e/ou limpeza de rotinas, que te dão um alto nível de controle sobre o ambiente no qual seus testes são executados.

De novo, lembre-se de que você pode usar ambos os sistemas lado a lado (até mesmo em uma mesma aplicação). No final, a maioria dos projetos provavelmente acabará utilizando ambos. Cada um se destaca em diferentes circunstâncias.

Rodando os testes

Uma vez que você escreveu os testes, execute-os utilizando o utilitário do seu projeto `manage.py`:

```
$ ./manage.py test
```

Por padrão, isso executará cada teste em cada aplicação em `INSTALLED_APPS`. Se você só quer rodar os testes para uma aplicação em particular, adicione o nome da aplicação à linha de comando. Por exemplo, se seu `INSTALLED_APPS` contém `'myproject.polls'` e `'myproject.animals'`, você pode rodar somente os testes unitários de `myproject.animals` com este comando:

```
# ./manage.py test animals
```

Note que utilizamos `animals`, e não `myproject.animals`. Agora você pode escolher qual teste rodar. Se você utiliza testes unitários, em vez de doctests, você pode ser ainda *mais* específico na escolha de quais testes executar. Para rodar um único teste em uma aplicação (por exemplo, o `AnimalTestCase` descrito na seção “Escrevendo testes unitários”), adicione o nome do test case à linha de comando:

```
$ ./manage.py test animals.AnimalTestCase
```

E pode ficar ainda mais granular que isso! Para rodar um *único* método de teste dentro de um test case, adicione o nome do método de teste:

```
$ ./manage.py test animals.AnimalTestCase.testFluffyAnimals
```

O banco de dados de teste

Testes que necessitam de uma base de dados (nomeadamente, testes de modelo) não usarão seu banco de dados “real” (produção). Um banco de dados vazio é criado em separado para os testes.

Independentemente de os testes passarem ou falharem, o banco de dados de teste é destruído quando todos os testes forem executados.

Por padrão, o nome deste banco de dados de teste é o valor da configuração `DATABASE_NAME` adicionando o prefixo `test_`. Quando um banco de dados SQLite é utilizado, os testes usarão bancos de dados na memória (ou seja, todo o banco será criado somente na memória, não utilizando nada do sistema de arquivos). Se você quiser usar um nome diferente para o banco de dados de teste, especifique o parâmetro de configuração `TEST_DATABASE_NAME`.

Além de utilizar um banco de dados separado, o executor de testes usará os mesmos parâmetros de banco de dados do seu arquivo de configuração: `DATABASE_ENGINE`, `DATABASE_USER`, `DATABASE_HOST`, etc. O banco de dados de teste é criado pelo usuário especificado em `DATABASE_USER`, então é necessário garantir que esta conta de usuário tem privilégios suficientes para criar um novo banco de dados no sistema. *Please, see the release notes* Para um controle apurado sobre a codificação de caractere de seu banco de dados de teste, use o parâmetro de configuração `TEST_DATABASE_CHARSET`. Se você está utilizando MySQL, você pode também usar o parâmetro `TEST_DATABASE_COLLATION` para controlar uma collation particular utilizada pelo banco de dados de teste. Veja a *documentação de configurações* para mais detalhes dessas configurações avançadas.

Outras condições de testes

Independentemente do valor do `DEBUG` do seu arquivo de configuração, todos os testes do Django rodam com `DEBUG=False`. Isto é para assegurar que a saída observada de seu código seja igual ao da aplicação em produção.

Entendendo a saída do teste

Quando roda seus testes, você visualiza um número de mensagens à medida que o executor de testes se prepara. Você pode controlar o nível de detalhe dessas mensagens com a opção de linha de comando `verbosity`:

```
Creating test database...
Creating table myapp_animal
Creating table myapp_mineral
Loading 'initial_data' fixtures...
No fixtures found.
```

Isso informa que o executor de testes está criando um banco de teste, como descrito na seção anterior.

Uma vez que o banco de testes foi criado, o Django rodará os seus testes. Se tudo correr bem, você verá algo do tipo:

```
-----
Ran 22 tests in 0.221s

OK
```

Se existirem falhas, entretanto, você verá os detalhes completos sobre quais testes falharam:

```
=====
FAIL: Doctest: ellington.core.throttle.models
-----
Traceback (most recent call last):
  File "/dev/django/test/doctest.py", line 2153, in runTest
    raise self.failureException(self.format_failure(new.getvalue()))
AssertionError: Failed doctest test for myapp.models
  File "/dev/myapp/models.py", line 0, in models

-----
File "/dev/myapp/models.py", line 14, in myapp.models
Failed example:
    throttle.check("actor A", "action one", limit=2, hours=1)
Expected:
```

```
True
Got:
False

-----
Ran 2 tests in 0.048s

FAILED (failures=1)
```

Uma explicação completa sobre essa saída de erro está fora do escopo deste documento, mas ela é bem intuitiva. Você pode consultar a documentação da biblioteca `unittest` do Python para maiores detalhes.

Note que o código retornado pelo script executor de testes é o número total de testes que falharam. Se todos os testes passarem, o código de retorno é 0. Este recurso é útil se você está usando script executor de testes em um script shell e precisa verificar o sucesso ou falha naquele nível.

Ferramentas de teste

O Django provê um pequeno conjunto de ferramentas que são uma verdadeira mão- na-roda na hora de escrever os testes.

O cliente de teste

O cliente de teste é uma classe Python que age como um navegador Web, permitindo a você testar suas views e interagir com suas aplicações feitas em Django programaticamente.

Algumas das coisas que você pode fazer com o cliente de teste são:

- Simular requisições GET e POST em uma URL e observar a resposta – tudo desde o nível baixo do HTTP (os cabeçalhos de resultado e códigos de status) até o conteúdo da página.
- Testar se a view correta é executada para uma dada URL.
- Testar que uma dada requisição é gerada por um determinado template, com um determinado contexto de template que contém certos valores.

Note que este cliente de teste não tem o propósito de ser um substituto para [Twill](#), [Selenium](#), ou outros frameworks “in-browser”. O cliente de teste do Django tem um foco diferente. Em resumo:

- Use o cliente de teste do Django para atestar que a view correta está sendo chamada e que a view está recebendo os dados de contexto corretos.
- Use frameworks in-browser como Twill e Selenium para testar HTML gerados e *comportamento* de páginas Web, ou seja, funcionalidade JavaScript.

Uma test suite completa deve usar uma combinação de ambos tipos de testes.

Visão geral e um exemplo rápido

Para usar o cliente de teste, instancie `django.test.client.Client` e acesse páginas Web:

```
>>> from django.test.client import Client
>>> c = Client()
>>> response = c.post('/login/', {'username': 'john', 'password': 'smith'})
>>> response.status_code
200
>>> response = c.get('/customer/details/')
>>> response.content
'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 ...'
```

Como esse exemplo sugere, você pode instanciar `Client` de dentro de uma sessão do interpretador interativo do Python.

Note algumas coisas importantes sobre como o cliente de teste funciona:

- O cliente de teste *não* requer que o servidor Web esteja rodando. Aliás, ele rodará muito bem sem nenhum servidor Web! Isso se deve ao fato de que ele evita o esforço extra do HTTP e lida diretamente com o framework Django. Isso faz com que os testes unitários rodem bem mais rápido.
- Ao acessar as páginas, lembre-se de especificar o *caminho* da URL, e não todo o domínio. Por exemplo, isto é correto:

```
>>> c.get('/login/')
```

E isto incorreto:

```
>>> c.get('http://www.example.com/login/')
```

O cliente de testes não é capaz de acessar páginas Web que não fazem parte do seu projeto Django. Se você precisa acessar outras páginas Web, utilize algum módulo da biblioteca padrão do Python como `urllib` ou `urllib2`.

- Para resolver URLs, o cliente de testes usa o `URLconf` que está especificado no parâmetro de configuração `ROOT_URLCONF`.
- Apesar de o exemplo acima funcionar no interpretador interativo do Python, algumas funcionalidades do cliente de teste, notadamente as relacionadas a templates, somente estão disponíveis *enquanto os testes estão sendo rodados*.

O motivo disso é porque o executor de testes do Django faz um pouco de magia negra para determinar qual template foi carregado por uma determinada view. Essa magia negra (essencialmente um patch no sistema de templates na memória) acontece somente durante a execução dos testes.

Fazendo requisições

Use a classe `django.test.client.Client` para fazer as requisições. Ela não requer argumentos em sua construção:

class Client

Uma vez que você tenha uma instância de `Client`, você pode chamar qualquer um dos seguintes métodos:

get (*path*, *data*=*{}*, ***extra*)

Faz uma requisição GET no *path* informado e devolve um objeto `Response`, que está documentado abaixo.

Os pares de chave-valor no dicionário *data* são usados para criar os dados que serão enviados via GET. Por exemplo:

```
>>> c = Client()
>>> c.get('/customers/details/', {'name': 'fred', 'age': 7})
```

...resultará em uma requisição GET equivalente a:

```
/customers/details/?name=fred&age=7
```

The *extra* keyword arguments parameter can be used to specify headers to be sent in the request. For example:

```
>>> c = Client()
>>> c.get('/customers/details/', {'name': 'fred', 'age': 7},
...      HTTP_X_REQUESTED_WITH='XMLHttpRequest')
```


...will send the HTTP header `HTTP_X_REQUESTED_WITH` to the details view, which is a good way to test code paths that use the `django.http.HttpRequest.is_ajax()` method.

post (*path*, *data*=*{}*, *content_type*=*MULTIPART_CONTENT*, ***extra*)

Faz uma requisição POST no *path* informado e devolve um objeto `Response`, que está documentado abaixo.

Os pares de chave-valor no dicionário *data* são usados para enviar os dados via POST. Por exemplo:

```
>>> c = Client()
>>> c.post('/login/', {'name': 'fred', 'passwd': 'secret'})
```

...resultará em uma requisição POST a esta URL:

```
/login/
```

...com estes dados de POST:

```
name=fred&passwd=secret
```

Se você informa o *content_type* (ex: `text/xml` para um XML), o conteúdo de *data* será enviado como está na requisição POST, utilizando *content_type* no cabeçalho HTTP `Content-Type`.

Se você não informa um valor para *content_type*, os valores em *data* serão transmitidos como um tipo de conteúdo `multipart/form-data`. Nesse caso, os pares chave-valor em *data* serão codificados como uma mensagem `multipart` e usados para criar os dados de POST.

Para enviar múltiplos valores para uma chave – por exemplo, para especificar as seleções para um `<select multiple>` – informe os valores como uma lista ou tupla para a chave. Por exemplo, este valor de *data* enviará três valores selecionados para o campo de nome `choices`:

```
{ 'choices': ('a', 'b', 'd') }
```

Enviar arquivos é um caso especial. Para postar um arquivo, você só precisa informar o nome do campo como chave, e um manipulador de arquivo apontando para o arquivo que deseja postar como valor. Por exemplo:

```
>>> c = Client()
>>> f = open('wishlist.doc')
>>> c.post('/customers/wishes/', {'name': 'fred', 'attachment': f})
>>> f.close()
```

(O nome `attachment` aqui não é relevante; use qualquer nome que o seu código de processamento de arquivo espera.)

Perceba que você deve fechar o arquivo manualmente depois que ele foi informado para o `post()`.

The *extra* argument acts the same as for `Client.get()`.

login (***credentials*)

Please, see the release notes Se seu site Django usa o *sistema de autenticação* e você precisa logar usuários, você pode usar o método `login()` do cliente de testes para simular o efeito de um usuário logando no site.

Depois de chamar este método, o cliente de testes terá todos os cookies e dados de sessão necessários para passar a qualquer teste baseado em login que faça parte de uma view.

O formato do argumento *credentials* depende de qual *backend de autenticação* você está usando (que é configurado pelo parâmetro de configuração `AUTHENTICATION_BACKENDS`). Se você está usando o backend padrão de autenticação do Django (`ModelBackend`), *credentials* deve ser o nome do usuário e a senha, informados como argumentos nomeados:

```
>>> c = Client()
>>> c.login(username='fred', password='secret')
>>> # Agora você pode acessar a view que somente está disponível
>>> # para usuários logados.
```

Se você está usando um backend de autenticação diferente, este método requer credenciais diferentes. Ele requer as mesmas credenciais que o método `authenticate()` do backend.

`login()` devolve `True` se as credenciais foram aceitas e o login ocorreu com sucesso.

Finalmente, você precisa lembrar-se de criar contas de usuários antes de utilizar este método. Como explicado acima, o executor de testes utiliza uma base de dados de teste, que não contém usuários criados. Como resultado, contas que são válidas no seu site em produção não funcionarão nas condições de teste. Você precisa criar os usuários como parte de sua test suite – tanto manualmente (utilizando a API de modelos do Django) ou com uma test fixture.

Lembre-se que se você quiser que seu usuários de teste tenha uma senha, você não pode setar a senha do usuário usando o atributo `password` diretamente – você deve usar a função `set_password()` para armazenar corretamente uma senha criptografada. Alternativamente, você pode usar o método helper `create_user()` para criar um novo usuário com a senha criptografada corretamente.

`logout()`

Please, see the release notes Se o seu site usa o *sistema de autenticação* do Django, o método `logout()` pode ser utilizado para simular o efeito de um usuário se deslogar do seu site.

Depois de chamar este método, o cliente de teste terá todos os cookies e dados de sessão retornados para os valores padrões. Requisições subsequentes parecerão vir de um usuário anônimo `AnonymousUser`.

Testando as respostas

Ambos os métodos `get()` e `post()` devolvem um objeto `Response`. Este objeto `Response` *não* é o mesmo que o objeto `HttpResponse` retornado pelas views do Django; o objeto de resposta do teste tem dados adicionais úteis para serem verificados pelo código de teste.

Especificamente, o objeto `Response` tem os seguintes atributos:

`class Response`

`client`

O cliente de teste que foi usado para fazer a requisição que resultou na resposta.

`content`

O corpo da resposta, como uma string. Este é o conteúdo final da página que foi gerada pela view, ou alguma mensagem de erro.

`context`

A instância `Context` que foi utilizada pelo template para produzir o conteúdo da resposta.

Se a página gerada utilizou múltiplos templates, então o `context` será uma lista de objetos `Context`, na ordem em que foram utilizados.

`request`

Os dados da requisição que provocaram a resposta.

`status_code`

O status da resposta HTTP, como um inteiro. Veja [RFC2616](#) para uma lista completa de códigos de status HTTP.

`template`

A instância de `Template` que foi utilizada para gerar o conteúdo final. Use `template.name` para obter o nome do arquivo do template, se o template foi carregado de um arquivo. (O nome é uma string como `'admin/index.html'`.)

Se a página gerada utilizou vários templates – ex: utilizando *herança de templates* – então `template` será uma lista de instâncias de `Template`, na ordem em que eles foram utilizados.

Você pode também usar uma sintaxe de dicionário no objeto de resposta para obter o valor de qualquer configuração nos cabeçalhos HTTP. Por exemplo, você pode determinar o conteúdo de uma resposta usando `response['Content-Type']`.

Exceções

Se você aponta o cliente de testes para uma view que lança uma exceção, esta exceção estará visível no test case. Você pode então usar blocos `try...catch` ou `unittest.TestCase.assertRaises()` para testar por exceções.

As únicas exceções que não estão visíveis ao cliente de teste são `Http404`, `PermissionDenied` e `SystemExit`. O Django captura estas exceções internamente e converte-as em códigos de resposta HTTP adequados. Nesses casos, você pode verificar `response.status_code` no seu código.

Estado persistente (persistent state)

O cliente de teste é “stateful”, ou seja, mantém o estado. Se uma resposta retorna um cookie, então esse cookie será armazenado no cliente de teste e enviado com todas as requisições subsequentes de `get()` e `post()`.

Políticas de expiração desses cookies não são seguidas. Se você quiser que um cookie expire, delete-o manualmente ou crie uma nova instância de `Client` (o que irá deletar todos os cookies).

Um cliente de teste possui dois atributos que armazenam informações de estado persistente. Você pode acessar essas propriedades como parte de uma condição de teste.

`Client.cookies`

Um objeto `SimpleCookie` Python, contendo os valores atuais de todos os cookies do cliente. Veja a [Documentação do módulo cookie](#) para saber mais.

`Client.session`

A dictionary-like object containing session information. See the [session documentation](#) for full details. Um objeto que se comporta como dicionário contendo informações da sessão. Veja a [documentação da sessão](#) para maiores detalhes.

Exemplo

Segue um teste unitário simples usando o cliente de teste:

```
import unittest
from django.test.client import Client

class SimpleTest(unittest.TestCase):
    def setUp(self):
        # Cada teste precisa de um cliente.
        self.client = Client()

    def test_details(self):
        # Faz uma requisição GET.
        response = self.client.get('/customer/details/')

        # Verifica se a resposta foi 200 OK.
        self.failUnlessEqual(response.status_code, 200)

        # Verifica se o contexto utilizado contém 5 customers.
        self.failUnlessEqual(len(response.context['customers']), 5)
```

TestCase

Uma classe de teste unitário normal do Python estende uma classe base `unittest.TestCase`. O Django provê uma extensão desta classe:

class TestCase

Esta classe provê algumas capacidades adicionais que podem ser úteis para testar site Web.

Converter um `unittest.TestCase` normal para um Django `TestCase` é fácil: basta mudar a classe base de seu teste de `unittest.TestCase` para `django.test.TestCase`. Todas as funcionalidades padrões de teste unitário do Python continuarão disponíveis, mas serão aumentadas com algumas adições úteis.

O cliente de teste default

Please, see the release notes

TestCase.client

Cada teste em uma instância de `django.test.TestCase` tem acesso a uma instância do cliente de testes do Django. Esse cliente pode ser acessado como `self.client`. O cliente é recriado para cada teste, então você não tem de se preocupar sobre o estado (como os cookies) ser levado de um teste a outro.

Isso significa que, em vez de instanciar um `Client` em cada teste:

```
import unittest
from django.test.client import Client

class SimpleTest(unittest.TestCase):
    def test_details(self):
        client = Client()
        response = client.get('/customer/details/')
        self.failUnlessEqual(response.status_code, 200)

    def test_index(self):
        client = Client()
        response = client.get('/customer/index/')
        self.failUnlessEqual(response.status_code, 200)
```

...você só precisa se referir a `self.client`, como:

```
from django.test import TestCase

class SimpleTest(TestCase):
    def test_details(self):
        response = self.client.get('/customer/details/')
        self.failUnlessEqual(response.status_code, 200)

    def test_index(self):
        response = self.client.get('/customer/index/')
        self.failUnlessEqual(response.status_code, 200)
```

Carga de fixture

TestCase.fixtures

Um teste para um site Web com banco de dados não tem muita utilidade se não existir dados no banco. Para facilitar a carga destes dados no banco, a classe `TestCase` do Django proporciona uma maneira de carregar **fixtures**.

Uma fixture é uma coleção de dados que o Django sabe como importar para um banco de dados. Por exemplo, se o seu site tem contas de usuários, você pode configurar uma fixture de usuários no intuito de popular seu banco durante os testes.

O método mais simples e direto de criar uma fixture é usar o comando `manage.py dumpdata`. Isso assume que você já tem algum dado em seu banco. Veja a documentação do `dumpdata` para mais detalhes.

Note: Se você já rodou alguma vez o `manage.py syncdb`, você já usou fixture sem nem mesmo saber! Quando você dá um `syncdb` no banco de dados pela primeira vez, o Django instala uma fixture chamada `initial_data`. Isso dá a possibilidade de popular um banco de dados novo com qualquer dado relacional, como um conjunto padrão de categorias, por exemplo.

Fixtures com outros nomes podem sempre ser instaladas manualmente usando o comando `manage.py loaddata`.

Uma vez que você criou uma fixture e colocou em algum lugar em seu projeto Django, você pode utilizá-la nos seus testes unitários especificando um atributo de classe `fixtures` na sua subclasse de `django.test.TestCase`:

```
from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    fixtures = ['mammals.json', 'birds']

    def setUp(self):
        # Definições de testes como anteriormente.

    def testFluffyAnimals(self):
        # Um teste que usa fixtures.
```

Eis o que acontecerá nesse caso:

- No início de cada test case, antes de `setUp()` ser rodado, o Django limpará o banco de dados, retornando um banco de dados no estado que estava diretamente após a chamada de `syncdb`.
- Em seguida, todas as fixtures nomeadas são instaladas. Nesse exemplo, o Django instalará qualquer fixture JSON chamada `mammals`, seguida por qualquer fixture chamada `birds`. Veja a documentação do `loaddata` para mais detalhes sobre a definição e a instalação de fixtures.

Esse procedimento de limpeza/carga é repetido para cada teste no test case, então você pode ter certeza de que o resultado de um teste não será afetado por outro teste, ou pela ordem de execução dos testes.

Configuração do URLconf

Please, see the release notes

`TestCase.urls`

Se a sua aplicação possui views, você pode querer incluir testes que utilizam o cliente de testes para exercitá-las. Entretanto, um usuário final é livre para configurar as views em sua aplicação em qualquer URL de sua preferência. Isso significa que seus testes não podem contar com o fato de que suas views estarão disponíveis em uma URL em particular.

A fim de proporcionar URLs confiáveis para seu teste, `django.test.TestCase` tem a capacidade de customizar a configuração do URLconf pelo período de execução dos testes. Se sua instância de `TestCase` define o atributo `urls`, o `TestCase` usará os valores deste atributo como o `ROOT_URLCONF` pelo período de execução do teste.

Por exemplo:

```
from django.test import TestCase

class TestMyViews(TestCase):
    urls = 'myapp.test_urls'
```

```
def testIndexPageView(self):
    # Aqui você testará sua view usando ``Client``.
```

Esse test case utilizará o conteúdo de `myapp.test_urls` como o `URLconf` durante a execução do teste.

Esvaziando a caixa de saída de teste

Please, see the release notes Se você usa a classe Django `TestCase`, o executor de testes limpará o conteúdo da caixa de saída de e-mail no início de cada teste.

Para mais detalhes sobre os serviços de e-mail durante os teste, veja *Serviços de e-mail*.

Assertions

Please, see the release notes Como numa classe Python `unittest.TestCase` normal que implementa métodos de asserção como `assertTrue` e `assertEquals`, a classe customizada `TestCase` do Django disponibiliza alguns métodos de asserção customizados que são úteis para testar aplicações Web:

`TestCase.assertContains (response, text, count=None, status_code=200)`

Testa se uma instância de `Response` produziu o `status_code` informado e que o `text` aparece no conteúdo da resposta. Se `count` é fornecido, `text` deve ocorrer exatamente `count` vezes na resposta.

`TestCase.assertNotContains (response, text, status_code=200)`

Testa se uma instância de `Response` produziu o `status_code` informado e que o `text` não aparece no conteúdo da resposta.

`TestCase.assertFormError (response, form, field, errors)`

Testa se um campo no formulário lança a lista de erros fornecida quando gerado no formulário.

`form` é o nome da instância de `Form` informada ao contexto do template.

`field` é o nome do campo no formulário para verificar. Se `field` tem um valor de `None`, erros não relacionados a campos (erros que você pode acessar via `form.non_field_errors()`) serão verificados.

`errors` é uma string de erro, ou uma lista de strings de erro, que são esperados como resultado da validação do formulário.

`TestCase.assertTemplateUsed (response, template_name)`

Testa se o template com o nome informado foi usado na geração da resposta.

O nome é uma string como `'admin/index.html'`.

`TestCase.assertTemplateNotUsed (response, template_name)`

Testa se o template com o nome informado não foi usado na geração da resposta.

`TestCase.assertRedirects (response, expected_url, status_code=302, target_status_code=200)`

Teste se a resposta devolve um status de redirecionamento `status_code`, se redirecionou para a URL `expected_url` (incluindo quaisquer dados GET), e a página subsequente foi recebida com o `target_status_code`.

Serviços de e-mail

Please, see the release notes Se alguma de suas views manda e-mail usando a *Funcionalidade de e-mail do Django*, você provavelmente não quer mandar e-mail cada vez que você roda um teste utilizando a view. Por esse motivo, o executor de testes do Django automaticamente redireciona todos e-mails enviados por meio do Django para uma caixa de saída fictícia. Isso deixa você testar cada aspecto do envio de e-mail – do número de mensagens enviadas ao conteúdo de cada mensagem – sem ter de enviar as mensagens de verdade.

O executor de testes consegue fazer isso de forma transparente trocando a classe `<django.core.mail.SMTPConnection>` normal por uma versão diferente. (Não se preocupe – isso não tem efeito em quaisquer outros meios de envio de e-mail fora do Django, como o servidor de e-mail de sua máquina, se estiver rodando um).

`django.core.mail.outbox`

Durante a execução dos testes, cada e-mail de saída é gravado em `django.core.mail.outbox`. Esta é uma lista simples de todas as instâncias de `<django.core.mail.EmailMessage>` que foram enviadas. Ela não existem nas condições normais de execução, ou seja, quando você não está rodando testes unitários. A caixa de saída é criada durante a configuração do teste, junto com a `<django.core.mail.SMTPConnection>` fictícia. Quando o framework de teste é encerrado, a classe `<django.core.mail.SMTPConnection>` padrão é restaurada, e a caixa de saída de teste é destruída.

The `outbox` attribute is a special attribute that is created *only* when the tests are run. It doesn't normally exist as part of the `django.core.mail` module and you can't import it directly. The code below shows how to access this attribute correctly.

Aqui vai um exemplo de teste que verifica o tamanho e conteúdo de `django.core.mail.outbox`:

```
from django.core import mail
from django.test import TestCase

class EmailTest(TestCase):
    def test_send_email(self):
        # Envia mensagem.
        mail.send_mail('Assunto aqui', 'Aqui vai a mensagem.',
                       'from@example.com', ['to@example.com'],
                       fail_silently=False)

        # Verifica se uma mensagem foi enviada.
        self.assertEqual(len(mail.outbox), 1)

        # Verifica se o assunto da mensagem é igual
        self.assertEqual(mail.outbox[0].subject, 'Assunto aqui')
```

Como dito *anteriormente*, a caixa de saída de teste é esvaziada no início de cada teste em um `TestCase` Django. Para esvaziar a caixa de saída manualmente, atribua uma lista vazia para `mail.outbox`:

```
from django.core import mail

# Esvazia a caixa de saída
mail.outbox = []
```

Utilizando frameworks de testes diferentes

Obviamente, `doctest` e `unittest` não são os únicos frameworks de testes Python. Apesar de o Django não suportar explicitamente frameworks alternativos, ele provê uma maneira de invocar testes construídos para um framework alternativo como se fossem testes Django normais.

Quando você roda `./manage.py test`, o Django procura a configuração `TEST_RUNNER` para determinar o que fazer. Por padrão, o `TEST_RUNNER` aponta para `'django.test.simple.run_tests'`. Este método define o comportamento padrão dos testes no Django. Esse comportamento envolve:

1. Fazer uma configuração global antes do teste.
2. Criar o banco de dados de teste.
3. Executar `syncdb` para instalar os modelos e dados iniciais no banco de dados de teste.
4. Procurar testes unitários e doctests nos arquivos `models.py` e `tests.py` em cada aplicação instalada.
5. Rodar os testes unitários e doctests que forem encontrados.
6. Destruir o banco de dados de teste.
7. Executar um encerramento global após o término dos testes.

Se você define seu próprio método executor de testes e aponta o `TEST_RUNNER` para este método, o Django rodará o seu executor de testes toda vez que você rodar `./manage.py test`. Desta maneira, é possível usar qualquer framework de testes que possa ser executado a partir de um código Python.

Definindo o executor de testes

Please, see the release notes Por convenção, um executor de testes deve ser chamado `run_tests`. O único requisito é que ele tenha os mesmos argumentos que o executor de testes do Django:

run_tests (*test_labels*, *verbosity=1*, *interactive=True*, *extra_tests=[]*)

`test_labels` é uma lista de strings descrevendo os testes a serem rodados. Cada string pode ter uma das três formas a seguir:

- `app.TestCase.test_method` – Executa um único método de teste em um test case.
- `app.TestCase` – Executa todos os métodos de teste em um test case.
- `app` – Procura e roda todos os testes na aplicação nomeada.

Se `test_labels` tem o valor `None`, o executor de testes deve procurar e rodar os testes para todas as aplicações em `INSTALLED_APPS`.

`verbosity` determina a quantidade de notificações e informações de debug que serão apresentadas no console; 0 significa sem saída, 1 é saída normal, e 2 é saída detalhada.

Se `interactive` é `True`, a test suite tem permissão para pedir ao usuário por instruções quando é executada. Um exemplo desse comportamento seria pedir permissão para excluir um teste existente do banco de dados. Se `interactive` é `False`, a test suite deve ser capaz de rodar sozinha, sem intervenção manual.

`extra_tests` é uma lista de instâncias extra de `TestCase` para adicionar à suite, que é rodada pelo executor de testes. Esses testes extras são executados além dos descobertos nos módulos listados em `module_list`.

Este método deve retornar o número de testes que falharam.

Utilitários de testes

Para ajudar na criação de seu próprio executor de testes, o Django possui alguns métodos utilitários no módulo `django.test.utils`.

setup_test_environment ()

Executa quaisquer configurações prévias aos testes, como a instalação de instrumentação para o sistema de templates e a configuração do `SMTPConnection` fictício.

teardown_test_environment ()

Executa quaisquer tarefas globais de encerramento após o término dos testes, como remover a magia negra do sistema de templates e restaurar os serviços normais de e-mail.

The creation module of the database backend (`connection.creation`) also provides some utilities that can be useful during testing.

create_test_db (*verbosity=1*, *autoclobber=False*)

Cria um novo banco de dados e roda o `syncdb` nele.

`verbosity` tem o mesmo comportamento que em `run_tests()`.

`autoclobber` descreve o comportamento que acontecerá se um banco de dados com o mesmo nome que o banco de teste é encontrado:

- Se `autoclobber` é `False`, será pedido ao usuário para aprovar a destruição do banco de dados existente. `sys.exit` é chamado se o usuário não aprovar.
- Se `autoclobber` é `True`, o banco de dados será destruído sem consultar o usuário.

Returns the name of the test database that it created.

`create_test_db()` tem o efeito colateral de modificar `settings.DATABASE_NAME` para bater com o nome do banco de dados de teste. `create_test_db()` agora retorna o nome do banco de dados de teste.

`destroy_test_db`(*old_database_name*, *verbosity=1*)

Destrói o banco de dados cujo nome está no parâmetro de configuração `DATABASE_NAME` e restaura o valor de `DATABASE_NAME` para o nome fornecido.

verbosity tem o mesmo comportamento que em `run_tests()`.

Autenticação de Usuário no Django

O Django vem com um sistema de autenticação de usuário. Ele manipula contas de usuário, grupos, permissões e sessões de usuário baseados em cookie. Este documento explica como as coisas funcionam.

Visão geral

O sistema de autenticação consiste em:

- Usuários
- Permissões: Flags binário (yes/no) designando quando um usuário pode executar uma certa tarefa.
- Grupos: Uma forma genérica de aplicar labels e permissões para mais de um usuário.
- Mensagens: Uma forma simples de enfileirar mensagens para um dado usuário.

Instalação

O suporte de autenticação é empacotado como uma aplicação Django em `django.contrib.auth`. Para instalá-lo, faça o seguinte:

1. Coloque `'django.contrib.auth'` na sua configuração `INSTALLED_APPS`.
2. Execute o comando `manage.py syncdb`.

Note que o arquivo padrão `settings.py` criado pelo `django-admin.py startproject` inclui `'django.contrib.auth'` no `INSTALLED_APPS` por conveniência. Se seu `INSTALLED_APPS` já contém `'django.contrib.auth'`, sinta-se livre para executar o `manage.py syncdb` novamente; você pode rodar este comando quantas vezes você quiser, e cada vez ele somente instalará o que for necessário.

O comando `syncdb` cria as tabelas necessárias no banco de dados, cria objetos de permissão para todas as aplicações instaladas que precisem deles, e abre um prompt para você criar uma conta de super usuário na primeira vez em que rodá-lo.

Uma vez que você tenha feito estes passos, é isso.

Usuários

`class models.User`

Referência de API

Campos

`class models.User`

O objetos *User* possuem os seguintes campos:

username

Obrigatório. 30 caracteres ou menos. Somente caracteres alfanuméricos (letras, dígitos e underscores).

first_name

Opcional. 30 caracteres ou menos.

last_name

Opcional. 30 caracteres ou menos.

email

Opcional. Endereço de e-mail.

password

Obrigatório. Um hash da senha, e metadados sobre a senha. (O Django não armazena a senha pura.) Senhas puras podem ser arbitrariamente longas e podem conter qualquer caracter. Veja a seção “Senhas” abaixo.

is_staff

Booleano. Determina se este usuário pode acessar o site admin.

is_active

Booleano. Determina se esta conta de usuário deve ser considerada ativa. Seta este flag para `False` ao invés de deletar as contas.

Isto não controla se o usuário pode ou não logar-se. Nada no caminho da autenticação verifica o flag `is_active`, então se você deseja rejeitar um login baseado no `is_active` estando `False`, cabe a você verificar isto no seu próprio view de login. Entretanto, verificação de permissões usando métodos como `has_perm()` averiguam este flag e sempre retornarão `False` para usuários inativos.

is_superuser

Booleano. Determina que este usuário tem todas as permissões sem explicitamente atribuí-las.

last_login

Um datetime do último login do usuário. É setado para a data/hora atual por padrão.

date_joined

Um datetime determinando quando a conta foi criada. É setada com a data/hora atual por padrão quando a conta é criada.

Métodos

`class models.User`

Os objetos *User* tem dois campos muitos-para-muitos: `models.User.groups` e `user_permissions`. O objetos *User* podem acessar seus objetos relacionados da mesma forma como qualquer outro *models do Django*:

```
myuser.groups = [group_list]
myuser.groups.add(group, group, ...)
myuser.groups.remove(group, group, ...)
myuser.groups.clear()
myuser.user_permissions = [permission_list]
```

```
myuser.user_permissions.add(permission, permission, ...)
myuser.user_permissions.remove(permission, permission, ...)
myuser.user_permissions.clear()
```

Além destes métodos automáticos da API, os objetos *User* tem os seguintes métodos customizados:

is_anonymous()

Sempre retorna *False*. Esta é a forma de diferenciar os objetos *User* e *AnonymousUser*. Geralmente, você deve preferir usar o método *is_authenticated()* para isto.

is_authenticated()

Sempre retorna *True*. Esta é a forma de dizer se o usuário foi autenticado. Isto não implica em quaisquer permissões, e não verifica se o usuário está ativo - ele somente indica que o usuário forneceu um nome e senha válidos.

get_full_name()

Retorna o *first_name* mais o *last_name*, com um espaço entre eles.

set_password(raw_password)

Seta a senha do usuário a partir de uma dada string, preocupe-se com em gerar um hash da senha. Não salva objeto *User*.

check_password(raw_password)

Retorna *True* se a string é a senha correta para o usuário. (Este se encarrega de tirar o hash da senha e fazer a comparação.)

set_unusable_password()

Please, see the release notes Marca o usuário como não tendo senha definida. Isto não é o mesmo que ter uma senha em branco. O *check_password()* para este usuário nunca retornará *True*. Não salva o objeto *User*.

Você pode precisar disto se a autenticação para sua aplicação ocorre contra uma fonte externa como um diretório LDAP.

has_usable_password()

Please, see the release notes Retorna *False* se *set_unusable_password()* foi chamado para este usuário.

get_group_permissions()

Retorna uma lista de strings de permissão que o usuário tem, através de seus grupos.

get_all_permissions()

Retorna uma lista de strings de permissão que o usuário possui, ambos através de permissões de grupo e usuário.

has_perm(perm)

Retorna *True* se o usuário tem uma permissão específica, onde *perm* é no formato "<nome da aplicação>.<nome do model em minúsculo>". Se o usuário é inativo, este método sempre retornará *False*.

has_perms(perm_list)

Retorna *True* se o usuário tem cada uma das permissões especificadas, onde cada permissão está no formato "*package.codename*". Se o usuário estiver inativo, este método sempre retornará *False*.

has_module_perms(package_name)

Retorna *True* se o usuário tem alguma permissão no dado pacote (a label da aplicação Django). Se o usuário está inativo, este método sempre retornará *False*.

get_and_delete_messages()

Retorna uma lista de objetos *Message* da fila do usuário e deleta as mensagens da fila.

email_user(subject, message, from_email=None)

Envia um e-mail para o usuário. Se *from_email* é *None*, o Django usa o *DEFAULT_FROM_EMAIL*.

get_profile()

Retorna um profile específico de site para este usuário. Lança `django.contrib.auth.models.SiteProfileNotAvailable` se o site atual não permitir profiles. Para informações de como definir um profile de usuário para um site específico, veja a seção sobre *armazenando informações adicionais de usuário* abaixo.

Gerente de funções

class models.UserManager

O model *User* tem um gerenciador customizado que possui os seguintes helpers:

create_user(username, email, password=None)

Cria, salva e retorna um *User*. Os atributos *username*, *email* e *password* são setados com os dados fornecidos, e o *User* recebe *is_active=True*.

Se nenhuma senha é fornecida, *set_unusable_password()* será chamado.

Veja *Criando usuários* para mais exemplos de uso.

make_random_password(length=10, allowed_chars='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ')

Retorna uma senha randômica com o comprimento e string, de caracteres permitidos, fornecida. (Note que o valor padrão de *allowed_chars* não contém letras que causem confusão, incluindo:

- i, l, I, e 1 (letra minúscula i, letra minúscula L, letra maiúscula i, e o número um)
- o, O, e 0 (letra maiúscula o, letra minúscula o, e zero)

Uso básico

Criando usuários

A forma mais básica de criar usuário é usar o helper *create_user()* que vem com o Django:

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('john', 'lennon@thebeatles.com', 'johnpassword
↪')

# Neste ponto, user é um objeto User que já foi salvo no banco de dados.
# Você pode continuar a mudar seus atributos se você quiser mudar outros
# campos.
>>> user.is_staff = True
>>> user.save()
```

Você pode também criar usuários usando o site admin do Django. Assumindo que você o tenha habilitado e adicionado-o a URL `/admin/`, a página “Add user” está em `/admin/auth/user/add`. Você deve também ver um link para “Users” na seção “Auth” da página principal do admin. A página “Add user” do admin é diferente de uma página normal do admin, pois exige que você escolha um nome de usuário e senha antes de permitir você editar o resto dos do usuário.

Também perceba: se você quiser que sua própria conta esteja habilitada a criar usuários usando o site admin do Django, você precisará dar a você mesmo permissão para adicionar e mudar usuários (i.e., as permissões “Add user” e “Change user”). Se sua conta tem permissão para adicionar usuários mas não para mudá-las, você não será hábil a adicionar usuários. Porque? Por que se você tem permissão para adicionar usuários, você tem o poder de criar super usuários, que podem então, por sua vez, mudar outros usuários. Então o Django requer permissão para adicionar e mudar como uma ligeira medida de segurança.

Mudando senhas

Mude uma senha com o *set_password()*:

```
>>> from django.contrib.auth.models import User
>>> u = User.objects.get(username__exact='john')
>>> u.set_password('new password')
>>> u.save()
```

Não sete o atributo `password` diretamente a menos que você saiba o que está fazendo. Isto é explicado na próxima seção.

Senhas

O atributo `password` de um objeto `User` é uma string neste formato:

```
hashtype$salt$hash
```

Que é o hastype, salt e hash, separados por um caracter dolar ('\$').

Hashtype é cada `sha1` (padrão), `md5` ou `crypt` – o algoritmo usado para construir o hash sem volta da senha. O salt é uma string randômica que salga a senha pura criando o hash. Note que o método `crypt` é somente suportado em plataformas que possuem o módulo padrão do Python `crypt` disponível. Suporte para o módulo `crypt` é novo no Django 1.0. Por exemplo:

```
sha1$a1976$a36cc8cbf81742a8fb52e221aaeab48ed7f58ab4
```

As funções `set_password()` e `check_password()` manipulam a configuração e verificam estes valores por trás das cenas.

Nas versões anteriores do Django, como a 0.90, usavá-se simples hashes MD5 sem salt nas senhas. Por compatibilidade descendentes, estes ainda são suportados; eles serão convertidos automaticamente para o novo estilo na primeira vez que o `check_password()` funcionar corretamente para um certo usuário.

Usuários Anonymous

`class models.AnonymousUser`

O `AnonymousUser` é uma classe que implementa a interface `User`, com estas diferenças:

- `id` é sempre `None`.
- `is_staff` e `is_superuser` são sempre `False`.
- `is_active` é sempre `False`.
- `groups` e `user_permissions` são sempre vazios.
- `is_anonymous()` retorna `True` ao invés de `False`.
- `is_authenticated()` retorna `False` ao invés de `True`.
- `has_perm()` sempre retorna `False`.
- `set_password()`, `check_password()`, `save()`, `delete()`, `set_groups()` e `set_permissions()` lançam um `NotImplementedError`.

Ná prática, você provavelmente não precisará usar objetos `AnonymousUser` você mesmo, mas eles serão utilizados por requisições Web, como explicado na próxima seção.

Criando superusers

O comando `manage.py createsuperuser` é novo. O `manage.py syncdb` conduz você a criar um superuser na primeira vez que você rodá-lo depois de adicionar `'django.contrib.auth'` ao seu `INSTALLED_APPS`. Se você precisa criar um superuser um tempo depois, você pode usar o utilitário de linha de comando:

```
manage.py createsuperuser --username=joe --email=joe@example.com
```

Você será perguntado por uma senha. Depois que você entra com uma, o usuário será criado imediatamente. Se você deixar desligado as opções `--username` ou `--email`, ele pedirá a você estes valores.

Se você estiver usando uma versão antiga do Django, a forma antiga de criar um superuser na linha de comando ainda funciona:

```
python /path/to/django/contrib/auth/create_superuser.py
```

...onde `/path/to` é o caminho para a base de código do Django dentro do seu sistema de arquivos. O comando `manage.py` é preferido porque ele adivinha o caminho correto do ambiente por você.

Armazenando informações adicionais sobre usuários

Se você quiser armazenar informações adicionais relacionadas aos seus usuários, o Django fornece um método para especificar um model relacionado específico do site – denominado um “user profile” – para este propósito.

Para fazer uso deste recurso, devina um model com campos para as informações adicionais que você desejaria armazenar, ou métodos adicionais que você gostaria de ter disponível, e também adicione uma `ForeignKey` de seu model para o model `User`, especificado com `unique=True` para assegurar que somente uma instância seja criada para cada `User`.

Para indicar que este model é um model de “user profile” para um dado site, preencha no `settings.py` o `AUTH_PROFILE_MODULE` com a string contendo os seguintes itens, separados por um ponto:

1. O nome da aplicação (case sensitive) em que o model do “user profile” é definido (em outras palavras, o nome que foi passado para o `manage.py startapp` para criar a aplicação).
2. O nome da classe do model (não case sensitive).

Por exemplo, se o model do profile era uma classe chamada `UserProfile` e foi definida dentro de uma aplicação chamada `accounts`, a configuração apropriada seria:

```
AUTH_PROFILE_MODULE = 'accounts.UserProfile'
```

Quando um model de “user profile” foi definido e especificado desta maneira, cada objeto `User` terá um método – `get_profile()` – que retorna a instância do model “user profile” associado com o `User`.

O método `get_profile()` não cria o profile, se ele não existe. Você precisa registrar um manipulador para o sinal `django.db.models.signals.post_save` sobre o model `User`, e, no manipulador, se `created=True`, criar o profile associado.

Para mais informações, veja o [Capítulo 12 do livro do Django](#).

Autenticação em requisições Web

Até agora, este documento ocupou-se com as APIs de baixo nível para manipular objetos relacionados com autenticação. Num nível mais alto, o Django pode ligar este framework de autenticação dentro de seu sistema de *objetos de requisição*.

Primeiro, instale os middlewares `SessionMiddleware` e `AuthenticationMiddleware` adicionando-os ao seu `MIDDLEWARE_CLASSES`. Veja a *documentação de sessões* para mais informações.

Uma vez que tenha estes middlewares instalados, você estará apto a acessar `request.user` nos views. O `request.user` dará a você um objeto `User` representando o usuário atualmente logado. Se um usuário não estiver logado, o `request.user` conterá uma instância do `AnonymousUser` (veja a seção anterior). Você pode utilizar `is_authenticated()`, tipo:

if `request.user.is_authenticated()`: # Faça algo para usuários autenticados.

else: # Faça algo para usuários anônimos.

Como logar um usuário

O Django fornece duas funções no `django.contrib.auth`: `authenticate()` e `login()`.

`authenticate()`

Para autenticar um dado username e senha, use `authenticate()`. Ele recebe dois argumentos nomeados, `username` e `password`, e retorna um objeto `User` se a senha for válida para este username. Se a senha for inválida `authenticate()` retorna `None`. Exemplo:

```
from django.contrib.auth import authenticate
user = authenticate(username='john', password='secret')
if user is not None:
    if user.is_active:
        print "Você forneceu um username e senha corretos!"
    else:
        print "Sua conta foi desabilitada!"
else:
    print "Seu username e senha estavam incorretos."
```

`login()`

Para logar um usuário, em uma view, use `login()`. ele recebe um objeto `HttpRequest` e objeto `User`. A função `login()` salva o ID do usuário na sessão, usando o framework de sessão do Django, então, como mencionado acima, você precisará assegurar-se de ter o middleware de sessão instalado.

Este exemplo mostra como você pode usar ambos `authenticate()` e `login()`:

```
from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(username=username, password=password)
    if user is not None:
        if user.is_active:
            login(request, user)
            # Redirecione para uma página de sucesso.
        else:
            # Retorna uma mensagem de erro de 'conta desabilitada' .
    else:
        # Retorna uma mensagem de erro 'login inválido'.
```

Calling `authenticate()` first

Quando você está manualmente logando um usuário, você *deve* chamar `authenticate()` antes de chamar `login()`. A função `authenticate()` seta um atributo sobre o `User` observando que o backend de autenticação, autenticou o usuário com sucesso (veja a [documentação de backends](#) para detalhes), e esta informação é necessária mais tarde, durante o processo de login.

Verificando uma senha de usuário manualmente

`check_password()`

Se você gostaria de autenticar manualmente um usuário comparando uma senha, em texto plano, com a senha em hash no banco de dados, use a função conveniente `django.contrib.auth.models.check_password()`. Ela recebe dois argumentos: a senha em texto plano, e o valor completo do campo `password` no banco de dados, eles serão conferidos, e retornará `True` se combinarem, ou `False` caso contrário.

Como deslogar um usuário

`logout()`

Para deslogar um usuário que esteja logado via `django.contrib.auth.login()`, use `django.contrib.auth.logout()` dentro de seu view. Ele recebe um objeto `HttpRequest` e não retorna nada. Exemplo:

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    # Redirecione para uma página de sucesso.
```

Note que `logout()` não gera qualquer erro se o usuário não estiver logado. Chamar `logout()` agora limpa os dados da sessão. Quando você chama `logout()`, os dados da sessão para a requisição atual são completamente apagados. Todo dado existente é removido. Isto é para prevenir que outra pessoa, que use o mesmo navegador, após logar-se acesse dados da sessão do usuário anterior. Se você deseja colocar algo na sessão que estará disponível para o usuário imediatamente após deslogar, faça isto *depois* de chamar `django.contrib.auth.logout()`.

Limitando acesso para usuários logados

O caminho bruto

O simples, caminho bruto para limitar o acesso a páginas é usar o método `request.user.is_authenticated()` e redicionar o usuário para página de login:

```
from django.http import HttpResponseRedirect

def my_view(request):
    if not request.user.is_authenticated():
        return HttpResponseRedirect('/login/?next=%s' % request.path)
    # ...
```

...ou mostrar uma mensagem de erro:

```
def my_view(request):
    if not request.user.is_authenticated():
        return render_to_response('myapp/login_error.html')
    # ...
```

O decorador `login_required`

`decorators.login_required()`

Como um atalho, você pode usar o conveniente decorador `login_required()`:

```
from django.contrib.auth.decorators import login_required

def my_view(request):
    # ...
my_view = login_required(my_view)
```

Aqui um exemplo equivalente, usando a sintaxe mais compacta de decorador introduzido no Python 2.4:

```
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    # ...
```

O `login_required()` também recebe um parâmetro opcional `redirect_field_name`. Exemplo:

```
from django.contrib.auth.decorators import login_required

def my_view(request):
    # ...
my_view = login_required(redirect_field_name='redirect_to')(my_view)
```

Novamente, um exemplo equivalente da sintaxe mais compacta do decorador introduzido no Python 2.4:

```
from django.contrib.auth.decorators import login_required

@login_required(redirect_field_name='redirect_to')
def my_view(request):
    # ...
```

O `login_required()` faz o seguinte:

- Se o usuário não estiver logado, redireciona-o para `settings.LOGIN_URL` (/accounts/login/ por padrão), passando a URL absoluta atual na query string como `next` ou o valor do `redirect_field_name`. Por exemplo: /accounts/login/?next=/pools/3/.
- Se o usuário estiver logado, executa o view normalmente. O código do view é livre para assumir que o usuário está logado.

Note que você precisará mapear o view apropriado do Django para `settings.LOGIN_URL`. Por exemplo, usando o padrão, adicione a seguinte linha no seu URLconf:

```
(r'^accounts/login/$', 'django.contrib.auth.views.login'),
```

`views.login(request[, template_name, redirect_field_name])`

O que o `django.contrib.auth.views.login` faz:

- Se chamado via GET, ele mostra um formulário de login que posta para mesma URL. Mais sobre isso daqui a pouco.
- Se chamado via POST, ele tenta logar o usuário. Se o login é feito, o view redireciona para a URL especificada em `next`. Se `next` não é fornecido, ele redireciona para `settings.LOGIN_REDIRECT_URL` (que é por padrão /accounts/profile/). Se o login não é feito, ele mostra novamente o formulário de login.

É de sua responsabilidade fornecer ao formulário de login um template chamado `registration/login.html` por padrão. A este template é passado quatro variáveis de contexto:

- `form`: Um objeto `Form` representando o formulário de login. Veja a [documentação do forms](#) para saber mais sobre objetos `Form`.
- `next`: A URL para redirecionar depois que o login for realizado com sucesso. Ele pode conter uma query string, também.
- `site`: O Site atual, de acordo com a configuração `SITE_ID`. Se você não tiver o framework site instalado, isto setará uma instância de `RequestSite`, que deriva do nome do site e domínio vindos do `HttpRequest` atual.
- `site_name`: Um alias para `site.name`. Se você não tiver o framework site instalado, isto setará o valor de `request.META['SERVER_NAME']`. Para saber mais sobre sites, veja [The “sites” framework](#).

Se você preferir não chamar o template `registration/login.html`, você pode passar o parâmetro `template_name` via argumentos extras para o view, no seu URLconf. Por exemplo, esta linha do URLconf poderia usar `myapp/login.html` em seu lugar:

```
(r'^accounts/login/$', 'django.contrib.auth.views.login', {'template_name':
    ↪ 'myapp/login.html'}),
```

Você pode também especificar o nome do campo GET que contém a URL para redirecioná-lo depois que login passar `redirect_field_name` para o view. Por padrão, o campo é chamado de `next`.

Aqui temos uma amostra de template `registration/login.html` que você pode usar como um ponto de início. Ele assume que você tem um template `base.html` que define um bloco `content`:

```
{% extends "base.html" %}

{% block content %}

{% if form.errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}

<form method="post" action="{% url django.contrib.auth.views.login %}">
<table>
<tr>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</tr>
<tr>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</tr>
</table>

<input type="submit" value="login" />
<input type="hidden" name="next" value="{{ next }}" />
</form>

{% endblock %}
```

Outros view embutidos

Além do view `login()`, o sistema de autenticação inclui uns outros views embutidos, localizados em `django.contrib.auth.views`:

`views.logout(request[, next_page, template_name])`
Desloga um usuário.

Argumentos opcionais:

- `next_page`: A URL para redirecionar o usuário depois de deslogá-lo.
- `template_name`: O nome completo do template que será mostrado depois de deslogar o usuário. Será utilizado o padrão `registration/logged_out.html` se nenhum argumento for fornecido.

Contexto do template:

- `title`: A string “Logged out”, localizada.

`views.logout_then_login(request[, login_url])`
Desloga um usuário, e então redireciona-o para a página de login.

Argumentos opcionais:

- `login_url`: a URL da página de login para redirecioná-lo. Será o padrão `settings.LOGIN_URL` se não for fornecido.

`views.password_change(request[, template_name, post_change_redirect])`
Permite um usuário mudar sua senha.

Argumentos opcionais:

- `template_name`: o nome completo do template que mostrará o formulário que muda a senha. Será o padrão `registration/password_change_form.html` se não for fornecido.

- `post_change_redirect`: a URL para redirecioná-lo depois da mudança da senha.

Contexto do template:

- `form`: O formulário de mudança de senha.

`views.password_change_done(request[, template_name])`
a página mostrada depois que um usuário muda sua senha.

Argumentos opcionais:

- `template_name`: o nome completo do template a se usar. Este será o padrão `registration/password_change_done.html` se não for fornecido.

`views.password_reset(request[, is_admin_site, template_name, email_template_name, password_reset_form, token_generator, post_reset_redirect])`

Permite um usuário resetar sua senha, e enviá-lo uma nova senha por e-mail.

Argumentos opcionais:

- `template_name`: O nome completo do template a se usar para mostrar o formulário de reset. Este será o padrão `registration/password_reset_form.html` se não for fornecido.
- `email_template_name`: O nome completo do template a se usar para gerar o e-mail com a nova senha. Este será o padrão `registration/password_reset_email.html` se não for fornecido.
- `password_reset_form`: Form que será usado para setar a senha. O padrão é `SetPasswordForm`.
- `token_generator`: Instância de classe para verificar a senha. Este será o padrão `default_token_generator`, é uma instância do `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `post_reset_redirect`: A URL para redirecionar depois de uma mudança de senha com sucesso.

Contexto do template:

- `form`: O formulário para resetar a senha do usuário.

`views.password_reset_done(request[, template_name])`
A páina mostrada depois que um usuário reseta sua senha.

Argumentos opcionais:

- `template_name`: O nome completo do template a se usar. Este será o padrão `registration/password_reset_done.html` se não for fornecido.

`views.redirect_to_login(next[, login_url, redirect_field_name])`

Redireciona para a página de login, e então volta para outra URL depois de um login bem sucedido.

Argumentos obrigatórios:

- `next`: A URL para redirecioná-lo depois de um login bem sucedido.

Argumentos opcionais:

- `login_url`: A URL da página de login para redirecioná-lo. Este será o padrão `settings.LOGIN_URL` se não for fornecido.
- `redirect_field_name`: O nome do campo GET contendo a URL para ser redirecionada depois de deslogar. Sobrescreve `next` se o dado parâmetro GET for passado.

`password_reset_confirm(request[, uidb36, token, template_name, token_generator, set_password_form, post_reset_redirect])`

Apresenta um formulário para introduzir uma nova senha.

Argumentos opcionais:

- `uidb36`: O id o usuário codificado na base 36. Por padrão é `None`.
- `token`: O token para verificar que a senha é válida. O padrão será `None`.

- `template_name`: O nome completo do template que mostrara o view de confirmação de senha. O valor padrão é `registration/password_reset_confirm.html`.
- `token_generator`: Instância da classe que verificar a senha. Este será por padrão `default_token_generator`, ele é uma instância do `django.contrib.auth.tokens.PasswordResetTokenGenerator`.
- `set_password_form`: Form que será usado para setar a senha. Este será por padrão `SetPasswordForm`.
- `post_reset_redirect`: A URL para redirecioná-lo depois da resetar a senha. Este será por padrão `None`.

`password_reset_complete` (*request* [, *template_name*])

Apresenta um view que informa ao usuário que a senha foi mudada com sucesso.

Argumentos opcionais:

- `template_name`: O nome completo do template para mostrar no view. Este será por padrão `registration/password_reset_complete.html`.

Fomulários embutidos

Se você não quiser usar os views embutidos, mas quiser o conforto de não ter que escrever formulários para esta funcionalidade, o sistema de autenticação fornece vários formulários embutidos localizados em `django.contrib.auth.forms`:

class `AdminPasswordChangeForm`

Um formulário usado na interface de administração para mudar a senha do usuário.

class `AuthenticationForm`

Um formulário para logar um usuário.

class `PasswordChangeForm`

Um formulário para permitir a um usuário mudar sua senha.

class `PasswordResetForm`

Um formulário para resetar uma senha de usuário e enviar um email com a nova senha para ele.

class `SetPasswordForm`

Um formulário que deixa um usuário mudar sua senha sem introduzir a senha antiga.

class `UserChangeForm`

Um formulário usado no admin para mudar informações de um usuário e suas permissões.

class `UserCreationForm`

Um formulário para criar um novo usuário.

Limitando o acesso para usuário logados que passam um teste

Para limitar o acesso baseado em certas permissões ou algum outro teste, você faria essencialmente a mesma coisa como descrito na seção anterior.

A forma simples é executar seu teste sobre `request.user` no view diretamente. Por exemplo, este view verifica para estar certo de que o usuário está logado e tem a permissão `polls.can_vote`:

```
def my_view(request):
    if not (request.user.is_authenticated() and request.user.has_perm('polls.can_
↪vote')):
        return HttpResponse("You can't vote in this poll.")
    # ...
```

`decorators.user_passes_test` ()

Como um atalho, você pode usar o prático decorador `user_passes_test`:

```
from django.contrib.auth.decorators import user_passes_test

def my_view(request):
    # ...
my_view = user_passes_test(lambda u: u.has_perm('polls.can_vote'))(my_view)
```

Nós estamos usando este teste particular como um exemplo relativamente simples. No entanto, se você só quer testar se uma permissão é disponível para o usuário, você pode usar o decorador `permission_required()`, descrito mais tarde neste documento.

Aqui tem a mesma coisa, usando a sintaxe de decorador do Python 2.4:

```
from django.contrib.auth.decorators import user_passes_test

@user_passes_test(lambda u: u.has_perm('polls.can_vote'))
def my_view(request):
    # ...
```

O `user_passes_test()` recebe um argumento obrigatório: uma função que recebe um objeto `User` e retorna True se o usuário tem permissão para ver a página. Notee que o `user_passes_test()` não verifica automaticamente se o `User` não é anônimo.

O `user_passes_test()` recebe um argumento opcional `login_url`, que permite você especificar a URL para sua página de login (`settings.LOGIN_URL` por padrão). Example in Python 2.3 syntax:

```
from django.contrib.auth.decorators import user_passes_test

def my_view(request):
    # ...
my_view = user_passes_test(lambda u: u.has_perm('polls.can_vote'), login_url='/login/') (my_view)
```

Exemplo na sintaxe do Python 2.4:

```
from django.contrib.auth.decorators import user_passes_test

@user_passes_test(lambda u: u.has_perm('polls.can_vote'), login_url='/login/')
def my_view(request):
    # ...
```

O decorador `permission_required`

`decorators.permission_required()`

Esta é uma tarefa relativamente comum, checar se um usuário tem uma permissão em particular. Por esta razão, o Django provê um atalho para este caso: o decorador `permission_required()`. Usando este decorador, o exemplo anterior pode ser escrito assim:

```
from django.contrib.auth.decorators import permission_required

def my_view(request):
    # ...
my_view = permission_required('polls.can_vote')(my_view)
```

Como no método `User.has_perm()`, nomes de permissões tem a forma "<nome da aplicação>. <nome do model em minúsculo>" (i.e. `polls.choice` para um model `Choice` na aplicação `polls`).

Perceba que `permission_required()` também recebe um parametro opcional `login_url`. Exemplo:

```
from django.contrib.auth.decorators import permission_required

def my_view(request):
    # ...
my_view = permission_required('polls.can_vote', login_url='/loginpage/') (my_
    ↪view)
```

Como no decorador `login_required()`, o padrão para `login_url` é `settings.LOGIN_URL`.

Limitando o acesso à visões genéricas

Para limitar o acesso a uma *visão genérica*, escreva um fino wrapper a volta do view, e aponte seu URLconf para seu wrapper ao invés da visão genérica. Por exemplo:

```
from django.views.generic.date_based import object_detail

@login_required
def limited_object_detail(*args, **kwargs):
    return object_detail(*args, **kwargs)
```

Permissões

O Django vem com um sistema simples de permissões. Ele fornece uma forma de atribuir permissões para usuários e grupos de usuários específicos.

Ele é usado pelo admin do Django, mas você também é bem-vindo a usá-lo no seu próprio código.

O admin do Django usa as seguintes permissões:

- Access to view the “add” form and add an object is limited to users with the “add” permission for that type of object.
- Access to view the change list, view the “change” form and change an object is limited to users with the “change” permission for that type of object.
- Access to delete an object is limited to users with the “delete” permission for that type of object.

Permissões são setadas globalmente para um tipo de objeto, não para uma instância de objeto específica. Por exemplo, é possível dizer “Maria pode mudar notícias”, mas não é atualmente possível dizer “Maria pode mudar notícias, mas somente as que ela mesma criou” ou “Maria pode somente mudar notícias que tenham um certo status, data de publicação ou ID.” Esta última funcionalidade é algo que os desenvolvedores do Django estão discutindo atualmente.

Permissões padrão

Quando `django.contrib.auth` é listado no seu `INSTALLED_APPS`, ele garantirá três permissões padrão – adicionar, editar, deletar – são criadas para cada model do Django definidos em suas aplicações instaladas.

Estas permissões serão criadas quando você rodar `manage.py syncdb`; na primeira vez que você rodar `syncdb` depois de adicionar o `django.contrib.auth` ao `:setting:INSTALLED_APPS`, as permissões padrão serão criadas para todos os models instalados anteriormente, assim como para qualquer novo model que seja criado. Logo, ele cria permissões padrão para novos models toda vez que você executa `manage.py syncdb`.

Permissões customizadas

Para criar permissões customizadas para um dado objeto de model, use o *atributo Meta do model*, `permissions`.

Este model de exemplo cria três permissões customizadas:

```
class USCitizen(models.Model):
    # ...
    class Meta:
        permissions = (
            ("can_drive", "Can drive"),
            ("can_vote", "Can vote in elections"),
            ("can_drink", "Can drink alcohol"),
        )
```

A única coisa que isto faz é criar estas permissões extra quando você executa o `manage.py syncdb`.

Referência de API

`class models.Permission`

Assim como usuários, as permissões são implementadas num model do Django que fica em `django/contrib/auth/models.py`.

Campos

Permission objects have the following fields:

`models.Permission.name`

Obrigatório. 50 caracteres ou menos. Exemplo: 'Can vote'.

`models.Permission.content_type`

Obrigatório. Uma referência da tabela de banco de dados `django_content_type`, que contém um dado para cada model do Django instalado.

`models.Permission.codename`

Obrigatório. 100 caracteres ou menos. Exemplo: 'can_vote'.

Métodos

O objetos *Permission* tem um método de acesso a dados padrão como qualquer outro *model do Django*.

Dados de autenticação em templates

O usuário atualmente logado e suas permissões estão disponíveis no *contexto do template* quando você usa o `RequestContext`.

Technicality

Tecnicamente, estas variáveis são somente disponibilizadas no contexto do template se você usar `RequestContext` e se sua configuração `TEMPLATE_CONTEXT_PROCESSORS` contiver `"django.core.context_processors.auth"`, que é o padrão. Para mais, veja a *documentação do RequestContext*.

Usuários

Quando se renderiza um template `RequestContext`, o usuário atualmente logado, qualquer instância de *User*, é armazenada na variável de template `{{ user }}`:


```
{% if user.is_authenticated %}
    <p>Welcome, {{ user.username }}. Thanks for logging in.</p>
{% else %}
    <p>Welcome, new user. Please log in.</p>
{% endif %}
```

Esta variável de contexto de template não fica disponível se um `RequestContext` não estiver sendo utilizado.

Permissões

As permissões do usuário atualmente logado são armazenadas na variável de template `{{ perms }}`. Isto é uma instância de `django.core.context_processors.PermWrapper`, que é um proxy de permissões amigável para templates.

No objeto `{{ perms }}`, na forma de atributo singular, é um proxy para o `User.has_module_perms`. Este exemplo mostraria `True` se o usuário logado tivesse qualquer permissão na aplicação `foo`:

```
{{ perms.foo }}
```

Na forma de atributo com segundo nível é um proxy para `User.has_perm`. Este exemplo poderia mostrar `True` se o usuário logado tivesse a permissão `foo.can_vote`:

```
{{ perms.foo.can_vote }}
```

Deste modo, você pode checar permissões no template usando declarações `{% if %}`:

```
{% if perms.foo %}
    <p>Você tem permissão para fazer algo na aplicação foo.</p>
    {% if perms.foo.can_vote %}
        <p>Você pode votar!</p>
    {% endif %}
    {% if perms.foo.can_drive %}
        <p>Você pode dirigir!</p>
    {% endif %}
{% else %}
    <p>Você não tem permissão para fazer nada na aplicação foo.</p>
{% endif %}
```

Grupos

Grupos são uma forma genérica de categorizar usuários e então aplicar permissões, ou algum outro rótulo, para estes usuários. Um usuário pode pertencer a qualquer quantidade de grupos.

Um usuário em um grupo automaticamente possui as permissões garantidas para esse grupo. Por exemplo, se o grupo `Editores do site` tem permissão `can_edit_home_page`, qualquer usuário neste grupo terá esta permissão.

Além das permissões, grupos são uma forma conveniente de categorizar usuários e dar-lhes algum rótulo, ou funcionalidade aplicada. Por exemplo, você poderia criar um grupo `'Usuários especiais'`, e você poderia escrever seu código que poderia, dizer, dar-lhes acesso a uma porção de usuários que são membros do seu site, ou enviar-lhes mensagens de e-mail privadas.

Mensagens

O sistema de mensagens é uma forma leve de enfileirar mensagens para certos usuários.

Uma mensagem é associada com um `User`. Não há conceito de expiração ou timestamps.

As mensagens são usadas pelo admin do Django depois de ações bem sucedidas. Por exemplo, "A enquete Foo foi criada com sucesso." é uma mensagem.

A API é simples:

```
models.User.message_set.create(message)
```

Para criar uma nova mensagem, use `user_obj.message_set.create(message='message_text')`.

To retrieve/delete messages, use `Para receber/deletar mensagens, use user_obj.get_and_delete_messages(), que retorna uma lista de objetos Message da fila do usuário (se tiver alguma) e deleta a mensagem da fila.`

Neste exemplo de view, o sistema salva uma mensagem para o usuário depois de criar uma playlist:

```
def create_playlist(request, songs):
    # Cria a playlist com as músicas fornecidas.
    # ...
    request.user.message_set.create(message="Sua playlist foi adicionada com_
↪sucesso.")
    return render_to_response("playlists/create.html",
                              context_instance=RequestContext(request))
```

Quando você usa o RequestContext, o usuário atualmente logado e suas mensagens são disponibilizadas no *contexto do template* como uma variável de template `{{ messages }}`. Aqui tem um exemplo de código de template que mostra mensagens:

```
{% if messages %}
<ul>
    {% for message in messages %}
    <li>{{ message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

Note que RequestContext chama `get_and_delete_messages()` no plano de fundo, então qualquer mensagem será deletada mesmo que você não a tenha mostrado.

Finalmente, repare que este framework de mensagem somente funciona com usuário no banco de dados. Para enviar mensagens para usuários anônimos, use o *framework de sessão*.

Other authentication sources

The authentication that comes with Django is good enough for most common cases, but you may have the need to hook into another authentication source – that is, another source of usernames and passwords or authentication methods.

For example, your company may already have an LDAP setup that stores a username and password for every employee. It'd be a hassle for both the network administrator and the users themselves if users had separate accounts in LDAP and the Django-based applications.

So, to handle situations like this, the Django authentication system lets you plug in another authentication sources. You can override Django's default database-based scheme, or you can use the default system in tandem with other systems.

Especificando backends de autenticação

Por trás das cenas, o Django mantém uma lista de “backends de autenticação” que ele verifica para autenticação. Quando alguém chama `django.contrib.auth.authenticate()` – como descrito no *como logar um usuário* acima – o Django tenta autenticar através de todos estes backends de autenticação. Se o primeiro método de autenticação falha, o Django tenta o segundo, e segue assim até que tenha tentado todos.

A lista de backends de autenticação para usar é especificada no `AUTHENTICATION_BACKENDS`. Este deve ser uma tupla com caminhos Python que apontam para as classes que sabem como autenticar. Essas classes podem estar em qualquer lugar no caminho do Python.

Por padrão, `AUTHENTICATION_BACKENDS` é setado para:

```
('django.contrib.auth.backends.ModelBackend',)
```

Que é o esquema básico de autenticação que verifica o banco de dados de usuários do Django.

A ordem no `AUTHENTICATION_BACKENDS` importa, então se o mesmo nome de usuário e senha são válidos para vários backends, o Django irá parar o processamento no primeiro resultado positivo.

Note: Uma vez que um usuário fora autenticado, o Django armazena qual backend foi utilizado para autenticar o usuário na sessão do mesmo, e re-usa o mesmo backend para tentativas subsequentes de autenticação deste usuário. Isto efetivamente significa que a fonte de autenticação são chacheadas, então se você mudar o `AUTHENTICATION_BACKENDS`, precisará limpar os dados da sessão caso necessite forçar a re-autenticação do usuário usando diferentes métodos. Uma forma simples para fazer isto é simplesmente executar `Session.objects.all().delete()`.

Escrevendo um backend de autenticação

Um backend de autenticação é uma classe que implementa dois métodos: `get_user(user_id)` e `authenticate(**credentials)`.

O método `get_user` recebe um `user_id` – que poderia ser um nome de usuário, ID do banco de dados ou o que seja – e retorna um objeto `User`.

O método `authenticate` recebe credenciais como argumentos nomeados. Na maioria das vezes, ele parecerá com isto:

```
class MyBackend:
    def authenticate(self, username=None, password=None):
        # Verifica o nome/senha e retorna um User.
```

Mas ele poderia também autenticar um token, tipo:

```
class MyBackend:
```

```
    def authenticate(self, token=None): # Verifica o token e retorna um User.
```

Either way, `authenticate` should check the credentials it gets, and it should return a `User` object that matches those credentials, if the credentials are valid. If they're not valid, it should return `None`.

De ambas as formas, o `authenticate` deve verificar as referências que receber, e deve retornar um objeto `User` que combina com essas referências, se as referências forem válidas. Se elas não forem válidas, ele deve retornar `None`.

O sistema de administração do Django é hermeticamente acoplado ao objeto `User` do Django descrito no início deste documento. Por agora, a melhor forma de tratar isto é criar um objeto `User` do Django para cada usuário existente para seu backend (e.g. no seu diretório LDAP, seu banco de dados SQL externo, etc.) Você pode também escrever um script para fazer isto com antecedência, ou seu método `authenticate` pode fazê-lo na primeira vez que o usuário se logar.

Aqui tem um exemplo de backend que autentica usando as variáveis `username` e `password` definidas no seu arquivo `settings.py` e cria um objeto `User` na primeira vez que autentica:

```
from django.conf import settings
from django.contrib.auth.models import User, check_password

class SettingsBackend:
    """
```

```

Autentica usando as configurações ADMIN_LOGIN e ADMIN_PASSWORD.

Use o nome de login, e um hash da senha. Por exemplo:

ADMIN_LOGIN = 'admin'
ADMIN_PASSWORD = 'sha1$4e987$afbcf42e21bd417fb71db8c66b321e9fc33051de'
"""

def authenticate(self, username=None, password=None):
    login_valid = (settings.ADMIN_LOGIN == username)
    pwd_valid = check_password(password, settings.ADMIN_PASSWORD)
    if login_valid and pwd_valid:
        try:
            user = User.objects.get(username=username)
        except User.DoesNotExist:
            # Cria um usuário. Note que nós podemos setar uma senha
            # para tudo, porque ela não será verificad; o password
            # vindo do settings.py será.
            user = User(username=username, password='get from settings.py')
            user.is_staff = True
            user.is_superuser = True
            user.save()
        return user
    return None

def get_user(self, user_id):
    try:
        return User.objects.get(pk=user_id)
    except User.DoesNotExist:
        return None

```

Manipulando autorização em backends customizados

Backends customizados de autenticação pode fornecer suas próprias permissões.

O model do usuário delegará funções de permissão (`get_group_permissions()`, `get_all_permissions()`, `has_perm()`, e `has_module_perms()`) para qualquer backend de autenticação que implemente estas funções.

As permissões dadas para o usuário serão o super conjunto de todas as permissões retornadas por todos os backends. Isto é, o Django concede uma permissão para um usuário que qualquer backend concederia.

O simples backend acima poderia aplicar permissões para o mágico admin de forma muito simples:

```

class SettingsBackend:

    # ...

    def has_perm(self, user_obj, perm):
        if user_obj.username == settings.ADMIN_LOGIN:
            return True
        else:
            return False

```

Isto dá permissões completas ao usuário acessado no exemplo acima. Advertindo que as todas as funções de autenticação do backend recebem o objeto do usuário como um argumento, e eles também aceitam os mesmos argumentos fornecidos para as funções associadas do `django.contrib.auth.models.User`.

Uma implementação completa de autorização pode ser encontrada em `django/contrib/auth/backends.py`, que é o backend padrão e consulta a tabela `auth_permission` a maior parte do tempo.

O framework de cache do Django

Uma dilema essencial dos sites dinâmicos vem a ser o próprio fato de serem dinâmicos. Cada vez que um usuário requisita uma página, o servidor web faz todo o tipo de cálculos – consultas a bancos de dados, renderização de templates e lógica de negócio – para criar a página que o seu visitante vê. Isso tem um custo de processamento muito maior que apenas a leitura de arquivos estáticos no disco.

Para a maior parte dos aplicativos Web, esse overhead não é um problema. A maior parte das aplicações Web não são o washingtonpost.com ou oouslashdot.org; são simplesmente sites pequenos a médio com tráfego equivalente. Mas para aplicações de porte mério para grante, é essencial eliminar toda a sobrecarga possível.

É onde entra o cache.

Fazer o cache de algo é gravar o resultado de um cálculo custoso para que você não tenha de executar o cálculo da próxima vez. Aqui está um pseudocódigo explicando como isso funcionaria para uma página Web gerada dinamicamente:

```
tente encontrar uma página no cache para tal URL
se a página estiver no cache:
    retorne a página do cache
se não:
    gere a página
    guarde a página gerada no cache (para a próxima vez)
    retorne a página gerada
```

O Django vem com um sistema de cache robusto que permite que você guarde as páginas dinâmicas para que elas não tenham de ser calculadas a cada requisição. Por conveniência, Django oferece diferentes níveis de granularidade de cache: Você pode fazer o cache da saída de views específicas, você pode fazer o cache somente das partes que são difíceis de produzir, ou pode fazer o cache do site inteiro.

O Django também trabalha com caches do tipo “upstream”, como o Squid (<http://www.squid-cache.org/>) e cache baseado em navegador. Esses são tipos de cache que você não controla diretamente mas para os quais fornece dicas (via cabeçalhos HTTP) sobre quais partes do seu site devem ser mantidas em cache, e como.

Configurando o cache

O sistema de cache requer uma pequena configuração. Você deve informar ao Django onde os seus dados em cache estarão – se em um banco de dados, no sistema de arquivos ou diretamente na memória. Essa é uma decisão importante que afeta a performance do seu cache; sim, alguns tipos de cache são mais rápidos que outros.

Sua preferência de cache vai na configuração `CACHE_BACKEND` no seu arquivo de configurações. Aqui vai uma explicação de todos os valores disponíveis para `CACHE_BACKEND`.

Memcached

De longe, o mais rápido e mais eficiente tipo de cache disponível no Django, Memcached é um framework de cache inteiramente baseado em memória originalmente desenvolvido para lidar com as altas cargas no LiveJournal.com e subsequentemente tornada open-sourced pela Danga Interactive. É usado por sites como o Facebook ou Wikipedia para reduzir o acesso a banco de dados e aumentar a performance do site drasticamente.

O Memcached está disponível de graça em <http://danga.com/memcached/>. Ele executa como um daemon, para o qual é alocada uma quantidade específica de RAM. Tudo o que ele faz é fornecer uma interface rápida para adição, busca e remoção de dados arbitrários do cache. Todos os dados são gravados diretamente na memória, então não existe sobrecarga de banco de dados ou uso do filesystem.

Após a instalação do Memcached, você precisa instalar as bibliotecas do Memcached para Python, que não estão empacotadas com o Django. Duas versões estão disponíveis. Selecione e instale *um* dos seguintes módulos:

- A opção mais rápida é um módulo chamado `cmemcache`, disponível em <http://gijsbert.org/cmcache/>.
- Se você não pode instalar o `cmemcache`, você pode instalar o `python-memcached`, disponível em <ftp://ftp.tummy.com/pub/python-memcached/>. Se essa URL não é mais válida, apenas vá ao site do Memcached (<http://www.danga.com/memcached/>) e obtenha as bibliotecas Python da seção “Client APIs”.

A opção `cmemcache` é nova no 1.0. Anteriormente, somente o `python-memcached` era suportado. Para usar o Memcached com Django, configure o `CACHE_BACKEND` para `memcached://ip:port/`, onde `ip` é o endereço IP do daemon do Memcached e o `port` é a porta onde o Memcached está rodando.

Nesse exemplo, o Memcached está rodando em localhost (127.0.0.1) na porta 11211:

```
CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

Uma característica excelente do Memcached é sua habilidade de compartilhar o cache em diversos servidores. Isso significa que você pode executar daemons do Memcached em diversas máquinas, e o programa irá tratar o grupo de máquinas como um *único* cache, sem a necessidade de duplicar os valores do cache em cada máquina. Para aproveitar essa funcionalidade, inclua todos os endereços de servidores em `CACHE_BACKEND`, separados por ponto e vírgula.

Nesse exemplo, o cache é compartilhado por instâncias do Memcached rodando nos endereços IP 172.19.26.240 (porta 11211) e 172.19.26.242 (porta 11212), e 172.19.26.244 (porta 11213):

```
CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11212;172.19.26.↪244:11213/'
```

O cache baseado em memória tem uma desvantagem: Como os dados em cache estão na memória, serão perdidos se o seu servidor travar. Claramente, a memória não foi feita para armazenamento permanente de dados, então não confie no cache em memória como sua única fonte de armazenamento de dados. Na verdade, nenhum dos backends de cache do Django deve ser usado para armazenamento permanente – devem ser usados como soluções para cache, e não armazenamento – mas reafirmamos isso aqui porque o cache em memória é realmente temporário.

Cache em banco de dados

Para usar uma tabela do banco de dados como o seu backend de cache, primeiro crie uma tabela de cache em seu banco de dados com o seguinte comando:

```
python manage.py createcachetable [cache_table_name]
```

...onde `[cache_table_name]` é o nome da tabela no banco de dados a ser criada. (Esse nome pode ser qualquer um, desde que seja um nome válido de tabela e esse nome ainda não esteja sendo usado no seu banco de dados.) Esse comando cria uma única tabela no seu banco de dados, no formato apropriado que o sistema de cache em banco de dados do Django espera encontrar.

Uma vez que você tenha configurado a tabela do banco de dados, configure o seu `CACHE_BACKEND` para `"db://tablename"`, onde `tablename` é o nome da tabela. Nesse exemplo, o nome da tabela de cache é `my_cache_table`:

```
CACHE_BACKEND = 'db://my_cache_table'
```

O cache em banco de dados usará o mesmo banco de dados especificado em seu arquivo de configuração. Você não pode usar um banco de dados diferente para sua tabela de cache.

O cache em banco de dados funciona melhor se você tem um servidor de banco de dados rápido e bem indexado.

Cache em sistema de arquivos

Para gravar os itens em cache no sistema de arquivos, use o tipo de cache `"file://"` para `CACHE_BACKEND`. Por exemplo, para salvar os dados do cache em `/var/tmp/django_cache`, use essa configuração:

```
CACHE_BACKEND = 'file:///var/tmp/django_cache'
```

Perceba que existem três barras próximas ao início do exemplo. As primeiras são para o `file://`, e a terceira é o primeiro caractere do caminho do diretório, `/var/tmp/django_cache`. Se você está no Windows, coloque a letra do drive após o `file://`, dessa forma:

```
file://c:/foo/bar
```

O caminho do diretório deve ser absoluto – isso é, ele deve iniciar na raiz do seu sistema de arquivos. Não faz diferença se você põe a barra no final da configuração.

Assegure-se que o diretório apontado por essa configuração exista e tenha permissões de leitura e escrita pelo usuário do sistema que executa o seu servidor web. Continuando o exemplo acima, se o seu servidor web roda como o usuário `apache`, tenha certeza que o diretório `/var/tmp/django_cache` exista e tenha permissão de leitura e escrita pelo usuário `apache`.

Cada valor de cache será gravado em um arquivo separado cujos conteúdos são os dados servidos pelo cache em um formato serializado (“pickled”), usando o módulo `pickle` do Python. Cada nome de arquivo é a chave do cache, escapado para uso seguro em sistema de arquivos.

Cache em memória local

Se você quer as vantagens de velocidade de executar um cache em memória mas não pode executar o Memcached, considere o backend de cache em memória local. Esse cache é multi-processo e thread-safe. Para usá-lo, configure o `CACHE_BACKEND` para `"locmem:///"`. Por exemplo:

```
CACHE_BACKEND = 'locmem:///'
```

Note que cada processo irá ter sua própria instância privada de cache, o que significa que nenhum cache entre processos é possível. Isso obviamente significa que o cache em memória local não é muito eficiente em termos de memória, então provavelmente não é uma boa escolha para um ambiente de produção. É bom para desenvolvimento.

Cache falso (para desenvolvimento)

Finalmente, o Django vem com um cache “falso” que não faz cache realmente – ele apenas implementa a interface de cache sem fazer nada realmente.

Isso é útil se você tem um site em produção que usa cacheamento pesado em vários lugares, mas em desenvolvimento ou no ambiente de testes você não quer usar cache e não quer mudar o código para lidar com o último caso em especial. Para ativar o cache falso, configure o `CACHE_BACKEND` assim:


```
CACHE_BACKEND = 'dummy:///'
```

Usando um backend de cache personalizado

Please, see the release notes Apesar do Django suportar diversos sistemas de cache diferentes, algumas vezes você pode querer usar algum backend de cache personalizado. Para usar um backend externo de cache com o Django, use um caminho de importação de módulos do Python como a parte do esquema (a parte que vem antes do dois pontos inicial) da URI do `CACHE_BACKEND`, assim:

```
CACHE_BACKEND = 'path.to.backend:/'
```

Se você está construindo o seu próprio backend, você pode usar os backends padrão de cache como implementações de referência. Você irá encontrar o código no diretório `django/core/cache/backends/` dos fontes do Django.

Nota: Você deveria usar os backends de cache incluídos com o Django, a não ser que você tenha uma razão muito boa, como um host que não os suporta. Eles foram bem testados e são fáceis de usar.

Argumentos do `CACHE_BACKEND`

Cada tipo de cache pode receber argumentos. Eles são informados em um estilo semelhante a query-string na configuração `CACHE_BACKEND`. Os seguintes argumentos são válidos:

- **timeout:** O timeout padrão, em segundos, a ser usado para o cache. O padrão é 300 segundos (5 minutos).
- **max_entries:** Para os backends `locmem`, `filesystem` e `database`, o número máximo de entradas permitidas no cache antes dos valores antigos serem removidos. O valor padrão desse argumento é 300.
- **cull_frequency:** A fração de entradas que são limpas do cache quando `max_entries` é atingido. A razão real é `1/cull_percentage`, então configure o `cull_percentage=2` para limpar metade das entradas quando o valor de `max_entries` for atingido.

Um valor de 0 para o `cull_frequency` significa que o cache todo será limpo quando `max_entries` for atingido. Isso torna a limpeza *muito* mais rápida, a custo de mais perdas no cache.

Nesse exemplo, o `timeout` está configurado para 60:

```
CACHE_BACKEND = "memcached://127.0.0.1:11211/?timeout=60"
```

Nesse exemplo, o `timeout` é 30 e `max_entries` é 400:

```
CACHE_BACKEND = "locmem:///?timeout=30&max_entries=400"
```

Argumentos inválidos são silenciosamente ignorados, assim como valores inválidos para argumentos conhecidos.

O cache por site

(versões anteriores do Django forneciam apenas um único `CacheMiddleware` no lugar das duas partes descritas abaixo). Uma vez que o cache esteja configurado, a forma mais simples de usá-lo é fazer o cache do seu site inteiro. Você precisa adicionar `'django.middleware.cache.UpdateCacheMiddleware'` e `'django.middleware.cache.FetchFromCacheMiddleware'` as suas configurações de `MIDDLEWARE_CLASSES`, como nesse exemplo:

```
MIDDLEWARE_CLASSES = (
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
)
```

Note: No, isso não é um erro de digitação: o middleware “update” deve aparecer primeiro na list, e o middleware “fetch” por último. Os detalhes são um pouco obscuros, mas veja *Ordem das MIDDLEWARE_CLASSES* abaixo se você quiser entender os detalhes.

Então, adicione as configurações necessárias ao seu arquivo de configurações do Django:

- `CACHE_MIDDLEWARE_SECONDS` – O número de segundos em que cada página deve permanecer em cache.
- `CACHE_MIDDLEWARE_KEY_PREFIX` – Se o cache é compartilhado entre diversos sites usando a mesma instalação do Django, configure isso para o nome do site, ou alguma outra string única que identifique essa instância do Django, para prevenir colisões de cache. Use uma string vazia se você não se importa.

O middleware de cache faz o cache de cada página que não tenha parâmetros de GET ou POST. Opcionalmente, se a configuração `CACHE_MIDDLEWARE_ANONYMOUS_ONLY` for `True`, somente requisições anônimas (i.e., aquelas feitas por usuários não autenticados) serão cacheadas. Esse é um modo simples e eficiente para desabilitar o cache para quaisquer páginas específicas de usuários (incluindo a interface de administração do Django). Note que se você usou o `CACHE_MIDDLEWARE_ANONYMOUS_ONLY`, você deve certificar-se de ter ativado o `AuthenticationMiddleware`.

Adicionalmente, o middleware de cache adiciona alguns cabeçalhos em cada `HttpResponse`:

- Configura o cabeçalho `Last-Modified` para a data/hora atuais quando uma nova versão da página (não cacheada) é resquitada.
- Configura o cabeçalho `Expires` para a hora atual mais o valor definido em `CACHE_MIDDLEWARE_SECONDS`.
- Configura o cabeçalho `Cache-Control` para dar um max age para a página – novamente, da configuração `CACHE_MIDDLEWARE_SECONDS`.

Veja *Middleware* para mais sobre middlewares. *Please, see the release notes* Se uma visão configura o seu próprio tempo de expiração de cache (i.e. tem uma seção `max-age` no seu cabeçalho `Cache-Control`) então a página será mantida em cache até o tempo de expiração, ao invés de `CACHE_MIDDLEWARE_SECONDS`. Usando os decoradores em `django.views.decorators.cache` você pode facilmente configurar o tempo de expiração de uma visão (usando o decorador `cache_control`) ou desabilitar o cache para uma visão (usando o decorador `never_cache`). Veja a seção *usando outros cabeçalhos* para mais informações sobre esses decoradores.

O cache por visão

Uma forma mais granular de usar o framework de cache é fazer o cache da saída de visões individuais. `django.views.decorators.cache` define um decorador `cache_page` que irá automaticamente fazer o cache da resposta da visão para você. É fácil de usar:

```
from django.views.decorators.cache import cache_page

def my_view(request):
    ...

my_view = cache_page(slashdot_this, 60 * 15)
```

Ou, usando a sintaxe de decorador do Python 2.4:

```
@cache_page(60 * 15)
def slashdot_this(request):
    ...
```

`cache_page` recebe um único argumento: o timeout do cache, em segundos. No exemplo acima, o resultado da visão `slashdot_this()` será mantido em cache por 15 minutos. (Note que escrevemos isso como `60 * 15` visando legibilidade. `60 * 15` será avaliado para 900 – isso é, 15 minutos multiplicados por 60 segundos por minuto.)

O cache por visão, como o cache por site, é indexado de acordo com a URL. Se múltiplas URLs apontam para a mesma visão, cada URL será cacheada separadamente. Continuando o exemplo `my_view`, se o seu URLconf for semelhante a isso:

```
urlpatterns = ('',
               (r'^foo/(\d{1,2})/$', my_view),
               )
```

então requisições para `/foo/1/` e `/foo/23/` serão cacheadas separadamente, conforme você pode esperar. Mas uma vez que uma URL em particular (ex: `/foo/23/`) tenha sido requisitada, acessos subsequentes a essa URL usarão o cache.

Especificando o cache por visão no URLconf

Os exemplos na seção anterior tem hard-coded o fato que a visão é cacheada, porque `cache_page` altera a função `my_view` in place. Esse método acopla suas visões ao sistema de cache, o que não é o ideal por diversas razões. Por exemplo, você pode querer reusar as funções de visão em outro site, sem cache, ou você pode querer distribuir suas visões para pessoas que podem querer usá-las sem cache. A solução para esses problemas é especificar o cache por visão no URLconf ao invés de configurá-lo nas próprias funções.

Fazer isso é fácil: simplesmente embrulhe a função de visão com `cache_page` quando você se referir a ela no URLconf. Aqui está a nossa já conhecida URLconf:

```
urlpatterns = ('',
               (r'^foo/(\d{1,2})/$', my_view),
               )
```

Aqui está a mesma coisa, com `my_view` embrulhada em `cache_page`:

```
from django.views.decorators.cache import cache_page

urlpatterns = ('',
               (r'^foo/(\d{1,2})/$', cache_page(my_view, 60 * 15)),
               )
```

Se você usar esse método, não se esqueça de importar `cache_page` dentro de seu URLconf.

Cache de fragmento de template

Please, see the release notes Se você quer ainda mais controle, você pode também fazer cache de fragmentos de template usando a tag de template `cache`. Para permitir que seu template use essa tag, coloque `{% load cache %}` perto do topo do seu template.

A tag de template `{% cache %}` faz o cache do conteúdo do bloco por um certo período de tempo. Recebe ao menos dois argumentos: o timeout do cache, em segundos, e o nome que será dado ao cache de fragmento. Por exemplo:

```
{% load cache %}
{% cache 500 sidebar %}
```

```
.. sidebar ..
{% endcache %}
```

Algumas vezes você pode querer adicionar ao cache diversas cópias de um fragmento dependendo de algum dado dinâmico que apareça dentro do fragmento. Por exemplo, você pode querer uma cópia separada da barra lateral usada no exemplo anterior para cada usuário do seu site. Faça isso passando argumentos adicionais a template tag `{% cache %}` que unicamente identifiquem esse cache de fragmento:

```
{% load cache %}
{% cache 500 sidebar request.user.username %}
    .. sidebar for logged in user ..
{% endcache %}
```

É perfeitamente válido especificar mais de um argumento para identificar o fragmento. Simplesmente passe tantos argumentos quanto você precisar para `{% cache %}`.

O timeout de cache pode ser uma variável de template, desde que a variável de template seja um valor inteiro. Por exemplo, se a variável de template `my_timeout` está configurada para o valor 600, então os dois exemplos a seguir são equivalentes:

```
{% cache 600 sidebar %} ... {% endcache %}
{% cache my_timeout sidebar %} ... {% endcache %}
```

Essa característica é útil para evitar repetição em seus templates. Você pode configurar o timeout em uma variável, em um lugar, e apenas reutilizar esse valor.

A API de baixo nível do cache

Algumas vezes, o cache de uma página completa renderizada não é de grande valor, é pode ser até inconveniente.

Talvez, por exemplo, o seu site inclua uma visão cujos resultados dependam de algumas consultas intensivas ao banco de dados, cujos resultados mudem em diferentes intervalos. Nesse caso, não seria ideal usar o cache completo de página que as estratégias de cache por site ou por visão oferecem, porque você não quer fazer o cache do resultado completo (já que alguns dados mudam frequentemente), mas você ainda gostaria de manter em cache os resultados que mudam com menos frequência.

Para casos como esse, o Django expõe uma API simples de baixo nível de cache. Você pode usar essa API para guardar objetos no cache com quaisquer níveis de granularidade que você quiser. Você pode cacher quaisquer objetos Python que podem ser serializados com segurança: strings, dicionários, listas ou objetos do modelo, e por aí vai. (A maioria dos objetos comum do Python podem ser serializados através de pickle; veja a documentação do Python para mais informações sobre o pickling.)

Por exemplo, você pode descobrir que basta fazer o cache do resultado de uma consulta intensiva ao banco de dados. Em casos assim, você pode usar a API de baixo nível de cache para guardar os objetos em qualquer nível de granularidade que você quiser.

O módulo de cache, `django.core.cache` tem um objeto `cache` que é criado automaticamente a partir da configuração `CACHE_BACKEND`:

```
>>> from django.core.cache import cache
```

A interface básica é `set(key, value, timeout_seconds)` e `get(key)`:

```
>>> cache.set('my_key', 'hello, world!', 30)
>>> cache.get('my_key')
'hello, world!'
```

O argumento `timeout_seconds` é opcional e o seu padrão é o mesmo do argumento `timeout` na configuração `CACHE_BACKEND` (explicada acima).

Se o objeto não existe no cache, `cache.get()` retorna `None`:

```
# Espere 30 segundos para 'my_key' expirar...

>>> cache.get('my_key')
None
```

We advise against storing the literal value `None` in the cache, because you won't be able to distinguish between your stored `None` value and a cache miss signified by a return value of `None`.

`cache.get()` pode ter um argumento padrão. Isso especifica qual valor retornar se o objeto não existe no cache:

```
>>> cache.get('my_key', 'has expired')
'has expired'
```

Please, see the release notes Para adicionar uma cache somente se ela ainda não existir, use o método `add()`. Ele recebe os mesmos parâmetros que `set()`, mas não irá tentar atualizar o cache se a chave especificada já estiver presente:

```
>>> cache.set('add_key', 'Initial value')
>>> cache.add('add_key', 'New value')
>>> cache.get('add_key')
'Initial value'
```

Se você precisa saber se `add()` salvou um valor no cache, você pode verificar o valor retornado. O retorno será `True` se o valor foi gravado, e `False` se nada foi gravado.

Existe também uma interface chamada `get_many()` que acessa o cache apenas uma vez. `get_many()` retorna um dicionário com todas as chaves que você pediu que estavam no cache e não estavam expiradas):

```
>>> cache.set('a', 1)
>>> cache.set('b', 2)
>>> cache.set('c', 3)
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

Finalmente, você pode remover chaves diferentes com `delete()`. Essa é uma forma fácil de limpar o cache para um objeto em particular:

```
>>> cache.delete('a')
```

Please, see the release notes Você pode também incrementar ou decrementar uma chave que já existe usando os métodos `incr()` ou `decr()`, respectivamente. Por padrão, o valor existente em cache será incrementado ou decrementado em 1. Outros valores para incremento/decremento podem ser especificados fornecendo um argumento para a chamada `increment/decrement`. Um `ValueError` será lançado se você tentar incrementar ou decrementar uma chave de cache não existente.:

```
>>> cache.set('num', 1)
>>> cache.incr('num')
2
>>> cache.incr('num', 10)
12
>>> cache.decr('num')
11
>>> cache.decr('num', 5)
6
```

Note: `incr()/decr()` não tem garantia de serem atômicos. Nos backends que suportam incremento/decremento atômico (mais notavelmente, o backend `memcached`), operações de incremento e decremento serão atômicas. Porém, se o backend não provê nativamente uma operação de incremento/decremento, ela será implementada usando uma operação obter/atualizar de dois passos.

Caches Upstream

Até agora, esse documento concentrou-se no cache de seus *próprios* dados. Mas outro tipo de cache é relevante para o desenvolvimento Web: o cache executado por cache “upstream”. Esses são sistemas que fazem o cache de páginas para usuários antes mesmo da requisição chegar ao seu Web site.

Aqui estão alguns exemplos de cache upstream:

- Seu ISP pode fazer o cache de certas páginas, então se você requisitou uma página de <http://example.com/>, seu ISP poderia enviar a página sem ter de acessar o example.com diretamente. Os mantenedores de example.com não tem conhecimento desse cache; O ISP fica entre example.com e o seu navegador Web, lidando com todo o cache de forma transparente.
- Seu Web site Django pode ficar atrás de um *proxy cache* como o proxy Squid (<http://www.squid-cache.org/>), que faz cache de páginas para performance. Nesse caso, cada requisição seria tratada primeiramente pelo proxy e seria passada para a sua aplicação somente se necessário.
- Seu navegador web também faz cache. Se uma página envia os cabeçalhos apropriados, seu navegador irá usar a cópia local (em cache) para requisições subsequentes para essa página, sem nem mesmo contatar a página Web novamente para ver se o conteúdo mudou.

Caches upstream são um potencializador de eficiência interessante, mas há um perigo nisso: muitos conteúdos de páginas web diferem baseados na autenticação e em diversas outras variáveis, e sistemas de cache que servem páginas baseadas em URL cegamente poderiam expor dados incorretos ou sensíveis para visitantes subsequentes a essas páginas.

Por exemplo, digamos que você opere um serviço de webmail, e o conteúdo da página “inbox” obviamente depende do usuário que está logado. Se um ISP cegamente cacheia seu site, então o primeiro usuário que logou através desse ISP poderia ter seu conteúdo da caixa de entrada exposto aos próximos visitantes do site. Isso não é legal.

Felizmente, o HTTP fornece uma solução para esse problema: um número de cabeçalhos HTTP existem para instruir os cache upstream a diferir seus conteúdos de cache dependendo de certas variáveis definidas, e como dizer aos mecanismos de cache como não cachear páginas particulares. Veremos alguns desses cabeçalhos na seção a seguir.

Usando cabeçalhos Vary

O cabeçalho `Vary` define quais cabeçalhos de requisição um mecanismo de cache deve levar em conta ao construir sua chave de cache. Por exemplo, se o conteúdo do site de uma página web depende da preferência de idioma do usuário, dizemos que a página “varia no idioma.”

Por padrão, o sistema de cache do Django cria suas chaves de acesso usando o caminho da requisição (ex: `"/stories/2005/jun/23/bank_robbed/"`). Isso significa que cada requisição a essa URL irá usar a mesma versão do cache, não importando as diferentes de agente de usuário, como cookies ou preferência de idioma. Porém, se essa página produz conteúdos diferentes baseados em alguma diferença de cabeçalhos de requisição – como um cookie, ou um idioma, ou um agente de usuário – você irá precisar usar o cabeçalho `Vary` para dizer aos mecanismos de cache que a saída da página depende dessas coisas.

Para fazer isso no Django, use o decorador de visão de conveniência `vary_on_headers`, assim:

```
from django.views.decorators.vary import vary_on_headers

# Sintaxe do Python 2.3.
def my_view(request):
    # ...
my_view = vary_on_headers(my_view, 'User-Agent')

# Sintaxe de decorador do Python 2.4 ou superior.
@vary_on_headers('User-Agent')
```

```
def my_view(request):  
    # ...
```

Nesse caso, um mecanismo de cache (como o próprio middleware de cache do Django) irá cachear uma versão diferente da página para cada agente de usuário único.

A vantagem de usar o decorador `vary_on_headers` ao invés de configurar o cabeçalho `Vary` manualmente (usando algo como `response['Vary'] = 'user-agent'`) é que o decorador *adiciona* algo ao cabeçalho `Vary` (que pode já existir), ao invés de sobrescrevê-lo e potencialmente apagar algo que já estava lá.

Você pode passar diversos cabeçalhos para `vary_on_headers()`:

```
@vary_on_headers('User-Agent', 'Cookie')  
def my_view(request):  
    # ...
```

Isso diz aos cache upstream para variar em *ambos* os cabeçalhos, o que significa que cada combinação de agente de usuário e cookie irá ter seu próprio valor de cache. Por exemplo, uma requisição com o agente de usuário Mozilla e o valor de cookie `foo=bar` será considerado diferente de uma requisição com um agente de usuário Mozilla e o valor de cookie `foo=ham`.

Como a variação baseada em cookie é um caso comum, existe um decorador `vary_on_cookie`. Essas duas views são equivalentes:

```
@vary_on_cookie  
def my_view(request):  
    # ...  
  
@vary_on_headers('Cookie')  
def my_view(request):  
    # ...
```

Os cabeçalhos que você passa para `vary_on_headers` não são sensíveis a caso. `"User-Agent"` é a mesma coisa que `"user-agent"`.

Você pode também usar uma função auxiliar, `django.utils.cache.patch_vary_headers`, diretamente. Essa função configura ou adiciona ao cabeçalho `Vary`. Por exemplo:

```
from django.utils.cache import patch_vary_headers  
def my_view(request):  
    # ...  
    response = render_to_response('template_name', context)  
    patch_vary_headers(response, ['Cookie'])  
    return response
```

`patch_vary_headers` recebe uma instância de `HttpResponse` como seu primeiro argumento e uma lista/tupla de nomes de cabeçalho não sensíveis a caso como seu segundo argumento.

Para mais sobre os cabeçalhos `Vary`, veja a [especificação oficial do Vary](#).

Controlando o cache: Usando outros cabeçalhos

Outros problemas com o cache são a privacidade de dados e a questão de onde os dados devem ser armazenados em uma cascata de caches.

Um usuário normalmente encontra dois tipos de cache: o do seu próprio navegador (um cache privado) e o do seu provedor (um cache público). Um cache público é usado por muitos usuários e controlado por mais alguém. Isso coloca problemas com dados sensíveis—você não quer, por exemplo, que o número da sua conta bancária seja gravada em um cache público. Assim, aplicações web precisam de uma forma de dizer aos caches quais dados são privados e quais dados são públicos.

A solução é indicar que o cache de página seja “particular”. Para fazer isso no Django, use o decorador de visão `cache_control`. Exemplo:

```
from django.views.decorators.cache import cache_control

@cache_control(private=True)
def my_view(request):
    # ...
```

Esse decorador cuida de enviar o cabeçalho HTTP apropriado nos bastidores.

Existem algumas outras formas de controlar os parâmetros de cache. Por exemplo, o HTTP permite que aplicações façam o seguinte:

- Define o tempo máximo que uma página deva permanecer em cache.
- Especifica se um cache deve sempre verificar por novas versões da página, exibindo o conteúdo em cache apenas quando não houve mudanças. (Alguns caches podem entregar conteúdo em cache mesmo se a página no servidor mudou – simplesmente porque a cópia do cache ainda não expirou.)

No Django, use o decorador de visão `cache_control` para especificar esses parâmetros de cache. Nesse exemplo, `cache_control` diz ao cache para revalidar o cache em cada acesso e para guardar versões em cache em no máximo por 3600 segundos:

```
from django.views.decorators.cache import cache_control

@cache_control(must_revalidate=True, max_age=3600)
def my_view(request):
    # ...
```

Qualquer diretiva HTTP Cache-Control é válida em `cache_control()`. Aqui está uma lista completa:

- `public=True`
- `private=True`
- `no_cache=True`
- `no_transform=True`
- `must_revalidate=True`
- `proxy_revalidate=True`
- `max_age=num_seconds`
- `s_maxage=num_seconds`

Para explicação das diretivas HTTP de Cache-Control, veja a [Especificação de Cache-Control](#).

(Note que o middleware de cache já configura o tempo máximo (`max-age`) do cache com o valor da configuração `CACHE_MIDDLEWARE_SETTINGS`. Se você usa um `max_age` personalizado no decorador `cache_control`, o decorador terá precedência sobre a configuração, e os valores do cabeçalho serão mesclados corretamente.)

Se você quer usar cabeçalhos para desabilitar todo o cache, `django.views.decorators.cache.never_cache` é um decorador de visão que adiciona cabeçalhos que garantem que a resposta não será posta em cache por navegadores ou outros caches. Exemplo:

```
from django.views.decorators.cache import never_cache

@never_cache
def myview(request):
    # ...
```


Outras otimizações

O Django vem com alguns outros middlewares que podem ajudar a otimizar a performance do seu aplicativo:

- `django.middleware.http.ConditionalGetMiddleware` adiciona suporte para GET condicional em navegadores modernos, baseado nos cabeçalhos ETag e Last-Modified.
- `django.middleware.gzip.GZipMiddleware` comprime conteúdo para todos os navegadores modernos reduzindo o uso de banda e o tempo de transferência.

Ordem das MIDDLEWARE_CLASSES

Se você está usando o middleware de cache, é importante colocar cada parte dentro do lugar correto nas configurações de `MIDDLEWARE_CLASSES`. Isso porque o cabeçalho de cache precisa saber por quais cabeçalhos ele irá variar o armazenamento de cache. Middleware sempre adiciona algo ao cabeçalho `Vary` quando ele pode.

`UpdateCacheMiddleware` executa durante a fase de resposta, onde os middlewares são executados em ordem reversa, então um item no topo da lista sempre executa *por último* durante a fase de resposta. Assim, você precisa garantir que `UpdateCacheMiddleware` apareça *antes* de quaisquer outros middlewares que possam adicionar algo ao cabeçalho `Vary`. Os seguintes módulos de middleware fazem isso so:

- `SessionMiddleware` adiciona `Cookie`
- `GZipMiddleware` adiciona `Accept-Encoding`
- `LocaleMiddleware` adiciona `Accept-Language`

`FetchFromCacheMiddleware`, por outro lado, executa durante a fase de requisição, onde os middlewares são aplicados do primeiro ao último, então um item no topo da lista executa *primeiro* durante a fase de requisição. O `FetchFromCacheMiddleware` também precisa executar depois de alguns outros middlewares atualizarem o cabeçalho `Vary`, assim `FetchFromCacheMiddleware` deve ser configurado *após* qualquer item que faça isso.

Sending e-mail

Although Python makes sending e-mail relatively easy via the `smtplib` library, Django provides a couple of light wrappers over it, to make sending e-mail extra quick.

The code lives in a single module: `django.core.mail`.

Quick example

In two lines:

```
from django.core.mail import send_mail

send_mail('Subject here', 'Here is the message.', 'from@example.com',
          ['to@example.com'], fail_silently=False)
```

Mail is sent using the SMTP host and port specified in the `EMAIL_HOST` and `EMAIL_PORT` settings. The `EMAIL_HOST_USER` and `EMAIL_HOST_PASSWORD` settings, if set, are used to authenticate to the SMTP server, and the `EMAIL_USE_TLS` setting controls whether a secure connection is used.

Note: The character set of e-mail sent with `django.core.mail` will be set to the value of your `DEFAULT_CHARSET` setting.

send_mail()

The simplest way to send e-mail is using the function `django.core.mail.send_mail()`. Here's its definition:

```
send_mail(subject, message, from_email, recipient_list, fail_silently=False,
           auth_user=None, auth_password=None)
```

The `subject`, `message`, `from_email` and `recipient_list` parameters are required.

- `subject`: A string.
- `message`: A string.

- `from_email`: A string.
- `recipient_list`: A list of strings, each an e-mail address. Each member of `recipient_list` will see the other recipients in the “To:” field of the e-mail message.
- `fail_silently`: A boolean. If it’s `False`, `send_mail` will raise an `smtplib.SMTPException`. See the [smtplib docs](#) for a list of possible exceptions, all of which are subclasses of `SMTPException`.
- `auth_user`: The optional username to use to authenticate to the SMTP server. If this isn’t provided, Django will use the value of the `EMAIL_HOST_USER` setting.
- `auth_password`: The optional password to use to authenticate to the SMTP server. If this isn’t provided, Django will use the value of the `EMAIL_HOST_PASSWORD` setting.

send_mass_mail()

`django.core.mail.send_mass_mail()` is intended to handle mass e-mailing. Here’s the definition:

```
send_mass_mail (datatuple, fail_silently=False, auth_user=None, auth_password=None)
```

`datatuple` is a tuple in which each element is in this format:

```
(subject, message, from_email, recipient_list)
```

`fail_silently`, `auth_user` and `auth_password` have the same functions as in `send_mail()`.

Each separate element of `datatuple` results in a separate e-mail message. As in `send_mail()`, recipients in the same `recipient_list` will all see the other addresses in the e-mail messages’ “To:” field.

send_mass_mail() vs. send_mail()

The main difference between `send_mass_mail()` and `send_mail()` is that `send_mail()` opens a connection to the mail server each time it’s executed, while `send_mass_mail()` uses a single connection for all of its messages. This makes `send_mass_mail()` slightly more efficient.

mail_admins()

`django.core.mail.mail_admins()` is a shortcut for sending an e-mail to the site admins, as defined in the [ADMINS](#) setting. Here’s the definition:

```
mail_admins (subject, message, fail_silently=False)
```

`mail_admins()` prefixes the subject with the value of the [EMAIL_SUBJECT_PREFIX](#) setting, which is “[Django] ” by default.

The “From:” header of the e-mail will be the value of the [SERVER_EMAIL](#) setting.

This method exists for convenience and readability.

mail_managers() function

`django.core.mail.mail_managers()` is just like `mail_admins()`, except it sends an e-mail to the site managers, as defined in the [MANAGERS](#) setting. Here’s the definition:

```
mail_managers (subject, message, fail_silently=False)
```

Examples

This sends a single e-mail to `john@example.com` and `jane@example.com`, with them both appearing in the “To:”:

```
send_mail('Subject', 'Message.', 'from@example.com',
         ['john@example.com', 'jane@example.com'])
```

This sends a message to `john@example.com` and `jane@example.com`, with them both receiving a separate e-mail:

```
datatuple = (
    ('Subject', 'Message.', 'from@example.com', ['john@example.com']),
    ('Subject', 'Message.', 'from@example.com', ['jane@example.com']),
)
send_mass_mail(datatuple)
```

Preventing header injection

Header injection is a security exploit in which an attacker inserts extra e-mail headers to control the “To:” and “From:” in e-mail messages that your scripts generate.

The Django e-mail functions outlined above all protect against header injection by forbidding newlines in header values. If any subject, `from_email` or `recipient_list` contains a newline (in either Unix, Windows or Mac style), the e-mail function (e.g. `send_mail()`) will raise `django.core.mail.BadHeaderError` (a subclass of `ValueError`) and, hence, will not send the e-mail. It’s your responsibility to validate all data before passing it to the e-mail functions.

If a message contains headers at the start of the string, the headers will simply be printed as the first bit of the e-mail message.

Here’s an example view that takes a subject, message and `from_email` from the request’s POST data, sends that to `admin@example.com` and redirects to “/contact/thanks/” when it’s done:

```
from django.core.mail import send_mail, BadHeaderError

def send_email(request):
    subject = request.POST.get('subject', '')
    message = request.POST.get('message', '')
    from_email = request.POST.get('from_email', '')
    if subject and message and from_email:
        try:
            send_mail(subject, message, from_email, ['admin@example.com'])
        except BadHeaderError:
            return HttpResponse('Invalid header found.')
        return HttpResponseRedirect('/contact/thanks/')
    else:
        # In reality we'd use a form class
        # to get proper validation errors.
        return HttpResponse('Make sure all fields are entered and valid.')
```

The EmailMessage and SMTPConnection classes

Please, see the release notes Django’s `send_mail()` and `send_mass_mail()` functions are actually thin wrappers that make use of the `EmailMessage` and `SMTPConnection` classes in `django.core.mail`. If you ever need to customize the way Django sends e-mail, you can subclass these two classes to suit your needs.

Note: Not all features of the `EmailMessage` class are available through the `send_mail()` and related

wrapper functions. If you wish to use advanced features, such as BCC'ed recipients, file attachments, or multi-part e-mail, you'll need to create `EmailMessage` instances directly.

This is a design feature. `send_mail()` and related functions were originally the only interface Django provided. However, the list of parameters they accepted was slowly growing over time. It made sense to move to a more object-oriented design for e-mail messages and retain the original functions only for backwards compatibility.

In general, `EmailMessage` is responsible for creating the e-mail message itself. `SMTPConnection` is responsible for the network connection side of the operation. This means you can reuse the same connection (an `SMTPConnection` instance) for multiple messages.

EmailMessage Objects

class `EmailMessage`

The `EmailMessage` class is initialized with the following parameters (in the given order, if positional arguments are used). All parameters are optional and can be set at any time prior to calling the `send()` method.

- `subject`: The subject line of the e-mail.
- `body`: The body text. This should be a plain text message.
- `from_email`: The sender's address. Both `fred@example.com` and `Fred <fred@example.com>` forms are legal. If omitted, the `DEFAULT_FROM_EMAIL` setting is used.
- `to`: A list or tuple of recipient addresses.
- `bcc`: A list or tuple of addresses used in the "Bcc" header when sending the e-mail.
- `connection`: An `SMTPConnection` instance. Use this parameter if you want to use the same connection for multiple messages. If omitted, a new connection is created when `send()` is called.
- `attachments`: A list of attachments to put on the message. These can be either `email.MIMEBase.MIMEBase` instances, or `(filename, content, mimetype)` triples.
- `headers`: A dictionary of extra headers to put on the message. The keys are the header name, values are the header values. It's up to the caller to ensure header names and values are in the correct format for an e-mail message.

For example:

```
email = EmailMessage('Hello', 'Body goes here', 'from@example.com',
                    ['to1@example.com', 'to2@example.com'], ['bcc@example.com'],
                    headers = {'Reply-To': 'another@example.com'})
```

The class has the following methods:

- `send(fail_silently=False)` sends the message, using either the connection that is specified in the `connection` attribute, or creating a new connection if none already exists. If the keyword argument `fail_silently` is `True`, exceptions raised while sending the message will be quashed.
- `message()` constructs a `django.core.mail.SafeMIMEText` object (a subclass of Python's `email.MIMEText.MIMEText` class) or a `django.core.mail.SafeMIMEMultipart` object holding the message to be sent. If you ever need to extend the `EmailMessage` class, you'll probably want to override this method to put the content you want into the MIME object.
- `recipients()` returns a list of all the recipients of the message, whether they're recorded in the `to` or `bcc` attributes. This is another method you might need to override when subclassing, because the SMTP server needs to be told the full list of recipients when the message is sent. If you add another way to specify recipients in your class, they need to be returned from this method as well.
- `attach()` creates a new file attachment and adds it to the message. There are two ways to call `attach()`:
 - You can pass it a single argument that is an `email.MIMEBase.MIMEBase` instance. This will be inserted directly into the resulting message.

- Alternatively, you can pass `attach()` three arguments: `filename`, `content` and `mimetype`. `filename` is the name of the file attachment as it will appear in the e-mail, `content` is the data that will be contained inside the attachment and `mimetype` is the optional MIME type for the attachment. If you omit `mimetype`, the MIME content type will be guessed from the filename of the attachment.

For example:

```
message.attach('design.png', img_data, 'image/png')
```

- `attach_file()` creates a new attachment using a file from your filesystem. Call it with the path of the file to attach and, optionally, the MIME type to use for the attachment. If the MIME type is omitted, it will be guessed from the filename. The simplest use would be:

```
message.attach_file('/images/weather_map.png')
```

Sending alternative content types

It can be useful to include multiple versions of the content in an e-mail; the classic example is to send both text and HTML versions of a message. With Django's e-mail library, you can do this using the `EmailMultiAlternatives` class. This subclass of `EmailMessage` has an `attach_alternative()` method for including extra versions of the message body in the e-mail. All the other methods (including the class initialization) are inherited directly from `EmailMessage`.

To send a text and HTML combination, you could write:

```
from django.core.mail import EmailMultiAlternatives

subject, from_email, to = 'hello', 'from@example.com', 'to@example.com'
text_content = 'This is an important message.'
html_content = '<p>This is an <strong>important</strong> message.</p>'
msg = EmailMultiAlternatives(subject, text_content, from_email, [to])
msg.attach_alternative(html_content, "text/html")
msg.send()
```

By default, the MIME type of the `body` parameter in an `EmailMessage` is `"text/plain"`. It is good practice to leave this alone, because it guarantees that any recipient will be able to read the e-mail, regardless of their mail client. However, if you are confident that your recipients can handle an alternative content type, you can use the `content_subtype` attribute on the `EmailMessage` class to change the main content type. The major type will always be `"text"`, but you can change it to the subtype. For example:

```
msg = EmailMessage(subject, html_content, from_email, [to])
msg.content_subtype = "html" # Main content is now text/html
msg.send()
```

SMTPConnection Objects

class SMTPConnection

The `SMTPConnection` class is initialized with the host, port, username and password for the SMTP server. If you don't specify one or more of those options, they are read from your settings file.

If you're sending lots of messages at once, the `send_messages()` method of the `SMTPConnection` class is useful. It takes a list of `EmailMessage` instances (or subclasses) and sends them over a single connection. For example, if you have a function called `get_notification_email()` that returns a list of `EmailMessage` objects representing some periodic e-mail you wish to send out, you could send this with:

```
connection = SMTPConnection() # Use default settings for connection
messages = get_notification_email()
connection.send_messages(messages)
```

Testing e-mail sending

There are times when you do not want Django to send e-mails at all. For example, while developing a website, you probably don't want to send out thousands of e-mails – but you may want to validate that e-mails will be sent to the right people under the right conditions, and that those e-mails will contain the correct content.

The easiest way to test your project's use of e-mail is to use a “dumb” e-mail server that receives the e-mails locally and displays them to the terminal, but does not actually send anything. Python has a built-in way to accomplish this with a single command:

```
python -m smtpd -n -c DebuggingServer localhost:1025
```

This command will start a simple SMTP server listening on port 1025 of localhost. This server simply prints to standard output all email headers and the email body. You then only need to set the `EMAIL_HOST` and `EMAIL_PORT` accordingly, and you are set.

For more entailed testing and processing of e-mails locally, see the Python documentation on the [SMTP Server](#).

Internacionalização

O Django tem um suporte completo para internacionalização dos textos dos códigos e templates. Aqui é mostrado como isto funciona.

Introdução

O objetivo da internacionalização é permitir que uma única aplicação web ofereça seus conteúdos e funcionalidades em múltiplas linguagens.

Você, o desenvolvedor Django, pode conseguir isto adicionando uns poucos hooks ao seu código Python e templates. Estes hooks, chamados **translation strings**, dizem ao Django: “Este texto deve ser traduzido para o idioma do usuário, se a tradução para este texto estiver disponível nesta língua.”

O Django se preocupa em usar estes hooks para traduzir as aplicações Web, em tempo de execução, conforme as preferências de idioma do usuário.

Essencialmente, o Django faz duas coisas:

- Ele permite que o desenvolvedor ou autor de templates especifique quais partes de sua aplicação podem ser traduzidas.
- Ele usa os hooks para traduzir a aplicação web para um usuário em particular de acordo com sua preferência.

Se você não necessita de internacionalização em sua aplicação

Os hooks de internacionalização do Django estão habilitados por padrão, o que significa que existe um consumo de processamento por parte do `i18n` em certas partes do framework. Se você não usa internacionalização, você deve gastar 2 segundos para setar `USE_I18N = False` no seu arquivo de configuração. Se o `USE_I18N` for `False`, então o Django irá realizar algumas otimizações, como não carregar o maquinário para internacionalização.

Você provavelmente desejará remover `'django.core.context_processors.i18n'` de sua configuração do `TEMPLATE_CONTEXT_PROCESSORS`.

Se você necessita de internacionalização: três passos

1. Insira as translation strings em seu código Python e nos templates.
2. Traduza essas strings, para as línguas que você quer suportar.
3. Ative o middleware de localização nas configurações do Django.

..admonition:: Por trás das cenas

O mecanismo de tradução do Django usa o módulo padrão `gettext` que acompanha o Python.

1. Como especificar as translation strings

Translation strings especificam “Este texto deve ser traduzido.” Estas strings podem aparecer no seu código Python e templates. É de sua responsabilidade marcar as strings traduzíveis; o sistema somente consegue traduzir o que ele sabe que tem que traduzir.

No código Python

Tradução padrão

Especifique uma translation string usando a função `ugettext()`. É convenção importá-la como um alias mais curto, o `_`, para facilitar a escrita.

Note: O módulo `gettext` da biblioteca padrão do Python instala uma função `_()` no namespace global, que funciona como um alias para `gettext()`. No Django, nós escolhemos não seguir esta prática, por algumas razões:

1. Para suporte a caracteres internacionais (Unicode), `ugettext()` é mais útil do que `gettext()`. Algumas vezes, você deve usar `ugettext_lazy()` como o método padrão de tradução para um arquivo em particular. Sem o `_()` no namespace global, o desenvolvedor tem que pensar sobre qual seria a função de tradução mais adequada.
 2. O caractere sublinhado (`_`) é usado para representar “o resultado prévio” no shell interativo do Python e nos testes doctest. Instalando uma função global `_()` causa interferências. Se você importa explicitamente a função `ugettext()` como `_()`, você evita esse problema.
-

Neste exemplo, o texto “Welcome to my site.” é marcado como uma translation string:

```
from django.utils.translation import ugettext as _

def my_view(request):
    output = _("Welcome to my site.")
    return HttpResponse(output)
```

Obviamente, você poderia ter feito isso sem usar o alias. Esse exemplo é idêntico ao anterior:

```
from django.utils.translation import ugettext

def my_view(request):
    output = ugettext("Welcome to my site.")
    return HttpResponse(output)
```

A tradução funciona sobre valores computados. Este exemplo é igual aos dois anteriores:

```
def my_view(request):
    words = ['Welcome', 'to', 'my', 'site.']
    output = _(' '.join(words))
    return HttpResponse(output)
```

A tradução funciona sobre variáveis. Novamente, um exemplo idêntico:

```
def my_view(request):
    sentence = 'Welcome to my site.'
    output = _(sentence)
    return HttpResponse(output)
```

(A limitação em usar variáveis ou valores computados, como nos dois últimos exemplos, é que o utilitário de detecção de translation strings do Django, `make-messages.py`, não será capaz de encontrar estas strings. Mais sobre `make-messages.py` depois.)

As strings que você passa para `_()` ou `gettext()` podem ter marcadores, especificados como a sintaxe de interpolação de named-string padrão do Python. Exemplo:

```
def my_view(request, m, d):
    output = _('Today is %(month)s, %(day)s.') % {'month': m, 'day': d}
    return HttpResponse(output)
```

Essa técnica permite que traduções de linguagens específicas reordenem o texto placeholder. Por exemplo, uma tradução em inglês poderia ser "Today is November, 26.", enquanto uma tradução em espanhol pode ser "Hoy es 26 de Noviembre." – com os placeholders (o mês e o dia) trocados de posição.

Por esta razão, você deve usar a interpolação por strings nomeadas (ex., `%(name)s`) ao invés de interpolação posicional (e.g., `%s` ou `%d`) sempre que você tiver mais de um parâmetro. Se você usou interpolação posicional, as traduções não poderiam ser capazes de reordenar o texto.

Marcando strings como no-op

Use a função `django.utils.translation.gettext_noop()` para marcar a string como uma translation string sem traduzi-la. A string é traduzida no final a partir de uma variável.

Use isto se você tem strings constantes que devem ser armazenadas no idioma original porque eles são trocados pelo sistema ou usuários – como uma string no banco de dados –, mas deve ser traduzida em um possível último momento, como quando a string é apresentada para o usuário.

Tradução tardia

Use a função `django.utils.translation.gettext_lazy()` para traduzir strings tardiamente – quando o valor é acessado em vez de quando a função `gettext_lazy()` é chamada.

Por exemplo, para traduzir uma string `help_text` de um modelo, faça o seguinte:

```
from django.utils.translation import gettext_lazy

class MyThing(models.Model):
    name = models.CharField(help_text=gettext_lazy('This is the help text'))
```

Neste exemplo, `gettext_lazy()` armazena uma referência preguiçosa da string – não da tradução real. A tradução por si só será feita quando a string é usada num contexto de string, como a renderização de templates na administração do Django.

The result of a `gettext_lazy()` call can be used wherever you would use a unicode string (an object with type `unicode`) in Python. If you try to use it where a bytestring (a `str` object) is expected, things will not work as expected, since a `gettext_lazy()` object doesn't know how to convert itself to a bytestring. You can't use a unicode string inside a bytestring, either, so this is consistent with normal Python behavior. For example:

```
# This is fine: putting a unicode proxy into a unicode string.
u"Hello %s" % ugettext_lazy("people")

# This will not work, since you cannot insert a unicode object
# into a bytestring (nor can you insert our unicode proxy there)
"Hello %s" % ugettext_lazy("people")
```

If you ever see output that looks like "hello <django.utils.functional...>", you have tried to insert the result of `ugettext_lazy()` into a bytestring. That's a bug in your code.

Se você não gosta do nome verboso `ugettext_lazy`, pode usar somente o alias `_` (underscore), então:

```
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(help_text=_('This is the help text'))
```

Sempre use traduções tardias nos *modelos do Django*. Nomes de campos e nomes de tabelas devem ser macados para tradução (do contrário eles não serão traduzidos na interface administrativa). Isso significa a escrita das opções `verbose_name` e `verbose_name_plural` na classe `Meta`, veja, ao invés de confiar na determinação do `verbose_name` e `verbose_name_plural` pelo Django, através do nome da classe do modelo:

```
from django.utils.translation import ugettext_lazy as _

class MyThing(models.Model):
    name = models.CharField(_('name'), help_text=_('This is the help text'))
    class Meta:
        verbose_name = _('my thing')
        verbose_name_plural = _('mythings')
```

Pluralização

Use a função `django.utils.translation.ungettext()` para especificar mensagens pluralizadas.

`ungettext` takes three arguments: the singular translation string, the plural translation string and the number of objects.

This function is useful when your need you Django application to be localizable to languages where the number and complexity of **plural forms** is greater than the two forms used in English ('object' for the singular and 'objects' for all the cases where `count` is different from zero, irrespective of its value.)

For example:

```
from django.utils.translation import ungettext
def hello_world(request, count):
    page = ungettext('there is %(count)d object', 'there are %(count)d objects',
↳count) % {
        'count': count,
    }
    return HttpResponse(page)
```

In this example the number of objects is passed to the translation languages as the `count` variable.

Lets see a slightly more complex usage example:

```
from django.utils.translation import ungettext

count = Report.objects.count()
if count == 1:
    name = Report._meta.verbose_name
else:
    name = Report._meta.verbose_name_plural
```

```

text = ungettext(
    'There is %(count)d %(name)s available.',
    'There are %(count)d %(name)s available.',
    count
) % {
    'count': count,
    'name': name
}

```

Here we reuse localizable, hopefully already translated literals (contained in the `verbose_name` and `verbose_name_plural` model Meta options) for other parts of the sentence so all of it is consistently based on the cardinality of the elements at play.

Note: When using this technique, make sure you use a single name for every extrapolated variable included in the literal. In the example above note how we used the `name` Python variable in both translation strings. This example would fail:

```

from django.utils.translation import ungettext
from myapp.models import Report

count = Report.objects.count()
d = {
    'count': count,
    'name': Report._meta.verbose_name
    'plural_name': Report._meta.verbose_name_plural
}
text = ungettext(
    'There is %(count)d %(name)s available.',
    'There are %(count)d %(plural_name)s available.',
    count
) % d

```

You would get a `KeyError` Python exception at runtime, as in `'msgstr[0]', doesn't exist in 'msgid'` error when running `django-admin.py compilemessages` or a `KeyError` Python exception at runtime.

No código do template

Traduções nos *templates do Django* usam duas template tags e uma sintaxe ligeiramente diferente do código Python. Para dar acesso a essas tags no seu template, coloque `{% load i18n %}` no topo do mesmo.

A template tag `{% trans %}` traduz uma string constante (entre aspas simples ou duplas) ou o conteúdo de uma variável:

```

<title>{% trans "This is the title." %}</title>
<title>{% trans myvar %}</title>

```

Se a opção `noop` está presente, a busca de variável ainda acontece, mas o texto original será retornado sem modificação. This is useful when “stubbing out” content that will require translation in the future:

```

<title>{% trans "myvar" noop %}</title>

```

Não é possível usar variáveis de template dentro de uma string com `{% trans %}`. Se suas traduções requerem variáveis string com placeholders, use `{% blocktrans %}`:

```

{% blocktrans %}This string will have {{ value }} inside.{% endblocktrans %}

```

Para traduzir uma expressão no template – digamos, usando filtros de templates – você necessita amarrar a expressão em uma variável local para usar dentro do bloco de tradução:

```
{% blocktrans with value|filter as myvar %}
This will have {{ myvar }} inside.
{% endblocktrans %}
```

Se você necessita amarrar mais que uma expressão dentro de uma tag `blocktrans`, separe-as com `and`:

```
{% blocktrans with book|title as book_t and author|title as author_t %}
This is {{ book_t }} by {{ author_t }}
{% endblocktrans %}
```

Para pluralizar, especifique ambas as formas, singular e plural, com a tag `{% plural %}`, que aparece dentro das tags `{% blocktrans %}` e `{% endblocktrans %}`. Exemplo:

```
{% blocktrans count list|length as counter %}
There is only one {{ name }} object.
{% plural %}
There are {{ counter }} {{ name }} objects.
{% endblocktrans %}
```

When you use the pluralization feature and bind additional values to local variables apart from the counter value that selects the translated literal to be used, have in mind that the `blocktrans` construct is internally converted to an `ungettext` call. This means the same [notes regarding ungettext variables](#) apply.

Cada `RequestContext` tem acesso a três variáveis de traduções específicas:

- `LANGUAGES` é uma lista de tuplas em que o primeiro elemento é o código da língua e o segundo é o nome da língua (traduzida no locale ativado no momento).
- `LANGUAGE_CODE` é o idioma corrente do usuário, na forma de string. Exemplo: `en-us`. (Veja [3. Como o Django descobre as preferencias de idioma](#), abaixo).
- `LANGUAGE_BIDI` é a direção do idioma corrente. Se `True`, a língua se escreve da direita para a esquerda, ex: Ebreu, Árabe. Se `False` da esquerda para a direita, ex: Inglês, Português, Francês, etc.

Se você não usa a extensão `RequestContext`, você pode acessar esses valores com três tags:

```
{% get_current_language as LANGUAGE_CODE %}
{% get_available_languages as LANGUAGES %}
{% get_current_language_bidi as LANGUAGE_BIDI %}
```

Essas tags também requerem um `{% load i18n %}`.

Os hooks de traduções estão disponíveis também dentro de qualquer tag de bloco de template que aceite strings constantes. Neste caso, use somente a sintaxe `_()` para especificar uma translation string:

```
{% some_special_tag _("Page not found") value|yesno:_("yes,no") %}
```

Neste caso, ambas as tags e filtros irão receber a string já traduzida, então elas não precisam se preocupar com as traduções.

Note: Nesse exemplo, a infraestrutura de tradução irá receber a string `"yes,no"`, não as strings individuais `"yes"` e `"no"`. A string traduzida precisa ter a vírgula para que o código do filter parsing saiba como dividir os argumentos. Por exemplo, um tradutor alemão poderia traduzir a string `"yes,no"` como `"ja,nein"` (mantendo a vírgula intacta).

Trabalhando com objetos de traduções tardias

O uso de `ugettext_lazy()` e `ungettext_lazy()` para marcar as strings nos models e funções utilitárias é uma operação comum. Quando você está trabalhando com esses objetos no seu código, você deve se assegurar que não os converteu em strings acidentalmente, porque eles devem ser convertidos o mais tardiamente possível (para que o efeito ocorra no local correto). Isso requer o uso de alguns helpers.

Juntando strings: `string_concat()`

O padrão do Python para junção de strings (`' '.join(...)`) não irá funcionar nas listas contendo objetos de tradução tardia. Em vez disso, você pode usar `django.utils.translation.string_concat`, que cria um objeto tardio que concatena seu conteúdo e os converte para strings somente quando o resultado é incluído em uma outra string. Por exemplo:

```
from django.utils.translation import string_concat
...
name = ugettext_lazy(u'John Lennon')
instrument = ugettext_lazy(u'guitar')
result = string_concat([name, ': ', instrument])
```

Nesse caso, a tradução tardia no `result` irá somente ser convertido para uma string quando `result` por si só for usado numa string (geralmente na hora de renderizar o template).

O decorador `allow_lazy()`

O Django oferece muitas funções utilitárias (particularmente `django.utils`) que recebem uma string como primeiro argumento e fazem algo com ela. Tais funções são usadas por meio de filtros nos templates, bem como em outros códigos.

Se você escrever suas próprias funções similares e tratar algumas traduções, você irá se deparar com um problema: o que fazer quando o primeiro argumento é um objeto de tradução tardia? Você não quer convertê-lo para uma string imediatamente, porque você pode estar usando esta função fora de uma view (and hence the current thread's locale setting will not be correct).

Para casos como este, use o decorador `django.utils.functional.allow_lazy()`. Ele modifica a função de modo que *caso* ela seja chamada com uma tradução tardia como primeiro argumento, a função de avaliação é mostrada até que necessite ser convertida para uma string.

Por exemplo:

```
from django.utils.functional import allow_lazy

def fancy_utility_function(s, ...):
    # Realiza alguma conversão sobre a string 's'
    ...
fancy_utility_function = allow_lazy(fancy_utility_function, unicode)
```

O decorador `allow_lazy()` recebe, além da função para decorar, um argumento extra (`*args`) especificando o(s) tipo(s) que a função original pode retornar. Geralmente, é suficiente incluir `unicode` ali, e assegurar que sua função retorne somente strings Unicode.

O uso desse decorador significa que você pode escrever suas funções e assumir que a entrada é uma string apropriada, e então adicionar um suporte para objetos de tradução tardia no final.

2. Como criar arquivos de linguagem

Uma vez que você tenha marcado suas strings para tradução, você precisa escrever (ou obter) as traduções em si. Aqui é mostrado como isto funciona.

Restrições de localidades

O Django não suporta a localização da sua aplicação em uma localidade para a qual ele mesmo não foi traduzido. Neste caso, ele irá ignorar os arquivos de tradução. Se você tentou isto e o Django suportou, você inevitavelmente verá uma mistura de palavras traduzidas (de sua aplicação) e strings em inglês (vindas do Django). Se você quiser suportar a sua aplicação para um local que não já não faz parte do Django, você precisa fazer pelo menos uma tradução mínima do Django core. See the relevant [LocaleMiddleware note](#) for more details.

Arquivos de mensagens

O primeiro passo é criar um **arquivo de mensagens** para um novo idioma. O arquivo de mensagens é um arquivo de texto simples, representando uma língua em particular, que contém todas as translation strings e como elas devem ser representadas para dada língua. Os arquivos de mensagens possuem a extensão `.po`.

O Django vem com uma ferramenta, `django-admin.py makemessages`, que automatiza a criação e manutenção destes arquivos.

Uma nota para veteranos do Django

A ferramenta antiga `bin/make-messages.py` foi movida para o comando `django-admin.py makemessages` para tornar o Django mais consistente.

Para criar ou atualizar um arquivo de mensagens, execute este comando:

```
django-admin.py makemessages -l de
```

...onde `de` é o código do idioma para o arquivo de mensagens que você deseja criar. O código do idioma, neste caso, está no formato de localidade. Por exemplo, `pt_BR` para Portugues do Brasil e `de_AT` para Alemão Austríaco.

O script deve ser executado em um destes três lugares:

- No diretório raiz do seu projeto Django.
- No diretório raiz de uma Django app.
- No diretório raiz do `django` (não um checkout do Subversion, mas em que esteja ligado ao `$PYTHONPATH` ou localizado em algum lugar sobre este caminho). This is only relevant when you are creating a translation for Django itself, see [Submitting and maintaining translations](#).

O script roda sobre todo código do seu projeto ou sobre todo o código da sua aplicação e extrai todas as strings marcadas para tradução. Ele cria (ou atualiza) o arquivo de mensagens no diretório `locale/LANG/LC_MESSAGES`. No exemplo `de`, o arquivo será `locale/de/LC_MESSAGES/django.po`.

Por padrão o `django-admin.py makemessages` examina todo arquivo que tem a extensão `.html`. Caso você queira mudar esse padrão, use a opção `--extension` ou `-e` para especificar a extensão dos arquivos a serem examinados:

```
django-admin.py makemessages -l de -e txt
```

Separe múltiplas extensões com vírgulas e/ou use `-e` ou `--extension` múltiplas vezes:

```
django-admin.py makemessages -l=de -e=html,txt -e xml
```

Quando estiver [criando catálogos de tradução para Javascript](#) você precisa usar o domínio `djangojs`, **não** `-e js`.

Sem gettext?

Se você não possui os utilitários do gettext instalados, o `django-admin.py makemessages` irá criar arquivos vazios. Se neste caso, você quiser instalar os utilitários do `gettext` ou somente copiar o arquivo com as mensagens em Inglês (`locale/en/LC_MESSAGES/django.po`) e usá-lo como ponto de partida; ele é apenas um arquivo vazio de tradução.

Trabalando no Windows?

Se você estiver usando Windwos precisará instalar os utilitários GNU gettext, somente assim o `django-admin makemessages` funcionará, veja [gettext no Windows](#) para mais informações.

O formato `.po` é bem simples. Cada arquivo `.po` contém um pouco de metadados, tais como as informações de contato do mantenedor da tradução, mas a maior parte do arquivo é uma lista de **mensagens** – um simples mapeamento entre as translation strings e o texto traduzido para um idioma em particular.

Por exemplo, se sua Django app contém uma translation string para o texto `"Welcome to my site."`, assim:

```
_("Welcome to my site.")
```

...então `django-admin.py makemessages` terá criado um arquivo `.po` contendo o seguinte conteúdo – uma mensagem:

```
#: path/to/python/module.py:23
msgid "Welcome to my site."
msgstr ""
```

Uma rápida explicação:

- `msgid` é a translation string, que aparece no seu código-fonte. Não toque nela!
- `msgstr` é onde se escreve a tradução. Ela é iniciada vazia, então é de sua responsabilidade mudá-la. Esteja certo de que as aspas não foram removidas.
- Por conveniência, cada mensagem inclui na forma de um comentário prefixado por `#`e` localizada acima da linha ``msgid` o nome do arquivo e número da linha de onde foi tirada a translation string.

Mensagens longas são um caso especial. Ali, a primeira string diretamente após `msgstr` (ou `msgid`) é uma string vazia. Em seguida, o conteúdo propriamente dito será escrito ao longo das próximas linhas como uma string por linha. Estas strings são diretamente concatenadas. Não esqueça dos espaços dentro das strings, caso contrário, elas irão aparecer todas juntas sem espaços em branco!

Pense no seu charset

Quando estiver criando um arquivo PO com seu editor de texto favorito, primeiro edite a linha do charset (procure por `"CHARSET"`) e mude-o para o charset que usará para editar o conteúdo. Devido à forma que a ferramenta `gettext` trabalha internamente, e porque nós queremos habilitar strings non-ASCII no core do Django e de suas aplicações, você **deve** usar UTF-8 como encoding de seus arquivos PO. Isto significa que todos estarão usando o mesmo encoding, o que é importante quando o Django processa os arquivos PO.

Para re-examinar todo código fonte e templates para novas translations strings e atualizar todos os arquivos de mensagens para **todas** as línguas, execute isto:

```
django-admin.py makemessages -a
```

Compilando arquivos de mensagens

Depois de criar seus arquivos de mensagens – e cada vez que você fizer alterações neles –, você necessitará compilá-los de uma forma eficiente, para que o `gettext` use-os. Faça isso com o utilitário `django-admin.py compilemessages`.

Essa ferramenta roda sobre todos os arquivos `.po` disponíveis e cria arquivos `.mo`, que são os binários otimizados para serem usados pelo `gettext`. No mesmo diretório de onde você rodou `django-admin.py makemessages`, execute `django-admin.py compilemessages`, desta forma:

```
django-admin.py compilemessages
```

É isso. Suas traduções estão prontas para serem usadas.

Uma nota para veteranos do Django

A ferramenta antiga `bin/compile-messages.py` foi movida para o comando `django-admin.py compilemessages` para tornar o Django mais consistente.

Trabalando no Windows?

Se você estiver usando Windows precisará instalar os utilitários GNU `gettext`, somente assim o `django-admin makemessages` funcionará, veja [gettext no Windows](#) para mais informações.

3. Como o Django descobre as preferencias de idioma

Uma vez que você tenha preparado suas traduções – ou se você somente quer usar as traduções que acompanham o Django – você somente precisará ativar a tradução para sua aplicação.

Por trás das cenas, o Django tem um modelo muito flexível de decisão sobre qual idioma será usado – em toda a instalação, para um usuário em particular, ou ambos.

Para configurar uma preferência de idioma para toda a instalação, configure o `:setting:LANGUAGE_CODE`. O Django usa esta língua como padrão de tradução – a tentativa final se não for encontrada outra tradução.

Se tudo o que você quer fazer é rodar o Django em seu idioma nativo, e um arquivo de linguagem disponível para sua língua, tudo que você precisa fazer é configurar `LANGUAGE_CODE`.

Se você quer deixar cada usuário individualmente especifique que idioma ele ou ela prefere, use `LocaleMiddleware`. O `LocaleMiddleware` habilita a seleção de idiomas baseado nos dados vindos pelo request, customizando o conteúdo para cada usuário.

Para usar o `LocaleMiddleware`, adicione `'django.middleware.locale.LocaleMiddleware'` em sua configuração de `MIDDLEWARE_CLASSES`. Como a ordem em que os middlewares são especificados importa, você deve seguir estas recomendações:

- Certifique-se de que é um dos primeiros middlewares instalados.
- Ele deve vir após `SessionMiddleware`, porque `LocaleMiddleware` faz uso de dados da sessão.
- Se você usa `CacheMiddleware`, coloque o `LocaleMiddleware` depois dele.

Por exemplo, seu `MIDDLEWARE_CLASSES` deve parecer com isso:

```
MIDDLEWARE_CLASSES = (  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.locale.LocaleMiddleware',  
    'django.middleware.common.CommonMiddleware',  
)
```

(Para mais sobre middleware, veja a :ref:`documentação do middleware <topics-http-middleware>`.)

O `LocaleMiddleware` tenta determinar o idioma do usuário a partir do seguinte algoritmo:

- Primeiro ele procura uma chave `django_language` na sessão do usuário corrente.

- Se falhar, ele procura por um cookie. *Please, see the release notes* No Django versão 0.96 e anteriores, o nome do cookie é imutável em `django_language`. No Django 1.0, o nome do cookie é configurado pela configuração `LANGUAGE_COOKIE_NAME`. (O nome padrão é `“django_language”`.)
- Se falhar, ele procura por `Accept-Language` no cabeçalho do HTTP. Este cabeçalho é enviado pelo seu navegador e diz ao servidor que idioma você prefere, na ordem de prioridade. O Django tenta cada língua do cabeçalho até que encontre uma que tenha traduções disponíveis.
- Se falhar, ele usa a configuração global `LANGUAGE_CODE`.

Notas:

- Em cada um desses lugares, o idioma de preferência é esperado no formato padrão da língua, como uma string. Por exemplo, Português Brasileiro é `pt-br`.
- Se uma língua de base está disponível, mas a sublíngua especificada não está, o Django usa a língua de base. Por exemplo, se um usuário especifica `de-at` (Alemão Austríaco), mas o Django somente tem de disponível, o Django usa `de`.
- Somente línguas listadas em `LANGUAGES` podem ser selecionadas. Se você quiser restringir o idioma para um subconjunto de línguas (porque sua aplicação não provê todos os idiomas), modifique o `LANGUAGES` para uma tupla de idiomas. Por exemplo:

```
LANGUAGES = (
    ('de', _('German')),
    ('en', _('English')),
)
```

Este exemplo restringe os idiomas disponíveis para seleção automática para Alemão e Inglês (e qualquer sublíngua, como `de-ch` ou `en-us`).

- Se você define uma configuração `LANGUAGES` customizada, como mostrado anteriormente, está tudo OK para marcar os idiomas como translation strings – mas use uma função “dummy” `gettext()`, não uma que esteja em `django.utils.translation`. Você nunca deve importar `django.utils.translation` dentro do arquivo `settings`, porque o módulo em si depende do `settings`, e isto poderia provocar um import circular.

A solução é usar uma função “dummy” `gettext()`. Aqui está um arquivo de configuração de exemplo:

```
gettext = lambda s: s

LANGUAGES = (
    ('de', gettext('German')),
    ('en', gettext('English')),
)
```

Com este arranjo, o `django-admin.py makemessages` ainda irá encontrar e marcar estas strings para tradução, mas a tradução não ocorrerá em tempo de execução – então, você deve lembrar de envolver as línguas no `gettext()` **real** em qualquer código que use `LANGUAGES` em tempo de execução.

- O `LocaleMiddleware` pode somente selecionar línguas que são providas pela base de tradução do Django. Se você quer prover traduções para sua aplicação que ainda não estão no conjunto de traduções do Django, você irá ter de prover pelo menos a tradução básica para tal língua. Por exemplo, o Django usa IDs de mensagens técnicas para traduzir formatos de data e tempo – então você precisará pelo menos destas traduções para que o seu sistema funcione corretamente.

Um bom começo é copiar o arquivo `.po` em inglês e traduzir pelo menos as mensagens técnicas – talvez as mensagens de validação, também.

Os IDs das mensagens técnicas são facilmente encontrados; todos eles estão em caixa alta. Você não traduz o ID da mensagem como nas outras mensagens, você provê o local correto variado a partir do valor em Inglês. Por exemplo, com `DATETIME_FORMAT` (ou `DATE_FORMAT` ou `TIME_FORMAT`), estas poderiam ser o formato da string que você quer usar em sua língua. O formato é idêntico ao formato usado pela template tag `now`.

Uma vez que `LocaleMiddleware` determine o idioma do usuário, ele disponibiliza esta preferência como `request.LANGUAGE_CODE` para cada `HttpRequest`. Sinta-se livre para ler este valor em sua visualização de código. Aqui está um pequeno exemplo:

```
def hello_world(request, count):
    if request.LANGUAGE_CODE == 'de-at':
        return HttpResponse("You prefer to read Austrian German.")
    else:
        return HttpResponse("You prefer to read another language.")
```

Note que, com tradução estática (sem-middleware), a língua está no `settings.LANGUAGE_CODE`, enquanto que a tradução dinâmica (com middleware), está em `request.LANGUAGE_CODE`.

Usando traduções em seus próprios projetos

O Django procura por traduções seguindo o seguinte algoritmo:

- Primeiro, ele procura por um diretório `locale` na pasta da aplicação que estiver sendo executada. Se ele encontra a tradução para a língua selecionada, a tradução será instalada.
- Depois, ele procura por um diretório `locale` no diretório do projeto. Se ele encontra uma tradução, a tradução será instalada.
- Finalmente, ele checa a base de traduções no `django/conf/locale`.

Deste jeito, você pode escrever aplicações que possuem suas próprias traduções, e sobrescrever as traduções padrões no caminho do seu projeto. Ou, você pode simplesmente construir um grande projeto de várias aplicações e colocar todas as traduções em um grande arquivo de mensagens do projeto. A escolha é sua.

Note: Se você tem o `settings` configurado manualmente, como descrito na [Usando o settings sem a configuração DJANGO_SETTINGS_MODULE](#), o diretório `locale` na pasta do projeto não será examinado, desde que o Django perca a habilidade de trabalhar fora da localização da pasta do projeto. (O Django normalmente usa a localização do arquivo `settings` para determinar isto, e um arquivo `settings` não existe se você está setando manualmente suas configurações.)

Todo arquivo repositório de mensagem é estruturado da mesma forma, a saber:

- `$APPPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- `$PROJECTPATH/locale/<language>/LC_MESSAGES/django.(po|mo)`
- All paths listed in `LOCALE_PATHS` in your settings file are searched in that order for `<language>/LC_MESSAGES/django.(po|mo)`
- `$PYTHONPATH/django/conf/locale/<language>/LC_MESSAGES/django.(po|mo)`

Para criar arquivos de mensagens, você usa a mesma ferramenta `django-admin.py makemessages` como nos arquivos de mensagem do Django. Você somente precisa estar no lugar certo – no diretório onde qualquer um dos dois – `conf/locale` (para o código do Django) ou `locale/` (para as mensagens de apps ou projeto) – estão localizados. E você usa o mesmo `compile-message.py` para gerar os arquivos binários `django.mo` que são usados pelo `gettext`.

Você pode também executar `django-admin.py compilemessages --settings=path.to.settings` para fazer o compilador processar todos os diretórios em seu `LOCALE_PATHS`.

Os arquivos de mensagens das aplicações são um pouco complicados de achar – eles precisam do `LocaleMiddleware`. Se você não usa o middleware, somente os arquivos de mensagens do Django e do projeto serão processados.

Finalmente, você deve ter pensado em algo para estruturar seus arquivos de tradução. Se suas aplicações precisam ser entregues para outros usuários e irão ser utilizadas em outros projetos, você pode querer usar as traduções

específicas de cada aplicação. Porém, usar as traduções de aplicações e traduções de projeto podem produzir alguns problemas com o `make-messages`: o `make-messages` irá visitar todos os diretórios abaixo do diretório corrente e então poderá colocar todos os IDs no arquivos de mensagens do projeto que já estão nos seus arquivos de mensagens das aplicações.

O caminho mais fácil é armazenar as aplicações que não fazem parte do projeto (que carrega suas próprias traduções) fora da árvore do projeto. Desta forma, o `django-admin.py makemessages` no nível do projeto irá traduzir somente o que estiver conectado dentro dele e não o que está distribuído independentemente.

O view redirecionador `set_language`

Por conveniência, o Django é acompanhado de um view, `django.view.i18n.set_language` que configura a preferência de idioma do usuário e redireciona para uma página anterior.

Para ativar este view, adicione a seguinte linha em seu `URLconf`:

```
(r'^i18n/', include('django.conf.urls.i18n')),
```

(Note que este exemplo torna o view disponível em `/i18n/setlang/`.)

O view espera uma requisição via método `POST`, com um parâmetro `language`. Se o suporte a sessão está habilitado, o view salva a escolha do idioma na sessão do usuário. Caso contrário, ele o salva em um cookie que tem por padrão o nome `django_language`. (O nome pode ser alterado por meio da configuração `LANGUAGE_COOKIE_NAME`).

Depois de salvar a opção de idioma, o Django redireciona o usuário, seguindo este algoritmo:

- O Django procura por um parâmetro `next` nos dados do `POST`.
- Se isso não existir, ou estiver vazio, o Django tenta a URL no cabeçalho `Referer`.
- Se estiver vazio – isto é, se o navegador do usuário suprimir o cabeçalho – então o usuário será redirecionado para `/` (a raiz do site) como um fallback.

Aqui está um exemplo de código de template em HTML:

```
.. code-block:: html+django
```

```
<form action="/i18n/setlang/" method="post"> <input name="next" type="hidden"
value="/next/page/" /> <select name="language"> {% for lang in LANGUAGES %} <option
value="{{ lang.0 }}">{{ lang.1 }}</option> {% endfor %} </select> <input type="submit"
value="Go" /> </form>
```

Traduções e JavaScript

Adicionar traduções para JavaScript apresenta alguns problemas:

- O código JavaScript não tem acesso a uma implementação do `gettext`.
- O código JavaScript não tem acesso aos arquivos `.po` ou `.mo`; eles precisam ser entregues por um servidor.
- Os catálogos de tradução para JavaScript devem ser mantidos tão pequenos quanto possível.

O Django provê uma solução integrada para estes problemas: ele passa a tradução dentro do JavaScript, então você pode chamar `gettext`, etc., de qualquer JavaScript.

O view `javascript_catalog`

A principal solução para estes problemas é o view `javascript_catalog`, que envia uma biblioteca de código JavaScript com funções que imitam a interface do `gettext`, mais um array com as `translation strings`. Estas

translation strings são capturadas das aplicações, do projeto ou do Django core, de acordo com o que você especificou em qualquer um dos `info_dict` ou na URL.

Faça desta forma:

```
js_info_dict = {
    'packages': ('your.app.package',),
}

urlpatterns = patterns('',
    (r'^jsi18n/$', 'django.views.i18n.javascript_catalog', js_info_dict),
)
```

Cada string no `packages` deve ser escrita na sintaxe pontuada do Python (o mesmo formato das `INSTALLED_APPS`) e devem se referir a um pacote que contenha um diretório `locale`. Se você especificar múltiplos pacotes, todos estes catálogos são combinados em um único catálogo. Isto é comum se você tem JavaScripts que usam strings de diferentes aplicações.

Você pode tornar o view dinâmico, colocando os pacotes no URL patterns:

```
urlpatterns = patterns('',
    (r'^jsi18n/(?P<packages>\S+?)/$', 'django.views.i18n.javascript_catalog'),
)
```

Com isto, você especifica os pacotes como uma lista de pacotes delimitados por sinais de `+` na URL. Esta é uma especialidade comum se sua página usa código de diferentes aplicações, se elas mudam frequentemente e você não quer baixar um catálogo enorme. Como uma medida de segurança, estes valores podem somente estar ou no `django.conf` ou em outro pacote configurado no `INSTALLED_APPS`.

Usando o catálogo de tradução do JavaScript

Para usar o catálogo, é só usar um script dinamicamente gerado, como a seguir:

```
<script type="text/javascript" src="{% url django.views.i18n.javascript_catalog %}"
↪"></script>
```

This uses reverse URL lookup to find the URL of the JavaScript catalog view. Quando o catálogo é carregado, seu código Javascript pode usar o padrão da interface de acesso do `gettext`:

```
document.write(gettext('this is to be translated'));
```

Existe também uma interface `ngettext`:

```
var object_cnt = 1 // or 0, or 2, or 3, ...
s = gettext('literal for the singular case',
    'literal for the plural case', object_cnt);
```

e ainda um função de interpolação de string:

```
function interpolate(fmt, obj, named);
```

A sintaxe de interpolação é emprestada do Python, então a função `interpolate` suporta ambas as interpolações, posicional e nomeada.

- Interpolação posicional: o `obj` contém um objeto Array de JavaScript e os valores dos elementos são então sequencialmente interpolados em seus lugares correspondendo aos marcadores `fmt` na mesma ordem em que aparecem. Por exemplo:

```
fmts = gettext('There is %s object. Remaining: %s',
    'There are %s objects. Remaining: %s', 11);
s = interpolate(fmts, [11, 20]);
// s is 'There are 11 objects. Remaining: 20'
```

- Interpolação nomeada: Este modo é selecionado passando um valor booleano para o parâmetro `named`, como `True`. O `obj` contém um objeto JavaScript ou um array associativo. Por exemplo:

```
d = {
    count: 10
    total: 50
};

fmts = ngettext('Total: %(total)s, there is %(count)s object',
'there are %(count)s of a total of %(total)s objects', d.count);
s = interpolate(fmts, d, true);
```

Você não deve passar por cima da interpolação de string. Embora isso ainda seja JavaScript, o código tem de fazer repetidas substituições por expressões-regulares. Isto não é tão rápido quanto a interpolação do Python, então mantenha esses casos onde você realmente precisa (por exemplo, na conjunção com `ngettext` para produzir pluralizações).

Criando catálogos de tradução para JavaScript

Para criar e atualizar os catálogos de traduções é da mesma forma como os outros catálogos de tradução do Django – com a ferramenta `django-admin.py makemessages`. A única diferença está em você passar um parâmetro `-d djangojs`, desta forma:

```
django-admin.py makemessages -d djangojs -l de
```

Isto poderia criar ou atualizar o catálogo de tradução para JavaScript em Alemão. Depois da atualização dos catálogos, simplesmente execute `django-admin.py compilemessages` da mesma forma que se faz com um catálogo normal do Django.

Especialidades da tradução do Django

Se você conhece `gettext`, você pôde notar estas especialidades na forma como o Django faz tradução:

- Os domínios de string são `django` ou `djangojs`. Estes domínios são utilizados para distinguir diferentes programas que armazenam seus dados em um arquivo comum de mensagens (usualmente em `/usr/share/locale/`). O domínio `django` é usado pelas translation strings no Python e em templates e é carregado dentro do catálogo de tradução geral. O domínio `djangojs` é utilizado somente para catálogos de tradução de JavaScript, a fim de torná-lo o menor possível.
- O Django não usa `xgettext` sozinho. Ele usa wrappers do Python em torno do `xgettext` e `msgfmt`. Isto é mais por conveniência.

gettext no Windows

Isso só é necessário para quem deseja extrair IDs de mensagens ou compilar arquivos de mensagens (`.po`). A tradução em si funciona somente envolvendo a edição dos arquivos desse tipo já existentes, mas se você deseja criar seus próprios arquivos de mensagens, ou quer testar ou compilar uma mudança num arquivo de mensagens, você precisará dos utilitários do `gettext`:

```
* Faça o download dos seguintes arquivos zip dos servidores do GNOME
  http://ftp.gnome.org/pub/gnome/binaries/win32/dependencies/ ou de um
  de seus espelhos_
* ``gettext-runtime-X.zip``
* ``gettext-tools-X.zip``

``X`` is the version number, we recomend using ``0.15`` or higher.
```

```
* Extract the contents of the ``bin`` directories in both files to the
  same folder on your system (i.e. ``C:\Program Files\gettext-utils``)

* Atualize o PATH do sistema:

  * ``Painel de Controle > Sistema > Avançado > Variáveis de Ambiente``
  * Na lista de ``Variáveis de Sistema``, clique ``Path``, clique ``Editar``
  * Adicionar ``;C:\Program Files\gettext-utils\bin`` no fim do campo
    ``Valor da Variável``
```

Você pode também usar os binários do gettext que você já possui, contanto que o comando `xgettext --version` funcione apropriadamente. Algumas versões 0.14.4 não possuem suporte para este comando. Não tente usar os utilitários de tradução do Django com um pacote gettext se o comando `xgettext --version` digitado no prompt de comando do Windows gera um popup dizendo “xgettext.exe gerou erros e será fechado pelo Windows”.

As fundações da paginação foram reformuladas, quase que por completo. O Django fornece algumas classes que ajudam a lidar com paginação de dados – isto é, dados que são divididos entre várias páginas, com links “Previous/Next”. Essas classes estão contidas no módulo `django/core/paginator.py`.

Exemplo

Dada a um *Paginator* uma lista de objetos, mais um número de itens que você gostaria que aparecessem em cada página, ele fornecerá os métodos necessários para acessar os itens de cada página:

```
>>> from django.core.paginator import Paginator
>>> objects = ['john', 'paul', 'george', 'ringo']
>>> p = Paginator(objects, 2)

>>> p.count
4
>>> p.num_pages
2
>>> p.page_range
[1, 2]

>>> page1 = p.page(1)
>>> page1
<Page 1 of 2>
>>> page1.object_list
['john', 'paul']

>>> page2 = p.page(2)
>>> page2.object_list
['george', 'ringo']
>>> page2.has_next()
False
>>> page2.has_previous()
True
>>> page2.has_other_pages()
True
>>> page2.next_page_number()
3
```



```
>>> page2.previous_page_number()
1
>>> page2.start_index() # O índice iniciado em 1 do primeiro item nesta página
3
>>> page2.end_index() # O índice iniciado em 1 do último item nesta página
4

>>> p.page(0)
Traceback (most recent call last):
...
EmptyPage: That page number is less than 1
>>> p.page(3)
Traceback (most recent call last):
...
EmptyPage: That page contains no results
```

Note: Note que você pode dar a um paginador uma lista ou tupla `Paginator`, uma `QuerySet` do Django, ou qualquer objeto com um método `count()` ou `__len__()`. Ao determinar o número de objetos contidos no objeto passado, o `Paginator` vai primeiro tentar chamar `count()`, depois irá tentar chamar `len()` se o objeto não tiver um método `count()`. Isso permite a objetos como o `QuerySet` do Django usarem um `count()` mais eficiente, quando disponível.

Usando Paginator em uma view

Aqui há um exemplo ligeiramente mais complexo do uso do `Paginator` em um view para paginar um queryset. Nós dizemos a ambos, view e o template que o acompanha, como você pode exibir resultados. Este exemplo assume que você tem um model `Contacts` que já foi importado.

A função view parece com isso:

```
from django.core.paginator import Paginator, InvalidPage, EmptyPage

def listing(request):
    contact_list = Contacts.objects.all()
    paginator = Paginator(contact_list, 25) # Mostra 25 contatos por página

    # Make sure page request is an int. If not, deliver first page.
    # Esteja certo de que o 'page request' é um inteiro. Se não, mostre a primeira
    # página.
    try:
        page = int(request.GET.get('page', '1'))
    except ValueError:
        page = 1

    # Se o page request (9999) está fora da lista, mostre a última página.
    try:
        contacts = paginator.page(page)
    except (EmptyPage, InvalidPage):
        contacts = paginator.page(paginator.num_pages)

    return render_to_response('list.html', {"contacts": contacts})
```

No template `list.html`, você poderá incluir a navegação entre as páginas, juntamente com qualquer informação interessante a partir dos próprios objetos:

```
{% for contact in contacts.object_list %}
    {# Cada "contato" é um objeto do model Contact. #}
    {{ contact.full_name|upper }}<br />
```

```

...
{% endfor %}

<div class="pagination">
  <span class="step-links">
    {% if contacts.has_previous %}
      <a href="?page={{ contacts.previous_page_number }}">anterior</a>
    {% endif %}

    <span class="current">
      Página {{ contacts.number }} of {{ contacts.paginator.num_pages }}.
    </span>

    {% if contacts.has_next %}
      <a href="?page={{ contacts.next_page_number }}">próxima</a>
    {% endif %}
  </span>
</div>

```

Objetos Paginator

A classe *Paginator* tem este construtor:

```
class Paginator (object_list, per_page, orphans=0, allow_empty_first_page=True)
```

Argumentos obrigatórios

object_list Uma lista, tupla, QuerySet do Django, ou outro objeto contável por um método `count()` ou `__len__()`.

per_page O número máximo de itens que serão incluídos na página, não incluindo orfãos (veja o argumento opcional `orphans` abaixo).

Argumentos opcionais

orphans O número mínimo de itens permitidos em uma última página, o padrão é zero. Use isto quando você não quiser ter uma última página com pouquíssimos itens. Se a última página poderá ter um número de itens menor que ou igual ao `orphans`, então estes itens serão adicionados na página anterior (que se torna a última página) ao invés de deixados em uma página só para eles. Por exemplo, com 23 itens, `per_page=10`, e `orphans=3`, haverá duas páginas; a primeira com 10 itens e a segunda (e última) com 13 itens.

allow_empty_first_page Permite ou não a primeira página ser vazia. Se for `False` e `object_list` estiver vazio, então um erro `EmptyPage` será levantado.

Métodos

`Paginator.page(number)`

Retorna um objeto `Page` com um determinado índice. Se a determinada página não existe, é lançado uma exceção `InvalidPage`.

Atributos

`Paginator.count`

O número total de objetos, através de todas as páginas.

Note: Quando se determina o número de objetos contidos no `object_list`, o `Paginator` irá primeiro tentar chamar `object_list.count()`. Se `object_list` não tem o método `count()`, então o `Paginator` tentará usar o `object_list.__len__()`. Isto permite objetos, como um `QuerySet` do Django, usar um método de `count()` mais eficiente quando disponível.

`Paginator.num_pages`

O número total de páginas.

`Paginator.page_range`

Um número começando com 1 de páginas, por exemplo, `[1, 2, 3, 4]`.

Exceções `InvalidPage`

O método `page()` lança a exceção `InvalidPage` se a página requisitada for inválida (ex., não for um inteiro) ou não conter objetos. Geralmente, isto é suficiente para se ter uma exceção `InvalidPage`, mas se você quizer mais granularidade, você pode utilizar qualquer uma dessas exceções:

`PageNotAnInteger` Lançada quando à `page()` é dado um valor que não é um inteiro.

`EmptyPage` Lançada quando à `page()` é dado um valor válido, mas não á objetos existentes naquela página.

Ambas as exceções são subclasses de `InvalidPage`, então você poder manipulá-las com um simples `except InvalidPage`.

Objetos `Page`

`Page(object_list, number, paginator):`

Você normalmente não irá construir `Pages` na mão – você vai obtê-los usando `Paginator.page()`.

Métodos

`Page.has_next()`

Retorna `True` se existe uma página subsequente.

`Page.has_previous()`

Retorna `True` se existe uma página anterior.

`Page.has_other_pages()`

Retorna `True` se existe uma página subsequente ou anterior.

`Page.next_page_number()`

Retorna o número da página subsequente. Note que este é um método “burro” e vai apenas retornar o número da página subsequente, a página existindo ou não.

`Page.previous_page_number()`

Retorna o número da página anterior. Note que este é um método “burro” e vai apenas retornar o número da página anterior, a página existindo ou não.

`Page.start_index()`

Retorna o índice iniciado em 1 do primeiro objeto na página, relativo a todos os objetos na lista do paginador. Por exemplo, quando se pagina uma lista com 5 objetos a 2 objetos por página, o `start_index()` da segunda página devolveria 3.

`Page.end_index()`

Retorna o índice iniciado em 1 do último objeto na página, relativo a todos os objetos na lista do paginador. Por exemplo, quando se pagina uma lista com 5 objetos a 2 objetos por página, o `end_index()` da segunda página devolveria 4.

Atributos

Page.**object_list**

A lista de objetos nesta página.

Page.**number**

O número da página com índice começando em 1.

Page.**paginator**

O *Paginator* associado ao objeto.

Seriando objetos do Django

O framework de serialização do Django provê um mecanismo para “traduzir” objetos do Django em outros formatos. Usualmente estes outros formatos serão baseados em texto e usados para enviar objetos do Django por um fio, mas é possível para um seriador manipular qualquer formato (baseado em texto ou não).

See also:

If you just want to get some data from your tables into a serialized form, you could use the `dumpdata` management command. Se você apenas quiser obter alguns dados de suas tabelas na forma seriada, você pode usar o comando `dumbdata`.

Seriando dados

No nível mais alto, serializar dados é uma operação muito simples:

```
from django.core import serializers
data = serializers.serialize("xml", SomeModel.objects.all())
```

Os argumentos para a função `serialize` são o formato para o qual os dados serão seriados (veja [Formatos de serialização](#)) e um `QuerySet` para serializar. (Na verdade, o segundo argumento pode ser qualquer iterador que fornece objetos Django, mas quase sempre será um `QuerySet`).

Você pode usar também um objeto seriador diretamente:

```
XMLSerializer = serializers.get_serializer("xml")
xml_serializer = XMLSerializer()
xml_serializer.serialize(queryset)
data = xml_serializer.getvalue()
```

Este é usual se você quiser serializar dados diretamente para um objeto arquivo (que inclua um `HttpResponse`):

```
out = open("file.xml", "w")
xml_serializer.serialize(SomeModel.objects.all(), stream=out)
```

Subconjunto de campos

Se você somente quiser um subconjunto de campos para serializar, você pode especificar um argumento `fields` para o serializador:

```
from django.core import serializers
data = serializers.serialize('xml', SomeModel.objects.all(), fields=('name', 'size
↪'))
```

Neste exemplo, somente os atributos `name` e `size` de cada `model` serão serializados.

Note: Dependendo do seu modelo, você pode descobrir que não é possível desserializar um modelo que teve apenas um subconjunto de seus campos serializados. Se um objeto serializado não especificar todos os campos que são requeridos pelo modo, o desserializador não será capaz de salvar instâncias desserializadas.

Models herdados

Se você tem um modelo que é definido usando uma *classe abstrata de base*, você não terá de fazer nada especial para serializar este modelo. Somente chamar o serializador sobre o objeto (ou objetos) que você deseja serializar, e a saída será uma representação completa do objeto serializado.

Porém, se você tem um modelo que usa *herança de multi-tabelas*, você também precisa serializar todos as classes que dão base ao modelo. Isto porque somente os campos que são localmente definidos no modelo serão serializados. Por exemplo, considere os seguintes modelos:

```
class Place(models.Model):
    name = models.CharField(max_length=50)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField()
```

Se você somente serializar o modelo `Restaurant`:

```
data = serializers.serialize('xml', Restaurant.objects.all())
```

os campos na saída serializada conterão somente atributos `serves_hot_dogs`. O atributo `name` da classe de base será ignorado.

A fim de permitir uma serialização completa para sua instância `Restaurant`, você precisará serializar o modelo `Place` também:

```
all_objects = list(Restaurant.objects.all()) + list(Place.objects.all())
data = serializers.serialize('xml', all_objects)
```

Desserializando dados

Desserialização de dados é também uma operação bastante simples:

```
for obj in serializers.deserialize("xml", data):
    do_something_with(obj)
```

Como você pode ver, a função `deserialize` recebe o mesmo argumento de formato que o `serialize`, uma string ou stream de dados, e retorna um iterador.

Entretanto, aqui fica um pouco complicado. Os objetos retornados pelo iterador do `deserialize` *não são* objetos simples do Django. Em vez disso, eles são instâncias especiais `DeserializedObject` que envolvem o objeto criado – mas não salvo – e toda relação de dados.

Chamando `DeserializedObject.save()` salva o objeto no banco de dados.

Isso garante que a desserialização é uma operação não destrutiva mesmo se o dado na sua representação não bata com a que estiver agora no seu banco de dados. Normalmente, trabalhar com estas instâncias de `DeserializedObject` parece algo como:

```
for deserialized_object in serializers.deserialize("xml", data):
    if object_should_be_saved(deserialized_object):
        deserialized_object.save()
```

Em outras palavras, o uso normal é examinar os objetos desserializados para ter certeza de que eles estão “adequados” a serem salvos antes de salvá-los. Logicamente, se você confiar na fonte de dados, poderá apenas salvar os objetos e seguir adiante.

O objeto Django em si pode ser inspecionado como `deserialized_object.object`.

Formatos de serialização

O Django suporta vários formatos de serialização, alguns que obrigam você instalar módulos de terceiros do Python:

Identificador	Informações
xml	Serializa para e do dialeto simples de XML.
json	Serializa para e do JSON (usando uma versão do simplejson empacotada com o Django).
python	Traduz para e dos objetos “simples” do Python (listas, dicionários, strings, etc.). Nem tudo dele é realmente útil, mas é usado como base para outros serializadores.
yaml	Serializa para YAML (YAML não é uma linguagem de marcação). Este serializador é somente disponível se o PyYAML estiver instalado.

Notas para formatos de serialização específicos

json

Se você esta usando dados UTF-8 (ou qualquer outra codificação não-ASCII) com o serializador JSON, você passar `ensure_ascii=False` como parâmetro para a chamada `serialize()`. Por outro lado, a saída não pode ser codificada corretamente.

Por exemplo:

```
json_serializer = serializers.get_serializer("json")()
json_serializer.serialize(queryset, ensure_ascii=False, stream=response)
```

O código do Django inclui o módulo [simplejson](#). Tenha cuidado que se você estiver serializando usando este módulo diretamente, nem todas as saídas do Django podem ser passadas sem modificação para o [simplejson](#). Em particular, *objetos de tradução tardios* precisam de um [codificador especial](#) escritos para eles. Algo como isto funcionará:

```
from django.utils.functional import Promise
from django.utils.encoding import force_unicode

class LazyEncoder(simplejson.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Promise):
            return force_unicode(obj)
        return super(LazyEncoder, self).default(obj)
```


O arquivo settings do Django contém toda configuração de sua instalação do Django. Este documento explica como o settings funciona e quais configurações estão disponíveis.

O básico

Um arquivo settings é só um módulo Python com variáveis a nível de módulo.

Aqui tem um exemplo de configurações:

```
DEBUG = False
DEFAULT_FROM_EMAIL = 'webmaster@example.com'
TEMPLATE_DIRS = ('/home/templates/mike', '/home/templates/john')
```

Because a settings file is a Python module, the following apply:

Pelo arquivo settings ser um módulo Python, o seguinte se aplica:

- Ele não permite erros de syntaxe do Python.
- Ele pode atribuir configurações dinamicamente usando a syntaxe normal do Python.

Por exemplo:

```
MY_SETTING = [str(i) for i in range(30)]
```

- Ele pode importar valores de outros arquivos settings.

Designando as configurações

Quando você usa o Django, você tem de informá-lo quais configurações você está usando. Faça isso usando uma variável de ambiente, `DJANGO_SETTINGS_MODULE`.

O valor do `DJANGO_SETTINGS_MODULE` deve ser na syntaxe de caminhos do Python, e.g. `mysite.settings`. Note que o módulo settings deve estar no [caminho de import](#) do Python.

O utilitário django-admin.py

Quando estiver usando o *django-admin.py*, você pode setar ambos, variável de ambiente, ou explicitamente passar o módulo settings toda que você rodar o utilitário.

Exemplo (Unix Bash shell):

```
export DJANGO_SETTINGS_MODULE=mysite.settings
django-admin.py runserver
```

Exemplo (Windows shell):

```
set DJANGO_SETTINGS_MODULE=mysite.settings
django-admin.py runserver
```

Use o argumento de linha de comando `--settings` para especificar o settings manualmente:

```
django-admin.py runserver --settings=mysite.settings
```

No servidor (mod_python)

No seu ambiente de servidor, você precisará dizer ao Apache/mod_python qual arquivo settings usar. Faça isso com SetEnv:

```
<Location "/mysite/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>
```

Leia a *documentação do mod_python do Django* para mais informações.

Configurações padrão

Um arquivo settings do Django não tem que definir quaisquer configurações se não for necessário. Cada configuração tem um valor padrão. Estes valores padrão ficam no módulo `django/conf/global_settings.py`.

Aqui tem o algoritmo que o Django usa ao compilar o settings:

- Carregar configurações do `global_settings.py`.
- Carregar configurações do arquivos settings especificado, sobrescrevendo as configurações se necessário.

Note que o arquivo settings *não* deve importar o `global_settings`, porque isto é redundante.

Vendo quais configurações você mudou

Há uma forma fácil de ver quais das suas configurações desviam do padrão. O comando `python manage.py diffsettings` mostra as diferenças entre as configurações atuais e do arquivo de configuração padrão do Django.

Para mais, veja a documentação do *diffsettings*.

Usando o settings em código Python

Em suas aplicações Django, use o settings importando o objeto `django.conf.settings`. Exemplo:

```
from django.conf import settings
```

if settings.DEBUG: # Faça algo

Note que o `django.conf.settings` não é um módulo – ele é um objeto. Então importar uma configuração individual não é possível:

```
from django.conf.settings import DEBUG # Isto não funcionará.
```

Também perceba que seu código *não* deve importar o `global_settings` ou o seu próprio arquivo `settings`. O `django.conf.settings` abstrai o conceito de configurações globais e específicas do site; ele apresenta em uma única interface. Ele também dissocia o código que utiliza as definições a partir da localização do seu `settings`.

Alterando o settings em tempo de execução

Você não deve alterar o settings nas suas aplicação em tempo de execução. Por exemplo, não faça isso em um view:

```
from django.conf import settings

settings.DEBUG = True # Não faça isso!
```

O único lugar onde você deve modificar as configurações é no arquivos `settings`.

Segurança

Por um arquivo `settings` conter informações sensíveis, como a senha do banco de dados, você deve fazer de tudo para evitar o limite de acesso a ele. Por exemplo, mude suas permissões de arquivo de forma que só você e seu usuário do servidor Web poderão lê-lo. Isso é excencialmente importante e um ambiente de hospedagem compartilhada.

Configurações disponíveis

Para uma lista completa das configurações disponíveis, veja a [referência do settings](#).

Criando seu próprio settings

Não há nada que impeça você de criar seu próprio settings, para sua aplicação Django. Só siga estas convenções:

- Nomes de configurações são todos em maiúsculo.
- Não reinvente uma configuração já existente.

Para configurações sequenciais, o próprio Django utiliza tuplas, no lugar de listas, mas esta é somente uma convenção.

Usando o settings sem a configuração DJANGO_SETTINGS_MODULE

Em alguns casos, você pode querer não passa a variável de ambiente `DJANGO_SETTINGS_MODULE`. Por exemplo, se você estiver usando o sistema de template somente, você provavelmente não vai querer ter que configurar uma variável de ambiente apontando apra o módulo `settings`.

Nestes casos, você pode configurar o settings do Django manualmente. Faça isso chamando:

```
django.conf.settings.configure(default_settings, **settings)
```

Exemplo:

```
from django.conf import settings

settings.configure(DEBUG=True, TEMPLATE_DEBUG=True,
    TEMPLATE_DIRS=('/home/web-apps/myapp', '/home/web-apps/base'))
```

Passe ao `configure()` tantos argumentos quantos você quiser, com cada argumento nomeado representando uma configuração e seu valor. Cada nome de argumento deve ser todo em maiúsculo, com o mesmo nome como descrito acima. Se uma configuração particular não for passada para o `configure()` e ela for necessária em algum ponto mais a frente, o Django usará o valor padrão.

Configurar o Django deste jeito é sobre tudo necessário – e, realmente, recomendado – quando você estiver usando uma parte do framework dentro de uma aplicação maior.

Consequentemente, quando configurado via `settings.configure()`, o Django não fará quaisquer modificações nas variáveis de ambiente do processo (veja a documentação do [TIME_ZONE](#) para saber porque isso normalmente ocorre). Ele assume que você já está no controle total do seu ambiente nestes casos.

Configurações padrão customizadas

Se você gostaria de valores padrão de outros lugares além do `django.conf.global_settings`, você pode passar num módulo ou classe que fornece as configurações padrão como o argumento `default_settings` (ou como o primeiro argumento posicional) na chamada do `configure()`.

Neste exemplo, configurações padrão são recebidas do `myapp_defaults`, e a configuração `DEBUG` é setada para `True`, indiferente do seu valor em `myapp_defaults`:

```
from django.conf import settings
from myapp import myapp_defaults

settings.configure(default_settings=myapp_defaults, DEBUG=True)
```

O exemplo a seguir, usa o `myapp_defaults` como um argumento posicional:

```
settings.configure(myapp_defaults, DEBUG = True)
```

Normalmente, você não precisará sobrescrever os valores padrão. Os padrões do Django são suficientemente fáceis de lidar, e você pode seguramente utilizá-los. Seja cuidadoso se você for passar um novo módulo padrão, e ele *substituir* inteiramente os valores padrão do Django, então você deve especificar um valor para toda configuração possível que poderia ser utilizada pelo código que você está importando. Confira a lista completa no `django.conf.settings.global_settings`.

O `configure()` ou `DJANGO_SETTINGS_MODULE` são requeridos

Se você não configurar a variável de ambiente `DJANGO_SETTINGS_MODULE`, você *deve* chamar `configure()` no mesmo ponto antes de usar qualquer código que leia configurações.

Se você não setar o `DJANGO_SETTINGS_MODULE` e não chamar o `configure()`, o Django lançará uma exceção `ImportError` na primeira vez que uma configuração for acessada.

Se você setar o `DJANGO_SETTINGS_MODULE`, acessar valores do `settings` de alguma maneira, e *então* chamar o `configure()`, o Django lançará um erro `RuntimeError` indicando que as configurações já foram carregadas.

Também, é um erro chamar o `configure()` mais de uma vez, ou chamar `configure()` depois que qualquer configuração fora acessada.

Em resumo é isto: Use exatamente um dos dois, `configure()` ou `DJANGO_SETTINGS_MODULE`. Não os dois, nem nenhum.

Sinais (Signals)

O Django inclui um “dispachador de sinal” que ajuda ao permitir que aplicações dissociadas sejam notificadas quando uma ação ocorre em qualquer lugar do framework. Em resumo, sinais permitem certos *remetentes* notificar um conjunto de *receptores* sobre alguma ação que tenha ocorrido. Eles são especialmente úteis quando muitas peças de código podem estar interessados nos mesmos eventos.

O Django fornece um *conjunto de sinais embutidos* que deixam o código do usuário ser notificado pelo próprio Django sobre certas ações. Estas incluem algumas notificações úteis:

- `django.db.models.signals.pre_save` & `django.db.models.signals.post_save`

Enviado antes ou depois que um método de model `save()` é chamado.

- `django.db.models.signals.pre_delete` & `django.db.models.signals.post_delete`

Enviado antes ou depois que um método de model `delete()` é chamado.

- `django.core.signals.request_started` & `django.core.signals.request_finished`

Enviado quando o Django inicia ou finaliza uma requisição HTTP.

Veja a *documentação de sinais embutidos* para uma lista completa, e uma explicação completa de cada sinal.

Você também pode *definir e enviar seus próprios sinais*; veja abaixo.

Ouvindo sinais

Para receber um sinal, você precisa registrar uma função *receptora* que será chamada quando o sinal é enviado. Vamos ver como isto funciona registrando um sinal que é chamado depois de cada requisição HTTP é finalizada. Nós conectaremos-nos ao sinal `request_finished`.

Funções receptoras

Primeiro, nós precisamos definir uma função receptora. Um receptor pode ser qualquer função Python ou método:

```
def my_callback(sender, **kwargs):  
    print "Request finished!"
```

Observe que a função recebe um argumento `sender`, junto com o argumento coringa (`**kwargs`); todo manipulador de sinal recebe estes argumentos.

Nós olharemos o remetente *um pouco depois*, mas agora olhe o argumento `**kwargs`. Todos os sinais enviam argumentos nomeados, e podem mudar seus argumentos nomeados a qualquer momento. Neste caso o `request_finished`, ele está documentado como se não enviasse argumentos, que significa que poderíamos ser tentados a escrever nosso manipulador do sinal como `my_callback(sender)`. Isso poderia estar errado – de fato, o Django irá lançar um erro se você fizer isso. Isto porque, em qualquer ponto, argumentos poderiam ser adicionados ao sinal, e o receptor deve ser capaz de manipular estes argumentos.

Conectando funções receptoras

Agora, nós precisaremos conectar nosso receptor ao sinal:

```
from django.core.signals import request_finished
request_finished.connect(my_callback)
```

Agora, nossa função `my_callback` será chamada a cada vez que uma requisição for finalizada.

Onde este código deve ficar?

Você poder colocar o código manipulador e de registro de sinal aonde você quiser. Entretanto, você precisará assegurar-se de que o módulo já tenha sido importado anteriormente, para que este manipulador de sinal seja registrado antes que qualquer sinal seja enviado. Isto faz do `models.py` de sua aplicação, um bom lugar para colocar o registro de manipuladores de sinais.

Conectando-se a sinais enviados por remetentes específicos

Alguns sinais são enviados muitas vezes, mas você somente é interessado em receber um certo sub-conjunto destes sinais. Por exemplo, considere o sinal `django.db.models.signals.pre_save` enviado antes de um model ser salvo. Na maioria das vezes, você não precisa saber quando *qualquer* model foi salvo – só quando um model *específico* é salvo.

Nestes casos, você pode registrá-lo para receber sinais enviados somente por um remetente em particular. No caso do `django.db.models.signals.pre_save`, o remetente será a classe model que estiver sendo salva, então você pode indicar que você somente quer sinais enviados por algum model:

```
from django.db.models.signals import pre_save
from myapp.models import MyModel

def my_handler(sender, **kwargs):
    ...

pre_save.connect(my_handler, sender=MyModel)
```

A função `my_handler` somente será chamada quando uma instância de `MyModel` for salva.

Sinais diferentes usam objetos diferentes como seus remetentes; você precisará consultar a *documentação de sinais embutidos* para detalhes de cada sinal em particular.

Definindo e enviando sinais

Sua aplicação pode tirar proveito da infraestrutura de sinais e prover seus próprios sinais.

Definindo sinais

`class Signal ([providing_args=list])`

Todos os sinais são instâncias de `django.dispatch.Signal`. O `providing_args` é uma lista de nomes de argumentos que o sinal fornecerá aos ouvintes.

Por exemplo:

```
import django.dispatch

pizza_done = django.dispatch.Signal(providing_args=["toppings", "size"])
```

Este declara um sinal `pizza_done` que fornecerá aos receptores os argumentos `toppings` e `size`.

Lembre-se de que você está autorizado a alterar esta lista de argumentos a qualquer momento, então obter o direito de API, na primeira tentativa, não é necessário.

Enviando sinais

`Signal.send(sender, **kwargs)`

Para enviar um sinal, chame `Signal.send()`. Você deve prover o argumento `sender`, e pode fornecer muitos outros argumentos nomeados conforme sua vontade.

Por exemplo, aqui mostramos como nosso envio de sinal `pizza_done`, pode ficar:

```
class PizzaStore(object):
    ...

    def send_pizza(self, toppings, size):
        pizza_done.send(sender=self, toppings=toppings, size=size)
        ...
```


Part III

Guias “how-to”

Aqui você irá encontrar respostas curtas para questões do tipo “Como eu faço...?”. Estes guias how-to cobrem tópicos em profundidade – você terá de encontrar materiais em *Usando o Django* e *Referência da API*. No entanto, estes guias irão ajudá-lo a realizar rapidamente tarefas comuns.

Autenticando no Apache usuários do banco de dados do Django

Fazer autenticação mantendo múltiplas bases de dados em sincronia é um problema comum quando se trabalha com o Apache, por isso você pode configurar o Apache para usar o *sistema de autenticação* do Django diretamente. Por exemplo, você poderia:

- Servir arquivos static/media diretamente do Apache somente para usuários autenticados.
- Acessar um repositório [Subversion](#) autenticado para usuários Django com uma certa permissão.
- Permitir certos usuários conectarem-se para um compartilhamento WebDAV criado com o `mod_dav`.

Configurando o Apache

Para autenticar-se na base de dados do Django a partir de um arquivo de configurações do Apache, você precisará utilizar a diretiva `PythonAuthenHandler` do `mod_python` junto com as tradicionais diretivas `Auth*` e `Require`:

```
<Location /example/>
    AuthType Basic
    AuthName "example.com"
    Require valid-user

    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonAuthenHandler django.contrib.auth.handlers.modpython
</Location>
```

Usando o manipulador de autenticação com o Apache 2.2

Se você está usando Apache 2.2, você necessitará seguir alguns passos.

Você precisará se assegurar de que o `mod_auth_basic` e `mod_authz_user` estão carregados. Esses devem ser compilados estaticamente no Apache, ou você pode usar `LoadModule` para carregar dinamicamente (como demonstram os exemplos abaixo desta nota).

Você também precisará inserir diretivas de configuração que impedem que o Apache tente utilizar outros módulos autenticação, bem como especificar a diretiva `AuthUserFile` e apontá-la para `/dev/null`. Dependendo dos módulos de autenticação que você tenha carregado, você pode precisar ter uma ou mais das seguintes diretivas:

```
AuthBasicAuthoritative Off
AuthDefaultAuthoritative Off
AuthzLDAPAuthoritative Off
AuthzDBMAuthoritative Off
AuthzDefaultAuthoritative Off
AuthzGroupFileAuthoritative Off
AuthzOwnerAuthoritative Off
AuthzUserAuthoritative Off
```

Uma configuração completa, com direções entre Apache 2.0 e Apache 2.2 marcadas em negrito, poderia parecer algo assim:

```
LoadModule auth_basic_module modules/mod_auth_basic.so
LoadModule authz_user_module modules/mod_authz_user.so
```

...

```
<Location /example/>
    AuthType Basic
    AuthName "example.com"
    AuthUserFile /dev/null
    AuthBasicAuthoritative Off
    Require valid-user

    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonAuthenHandler django.contrib.auth.handlers.modpython
</Location>
```

Por padrão, o manipulador de autenticação limitará o acesso para localização `/example/` para usuários marcados como staff. Você pode usar um conjunto de diretivas `PythonOption` para modificar esse comportamento:

PythonOption	Explicação
DjangoRequireStaffStatus	Se marcado como <code>on</code> somente usuários “staff” (i.e. aqueles com o flag <code>is_staff</code> setado) serão permitidos. O padrão é <code>on</code> .
DjangoRequireSuperuserStatus	Se setado como <code>on</code> somente super-usuários (i.e. aqueles com o flag <code>is_superuser</code> setado) serão permitidos. O padrão é <code>off</code> .
DjangoPermissionName	O nome de uma permissão de acesso a exigir. Veja custom permissions para mais informações. Por padrão não será exigida autorização expressa.

Note que algumas vezes o `SetEnv` não funciona bem com esta configuração do `mod_python`, por razões desconhecidas. Se você está encontrando problemas para fazer o `mod_python` reconhecer seu `DJANGO_SETTINGS_MODULE`, você pode configurá-lo usando `PythonOption` em vez de `SetEnv`. Portanto, estas duas diretivas do Apache são equivalentes:

```
SetEnv DJANGO_SETTINGS_MODULE mysite.settings
PythonOption DJANGO_SETTINGS_MODULE mysite.settings
```

Escrevendo commando customizados para o django-admin

Please, see the release notes As aplicações podem registrar suas próprias ações com `manage.py`. Por exemplo, você pode querer adicionar uma ação `manage.py` para uma aplicação Django que você está distribuindo.

Para fazer isto, é só adicionar um diretório `management/commands` na sua aplicação. Cada módulo Python neste diretório será auto-descoberto e registrado como um comando que pode ser executado como uma ação quando você roda `manage.py`:

```
blog/  
  __init__.py  
  models.py  
  management/  
    __init__.py  
    commands/  
      __init__.py  
      explode.py  
  views.py
```

Neste exemplo, o comando `explode` será disponibilizado para qualquer projeto que incluir a aplicação `blog` no `settings.INSTALLED_APPS`.

O módulo `explode.py` tem somente um requerimento – ele deve definir uma classe chamada `Command` que estende `django.core.management.base.BaseCommand`.

Para mais detalhes de como definir seus próprios comandos, olhe o código dos comandos do `django-admin.py`, em `/django/core/management/commands`.

Writing custom model fields

Please, see the release notes

Introduction

The *model reference* documentation explains how to use Django’s standard field classes – *CharField*, *DateField*, etc. For many purposes, those classes are all you’ll need. Sometimes, though, the Django version won’t meet your precise requirements, or you’ll want to use a field that is entirely different from those shipped with Django.

Django’s built-in field types don’t cover every possible database column type – only the common types, such as *VARCHAR* and *INTEGER*. For more obscure column types, such as geographic polygons or even user-created types such as *PostgreSQL custom types*, you can define your own Django *Field* subclasses.

Alternatively, you may have a complex Python object that can somehow be serialized to fit into a standard database column type. This is another case where a *Field* subclass will help you use your object with your models.

Our example object

Creating custom fields requires a bit of attention to detail. To make things easier to follow, we’ll use a consistent example throughout this document: wrapping a Python object representing the deal of cards in a hand of *Bridge*. Don’t worry, you don’t have to know how to play Bridge to follow this example. You only need to know that 52 cards are dealt out equally to four players, who are traditionally called *north*, *east*, *south* and *west*. Our class looks something like this:

```
class Hand(object):
    def __init__(self, north, east, south, west):
        # Input parameters are lists of cards ('Ah', '9s', etc)
        self.north = north
        self.east = east
        self.south = south
        self.west = west

    # ... (other possibly useful methods omitted) ...
```

This is just an ordinary Python class, with nothing Django-specific about it. We’d like to be able to do things like this in our models (we assume the *hand* attribute on the model is an instance of *Hand*):

```
example = MyModel.objects.get(pk=1)
print example.hand.north

new_hand = Hand(north, east, south, west)
example.hand = new_hand
example.save()
```

We assign to and retrieve from the `hand` attribute in our model just like any other Python class. The trick is to tell Django how to handle saving and loading such an object.

In order to use the `Hand` class in our models, we **do not** have to change this class at all. This is ideal, because it means you can easily write model support for existing classes where you cannot change the source code.

Note: You might only be wanting to take advantage of custom database column types and deal with the data as standard Python types in your models; strings, or floats, for example. This case is similar to our `Hand` example and we'll note any differences as we go along.

Background theory

Database storage

The simplest way to think of a model field is that it provides a way to take a normal Python object – string, boolean, `datetime`, or something more complex like `Hand` – and convert it to and from a format that is useful when dealing with the database (and serialization, but, as we'll see later, that falls out fairly naturally once you have the database side under control).

Fields in a model must somehow be converted to fit into an existing database column type. Different databases provide different sets of valid column types, but the rule is still the same: those are the only types you have to work with. Anything you want to store in the database must fit into one of those types.

Normally, you're either writing a Django field to match a particular database column type, or there's a fairly straightforward way to convert your data to, say, a string.

For our `Hand` example, we could convert the card data to a string of 104 characters by concatenating all the cards together in a pre-determined order – say, all the *north* cards first, then the *east*, *south* and *west* cards. So `Hand` objects can be saved to text or character columns in the database.

What does a field class do?

All of Django's fields (and when we say *fields* in this document, we always mean model fields and not *form fields*) are subclasses of `django.db.models.Field`. Most of the information that Django records about a field is common to all fields – name, help text, uniqueness and so forth. Storing all that information is handled by `Field`. We'll get into the precise details of what `Field` can do later on; for now, suffice it to say that everything descends from `Field` and then customizes key pieces of the class behavior.

It's important to realize that a Django field class is not what is stored in your model attributes. The model attributes contain normal Python objects. The field classes you define in a model are actually stored in the `Meta` class when the model class is created (the precise details of how this is done are unimportant here). This is because the field classes aren't necessary when you're just creating and modifying attributes. Instead, they provide the machinery for converting between the attribute value and what is stored in the database or sent to the *serializer*.

Keep this in mind when creating your own custom fields. The Django `Field` subclass you write provides the machinery for converting between your Python instances and the database/serializer values in various ways (there are differences between storing a value and using a value for lookups, for example). If this sounds a bit tricky, don't worry – it will become clearer in the examples below. Just remember that you will often end up creating two classes when you want a custom field:

- The first class is the Python object that your users will manipulate. They will assign it to the model attribute, they will read from it for displaying purposes, things like that. This is the `Hand` class in our example.
- The second class is the `Field` subclass. This is the class that knows how to convert your first class back and forth between its permanent storage form and the Python form.

Writing a field subclass

When planning your `Field` subclass, first give some thought to which existing `Field` class your new field is most similar to. Can you subclass an existing Django field and save yourself some work? If not, you should subclass the `Field` class, from which everything is descended.

Initializing your new field is a matter of separating out any arguments that are specific to your case from the common arguments and passing the latter to the `__init__()` method of `Field` (or your parent class).

In our example, we'll call our field `HandField`. (It's a good idea to call your `Field` subclass `<Something>Field`, so it's easily identifiable as a `Field` subclass.) It doesn't behave like any existing field, so we'll subclass directly from `Field`:

```
from django.db import models

class HandField(models.Field):
    def __init__(self, *args, **kwargs):
        kwargs['max_length'] = 104
        super(HandField, self).__init__(*args, **kwargs)
```

Our `HandField` accepts most of the standard field options (see the list below), but we ensure it has a fixed length, since it only needs to hold 52 card values plus their suits; 104 characters in total.

Note: Many of Django's model fields accept options that they don't do anything with. For example, you can pass both `editable` and `auto_now` to a `django.db.models.DateField` and it will simply ignore the `editable` parameter (`auto_now` being set implies `editable=False`). No error is raised in this case.

This behavior simplifies the field classes, because they don't need to check for options that aren't necessary. They just pass all the options to the parent class and then don't use them later on. It's up to you whether you want your fields to be more strict about the options they select, or to use the simpler, more permissive behavior of the current fields.

The `__init__()` method takes the following parameters:

- `verbose_name`
- `name`
- `primary_key`
- `max_length`
- `unique`
- `blank`
- `null`
- `db_index`
- `rel`: Used for related fields (like `ForeignKey`). For advanced use only.
- `default`
- `editable`
- `serialize`: If `False`, the field will not be serialized when the model is passed to Django's *serializers*. Defaults to `True`.

- `unique_for_date`
- `unique_for_month`
- `unique_for_year`
- `choices`
- `help_text`
- `db_column`
- `db_tablespace`: Currently only used with the Oracle backend and only for index creation. You can usually ignore this option.
- `auto_created`: True if the field was automatically created, as for the *OneToOneField* used by model inheritance. For advanced use only.

All of the options without an explanation in the above list have the same meaning they do for normal Django fields. See the *field documentation* for examples and details.

The `SubfieldBase` metaclass

As we indicated in the *introduction*, field subclasses are often needed for two reasons: either to take advantage of a custom database column type, or to handle complex Python types. Obviously, a combination of the two is also possible. If you're only working with custom database column types and your model fields appear in Python as standard Python types direct from the database backend, you don't need to worry about this section.

If you're handling custom Python types, such as our `Hand` class, we need to make sure that when Django initializes an instance of our model and assigns a database value to our custom field attribute, we convert that value into the appropriate Python object. The details of how this happens internally are a little complex, but the code you need to write in your `Field` class is simple: make sure your field subclass uses a special metaclass:

```
class django.db.models.SubfieldBase
```

For example:

```
class HandField(models.Field):
    __metaclass__ = models.SubfieldBase

    def __init__(self, *args, **kwargs):
        # ...
```

This ensures that the `to_python()` method, documented below, will always be called when the attribute is initialized.

Useful methods

Once you've created your `Field` subclass and set up the `__metaclass__`, you might consider overriding a few standard methods, depending on your field's behavior. The list of methods below is in approximately decreasing order of importance, so start from the top.

Custom database types

`db_type(self)`

Returns the database column data type for the `Field`, taking into account the current `DATABASE_ENGINE` setting.

Say you've created a PostgreSQL custom type called `mytype`. You can use this field with Django by subclassing `Field` and implementing the `db_type()` method, like so:

```
from django.db import models

class MytypeField(models.Field):
    def db_type(self):
        return 'mytype'
```

Once you have `MytypeField`, you can use it in any model, just like any other `Field` type:

```
class Person(models.Model):
    name = models.CharField(max_length=80)
    gender = models.CharField(max_length=1)
    something_else = MytypeField()
```

If you aim to build a database-agnostic application, you should account for differences in database column types. For example, the date/time column type in PostgreSQL is called `timestamp`, while the same column in MySQL is called `datetime`. The simplest way to handle this in a `db_type()` method is to import the Django settings module and check the `DATABASE_ENGINE` setting. For example:

```
class MyDateField(models.Field):
    def db_type(self):
        from django.conf import settings
        if settings.DATABASE_ENGINE == 'mysql':
            return 'datetime'
        else:
            return 'timestamp'
```

The `db_type()` method is only called by Django when the framework constructs the `CREATE TABLE` statements for your application – that is, when you first create your tables. It's not called at any other time, so it can afford to execute slightly complex code, such as the `DATABASE_ENGINE` check in the above example.

Some database column types accept parameters, such as `CHAR(25)`, where the parameter 25 represents the maximum column length. In cases like these, it's more flexible if the parameter is specified in the model rather than being hard-coded in the `db_type()` method. For example, it wouldn't make much sense to have a `CharMaxlength25Field`, shown here:

```
# This is a silly example of hard-coded parameters.
class CharMaxlength25Field(models.Field):
    def db_type(self):
        return 'char(25)'

# In the model:
class MyModel(models.Model):
    # ...
    my_field = CharMaxlength25Field()
```

The better way of doing this would be to make the parameter specifiable at run time – i.e., when the class is instantiated. To do that, just implement `django.db.models.Field.__init__()`, like so:

```
# This is a much more flexible example.
class BetterCharField(models.Field):
    def __init__(self, max_length, *args, **kwargs):
        self.max_length = max_length
        super(BetterCharField, self).__init__(*args, **kwargs)

    def db_type(self):
        return 'char(%s)' % self.max_length

# In the model:
class MyModel(models.Model):
    # ...
    my_field = BetterCharField(25)
```

Finally, if your column requires truly complex SQL setup, return `None` from `db_type()`. This will cause Django's SQL creation code to skip over this field. You are then responsible for creating the column in the right table in some other way, of course, but this gives you a way to tell Django to get out of the way.

Converting database values to Python objects

`to_python(self, value)`

Converts a value as returned by your database (or a serializer) to a Python object.

The default implementation simply returns `value`, for the common case in which the database backend already returns data in the correct format (as a Python string, for example).

If your custom `Field` class deals with data structures that are more complex than strings, dates, integers or floats, then you'll need to override this method. As a general rule, the method should deal gracefully with any of the following arguments:

- An instance of the correct type (e.g., `Hand` in our ongoing example).
- A string (e.g., from a deserializer).
- Whatever the database returns for the column type you're using.

In our `HandField` class, we're storing the data as a `VARCHAR` field in the database, so we need to be able to process strings and `Hand` instances in `to_python()`:

```
import re

class HandField(models.Field):
    # ...

    def to_python(self, value):
        if isinstance(value, Hand):
            return value

        # The string case.
        p1 = re.compile('{26}')
        p2 = re.compile('.')
        args = [p2.findall(x) for x in p1.findall(value)]
        return Hand(*args)
```

Notice that we always return a `Hand` instance from this method. That's the Python object type we want to store in the model's attribute.

Remember: If your custom field needs the `to_python()` method to be called when it is created, you should be using *The SubfieldBase metaclass* mentioned earlier. Otherwise `to_python()` won't be called automatically.

Converting Python objects to database values

`get_db_prep_value(self, value)`

This is the reverse of `to_python()` when working with the database backends (as opposed to serialization). The `value` parameter is the current value of the model's attribute (a field has no reference to its containing model, so it cannot retrieve the value itself), and the method should return data in a format that can be used as a parameter in a query for the database backend.

For example:

```
class HandField(models.Field):
    # ...

    def get_db_prep_value(self, value):
        return ''.join([''.join(l) for l in (value.north,
            value.east, value.south, value.west)])
```

```
get_db_prep_save (self, value)
```

Same as the above, but called when the Field value must be *saved* to the database. As the default implementation just calls `get_db_prep_value`, you shouldn't need to implement this method unless your custom field needs a special conversion when being saved that is not the same as the conversion used for normal query parameters (which is implemented by `get_db_prep_value`).

Preprocessing values before saving

```
pre_save (self, model_instance, add)
```

This method is called just prior to `get_db_prep_save()` and should return the value of the appropriate attribute from `model_instance` for this field. The attribute name is in `self.attname` (this is set up by Field). If the model is being saved to the database for the first time, the `add` parameter will be `True`, otherwise it will be `False`.

You only need to override this method if you want to preprocess the value somehow, just before saving. For example, Django's `DateTimeField` uses this method to set the attribute correctly in the case of `auto_now` or `auto_now_add`.

If you do override this method, you must return the value of the attribute at the end. You should also update the model's attribute if you make any changes to the value so that code holding references to the model will always see the correct value.

Preparing values for use in database lookups

```
get_db_prep_lookup (self, lookup_type, value)
```

Prepares the value for passing to the database when used in a lookup (a WHERE constraint in SQL). The `lookup_type` will be one of the valid Django filter lookups: `exact`, `iexact`, `contains`, `icontains`, `gt`, `gte`, `lt`, `lte`, `in`, `startswith`, `istartswith`, `endswith`, `iendswith`, `range`, `year`, `month`, `day`, `isnull`, `search`, `regex`, and `iregex`.

Your method must be prepared to handle all of these `lookup_type` values and should raise either a `ValueError` if the value is of the wrong sort (a list when you were expecting an object, for example) or a `TypeError` if your field does not support that type of lookup. For many fields, you can get by with handling the lookup types that need special handling for your field and pass the rest to the `get_db_prep_lookup()` method of the parent class.

If you needed to implement `get_db_prep_save()`, you will usually need to implement `get_db_prep_lookup()`. If you don't, `get_db_prep_value` will be called by the default implementation, to manage `exact`, `gt`, `gte`, `lt`, `lte`, `in` and `range` lookups.

You may also want to implement this method to limit the lookup types that could be used with your custom field type.

Note that, for `range` and `in` lookups, `get_db_prep_lookup` will receive a list of objects (presumably of the right type) and will need to convert them to a list of things of the right type for passing to the database. Most of the time, you can reuse `get_db_prep_value()`, or at least factor out some common pieces.

For example, the following code implements `get_db_prep_lookup` to limit the accepted lookup types to `exact` and `in`:

```
class HandField(models.Field):
    # ...

    def get_db_prep_lookup(self, lookup_type, value):
        # We only handle 'exact' and 'in'. All others are errors.
        if lookup_type == 'exact':
            return [self.get_db_prep_value(value)]
        elif lookup_type == 'in':
```



```
        return [self.get_db_prep_value(v) for v in value]
    else:
        raise TypeError('Lookup type %r not supported.' % lookup_type)
```

Specifying the form field for a model field

formfield(*self*, *form_class=forms.CharField*, ***kwargs*)

Returns the default form field to use when this field is displayed in a model. This method is called by the `ModelForm` helper.

All of the `kwargs` dictionary is passed directly to the form field's `Field__init__()` method. Normally, all you need to do is set up a good default for the `form_class` argument and then delegate further handling to the parent class. This might require you to write a custom form field (and even a form widget). See the [forms documentation](#) for information about this, and take a look at the code in [django.contrib.localflavor](#) for some examples of custom widgets.

Continuing our ongoing example, we can write the `formfield()` method as:

```
class HandField(models.Field):
    # ...

    def formfield(self, **kwargs):
        # This is a fairly standard way to set up some defaults
        # while letting the caller override them.
        defaults = {'form_class': MyFormField}
        defaults.update(kwargs)
        return super(HandField, self).formfield(**defaults)
```

This assumes we've imported a `MyFormField` field class (which has its own default widget). This document doesn't cover the details of writing custom form fields.

Emulating built-in field types

get_internal_type(*self*)

Returns a string giving the name of the `Field` subclass we are emulating at the database level. This is used to determine the type of database column for simple cases.

If you have created a `db_type()` method, you don't need to worry about `get_internal_type()` – it won't be used much. Sometimes, though, your database storage is similar in type to some other field, so you can use that other field's logic to create the right column.

For example:

```
class HandField(models.Field):
    # ...

    def get_internal_type(self):
        return 'CharField'
```

No matter which database backend we are using, this will mean that `syncdb` and other SQL commands create the right column type for storing a string.

If `get_internal_type()` returns a string that is not known to Django for the database backend you are using – that is, it doesn't appear in `django.db.backends.<db_name>.creation.DATA_TYPES` – the string will still be used by the serializer, but the default `db_type()` method will return `None`. See the documentation of `db_type()` for reasons why this might be useful. Putting a descriptive string in as the type of the field for the serializer is a useful idea if you're ever going to be using the serializer output in some other place, outside of Django.

Converting field data for serialization

value_to_string(*self*, *obj*)

This method is used by the serializers to convert the field into a string for output. Calling `Field._get_val_from_obj(obj)()` is the best way to get the value to serialize. For example, since our `HandField` uses strings for its data storage anyway, we can reuse some existing conversion code:

```
class HandField(models.Field):
    # ...

    def value_to_string(self, obj):
        value = self._get_val_from_obj(obj)
        return self.get_db_prep_value(value)
```

Some general advice

Writing a custom field can be a tricky process, particularly if you're doing complex conversions between your Python types and your database and serialization formats. Here are a couple of tips to make things go more smoothly:

1. Look at the existing Django fields (in `django/db/models/fields/__init__.py`) for inspiration. Try to find a field that's similar to what you want and extend it a little bit, instead of creating an entirely new field from scratch.
2. Put a `__str__()` or `__unicode__()` method on the class you're wrapping up as a field. There are a lot of places where the default behavior of the field code is to call `force_unicode()` on the value. (In our examples in this document, `value` would be a `Hand` instance, not a `HandField`). So if your `__unicode__()` method automatically converts to the string form of your Python object, you can save yourself a lot of work.

Writing a `FileField` subclass

In addition to the above methods, fields that deal with files have a few other special requirements which must be taken into account. The majority of the mechanics provided by `FileField`, such as controlling database storage and retrieval, can remain unchanged, leaving subclasses to deal with the challenge of supporting a particular type of file.

Django provides a `File` class, which is used as a proxy to the file's contents and operations. This can be subclassed to customize how the file is accessed, and what methods are available. It lives at `django.db.models.fields.files`, and its default behavior is explained in the [file documentation](#).

Once a subclass of `File` is created, the new `FileField` subclass must be told to use it. To do so, simply assign the new `File` subclass to the special `attr_class` attribute of the `FileField` subclass.

A few suggestions

In addition to the above details, there are a few guidelines which can greatly improve the efficiency and readability of the field's code.

1. The source for Django's own `ImageField` (in `django/db/models/fields/files.py`) is a great example of how to subclass `FileField` to support a particular type of file, as it incorporates all of the techniques described above.
2. Cache file attributes wherever possible. Since files may be stored in remote storage systems, retrieving them may cost extra time, or even money, that isn't always necessary. Once a file is retrieved to obtain some data about its content, cache as much of that data as possible to reduce the number of times the file must be retrieved on subsequent calls for that information.

Tags e filtros de template personalizados

Introdução

O sistema de template do Django vem com uma larga variedade de *tags e filtros embutidos* projetados para direcionar a lógica da apresentação que sua aplicação. Todavia, você pode se encontrar precisando de funcionalidades que não são cobertas pelo conjunto de primitivas de template. Você pode estender o motor de template definindo tags e filtros personalizados usando Python, e então torná-los disponíveis aos seus templates usando a tag `{% load %}`.

Layout do código

Tags e filtros de template customizados devem estar dentro de uma Django app. Se elas estão relacionadas a uma app existente, faz mais sentido estarem empacotados na app; caso contrário, você deveria criar uma nova app para armazená-los.

A app deve conter um diretório `templatetags`, no mesmo nível do `models.py`, `views.py`, etc. Se ele não existe ainda, é só criá-lo - não esqueça o arquivo `__init__.py` para assegurar que o diretório seja tratado como um pacote Python.

Suas tags e filtros personalizados ficaram nesse módulo dentro do diretório `templatetags`. O nome do arquivo do módulo é o nome que você usará para carregar as tags depois, então seja cuidadoso ao criar um nome que não colida com as tags e filtros de outras apps.

Por exemplo, se sua tags/filtros estão num arquivo chamado `poll_extras.py`, o layout de sua app deve parecer com isso:

```
polls/
  models.py
  templatetags/
    __init__.py
    poll_extras.py
  views.py
```

E nos seus templates você poderia usar o seguinte:

```
{% load poll_extras %}
```

A app que contém as tags personalizadas deve estar no `INSTALLED_APPS` para que a tag `{% load %}` funcione. Esta é uma medida de segurança: Ela permite que você armazene código Python para muitas bibliotecas de template num único servidor, sem permitir o acesso a todas elas para toda instalação do Django.

Não há limites de quantos módulos você pode colocar no pacote `templatetags`. Só tenha em mente que uma declaração `{% load %}` carregará tags/filtros para um dado nome de módulo Python, não o nome da app.

Para uma biblioteca de tag ser válida, o módulo deve conter uma variável chamada `register` que é uma instância do `templates.Library`, na qual todas as tags e filtros são registrados. Então, no topo de seu módulo, coloque o seguinte:

```
from django import template

register = template.Library()
```

Por trás das cenas

Para uma tonelada de exemplos, leia o código fonte dos filtros e tags padrão do Django. Eles estão em `django/templates/defaultfilters.py` e `django/template/defaulttags.py`, respectivamente.

Escrevendo filtros de template personalizados

Filtros personalizados são como funções Python que recebem um ou dois argumentos:

- O valor de uma variável (input) – não necessariamente uma string.
- O valor de um argumento – este pode ter um valor padrão, ou ser deixado de fora.

Por exemplo, no filtro `{{ var|foo:"bar" }}`, ao filtro `foo` seria passado a variável `var` e o argumento `"bar"`.

Funções filtro devem sem retornar algo. Elas não lançam exceções. Elas falham silenciosamente. No caso de um erro, elas devem retornar a entrada original ou uma string vazia – o que fizer mais sentido.

Aqui temos um exemplo de definição de filtro:

```
def cut(value, arg):
    "Remove todos os valores do arg da string fornecida."
    return value.replace(arg, '')
```

E aqui tem um exemplo de como este filtro poderia ser usado:

```
{{ somevariable|cut:"0" }}
```

A maioria dos filtros não recebem argumentos. Neste caso, é só deixar o argumento fora de sua função. Exemplo:

```
def lower(value): # Somente um argumento.
    "Converte uma string para minúsculo"
    return value.lower()
```

Filtros de template que esperam strings

Se você estiver escrevendo um filtro de template que somente espera uma string como o primeiro argumento, você deve usar o decorador `stringfilter`. Este converterá um objeto para seu valor em string antes de ser passado para a sua função:

```
from django.template.defaultfilters import stringfilter

@stringfilter
```

```
def lower(value):
    return value.lower()
```

Desta forma, você será capaz de passar, digamos, um inteiro para este filtro, e ele não causará um `AttributeError` (pois inteiros não possuem métodos `lower()`).

Registrando filtros personalizados

Uma vez que tenha escrito sua definição de filtro, você precisa registrá-lo na sua instância `Library`, para torná-lo disponível na linguagem de template do Django:

```
register.filter('cut', cut)
register.filter('lower', lower)
```

O método `Library.filter()` recebe dois argumentos:

1. O nome do filtro – uma string.
2. A função de compilação – uma função Python (não o nome da função como uma string).

Se você estiver usando Python 2.4 ou superior, você pode usar `register.filter()` como um decorador, no entanto:

```
@register.filter(name='cut')
@stringfilter
def cut(value, arg):
    return value.replace(arg, '')

@register.filter
@stringfilter
def lower(value):
    return value.lower()
```

Se você deixar desligado o argumento `name`, como no segundo exemplo acima, o Django usará o nome da função como o nome do filtro.

Filtros e auto-escaping

Please, see the release notes Quando estiver escrevendo um filtro personalizado, pense um pouco em como o filtro irá interagir com o comportamento de auto-escaping do Django. Note que há três tipos de strings que podem ser passadas por dentro de um código de template:

- **Strings puras** são tipos nativos do Python `str` ou `unicode`. Na saída, elas são escapadas se o auto-escaping estiver em efeito e apresenta-se inalterado.
- **Strings seguras** são strings que foram marcadas como seguras por um escape fora de tempo. Qualquer escape necessário já foi feito. Elas são comumente usadas para a saída que contém HTML puro que destina-se a ser interpretado, assim como está, no lado do cliente.

Internamente, estas strings são do tipo `SafeString` ou `SafeUnicode`. Elas compartilham uma classe básica comum de `SafeData`, então você pode testá-las usando um código como este:

```
if isinstance(value, SafeData):
    # Faça algo com a string "safe".
```

- **Strings marcadas como “precisando escapar”** são *sempre* escapadas na saída, indiferente se elas estão num bloco `autoescape` ou não. Essas strings são somente escapadas uma vez, contudo, mesmo se o auto-escaping se aplica.

Internamente, estas strings são do tipo `EscapeString` ou `EscapeUnicode`. Geralmente você não tem de se preocupar com eles; eles existem para implementação do filtro `escape`.

Código de filtros de template caem em uma de duas situações:

1. Seu filtro não introduz quaisquer caracteres HTML não seguros (<, >, ', " or &) no resultado que ainda não foi apresentado. Neste caso, você pode deixar o Django se preocupar com todo o auto-escaping por você. Tudo que você precisa fazer é colocar o atributo `is_safe` sobre o sua função filtro e setá-lo como `True`, assim:

```
@register.filter
def myfilter(value):
    return value
myfilter.is_safe = True
```

Este atributo diz ao Django que se uma string “safe” é passada para o seu filtro, o resultado será mantido “safe” e se uma string não-safe é passada, o Django automaticamente a escapará, se necessário.

Você pode pensar que nisso como significando “este filtro é seguro – ele não introduz qualquer possibilidade de HTML não seguro.”

A razão de `is_safe` ser necessário é porque há abundância de operadores de string normais que tornarão um objeto `SafeData` uma string normal `str` ou `unicode` e, ao invés de tentar pegá-los todos, o que será muito difícil, o Django repara o dano depois que o filtro foi completado.

Por exemplo, suponhamos que você tem um filtro que adiciona uma string `xx` no final de uma entrada. Já que isso não introduz caracteres HTML perigosos ao resultado (com exceção de alguns que já estavam presentes), você deve marcar seu filtro com `is_safe`:

```
@register.filter
def add_xx(value):
    return '%sxx' % value
add_xx.is_safe = True
```

Quando este filtro é usado num template onde auto-escaping está habilitado, o Django escapa a saída sempre que a entrada já não estiver marcada como “safe”.

Por padrão, `is_safe` é `False`, e você pode omiti-lo de qualquer filtro onde ele não é requerido.

Seja cuidadoso quando decidir se seu filtro realmente deixa strings safe como safe. Se estiver *removendo* caracteres, você pode inadvertidamente deixa tabs HTML desbalanceadas ou entidades no resultado. Por exemplo, removendo um `>` de uma entrada por tornar um `<a>` em `<a`, que seria necessário escapar na saída para evitar problemas. Similarmente, remover um ponto-e-vírgula (`;`) pode transformar um `&` em `&`, que não é uma entidade válida e isso precisa ser escapado. Na maioria dos casos não será tão complicado, mas mantenha o olho aberto quanto a problemas como esse ao revisar o seu código.

Marcando um filtro como `is_safe` forçará o valor de retorno do filtro a ser uma string. Se seu filtro deve retornar um booleano ou outro valor não string, marcando-o `is_safe` provavelmente terá consequências intencionais (como uma conversão do booleano `False` para uma string `'False'`).

2. Alternativamente, o código do seu filtro pode manualmente se preocupar com qualquer escape. Isso é necessário quando você estiver introduzindo uma nova marcação HTML dentro do resultado. Você quer marcar a saída de escape como seguro de forma que sua marcação HTML não foi escapada, então você precisará manipular a saída você mesmo.

Para marcar a saída como uma string safe, use `django.utils.safestring.mark_safe()`.

Seja cuidadoso, contudo. Você precisa fazer mais do que somente marcar a saída como safe. Você precisa assegurar-se de que realmente *é* safe, e o que você faz depende de se auto-escaping está em vigor. A idéia é escrever filtros que possam operar nos templates onde auto-escaping está ligado ou desligado para tornar as coisas mais fáceis para os seus autores de template.

A fim de seu filtro saber o estado de auto-escaping atual, configure o atributo `needs_autoescape` como `True` na sua função. (Se você não especificar este atributo, o padrão é `False`). Este atributo diz ao Django que sua função filtro espera um argumento extra, chamado `autoescape`, que é `True` se auto-escaping está em vigor é `False` caso contrário.

Por exemplo, vamos escrever um filtro que enfatiza o primeiro caracter de uma string:

```

from django.utils.html import conditional_escape
from django.utils.safestring import mark_safe

def initial_letter_filter(text, autoescape=None):
    first, other = text[0], text[1:]
    if autoescape:
        esc = conditional_escape
    else:
        esc = lambda x: x
    result = '<strong>%s</strong>%s' % (esc(first), esc(other))
    return mark_safe(result)
initial_letter_filter.needs_autoescape = True

```

O atributo `needs_autoescape` sobre a função `filter` e argumento `autoescape` significa que nossa função saberá se o escape automático está em efeito quando o filtro é chamado. Nós usamos `autoescape` para decidir se a entrada de dados precisa ser passada através do `django.utils.html.conditional_escape` ou não. (No último caso, nós só o usamos para identificar funções como a função “escape”.) A função `conditional_escape()` é como `escape()` exceto que somente escapa a entrada que **não** for uma instância de `SafeData`. Se um `SafeData` é passado para o `conditional_escape()`, os dados são retornados sem mudanças.

Finalmente, no exemplo acima, nós lembramos de marcar o resultado como `safe`, para que nosso HTML seja inserido diretamente dentro do template sem escape adicional.

Não há necessidade de se preocupar com o atributo `is_safe` neste caso (embora não inclua não doerá nada). Sempre que você manipular manualmente as questões de auto-escaping e retornar uma string `safe`, o atributo `is_safe` não mudará nada de qualquer forma.

Escrevendo tags de template personalizado

Tags são mais complexas que filtros, porque tags podem fazer qualquer coisa.

Uma rápida visão geral

Acima, nesse documento é explicado que o sistema de template funciona num processo de dois passos: compilação e renderização. Para definir uma tag de template personalizada, você especifica com a compilação funciona e como a renderização funciona.

Quando o Django compila um template, ele divide o texto do template puro dentro de “nodes”. Cada nodo é uma instância do `django.template.Node` e tem um método `render()`. Um template compilado é, simplesmente, uma lista de objetos `Node`. Quando você chama `render()` sobre um objeto de template compilado, o template chama `render()` em cada `Node` na sua lista, com o dado contexto. Os resultados são todos concatenados para formar a saída do template.

Deste modo, para definir uma tag de template personalizada, você especifica como o template puro é convertido num `Node` (a função compilação), e o que o método `render()` do nodo faz.

Escrevendo a função de compilação

Para cada template tag o parser de template encontra, ele chama uma função Python com o conteúdo da tag e o objeto parser em si. Esta função é responsável por retornar uma instância `Node` baseado nos conteúdos da tag.

Por exemplo, vamos escrever uma template tag, `{% current_time %}`, que mostra o data/hora atual, formatado de acordo com o parâmetro dado na tag, na [sintaxe do strftime](#). É uma boa idéia decidir a sintaxe da tag antes de qualquer outra coisa. No nosso caso, vamos dizer que a tag deve ser usada desta forma:

```
<p>A hora é {% current_time "%Y-%m-%d %I:%M %p" %}.</p>
```

O parser para esta função deve abarrar o parâmetro e criar um objeto `Node`:


```
from django import template
def do_current_time(parser, token):
    try:
        # split_contents() sabe que não é para dividir strings entre aspas.
        tag_name, format_string = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError, "%r tag requires a single argument" % token.contents.split()[0]
    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', "'")):
        raise template.TemplateSyntaxError, "%r tag's argument should be in quotes" % tag_name
    return CurrentTimeNode(format_string[1:-1])
```

Notas:

- O parser é o objeto parser de template. Não precisamos dele nesse exemplo.
- O token.contents é uma string de conteúdos puros da tag. No nosso exemplo, é 'current_time "%Y-%m-%d %I:%M %p"'.
- O método token.split_contents() quebra os argumentos nos espaços mantendo as strings entre aspas agrupadas. O mais simples token.contents.split() não seria tão robusto, e dividiria ingenuamente todos os espaços, incluindo aqueles dentro de strings entre aspas agrupadas. É uma boa idéia sempre usar token.split_contents().
- Essa função é responsável por lançar django.template.TemplateSyntaxError, com mensagens úteis, para qualquer erro de sintaxe.
- As exceções TemplateSystemError usam a variável tag_name. Não embute o nome da tag nas suas mensagens de erro, porque eles casam com o nome de sua função. O token.contents.split()[0] “sempre” será o nome da sua tag – mesmo quando a tag não tenha argumentos.
- A função retorna um CurrentTimeNode com tudo o que o nodo precisa saber sobre esta tag. Neste caso, é só passar o argumento – "%Y-%m-%d %I:%M %p". As aspas, no início e no final, da template tag são removidas no format_string[1:-1].
- O parseamento é de muito baixo nível. Os desenvolvedores do Django têm experimentado escrever pequenos frameworks sobre o este sistema de parse, usando técnicas como gramáticas EBNF, mas estes experimentos tornam o motor de template muito lento. Ele é de baixo nível porque é mais rápido.

Escrevendo o renderizador

O segundo passo em escrever tags personalizadas é definir uma subclasse Node que tem um método render().

Continuando o exemplo acima, nós precisamos definir CurrentTimeNode:

```
from django import template
import datetime
class CurrentTimeNode(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string
    def render(self, context):
        return datetime.datetime.now().strftime(self.format_string)
```

Notas:

- __init__() recebe o format_string do do_current_time(). Sempre passe quaisquer opções/parâmetros/argumentos para um Node via seu __init__().
- O método render() é onde o trabalho realmente acontece.
- O render() nunca deve lançar um TemplateSyntaxError ou qualquer outra exceção. Ele deve falhar silenciosamente, assim como os filtros de template o fazem.

Ultimamente, essa dissociação de compilação e renderização resulta num sistema de template eficiente, pois um template pode renderizar vários contextos sem precisar ser parsado múltiplas vezes.

Considerações do Auto-escaping

Please, see the release notes A saída de uma template tag *não* é automaticamente executada através de filtros de auto-escaping. Entretanto, ainda há algumas coisas que você deve ter em mente quando estiver escrevendo uma template tag.

Se a função `render()` de seu template armazena o resultado numa variável de contexto (ao invés de retornar o resultado numa string), ela deverá se preocupar em chamar `mark_safe()` se for apropriado. Quando a variável é finalmente renderizada, ela será afetada pela configuração auto-escape em efeito nessa hora, assim o conteúdo que deveria ser seguro além de escapado precisa ser marcado como tal.

Também, se sua template tag cria um novo contexto para executar algumas sub-renderizações, configure o atributo auto-escape para o valor do contexto atual. O método `__init__` para a classe `Context` recebe um parâmetro chamado `autoescape` que você pode usar para este propósito. Por exemplo:

```
def render(self, context):
    # ...
    new_context = Context({'var': obj}, autoescape=context.autoescape)
    # ... Faça algo com new_context ...
```

Isso não é uma situação muito comum, mas é útil se você estiver renderizando um template você mesmo. Por exemplo:

```
def render(self, context):
    t = template.loader.get_template('small_fragment.html')
    return t.render(Context({'var': obj}, autoescape=context.autoescape))
```

Se nós tivéssemos esquecido de passar o valor atual `context.autoescape` para nosso novo `Context` neste exemplo, os resultados teriam *sempre* de ser automaticamente escapados, o que pode não ser um comportamento desejável se a template tag é usada dentro de um bloco `{% autoescape off %}`.

Registrando a tag

Finalmente, registrar a tag com sua instância `Library` do módulo, como explicado in “Escrevendo filtros de template personalizados” acima. Exemplo:

```
register.tag('current_time', do_current_time)
```

O método `tag()` recebe dois argumentos:

1. O nome da template tag – uma string. Se isso for deixado de fora, o nome da função de compilação será usado.
2. A função de compilação – uma função Python (não o nome da função como uma string).

Como no registro de filtros, também é possível usar isso como um decorador, no Python 2.4 ou superior:

```
@register.tag(name="current_time")
def do_current_time(parser, token):
    # ...

@register.tag
def shout(parser, token):
    # ...
```

Se você deixa desligado o argumento `name`, como no segundo exemplo acima, o Django usará o nome da função como o nome da tag.

Passando variáveis de template para a tag

Embora você possa passar qualquer número de argumentos para uma template tag usando `token.split_contents()`, os argumentos são todos extraídos como strings literais. Um pouco mais de trabalho é necessária a fim de passar o conteúdo dinâmico (uma variável de template) para uma template tag como um argumento.

Enquanto nos exemplos anteriores foram formatados a hora atual dentro de uma string e retornado uma string, suponhamos que você queira passar um `DateTimeField` de um objeto e tem a template tag que formata essa data:

```
<p>Este post foi atualizado em {% format_time blog_entry.date_updated "%Y-%m-%d %I:↵%M %p" %}.</p>
```

Inicialmente, `token.split_contents()` retornará três valores:

1. O nome da tag `format_time`.
2. A string `"blog_entry.date_updated"` (sem as aspas em volta).
3. A string de formatação `"%Y-%m-%d %I:%M %p"`. O valor retornado de `split_contents()` incluirá as aspas em strings literais como esta.

Agora sua tag deve começar a parecer como isso:

```
from django import template
def do_format_time(parser, token):
    try:
        # split_contents() não sabe como separar strings com aspas.
        tag_name, date_to_be_formatted, format_string = token.split_contents()
    except ValueError:
        raise template.TemplateSyntaxError, "%r tag requires exactly two arguments
↵" % token.contents.split()[0]
        if not (format_string[0] == format_string[-1] and format_string[0] in ('"', '
↵')):
            raise template.TemplateSyntaxError, "%r tag's argument should be in quotes
↵" % tag_name
        return FormatTimeNode(date_to_be_formatted, format_string[1:-1])
```

Variable resolution has changed in the 1.0 release of Django. `template.resolve_variable()` has been deprecated in favor of a new `template.Variable` class. Você também tem de mudar o renderizador para receber os conteúdos atuais da propriedade `date_updated` de um objeto `blog_entry`. Isso pode ser realizado usando a classe `Variable()` em `django.template`.

Para usar a classe `Variable`, simplesmente instancie-o com o nome da variável a ser resolvida, e então chame `variable.resolve(context)`. Assim, por exemplo:

```
class FormatTimeNode(template.Node):
    def __init__(self, date_to_be_formatted, format_string):
        self.date_to_be_formatted = template.Variable(date_to_be_formatted)
        self.format_string = format_string

    def render(self, context):
        try:
            actual_date = self.date_to_be_formatted.resolve(context)
            return actual_date.strftime(self.format_string)
        except template.VariableDoesNotExist:
            return ''
```

A resolução de variável lançará uma exceção `VariableDoesNotExist` se ela não conseguir resolver a string passada no atual contexto da página.

Atalho para simple tags

Muitas templates tags recebe vários argumentos – strings ou variáveis de template – e retornam uma string depois de fazer algum processamento baseado unicamente no argumento de entrada e algumas informações externas. Por exemplo, a tag `current_time` que nós escrevemos acima é desta variedade: nós fornecemos a ela uma string de formatação, e ela retorna a hora como uma string.

Para facilitar a criação de tipos de tags, o Django fornece uma função helper, `simple_tag`. Esta função, que é um método de `django.template.Library`, recebe uma função que aceita qualquer quantidade de argumentos, a envolve com uma função `render` e de outras coisas necessárias mencionadas acima e a registra no sistema de template.

Nossa função `current_time` poderia, assim, ser escrita desta forma:

```
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)

register.simple_tag(current_time)
```

No Python 2.4, a sintaxe de decorador também funciona:

```
@register.simple_tag
def current_time(format_string):
    ...
```

Uma das coisas que você deve tomar nota sobre o helper `simple_tag`:

- Verificar o número de argumentos requeridos, etc., já é feito na hora que nossa função é chamada, então nós não precisamos fazer isso.
- As aspas em volta dos argumentos (se tive alguma) já foram retiradas, desta forma nós só recebemos uma string.
- Se o argumento for uma variável de template, à nossa função é passada o valor atual da variável, não a própria variável.

Quando sua template tag não precisa acessar o contexto atual, escrever uma função para trabalhar com os valores de entrada e usar o helper `simple_tag` é a forma mais fácil de criar uma nova tag.

Inclusion tags

Outro tipo comum de template tag é o tipo que mostra algum dados renderizando *outro* template. Por exemplo, a interface de administração do Django usa templates personalizados para mostrar botões na base das páginas de formulário “adicionar/atualizar”. Estes botões sempre são os mesmos, mas o alvo do link muda dependendo do objeto que estiver sendo editado – então eles são o caso perfeito para se usar um template que é preenchido com detalhes do objeto atual. (No caso do admin, isso é a tag `submit_row`.)

Esses tipos de tags são chamados “inclusion tags”.

Escrever inclusion tags é provavelmente melhor demonstrado com exemplo. Vamos escrever uma tag que mostra uma lista de escolhas para um dado objeto `Poll`, como a que foi criada no [tutorial](#). Nós usaremos a tag desta forma:

```
{% show_results poll %}
```

...e a saída será algo assim:

```
<ul>
  <li>Primeira escolha</li>
  <li>Segunda escolha</li>
  <li>Terceira escolha</li>
</ul>
```

Primeiro, definir a função que recebe o argumento e produz um dicionário de dados para o resultado. O ponto importante aqui é nós somente precisarmos retornar um dicionário, nada mais complexo. Isso será usado como um contexto de template para o template fragmentado. Exemplo:

```
def show_results(poll):
    choices = poll.choice_set.all()
    return {'choices': choices}
```

Próximo, criar o template usado para renderizar a saída da tag. Este template é uma funcionalidade fixa da tag: o escritor da tag o especifica, não o designer de template. Seguindo nosso exemplo, o template é muito simples:

```
<ul>
{% for choice in choices %}
    <li> {{ choice }} </li>
{% endfor %}
</ul>
```

Agora, criar e registrar a inclusion tag chamando o método `inclusion_tag()` de um objeto `Library`. Seguindo nosso exemplo, se o template acima estiver num arquivo chamado `results.html` num diretório que é procurado pelo carregador de template, nós registrariamos a tag desta forma:

```
# Aqui, register é uma instância do django.template.Library, como antes
register.inclusion_tag('results.html')(show_results)
```

Como sempre, a sintaxe de decorador do Python 2.4 também funciona, então poderíamos ter escrito:

```
@register.inclusion_tag('results.html')
def show_results(poll):
    ...
```

...ao criar a primeira função.

Algumas vezes, suas inclusion tags podem requerer um número grande de argumentos, causando dor aos autores de template ao passar-lhes todos os argumentos e lembrar de suas ordens. Para resolver isso, o Django fornece uma opção `takes_context` para inclusion_tags. Se você especificar `takes_context` ao criar uma template tag, a tag não terá argumentos requeridos, e a função Python subjacente terá um argumento – o contexto do template a partir de quando a tag foi chamada.

Por exemplo, digamos que você esteja escrevendo uma inclusion tag que sempre será usada no contexto que contém as variáveis `home_link` e `home_title` que apontam de volta a página principal. Aqui está como a função Python poderia parecer:

```
# O primeiro argumento *deve* ser chamado "context" aqui.
def jump_link(context):
    return {
        'link': context['home_link'],
        'title': context['home_title'],
    }
# Registra a tag personalizada como uma inclusion tag com takes_context=True.
register.inclusion_tag('link.html', takes_context=True)(jump_link)
```

(Note que o primeiro parâmetro para a função *deve* ser chamado `context`.)

Nessa linha `register.inclusion_tag()`, nós especificamos `takes_context=True` e o nome do template. Aqui temos como o template `link.html` pode parecer:

```
Pule direto para <a href="{{ link }}">{{ title }}</a>.
```

Então, em qualquer hora que você desejar usar essa tag personalizada, carregue sua biblioteca e chame-a sem quaisquer argumentos, dessa forma:

```
{% jump_link %}
```

Note que quando você estiver usando `takes_context=True`, não há necessidade de passar argumentos para a template tag. Ela automaticamente tem acesso ao contexto.

O parâmetro `takes_context` é por padrão `False`. Quando ele é configurado para `True`, a tag é passada o objeto de contexto, como nesse exemplo. Essa é a única diferença entre este caso e o exemplo anterior da `inclusion_tag`.

Configurando uma variável no contexto

O exemplo acima simplesmente mostra um valor. Geralmente, é mais flexível se suas template tags configurarem variáveis ou invés de mostrar valores. Desta forma os autores de templates podem reusar os valores que suas template tags criam.

Para setar uma variável no contexto, é só usar atribuição de dicionário no contexto de objeto no método `render()`. Aqui temos uma versão atualizada do `CurrentTimeNode` que configura uma variável de template `current_time` ao invés de mostrá-la:

```
class CurrentTimeNode2(template.Node):
    def __init__(self, format_string):
        self.format_string = format_string
    def render(self, context):
        context['current_time'] = datetime.datetime.now().strftime(self.format_
        ↪string)
        return ''
```

Note que `render()` retorna uma string vazia. O `render()` deve sempre retornar uma string. Se tudo o que a template tag faz é setar uma variável, o `render()` deve retornar uma string vazia.

Aqui temos como poderíamos usar esta nova versão de tag:

```
{% current_time "%Y-%M-%d %I:%M %p" %}<p>The time is {{ current_time }}.</p>
```

Porém, há um problema com `CurrentTimeNode2`: O nome da variável `current_time` é nativo. Isso significa que você precisará assegurar-se de que seu template não use `{{ current_time }}` em nenhum lugar mais, por que o `{% current_time %}` será cegamente sobrescrito pelo valor desta variável. Uma solução clara é fazer uma template tag especificar o nome da variável de saída, desta forma:

```
.. code-block:: html+django
```

```
{% get_current_time "%Y-%M-%d %I:%M %p" as my_current_time %} <p>The current time is {{
my_current_time }}.</p>
```

Para fazer isso, você precisa refatorar ambas funções de compilação e classe `Node`, tipo:

```
class CurrentTimeNode3(template.Node):
    def __init__(self, format_string, var_name):
        self.format_string = format_string
        self.var_name = var_name
    def render(self, context):
        context[self.var_name] = datetime.datetime.now().strftime(self.format_
        ↪string)
        return ''

import re
def do_current_time(parser, token):
    # Essa versão usa uma expressão regular para parsear a o conteúdo da tag.
    try:
        # Separando None == separando por espaços.
        tag_name, arg = token.contents.split(None, 1)
    except ValueError:
        raise template.TemplateSyntaxError, "%r tag requires arguments" % token.
    ↪contents.split()[0]
```

```
m = re.search(r'(.*) as (\w+)', arg)
if not m:
    raise template.TemplateSyntaxError, "%r tag had invalid arguments" % tag_
↪name
    format_string, var_name = m.groups()
    if not (format_string[0] == format_string[-1] and format_string[0] in ('"', '
↪'))):
        raise template.TemplateSyntaxError, "%r tag's argument should be in quotes
↪" % tag_name
    return CurrentTimeNode3(format_string[1:-1], var_name)
```

A diferença aqui é que o `do_current_time()` pega a string de formato e o nome da variável, passando ambos ao `CurrentTimeNode3`.

Parseando até outro tag de bloco

Template tags podem funcionar em conjunto. Por exemplo, a tag padrão `{% comment %}` esconde tudo até o `{% endcomment %}`. Para criar uma template tag como esta, use `parser.parse()` na sua função de compilação.

Aqui temos como a tag padrão `{% comment %}` é implementado:

```
def do_comment(parser, token):
    nodelist = parser.parse(('endcomment',))
    parser.delete_first_token()
    return CommentNode()

class CommentNode(template.Node):
    def render(self, context):
        return ''
```

O `parser.parse()` recebe uma tupla de nomes de tags de bloco “para parsear até ela”. Ela retorna uma instância do `django.template.NodeList`, que é uma lista de todos objetos `Node` que o parser encontrar “antes” de encontrar quaisquer tags chamadas na tupla.

No `"nodelist = parser.parse(('endcomment',))"` do exemplo acima, `nodelist` é uma lista de todos os nodos entre o `{% comment %}` e `{% endcomment %}`, não contando `{% comment %}` e `{% endcomment %}`.

Depois que `parser.parse()` é chamado, o parser ainda não “consumiu” a tag `{% endcomment %}`, assim o código precisa explicitamente chamar `parser.delete_first_token()`.

O `CommentNode.render()` simplesmente retorna uma string vazia. Qualquer coisa entre `{% comment %}` e `{% endcomment %}` é ignorado.

Parseando até outra tag de bloco, e salvando conteúdos

No exemplo anterior, `do_comment()` tudo descartado entre `{% comment %}` e `{% endcomment %}`. Ao invés de fazer isso, é possível fazer algo com o código entre as tags de bloco.

Por exemplo, há uma template tag personalizada, `{% upper %}`, que capitaliza tudo entre ela mesma e `{% endupper %}`.

Uso:

```
{% upper %}Isso irá aparecer em maiúsculas, {{ your_name }}.{% endupper %}
```

Como no exemplo anterior, nós usaremos `parser.parse()`. Mas dessa vez, nós passamos o resultado `nodelist` para o `Node`:

```
def do_upper(parser, token):
    nodelist = parser.parse(('endupper',))
    parser.delete_first_token()
    return UpperNode(nodelist)

class UpperNode(template.Node):
    def __init__(self, nodelist):
        self.nodelist = nodelist
    def render(self, context):
        output = self.nodelist.render(context)
        return output.upper()
```

O único conceito novo aqui é o `self.nodelist.render(context)` no `UpperNode.render()`.

Para mais exemplos de complexidade de renderização, vej o código fonte `{% if %}`, `{% for %}`, `{% ifequal %}` e `{% ifchanged %}`. Eles estão em `django/template/defaulttags.py`.

Escrevendo um sistema de armazenamento customizado

Se você precisar fornecer sistema de armazenamento customizado - um exemplo comum é armazenar arquivos em algum sistema remoto - você pode fazê-lo, definindo uma classe de armazenamento customizada. Você precisará seguir esses passos:

1. Seu mecanismo de armazenamento customizado deve ser uma subclasse de `django.core.files.storage.Storage`:

```
from django.core.files.storage import Storage

class MyStorage(Storage):
    ...
```

2. O Django deve ser capaz de instanciar o seu sistema de armazenamento sem quaisquer argumentos. Isto significa que qualquer definição deve ser obtida a partir de `django.conf.settings`:

```
from django.conf import settings
from django.core.files.storage import Storage

class MyStorage(Storage):
    def __init__(self, option=None):
        if not option:
            option = settings.CUSTOM_STORAGE_OPTIONS
        ...
```

3. Sua classe de armazenamento deve implementar os métodos `_open()` e `_save()`, juntamente com quaisquer outros métodos adequados para sua classe. Veja abaixo mais informações sobre esses métodos. Além disso, se a sua classe forcener um sistema de armazenamento local, ela deve sobrescrever o método `path()`.

Seu sistema de armazenamento personalizado pode sobrescrever qualquer um dos métodos de armazenamento explicados em *API de armazenamento de arquivos*., mas você **deve** implementar os seguintes métodos:

- `Storage.delete()`
- `Storage.exists()`
- `Storage.listdir()`
- `Storage.size()`
- `Storage.url()`

Geralmente você também vai querer usar ganchos concebidos especificamente para objetos de armazenamento personalizados. São eles:

`_open(name, mode='rb')`

Obrigatório.

Chamado por `Storage.open()`, esse é o verdadeiro mecanismo que a classe de armazenamento usa para abrir o arquivo. Ele deve retornar um objeto `File`, embora na maioria dos casos, você preferirá retornar alguma subclasse que implemente a lógica específica do backend do sistema de armazenamento.

`_save(name, content)`

Chamado por `Storage.save()`. O `name` já terá passado por `get_valid_name()` e `get_available_name()`, e o `content` será um objeto `File` em si.

Deve retornar o nome real do arquivo salvo (normalmente o `name` recebido, mas se o storage precisa mudar o nome do arquivo o novo nome deve ser retornado).

`get_valid_name(name)`

Retorna um nome de arquivo adequado para uso com o sistema de armazenamento subjacente. O argumento `name` passado para este método é o nome de arquivo original enviado para o servidor, depois de ter removido qualquer informação de caminho. Sobrescreva este método para customizar como caracteres fora do padrão são convertidos para nomes de arquivos válidos.

O código fornecido em `Storage` retém apenas caracteres alfanuméricos, pontos e sublinhas do arquivo original, removendo todo o resto.

`get_available_name(name)`

Retorna um nome de arquivo que está disponível no mecanismo de armazenamento, possivelmente levando em conta o nome de arquivo fornecido. O argumento `name` passado para este método já terá sido validado para um nome de arquivo válido para o sistema de armazenamento, de acordo com o método `get_valid_name()` descrito acima.

O código fornecido em `Storage` simplesmente adiciona sublinhas ao nome do arquivo até encontrar um que esteja disponível no diretório de destino.

Implantando o Django

O Django possui uma série de atalhos para tornar a vida do desenvolvedor web mais fácil, mas de nada vale se você não consegue realizar a implantação de seus sites facilmente. Desde o início do desenvolvimento do Django, o conceito de facilidade de implantação tem sido um alvo prioritário. Existem algumas boas soluções para se colocar um projeto Django em produção:

Como usar o Django com o Apache e mod_wsgi

A implantação do Django com [Apache](#) e [mod_wsgi](#) é a forma recomendada de ter o Django funcionando em produção.

O [mod_wsgi](#) é um módulo do Apache que pode ser usado para hospedar qualquer aplicação Python que suporte a [Interface WSGI do Python](#), incluindo o Django. O Django irá funcionar com qualquer versão do Apache que suporte o [mod_wsgi](#).

A [documentação oficial do mod_wsgi](#) é fantástica; é sua fonte para todos os detalhes sobre como usar o [mod_wsgi](#). você provavelmente irá querer iniciar com a [documentação de instalação e configuração](#).

Configuração Básica

Uma vez que você tenha o [mod_wsgi](#) instalado e ativado, edite o seu arquivo `httpd.conf` e adicione:

```
WSGIScriptAlias / /path/to/mysite/apache/django.wsgi
```

A primeira parte acima é a url na qual você estará servindo sua aplicação (`/` indica a url raiz), e a segunda parte é a localização de um “arquivo WSGI” – veja abaixo – no seu sistema, normalmente dentro do seu projeto. Isso diz ao Apache para servir quaisquer requisições abaixo de dada URL usando a aplicação WSGI definida por aquele arquivo.

Em seguida, vamos efetivamente criar essa aplicação WSGI, então crie o arquivo mencionado na segunda parte da linha `WSGIScriptAlias` e adicione:

```
import os
import sys

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'
```

```
import django.core.handlers.wsgi
application = django.core.handlers.wsgi.WSGIHandler()
```

Se o seu projeto não está no PYTHONPATH padrão você pode adicionar:

```
sys.path.append('/usr/local/django')
```

logo abaixo das linhas de `import` para colocar o seu projeto no path. Lembre-se de substituir o ‘mysite.settings’ como seu arquivo de configurações correto.

Veja a [Documentação do Apache/mod_python](#) para informações de como servir mídia estática, e a [documentação do mod_wsgi](#) para uma explicação de outras diretivas e opções de configuração que você pode usar.

Detalhes

Para maiores detalhes, veja a [documentação do mod_wsgi](#), que explica os itens acima em maior detalhamento, e lhe dá todas as possibilidades de opções que você tem ao fazer uma implantação no mod_wsgi.

Como usar o Django com o Apache e mod_python

O módulo `mod_python` para o `Apache` pode ser usado para fazer a implantação do Django em um servidor de produção, embora ele tenha sido praticamente superado pela opção mais simples [implantação com mod_wsgi](#).

O `mod_python` é semelhante (e inspirado) ao `mod_perl`: Ele embute o Python dentro do Apache e carrega o código Python na memória quando o servidor inicia. O Código fica na memória durante o tempo de vida do processo Apache, que leva a ganhos de performance significantes sobre outras opções de disposição de servidor.

O Django requer o Apache 2.x e o `mod_python` 3.x, e você deve usar o `MPM prefork`, ao invés do `MPM worker`.

See also:

- O apache é um animal grande e complexo, e esse documento dá apenas um panorama do que o Apache é capaz de fazer. Se você precisa de informação mais avançado sobre o Apache, não há fonte de informação melhor que [a própria documentação oficial do Apache](#)
- Você pode também estar interessado em [Como usar o Django com FastCGI, SCGI ou AJP](#).

Configuração Básica

Para configurar o Django com o `mod_python`, verifique primeiro se você tem o Apache instalado, Com o módulo `mod_python` ativado.

Edite o arquivo `httpd.conf` e adicione o seguinte:

```
<Location "/mysite/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonOption django.root /mysite
    PythonDebug On
</Location>
```

...e substitua o `mysite.settings` com o caminho de importação do Python para o seu arquivo de configuração do projeto Django.

Isso diz ao Apache: “Use o `mod_python` para qualquer URL em ou sobre ‘/mysite/’, usando o handler do `mod_python` do Django.” Ele passa o valor do `DJANGO_SETTINGS_MODULE` para que o `mod_python` saiba quais configurações usar. A `PythonOption django.root ...` é nova nessa versão. Como o `mod_python` não sabe que estamos servido esse site a partir do prefixo `/mysite/`, esse valor precisa ser passado para o Django

através do manipulador do `mod_python`, pela linha `PythonOption django.root ...`. O valor configurado nessa linha (o último item) deve ser igual ao texto informado na diretiva `<Location ...>`. O efeito disso é que o Django irá automaticamente remover a string `/mysite` no início de quaisquer URLs antes de fazer a verificação deles com os seus padrões em `URLConf`. Se você depois mudar o seu site para um endereço como `/mysite2`, você não irá precisar alterar nada, exceto a opção `django.root` no arquivo de configuração.

Ao usar o `django.root` você deve assegurar-se que o que resta, após a remoção do prefixo, inicie com uma barra. Seus padrões do `URLConf` que esperar uma barra inicial funcionaram corretamente. No exemplo acima, supondo que queiramos enviar uma URL como `/mysite/admin/` para o `/admin/`, precisamos remover a string `/mysite` do início, então esse deve ser o valor do `django.root`. Seria um erro usar o `/mysite/` (com uma barra final) nesse caso.

Repare que estamos usando a diretiva `<Location>`, não a diretiva `<Directory>`. Essa última é usada para apontar para lugares no sistema de arquivos, enquanto `<Location>` aponta para lugares específicos na estrutura de URL de um Web site. `<Directory>` seria sem significado aqui.

Além disso, se o seu projeto Django não está no `PYTHONPATH` padrão do seu computador, você terá de dizer ao `mod_python` onde o seu projeto pode ser encontrado:

```
<Location "/mysite/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonOption django.root /mysite
    PythonDebug On
    PythonPath ["'/path/to/project'"] + sys.path"
</Location>
```

O valor usado em `PythonPath` deve incluir o diretório pai de todos os módulos que você vai importar em sua aplicação. Deve também incluir o diretório pai da localização do `DJANGO_SETTINGS_MODULE`. Essa é exatamente a mesma situação do caminho do Python para uso iterativo. Sempre que você tenta importar algo, o Python irá percorrer todos os diretórios no `sys.path` em ordem, do primeiro ao último, e tentar importar de cada diretório até ter sucesso.

Verique se as permissões dos seus arquivos de código Python estejam configuradas de forma que o usuário do Apache (normalmente chamado de `apache` ou `httpd` na maioria dos sistemas) tenha acesso de leitura aos arquivos.

Um exemplo que pode tornar isso mais claro: suponha que você tenha algumas aplicações em `/usr/local/django-apps/` (por exemplo, `/usr/local/django-apps/weblog/` e assim por diante), seu arquivo de configurações está em `/var/www/mysite/settings.py` e você tem o `DJANGO_SETTINGS_MODULE` especificado como no exemplo acima. Nesse caso, você precisa escrever a sua diretiva `PythonPath` assim:

```
PythonPath ["'/usr/local/django-apps/', '/var/www'"] + sys.path"
```

Com esse caminho, `import weblog` e `import mysite.settings` irão funcionar. Se você tem `import blogroll` em algum lugar do seu código e `blogroll` está no diretório `weblog/`, você *também* precisa adicionar `/usr/local/django-apps/weblog/` ao seu `PythonPath`. Lembre-se: os **diretórios pai** de tudo o que você for importar diretamente devem estar no caminho do Python.

Note: Se você está usando o Windows, ainda assim recomendamos que você use barras normais nos nome dos caminhos, ainda que o Windows normalmente use as barras invertidas como seu separador nativo. O Apache sabe como converter a barra para o formato nativo, então esse jeito de lidar é mais portátil e mais fácil de ler. (E evita problemas esquisitos como ter de escapar as barras invertidas duas vezes.)

Isso é válido mesmo em um sistema Windows:

```
PythonPath ["'c:/path/to/project'"] + sys.path"
```

Você também pode adicionar diretivas como `PythonAutoReload Off` para performance. Veja a [documentação do mod_python](#) para uma lista completa de opções.

Note que você deve configurar `PythonDebug Off` em um servidor de produção. Se você deixar o `PythonDebug On`, seus usuários vão ver `tracebacks Python` feios (e reveladores) se algo der errado dentro do `mod_python`.

Reinicie o Apache, e quaisquer requisições feitas para `/mysite/` ou abaixo dele serão servidas pelo Django. Note que as `URLconfs` do Django não eliminam o “`/mysite/`” – elas recebem a URL completa.

Ao implantar sites Django no `mod_python`, você irá precisar reiniciar o Apache a cada vez que você fizer mudanças no seu código Python.

Múltiplas instalações do Django no mesmo Apache

É inteiramente possível executar múltiplas instalações do Django na mesma instância do Apache. Apenas use `VirtualHost` para isso, assim so:

```
NameVirtualHost *
```

```
<VirtualHost *>
    ServerName www.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</VirtualHost>

<VirtualHost *>
    ServerName www2.example.com
    # ...
    SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
</VirtualHost>
```

Se você precisa colocar duas instalações do Django dentro do mesmo `VirtualHost` (ou em blocos `VirtualHost` diferentes que compartilham o mesmo nome de servidor), você precisará tomar um cuidado especial para que o cache do `mod_python` não bagunce as coisas. Use a diretiva `PythonInterpreter` para dar a cada diretiva `<Location>` diferentes interpretadores separados:

```
<VirtualHost *>
    ServerName www.example.com
    # ...
    <Location "/something">
        SetEnv DJANGO_SETTINGS_MODULE mysite.settings
        PythonInterpreter mysite
    </Location>

    <Location "/otherthing">
        SetEnv DJANGO_SETTINGS_MODULE mysite.other_settings
        PythonInterpreter othersite
    </Location>
</VirtualHost>
```

Os valores para `PythonInterpreter` não importam muito, desde que eles sejam diferentes entre os blocos `Location`.

Executando um servidor de desenvolvimento com o mod_python

Se você usar o `mod_python` como seu servidor de desenvolvimento, você pode evitar o fardo de ter de reiniciar o servidor cada vez que seu código muda. Apenas configure `MaxRequestsPerChild 1` no seu arquivo `httpd.conf` para forçar o Apache a recarregar tudo de novo em cada requisição. Mas não faça isso no servidor de produção, ou revogaremos seus privilégios Django.

Se você é o tipo de programador que debuga desperando declarações `print` no meio do seu código, note que declarações `print` não tem efeito no `mod_python`; elas não aparecem no log do Apache, como você espera. Se

you need to print debug information in a configuration with `mod_python`, or do it like this:

```
assert False, the_value_i_want_to_see
```

Or add the debug documentation to the template of your page.

Servindo arquivos de mídia

Django does not serve media files by itself; it delegates this work to the Web server of your choice.

We recommend the use of a separate Web server – i.e., one that is not running Django – to serve media. Here are some good choices:

- [lighttpd](#)
- [Nginx](#)
- [TUX](#)
- Uma versão reduzida do [Apache](#)
- [Cherokee](#)

Se, porém, você não tem opção exceto servir arquivos de mídia no mesmo `VirtualHost` do Apache do Django, aqui está como você pode desligar o `mod_python` para uma parte em particular do site:

```
<Location "/media">
    SetHandler None
</Location>
```

Just change the `Location` to the URL root of your media files. You can also use the `<LocationMatch>` to match with a regular expression.

This example configures Django at the root of the site but explicitly disables Django for the subdirectory `media` and any URL that ends in `.jpg`, `.gif` or `.png`:

```
<Location "/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
</Location>

<Location "/media">
    SetHandler None
</Location>

<LocationMatch "\.(jpg|gif|png)$">
    SetHandler None
</LocationMatch>
```

Servindo os arquivos da administração

Note that the Django development server automatically serves the media files of the administration, but this is not automatic when you use another server configuration. You are responsible for configuring the Apache, or whatever other media server you are using, to serve the administration files.

The administration files are in the `(django/contrib/admin/media)` directory of the Django distribution.

Here are two recommended ways:

1. Create a symbolic link for the administration media files inside your document root. In this way, all your files related to Django – code and templates – are in a single place, and you will still be able to do a `svn update` on your code to get the latest versions of the templates of the administration, if they change.

2. Ou, copie os arquivos de mídia da administração de forma que eles fiquem dentro do document root do seu Apache.

Usando “eggs” com o mod_python

Se você instalou o Django de um Python [egg](#) ou está usando eggs no seu projeto Django, algumas configurações extra são requeridas. Crie um arquivo extra no seu projeto (ou em algum outro lugar) que contenha algo assim:

```
import os
os.environ['PYTHON_EGG_CACHE'] = '/some/directory'
```

Aqui, `/some/directory` é um diretório ao qual o servidor Apache tenha acesso de escrita. Ele será usado como o local de descompactação para qualquer código que os eggs precisem fazer.

Então você irá precisar dizer ao mod_python para importar esse arquivo antes de fazer qualquer coisa. Isso é feito usando a diretiva `PythonImport` do seu mod_python. Você precisa garantir que você especificou a diretiva `PythonInterpreter` do mod_python como descrito [acima](#) (nesse caso, você precisa fazer isso mesmo que você não esteja servindo múltiplas instalações). Então, adicione a linha `PythonImport` na configuração principal do servidor (i.e., fora das seções `Location` ou `VirtualHost`). Por exemplo:

```
PythonInterpreter my_django
PythonImport /path/to/my/project/file.py my_django
```

Note que você pode usar um caminho absoluto aqui (ou um caminho de importação com pontos normal), como descrito no [manual do mod_python](#). Nós usamos um caminho absoluto no exemplo acima porque se quaisquer modificações no Python path forem necessárias para acessar seu projeto, elas não terão sido feitas no momento em que a linha `PythonImport` é processada.

Tratamento de erros

Quando você usa o Apache/mod_python, os error serão capturados pelo Django – em outras palavras, eles não irão propagar para o nível do Apache nem aparecerão no `error_log` do Apache.

A exceção a isso é se algo estiver realmente quebrado na sua configuração do Django. Nesse caso, você verá uma página de “Erro Interno do Servidor” no seu navegador e o traceback completo do Python no seu arquivo `error_log` do Apache. O traceback no `error_log` ocupa diversas linhas. (Sim, isso é feio e bem chato de ler, mas é como o mod_python faz as coisas.)

Se você tiver uma falha de segmentação

Se o Apache causar uma falha de segmentação, existem duas razões prováveis, nenhuma das quais tem a ver com o Django em si.

1. Pode ser que o seu código Python está importando o módulo “pyexpat”, que pode causar conflito com a versão embutida no Apache. Para informação completa, veja [Expat fazendo o Apache travar](#).
2. Pode ser porque você está usando o mod_python e o mod_php na mesma instância do Apache, com o MySQL como seu banco de dados. Em alguns casos, isso causa um problema conhecido do mod_python devido a conflito de versões no PHP e no backend do MySQL do Python. Existem informação completa na [entrada da FAQ do mod_python](#).

Se você continuar a ter problemas configurando o mod_python, algo interessante a se fazer e conseguir um site mod_python cru rodando, sem o framework Django. Isso é um modo fácil de isolar problemas específicos do mod_python. [Fazendo o mod_python funcionar](#) detalha esse procedimento.

O próximo passo deve ser editar seu código de teste e adicionar um import de qualquer código específico do Django que você esteja usando – suas views, seus models, suas URLconf, sua configuração de RSS, etc. Coloque esses imports na sua função gerenciadora de teste e acesse sua URL de teste em um navegador. Se isso causa um crash, você confirmou que a importação de código Django que causa o problema. Gradualmente reduza o conjunto

de imports até que pare de travar, para que você encontre o módulo específico que causa o problema. Investigue cada módulo e veja seus imports, conforme necessário.

Como usar o Django com FastCGI, SCGI ou AJP

Apesar da instalação atualmente preferida para rodar o Django ser *Apache com `mod_wsgi`*, muitas pessoas usam hospedagem compartilhada, onde protocolos como FastCGI, SCGI ou AJP são a única opção viável. Em algumas instalações, estes protocolos pode fornecer uma performance melhor que o `mod_wsgi`.

Note

Este documento foca primariamente no FastCGI. Outros protocolos, como SCGI e AJP, são também suportados, através do pacote do Python `flup`. Veja a seção *Protocolos* abaixo para detalhes sobre SCGI e AJP.

Essencialmente, o FastCGI é uma forma eficiente de deixar uma aplicação externa servir páginas para um servidor Web. O servidor Web deleta as requisições Web vindas (via socket) para o FastCGI, que executa o código passando a resposta de volta ao servidor Web, que, por sua vez, passa-o de volta ao navegador do cliente.

Como `mod_python`, o FastCGI permite que o código fique na memória, permitindo requisições serem servidas sem tempo de inicialização. Diferentemente do `mod_python` (ou `mod_perl`), um processo FastCGI não roda dentro do processo do servidor Web, mas em um processo persistente separado.

Porque rodar código em um processo separado?

A modalidade tradicional `mod_*` no Apache embute várias linguagens de script (mais notavelmente PHP, Python e Perl) dentro do espaço do processo de seu servidor Web. Embora isto diminua o tempo de inicialização – porque o código não tem que ser lido em disco para cada requisição – ele vem com custo de uso de memória. Para `mod_python`, por exemplo, todo processo Apache carrega seu próprio interpretador Python, que consome um montante considerável de RAM.

Devido a natureza do FastCGI, ainda é possível ter processos que rodem sob uma conta de usuário diferente do processo do servidor Apache. O que é um grande benefício para segurança em sistemas compartilhados, por que significa que você pode assegurar o seu código de outros usuários.

Pré-requisito: flup

Antes que você possa começar a usar o FastCGI com o Django, você precisará instalar o `flup`, uma biblioteca Python para lidar com FastCGI. A versão 0.5 ou as mais recentes devem funcionar muito bem.

Iniciando seu servidor FastCGI

O FastCGI opera sobre um modelo cliente-servidor, e na maioria dos casos você estará iniciando o processo FastCGI em seu próprio. Seu servidor Web (seja o Apache, `lighttpd`, ou qualquer outro) somente contata seu processo Django-FastCGI quando o servidor precisa que uma página dinâmica seja carregada. Porque o daemon já está rodando com o código em memória, ele está pronto para servir uma resposta rapidamente.

Note

Se você está em sistema de hospedagem compartilhada, você provavelmente será forçado a usar o processo FastCGI gerenciado pelo servidor Web. Veja a seção abaixo, rodando o Django em processos gerenciados pelo servidor Web para mais informações.

Um servidor Web pode se conectar com um servidor FastCGI de uma das duas maneiras disponíveis: Ele pode usar cada um dos sockets de domínios Unix (um “pipe nomeado” em sistemas Win32), ou ele pode usar um socket

TCP. O que você escolher é de sua preferência; um socket TCP é normalmente mais fácil devido aos problemas de permissões.

Para iniciar seu servidor, primeiro mude dentro do diretório de seu projeto (aonde quer que seu *manage.py* esteja), e então rode o comando `runfcgi`:

```
./manage.py runfcgi [options]
```

Se você especificar `help` como a única opção depois de `runfcgi`, ele irá mostrar uma lista de todas as opções disponíveis.

Você precisará especificar ainda um socket, um protocolo ou ambos `host` e `porta`. Então, quando você configurar seu servidor Web, você somente precisará apontá-lo para o `host/port` ou socket que você especificou quando iniciou o servidor FastCGI. Veja os *Exemplos*, abaixo.

Protocolos

O Django suporta todos os protocolos que o *flup* suporta, *fastcgi*, *SCGI* e *AJP1.3* (o Apache JServ Protocol, versão 1.3). Selecione seu protocolo preferido usando a opção `protocol=<protocol_name>` com `./manage.py runfcgi` – onde `<protocol_name>` pode ser um dos: `fcgi` (o padrão), `scgi` ou `ajp`. Por exemplo:

```
./manage.py runfcgi protocol=scgi
```

Exemplos

Rodando um servidor em threads em uma porta TCP:

```
./manage.py runfcgi method=threaded host=127.0.0.1 port=3033
```

Rodando um servidor preforked em um socket de domínio Unix:

```
./manage.py runfcgi method=prefork socket=/home/user/mysite.sock pidfile=django.pid
```

Executando sem o daemonizing (rodar em background) o processo (bom para debugging):

```
./manage.py runfcgi daemonize=false socket=/tmp/mysite.sock maxrequests=1
```

Parando o daemon FastCGI

Se você tem o processo rodando em foreground, é bem fácil de pará-lo: Simplesmente pressionando `Ctrl-C` você parará e fechará o servidor FastCGI. Entretanto, quando você está com processos em background, você precisará recorrer ao comando `kill` do Unix.

Se você especificar a opção `pidfile` para o `runfcgi`, você poderá matar o daemon do processo FastCGI desta forma:

```
kill `cat $PIDFILE`
```

...onde `$PIDFILE` é o `pidfile` que você especificou.

Para facilitar o reinício do seu daemon FastCGI no Unix, tente este pequeno script shell:

```
#!/bin/bash

# Replace these three settings.
PROJDIR="/home/user/myproject"
PIDFILE="$PROJDIR/mysite.pid"
SOCKET="$PROJDIR/mysite.sock"
```

```
cd $PROJDIR
if [ -f $PIDFILE ]; then
    kill `cat -- $PIDFILE`
    rm -f -- $PIDFILE
fi

exec /usr/bin/env - \
    PYTHONPATH="../python:.." \
    ./manage.py runfcgi socket=$SOCKET pidfile=$PIDFILE
```

Configuração do Apache

Para usar o Django com o Apache e FastCGI, você precisará ter o Apache instalado e configurado, com `mod_fastcgi` instalado e habilitado. Consulte a documentação do Apache para instruções.

Uma vez que você tenha realizado a instalação, aponte o Apache para sua instância do Django FastCGI editando o arquivo `httpd.conf` (configuração do Apache). Você precisará fazer duas coisas:

- Usar a diretiva `FastCGIExternalServer` para especificar a localização do seu servidor FastCGI.
- Usar `mod_rewrite` para apontar as URLs para o FastCGI como se deve.

Especificando a localização do servidor FastCGI

A diretiva `FastCGIExternalServer` diz ao Apache como encontrar seu servidor FastCGI. Como a [documentação do FastCGIExternalServer](#) explica, você pode especificar ambos `socket` ou um `host`. Aqui há alguns exemplos de ambos:

```
# Conecta o FastCGI via socket / pipe nomeado.
FastCGIExternalServer /home/user/public_html/mysite.fcgi -socket /home/user/mysite.
↪sock

# Conecta o FastCGI via um host/port TCP.
FastCGIExternalServer /home/user/public_html/mysite.fcgi -host 127.0.0.1:3033
```

Em ambos os casos, o arquivo `/home/user/public_html/mysite.fcgi` na verdade não tem de existir. Ele é somente uma URL usada pelo servidor Web internamente – um hook para significando que as requisições para uma URL devem ser manipuladas pelo FastCGI. (Mais sobre isto na próxima seção.)

Usando o `mod_rewrite` para apontar URLs no FastCGI

O segundo passo é dizer ao Apache para usar o FastCGI para URLs que combinam com certos padrões. Para fazer isto, use o módulo `mod_rewrite` e redirecione as URLs para `mysite.fcgi` (ou o que você tenha especificado na diretiva `FastCGIExternalServer`, como explicado na seção anterior).

Neste exemplo, nós dizemos ao Apache para usar o FastCGI como manipulador de qualquer requisição que não represente um arquivo do sistema de arquivos e não com `/media/`. Este é provavelmente caso mais comum, se você estiver usando o admin do Django:

```
<VirtualHost 12.34.56.78>
    ServerName example.com
    DocumentRoot /home/user/public_html
    Alias /media /home/user/python/django/contrib/admin/media
    RewriteEngine On
    RewriteRule ^/(media.*)$ /$1 [QSA,L,PT]
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^/(.*)$ /mysite.fcgi/$1 [QSA,L]
</VirtualHost>
```

O Django automaticamente usará a versão pre-rewrite da URL quando estiver construindo URLs com a tag de template `{% url %}` (e métodos similares).

Configuração do lighttpd

O `lighttpd` é um servidor Web bem leve, comumente usado para servir arquivos estáticos. Ele suporta FastCGI nativamente e, esta, é uma boa escolha para servir ambas páginas estáticas e dinâmicas, se seu site não tem qualquer necessidade específica relacionada ao Apache.

Esteja certo que `mod_fastcgi` está na sua lista de módulos, em algum lugar depois de `mod_rewrite` e `mod_access`, mas não depois do `mod_accesslog`. Você provavelmente precisará do `mod_alias`, para servir as medias do admin. Add the following to your lighttpd config file:

```
server.document-root = "/home/user/public_html"
fastcgi.server = (
    "/mysite.fcgi" => (
        "main" => (
            # Use host / port instead of socket for TCP fastcgi
            # "host" => "127.0.0.1",
            # "port" => 3033,
            "socket" => "/home/user/mysite.sock",
            "check-local" => "disable",
        )
    ),
)
alias.url = (
    "/media" => "/home/user/django/contrib/admin/media/",
)
url.rewrite-once = (
    "^(/media.*)$" => "$1",
    "^/favicon\.ico$" => "/media/favicon.ico",
    "^(/.*)$" => "/mysite.fcgi$1",
)
```

Executando múltiplos sites Django em um lighttpd

O lighttpd deixa você usar “configuração condicional” para permitir que a configuração seja customizada por host. Para especificar múltiplos sites FastCGI, é só adicionar um bloco condicional na sua configuração do FastCGI para cada site:

```
# Se o hostname for 'www.example1.com'...
$HTTP["host"] == "www.example1.com" {
    server.document-root = "/foo/site1"
    fastcgi.server = (
        ...
    )
    ...
}

# Se o hostname for 'www.example2.com'...
$HTTP["host"] == "www.example2.com" {
    server.document-root = "/foo/site2"
    fastcgi.server = (
        ...
    )
    ...
}
```

Você pode também rodar múltiplas instalações do Django no mesmo site, simplesmente especificando múltiplas entradas na diretiva `fastcgi.server`. Adicione um FastCGI host para cada um.

Configuração do Cherokee

O Cherokee é um servidor Web muito rápido, flexível e fácil de configurar. Ele suporta tecnologias muito difundidas hoje: FastCGI, SCGI, PHP, CGI, SSI, TLS e conexões criptografadas por SSL, Virtual hosts, Authentication, encoding em tempo de execução, Load Balancing, compatível com arquivos de log do Apache, Data Base Balancer, Reverse HTTP Proxy e muito mais.

O projeto Cherokee provê uma documentação para [configurar o Django](#) com o Cherokee.

Rodando o Django em ambiente compartilhado com Apache

Muitos hospedeiros não permitem que você rode seus próprios daemons de servidor ou editar o arquivo `httpd.conf`. Nestes casos, ainda é possível rodar o Django usando processos gerados pelo servidor Web.

Note

Se você está usando processos gerados pelo servidor Web, como explicado nesta seção não há necessidade de você iniciar o servidor FastCGI você mesmo. O Apache gerará um número de processos, escalando como ele precisar.

No seu diretório Web raiz, adicione isto ao arquivo chamado `.htaccess`:

```
AddHandler fastcgi-script .fcgi
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ mysite.fcgi/$1 [QSA,L]
```

Então, crie um pequeno script para dizer ao Apache como gerar seu programa FastCGI. Crie um arquivo `mysite.fcgi` e coloque-o no seu diretório Web, e assegure-se de torná-lo executável:

```
#!/usr/bin/python
import sys, os

# Adicione um caminho customizado do Python.
sys.path.insert(0, "/home/user/python")

# Mudando para o diretório do seu projeto. (Opcional.)
# os.chdir("/home/user/myproject")

# Setar a variável de ambiente DJANGO_SETTINGS_MODULE.
os.environ['DJANGO_SETTINGS_MODULE'] = "myproject.settings"

from django.core.servers.fastcgi import runfastcgi
runfastcgi(method="threaded", daemonize="false")
```

Restartando o servidor gerado

Se você mudou qualquer código Python do seu site, você precisará dizer ao FastCGI que o código mudou. Mas não há necessidade de reiniciar o Apache neste caso. Ao invés, somente re-envie o `mysite.fcgi`, ou edite o arquivo, desta forma o timestamp do arquivo mudará. Quando o Apache ver que o arquivos foi atualizado, ele irá reiniciar sua aplicação Django para você.

Se você tem acesso ao terminal em um sistema Unix, você pode realizar isto facilmente usando o comando `touch`:

```
touch mysite.fcgi
```

Servindo arquivos de mídia da administração

Regardless of the server and configuration you eventually decide to use, you will also need to give some thought to how to serve the admin media files. The advice given in the *modpython* documentation is also applicable in the setups detailed above.

Forçando o prefixo de URL para um valor em particular

Como muitas destas soluções baseadas no FastCGI requerem re-escrita de URL em algum ponto dentro do servidor Web, a informação de caminho que o Django vê pode não ser a mesma da URL original que foi passada. Isto é um problema se a aplicação Django é servida sob um prefixo particular e você deseja que as URLs geradas com a tag `{% url %}` apareçam com o prefixo, ao invés da versão re-escrita, que pode conter por exemplo, `mysite.fcgi`.

O Django faz uma boa tentativa pra descobrir qual o verdadeiro nome de prefixo do script que deve ser. Em particular, se o servidor Web seta o `SCRIPT_URL` (específica para o `mod_rewrite` do Apache), ou `REDIRECT_URL` (setada por alguns servidores, incluindo Apache + `mod_rewrite` em algumas situações), o Django encontrará o prefixo original automaticamente.

Nos casos onde o Django não encontra o prefixo corretamente e onde você quer que o valor original seja usado nas URLs, você pode setar a configuração `FORCE_SCRIPT_NAME` no arquivo `settings` principal. Isto seta o nome do script uniformemente para toda URL servida via este arquivo `settings`. Assim, você precisará usar arquivos de configurações diferentes, se você quiser diferentes conjuntos de URLs para ter nomes de scripts diferentes neste caso, mas está é uma situação rara.

Como um exemplo de como usá-lo, se sua configuração do Django está servindo todas as URLs em `'/'` e você esperava usar esta configuração, você poderia setar `FORCE_SCRIPT_NAME = '/'` no seu arquivo `settings`.

Se você é novato na implantação do Django e/ou Python, nós recomendamos que você tente o *mod_wsgi* primeiro. Na maioria dos casos ele é a opção de implantação mais fácil, rápida e estável.

See also:

- O capítulo 12 do Django Book discute a implantação e especialmente a escalabilidade em maiores detalhes.

Reporte de erros via e-mail

Quando você está rodando um site público deve sempre desligar a configuração `DEBUG`. Isso irá fazer seu servidor operar muito mais rápido, e também evitará que usuários maliciosos vejam detalhes de sua aplicação que não devem ser revelados pelas páginas de erro.

Entretanto, rodar com o `DEBUG` como `False` significa que você nunca verá os erros gerados pelo seu site – todo mundo sempre verá somente a sua página de erros pública.

Erros de servidor

Quando `DEBUG` é `False`, o Django irá enviar um email para os usuários listados em `ADMIN` sempre que seu código lançar uma exceção não tratada e que resulta num “internal server error” (código de status HTTP 500). Isto dá aos administradores a notificação imediata de quaisquer erros. O `ADMINS` irá pegar uma descrição do erro, um traceback completo do Python, e detalhes sobre a requisição HTTP que causou o erro.

Por padrão, o Django irá mandar e-mails de `root@localhost`. Porém, alguns provedores de e-mail rejeitam todos os e-mails desse endereço. Para usar um endereço de remetente diferente, modifique a configuração `SERVER_EMAIL`.

Para desabilitar este comportamento, basta remover todas as entradas da configuração `ADMINS`.

Erros 404

O Django também pode ser configurado para enviar e-mail com erros de links quebrados (erros 404 “page not found”). O Django envia um email sobre os erros quando:

- `DEBUG` é `False`
- `SEND_BROKEN_LINK_EMAILS` é `True`
- Seu `MIDDLEWARE_CLASSES` inclui `CommonMiddleware` (o que é feito por padrão).

Se essas condições forem satisfeitas, o Django irá enviar um e-mail para os usuários listados em `MANAGERS` toda vez que seu código gerar um erro 404 e a requisição tiver um referente. (Ela não se preocupa em enviar e-mail de erros 404 que não têm um referente – normalmente esses são tentativas dos usuários de escrever URLs ou algum bot web quebrado).

Você pode dizer ao Django parar de reportar alguns erros 404 específicos apromirando as configurações `IGNORABLE_404_ENDS` e `IGNORABLE_404_STARTS`. Ambos devem ser uma tupla de strings. Por exemplo:

```
IGNORABLE_404_ENDS = ('.php', '.cgi')
IGNORABLE_404_STARTS = ('/phpmyadmin/',)
```

Neste exemplo, um error 404 para qualquer final de URL com `.php` ou `.cgi` não será reportado. Nem mesmo qualquer URL que comece com `/phpmyadmin/`.

O melhor caminho para desabilitar este comportamento é setar o `SEND_BROKEN_LINK_EMAILS` para `False`.

See also:

Você pode também configurar um reporte de erro customizado, personalizando uma parte do *exception middleware*. Se você escrever um manipulador de erro personalizado, será uma boa idéia emular o manipulador de erros nativo do Django e somente reportar/logar erros se o `DEBUG` for `False`.

Provendo dados iniciais para models

As vezes é útil povoar seu banco de dados com dados no momento da instalação de uma aplicação. Aqui há algumas formas de se fazer o Django criar estes dados automaticamente: você pode prover *dados iniciais usando fixtures* ou você pode prover *dados iniciais em SQL*.

Geralmente, usar um fixture é um método mais limpo desde que o banco de dados seja agnóstico, mas dados iniciais em SQL podem ser um pouco mais flexíveis.

Provendo dados iniciais com fixtures

Uma fixture é uma coleção de dados que o Django sabe como importar para dentro de um banco de dados. A forma mais simples de se criar uma fixture, se você já tem alguns dados, é usando o comando `manage.py dumpdata`. Ou, você pode escrever a mão; as fixtures podem ser escritas em XML, YAML, ou JSON. A *documentação de serialização* tem mais detalhes sobre cada um destes *formatos de serialização* suportados.

Como exemplo, aqui é mostrado uma fixture para um model simples `Person`, no formato JSON:

```
[
  {
    "model": "myapp.person",
    "pk": 1,
    "fields": {
      "first_name": "John",
      "last_name": "Lennon"
    }
  },
  {
    "model": "myapp.person",
    "pk": 2,
    "fields": {
      "first_name": "Paul",
      "last_name": "McCartney"
    }
  }
]
```

E aqui tem a mesma fixture como YAML:

```
- model: myapp.person
  pk: 1
  fields:
    first_name: John
    last_name: Lennon
- model: myapp.person
  pk: 2
  fields:
    first_name: Paul
    last_name: McCartney
```

Você armazenará estes dados em um diretório `fixtures` dentro de sua aplicação.

Carregar os dados é fácil: é só chamar `manage.py loaddata fixturename`, onde *fixturename* é o nome do arquivo fixture que você criou. Toda vez que você executar `loaddata` os dados serão lidos da fixture e recarregados no banco de dados. Note que isto significa que se você mudar algum dado criado por uma fixture e rodar `loaddata` novamente, ele irá sobrescrever a mudanças feitas.

Carregando automaticamente os dados iniciais de fixtures

Se você criar uma fixture chamada `initial_data` [`xml/yaml/json`], essas fixtures serão carregadas sempre que você rodar `syncdb`. Isto é extremamente conveniente, mas seja cuidadoso: lembre-se que os dados serão atualizado *toda vez* que voce executar `syncdb`. Então não use `initial_data` para dados que você deseja editar.

See also:

As fixtures são usadas também pelo *framework de testes* para ajudar a montar um ambiente consistente de testes.

Provendo dados iniciais em SQL

O Django provê um hook para passar SQL arbitrário para o banco de dados, que é executado somente depois de uma consulta `CREATE TABLE`, no momento em que você roda um `syncdb`. Você pode usar este hook para popular com dados padrões, ou também criar funções SQL, view, triggers, etc.

O hook é simples: o Django simplesmente procura por um arquivo chamado `sql/<modelname>.sql`, no diretório da aplicação, onde `<modelname>` é o nome do model em minúsculo.

Então, se você tem um model `Person` em uma aplicação chamada `myapp`, você poderia adicionar SQL arbitrário num arquivo `sql/person.sql` dentro do diretório `myapp`. Aqui tem um exemplo do que o arquivo pode conter:

```
INSERT INTO myapp_person (first_name, last_name) VALUES ('John', 'Lennon');
INSERT INTO myapp_person (first_name, last_name) VALUES ('Paul', 'McCartney');
```

Para cada arquivo SQL, se fornecido, é esperado que contenha consultas SQL válidas que irão inserir os dados desejados (e.g., consultas `INSERT` apropriadamente formadas, separadas por ponto-e-vírgula);

Os arquivos SQL são lidos pelos comandos `sqlcustom`, `sqlreset`, `sqlall` e `reset` no *manage.py*. Consulte a *documentação do manage.py* para mais informações.

Note que se você tem múltiplos arquivos de dados SQL, não há garantias da ordem em que eles serão executados. A única coisa que pode ser assumida, é que no momento em que todos forem executados, o banco de dados já estará com todas as suas tabelas criadas.

Dados SQL para um backend específico de banco de dados

Há também um hook para backend específico de dados SQL. Por exemplo, você pode ter separados arquivos com dados iniciais para PostgreSQL e MySQL. Para cada aplicação, o Django procura por um arquivo chamado

`<appname>/sql/<modelname>.<backend>.sql`, onde `<appname>` é o diretório de sua aplicação, `<modelname>` é o nome do model em minúsculo e `<backend>` é o valor de `DATABASE_ENGINE` no seu arquivo settings (e.g., `postgresql`, `mysql`).

O backend específico de dados SQL é executado antes do backend não-específico de dados SQL. Por exemplo, se sua aplicação contém os arquivos `sql/person.sql` e `sql/person.postgresql.sql` e você está instalando a aplicação sobre o PostgreSQL, o Django irá executar o conteúdo do `sql/person.postgresql.sql` primeiro, e então `sql/person.sql`.

Rodando Django no Jython

Jython é uma implementação de Python que roda na plataforma Java (JVM). Django roda sem hacks no Jython a partir da versão 2.5, o que significa que você pode fazer o deploy do Django em qualquer plataforma Java.

Esse documento vai orientá-lo em como executar o Django no Jython.

Instalando o Jython

Django funciona a partir da versão 2.5b3 do Jython. Baixe o Jython em <http://www.jython.org/>.

Criando um container com servlets

Se você só quiser fazer experimentos com Django, pule para a próxima seção; Django inclui um servidor web leve que você pode usar para testes, para que você não precisa configurar nada até que esteja pronto para deploy Django em produção

Se você quiser usar Django num site em produção, use um container Java com servlets, como o [Apache Tomcat](#). Aplicações JavaEE completas como o [GlassFish](#) ou [JBoss](#) também funcionam, se você precisar das funcionalidades extras que eles incluem.

Instalando o Django

O próximo passo será instalar o Django. É exatamente da mesma maneira a instalar com o Python padrão, então veja *Remova qualquer versão antiga do Django* e *Instalando o Django* para mais instruções.

Instalando as bibliotecas de suporte para a plataforma Jython

O projeto [django-jython](#) contém backends de banco de dados e comandos de gerência para o desenvolvimento com Django/Jython. Note que os backends embutidos do Django não irão funcionar no Jython.

Para instalá-lo, siga as [instruções de instalação](#) detalhas no site do projeto. Também leia a documentação dos [database backends](#) lá.

Diferenças do Django no Jython

Nesse ponto, Django no Jython deve parecer praticamente idêntico ao Django rodando no Python padrão. Porém, existem algumas diferenças a serem lembradas:

- Lembre-se de usar o comando `jython` ao invés de `python`. A documentação usa `python` por consistência, mas se você tiver usando Jython você precisará trocar `python` por `jython` mentalmente em todas as ocorrências
- Similarmente, você precisará usar a variável de ambiente `JYTHONPATH` ao invés de `PYTHONPATH`.

Integrando o Django com um banco de dados legado

Embora o Django seja mais adequado para o desenvolvimento de novas aplicações, é bem possível integrá-lo com bases de dados legadas. O Django inclui alguns utilitários para automatizar, tanto quanto possível, este processo.

Este documento assume que você sabe o básico sobre Django, pelo menos o que já foi discutido no [tutorial](#).

Uma vez que você tenha o Django configurado, siga esse processo geral para integrar com um banco de dados pré-existente.

Dê ao Django seus parâmetros do banco de dados

Você necessitará informar ao Django os seus parâmetros de conexão com o banco de dados, e qual tipo de banco de dados será usado. Faça isso editando estas configurações no seu [arquivo settings](#):

- `DATABASE_NAME`
- `DATABASE_ENGINE`
- `DATABASE_USER`
- `DATABASE_PASSWORD`
- `DATABASE_HOST`
- `DATABASE_PORT`

Geramento automático de modelos

O Django vem acompanhado de um utilitário, chamado `inspectdb`, que pode criar modelos a partir de um banco de dados existente. Você pode ver o que ele gera usando este comando:

```
python manage.py inspectdb
```

Salve a saída como um arquivo usando o redirecionamento de saída padrão do Unix:

```
python manage.py inspectdb > models.py
```


Esta funcionalidade é entendida como um shortcut (atalho), e não como um gerador definitivo de modelos. Veja a documentação do `inspectdb` para mais informações.

Uma vez que você tenha limpo seus modelos, nomeie o arquivo como `models.py` e coloque-o no pacote Python correspondente à sua aplicação. Então adicione sua aplicação nas configurações de `INSTALLED_APPS`.

Instale as tabelas do core do Django

Agora, execute o comando `syncdb` para instalar qualquer dado extra no banco de dados, como permissões de administração ou tipos de conteúdo:

```
python manage.py syncdb
```

Veja se funciona

Estes são os passos básicos – a partir daqui você terá de modificar os modelos gerados pelo Django até que eles funcionem da maneira que lhe é conveniente. Tente acessar seus dados por meio da API de banco de dados do Django, e tente editar objetos por meio do site de administração do Django.

Exportando CSV com o Django

Este documento explica como gerar CSV (Valores Separados por Vírgulas) dinamicamente usando views do Django. Para fazer isso, você pode utilizar a [biblioteca Python CSV](#) ou o sistema de templates do Django.

Python comes with a CSV library, `csv`. The key to using it with Django is that the `csv` module's CSV-creation capability acts on file-like objects, and Django's `HttpResponse` objects are file-like objects.

Usando a biblioteca Python CSV

O Python vem com uma biblioteca CSV, `csv`. A chave para usá-la com o Django é que a capacidade de criação de CSV do módulo `csv` recai em objetos que equivalem a arquivos, e objetos `HttpResponse` do Django são deste tipo.

Aqui tem um exemplo:

```
import csv
from django.http import HttpResponse

def some_view(request):
    # Cria o objeto HttpResponse com o cabeçalho CSV apropriado.
    response = HttpResponse(mimetype='text/csv')
    response['Content-Disposition'] = 'attachment; filename=somefilename.csv'

    writer = csv.writer(response)
    writer.writerow(['First row', 'Foo', 'Bar', 'Baz'])
    writer.writerow(['Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"])

    return response
```

O código e comentários devem ser auto-explicativos, mas algumas coisas merecem ser mencionadas:

- O response pega um `mimetype` especial, `text/csv`. Isso avisa aos navegadores que o documento é um arquivo CSV, em vez de um arquivo HTML. Se você deixar isso desligado, os navegadores provavelmente irão interpretar a saída com um HTML, o que resultará em feiúra, um assustador gobbledygook (deve ser um ser epacial) na sua janela do navegador.
- O response pega um cabeçalho adicional `Content-Disposition`, que contém o nome do arquivo CSV. Esse nome de arquivo é arbitrário; chame-o como quiser. Ele será usado pelos navegadores em caixas de diálogo “Salvar como...”, etc.

- Usar hooks na API de geração do CSV é fácil: Simplesmente passe `response` como o primeiro argumento para `csv.writer`. A função `csv.writer` espera um objeto semelhante a um arquivo e objetos `HttpResponse` são justamente isso.
- Para cada linha em seu arquivo CSV, chame `writer.writerow`, passando-o um objeto iterável como uma lista ou tupla.
- O módulo CSV se preocupa em escapar por você, então você não tem de se preocupar em escapar strings com aspas ou vírgulas. Somente passe para o `writerow()` suas strings normalmente, e ele irá fazer a coisa certa.

Usando o sistema de template

Alternativamente, você pode usar o *sistema de template do Django* para gerar CSV. Isso é mais baixo nível do que usar a biblioteca CSV, mas a solução é apresentada para fins de complementação.

A idéia aqui é passar uma lista de itens para o seu template e ter a saída separada por vírgulas através de uma tag de loop `for`.

Aqui vai um exemplo, que gera o mesmo arquivo CSV do outro exemplo:

```
from django.http import HttpResponse
from django.template import loader, Context

def some_view(request):
    # Cria o objeto HttpResponse com o cabeçalho CSV apropriado.
    response = HttpResponse(mimetype='text/csv')
    response['Content-Disposition'] = 'attachment; filename=somefilename.csv'

    # The data is hard-coded here, but you could load it from a database or
    # some other source.
    # Aqui os dados estão embutidos no código, mas você poderia carregá-lo
    # de um banco de dados ou de alguma outra fonte.

    csv_data = (
        ('First row', 'Foo', 'Bar', 'Baz'),
        ('Second row', 'A', 'B', 'C', '"Testing"', "Here's a quote"),
    )

    t = loader.get_template('my_template_name.txt')
    c = Context({
        'data': csv_data,
    })
    response.write(t.render(c))
    return response
```

This template is quite basic. It just iterates over the given data and displays a line of CSV for each row. It uses the `addslashes` template filter to ensure there aren't any problems with quotes.

A única diferença entre esse exemplo e o anterior, é que esse usa o carregador de templates em vez do módulo CSV. O resto do código – tais como o `mimetype='text/csv'` – é o mesmo.

Então crie um template `my_template_name.txt` com este código:

```
{% for row in data %}"{{ row.0|addslashes }}"', "{{ row.1|addslashes }}"', "{{ row.
↪2|addslashes }}"', "{{ row.3|addslashes }}"', "{{ row.4|addslashes }}"
{% endfor %}
```

Este template é muito básico. Ele somente itera sobre o dado passado e mostra uma linha de CSV para cada linha. Ele usa o filtro de template `addslashes` para assegurar que não haja quaisquer problemas com aspas.

Outro formato baseado em texto

Repare que não existe muita especificidade para CSV aqui – somente o formato de saída. Você pode usar qualquer destas técnicas para gerar saída de qualquer formato baseado em texto que você sonhar. Você pode também usar uma técnica similar para gerar dados binários; veja [Exportando PDFs com o Django](#) por exemplo.

Exportando PDFs com o Django

Este documento explora como exportar arquivos PDF dinamicamente usando views do Django. Isso é possível graças a uma excelente, biblioteca PDF de código-aberto para Python, chamada [ReportLab](#).

A vantagem de gerar dinamicamente arquivos PDF é que você pode criar PDFs customizados para diferentes propósitos – como para diferentes usuários ou diferentes partes de conteúdos.

Por exemplo, o Django foi usado no [kusports.com](#) para gerar versões de impressão customizadas dos torneios do NCAA, como arquivos PDF, para pessoas participantes no concurso March Madness.

Instalando o ReportLab

Baixe e instale a biblioteca ReportLab do site <http://www.reportlab.org/downloads.html>. O [guia de usuário](#) (não coincidentemente, um arquivo PDF) explica como instalá-lo.

Teste sua instalação importando-o no interpretador interativo do Python:

```
>>> import reportlab
```

Se o comando não mostrar nenhum erro, a instalação está funcionando.

Escreva o seu view

A chave para gerar PDFs dinamicamente com o Django é que a API do ReportLab trabalha sobre objetos que se comportam como arquivos, e objetos [HttpResponse](#) do Django são objetos assim.

Aqui vai um exemplo “Hello World”:

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Crie o objeto HttpResponse com o cabeçalho de PDF apropriado.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=somefilename.pdf'

    # Crie o objeto PDF, usando o objeto response como seu "arquivo".
```

```
p = canvas.Canvas(response)

# Desenhe coisas no PDF. Aqui é onde a geração do PDF acontece.
# Veja a documentação do ReportLab para a lista completa de
# funcionalidades.
p.drawString(100, 100, "Hello world.")

# Feche o objeto PDF, e está feito.
p.showPage()
p.save()
return response
```

O código e comentários devem ser auto-explicativos, mas umas poucas coisas merecem ser mencionadas:

- O response recebe um mimetype especial, `application/pdf`. Isso informa ao navegador que o documento é um arquivo PDF, em vez de um arquivo HTML. Se você não informar isto, o navegador irá provavelmente interpretar a saída como um HTML, o que poderia resultar em um resultado feio e assustador na janela do navegador.
- O response recebe um cabeçalho adicional `Content-Disposition`, que contém o nome do arquivo PDF. Esse nome de arquivo é arbitrário: chame-o como quiser. Ele será usado pelo navegador na caixa de diálogo “Salvar como...”, etc.
- O cabeçalho `Content-Disposition` começa com `'attachment; '` neste exemplo. Isso força o navegador a mostrar uma caixa de diálogo avisando/confirmando como manipular o documento, mesmo se um padrão está definido na máquina. Se você não informar o `'attachment'`, o navegador manipulará o PDF usando qualquer programa/plugin que ele tiver configurado para usar com PDFs. Veja como esse código se pareceria:

```
response['Content-Disposition'] = 'filename=algum_nomedearquivo.pdf'
```

- Usar hooks da API do ReportLab é fácil: somente passe `response` como o primeiro argumento para o `canvas.Canvas`. A classe `Canvas` recebe um objeto como arquivo, e objetos `HttpResponse` atendem a esse requisito.
- Note que todo método subsequente de geração de PDF é chamado no objeto PDF (neste caso, `p`) – não no `response`.
- Finalmente, é importante chamar `showPage()` e `save()` sobre o arquivo PDF.

PDFs complexos

Se você está criando um documento PDF complexo com o ReportLab, considere a utilização da biblioteca `cStringIO` como um manipulador temporário para o seu arquivo PDF. A biblioteca `cStringIO` fornece uma interface para objeto como arquivo que é particularmente eficiente. Aqui o exemplo acima “Hello World” é re-escrito para usar o `cStringIO`:

```
from cStringIO import StringIO
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def some_view(request):
    # Crie o objeto HttpResponse com o cabeçalho PDF apropriado.
    response = HttpResponse(mimetype='application/pdf')
    response['Content-Disposition'] = 'attachment; filename=somefilename.pdf'

    buffer = StringIO()

    # Crie um objeto PDF, usando o objeto StringIO como seu "arquivo."
    p = canvas.Canvas(buffer)
```

```
# Desenhe coisas no PDF. Aqui é onde a geração do PDF acontece.
# Veja a documentação do ReportLab para a lista completa de
# funcionalidades.
p.drawString(100, 100, "Hello world.")

# Feche o objeto PDF.
p.showPage()
p.save()

# Pegue o valor do buffer StringIO e escreva-o para o response.
pdf = buffer.getvalue()
buffer.close()
response.write(pdf)
return response
```

Mais recursos

- [PDFlib](#) é outra biblioteca de geração de PDF que tem binding para o Python. Para usá-lo com o Django, somente use o mesmo conceito explicado neste artigo.
- [Pisa HTML2PDF](#) é mais uma biblioteca de geração de PDF. Pisa é entregue com um exemplo de como integrá-la com o Django.
- [HTMLdoc](#) é um script de linha de comando que consegue converter HTML para PDF. Ele não possui uma interface Python, mas você consegue escapar do terminal usando `system` ou `popen` e recebendo a saída no Python.
- [forge_fdf no Python](#) é uma biblioteca que preenche formulários em PDF.

Outros formatos

Repare que não há muitos destes exemplos específicos de PDF – somente o que é usado o `reportlab`. Você pode usar uma técnica similar para gerar qualquer formato que possa ser gerado por uma biblioteca Python. Também veja [Exportando CSV com o Django](#) para outro exemplo e algumas técnicas que você pode usar quando gera formatos baseados em texto.

Como servir arquivos estáticos

Django, por si só, não serve arquivos estáticos (media), como imagens, folhas de estilo (CSS), ou vídeo. Esse trabalho é deixado para o servidor Web de sua escolha.

O motivo para isso é que servidores Web comuns, como Apache e [lighttpd](#) e [Cherokee](#), são muito mais apropriados para servir arquivos estáticos do que um framework de aplicações web.

Dito isso, Django suporta arquivos estáticos **durante o desenvolvimento**. Você pode usar a view `django.views.static.serve()` para servir arquivos estáticos.

See also:

Se você precisa servir mídia administrativa de uma localização não padrão, veja o parâmetro `--adminmedia` para `runserver`.

Não diga que não avisamos

Usar este método é **ineficiente** e **inseguro**. Não use isso em um ambiente de produção. Use apenas para desenvolvimento.

Para informações sobre servir arquivos estáticos em um ambiente de produção com Apache, veja a [documentação Django mod_python](#).

Como proceder

Aqui temos a definição formal da view `serve()`:

```
def serve(request, path, document_root, show_indexes=False):
```

Para usá-la, somente coloque isto na sua [URLconf](#):

```
(r'^site_media/(?P<path>.*)$', 'django.views.static.serve',
 {'document_root': '/path/to/media'}),
```

...onde `site_media` é a URL onde a sua mídia será mapeada, e `/path/to/media` é a pasta raiz para sua mídia. Isto irá chamar a view `serve()`, passando o caminho da URLconf e o parâmetro (obrigatório) `document_root`.

Dado o URLconf acima:

- O arquivo `/path/to/media/foo.jpg` será mapeado para a URL `/site_media/foo.jpg`.
- O arquivo `/path/to/media/css/mystyles.css` será mapeado para a URL `/site_media/css/mystyles.css`.
- O arquivo `/path/bar.jpg` não será acessível, pois não é mapeado para a pasta raiz.

É claro, não é obrigatória a utilização de uma string fixa para o valor `'document_root'`. Você pode querer ter uma variável em seu arquivo settings e usar o valor que estiver nela. Que permitirá a você e outros desenvolvedores trabalhar sobre o código facilmente, mudando o valor como quiserem. Por exemplo, se nós tivermos uma linha no `settings.py` que diz:

```
STATIC_DOC_ROOT = '/path/to/media'
```

...nós poderíamos escrever uma entrada *URLconf* desta forma:

```
from django.conf import settings
...
(r'^site_media/(?P<path>.*)$', 'django.views.static.serve',
 {'document_root': settings.STATIC_DOC_ROOT}),
```

Tome cuidado para não usar o mesmo caminho da sua configuração `ADMIN_MEDIA_PREFIX` (cujo padrão é `/media/`) já que isso irá sobrescrever sua entrada na configuração de URL.

Listagem de diretórios

Opcionalmente, você pode passar um parâmetro `show_indexes` para a view `serve()`. O valor padrão é `False`. Se for `True`, o Django exibirá uma listagem dos arquivos dos diretórios.

Exemplo:

```
(r'^site_media/(?P<path>.*)$', 'django.views.static.serve', {'document_root': '/
↪path/to/media', 'show_indexes': True}),
```

Você pode customizar a exibição do índice criando um template chamado `static/directory_index`. Esse template recebe dois objetos em seu contexto:

- `directory` – o nome do diretório (como string)
- `file_list` – a lista de nomes de arquivos (como strings) no diretório

Aqui está o template `static/directory_index.html` padrão:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
↪TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
  <meta http-equiv="Content-Language" content="en-us" />
  <meta name="robots" content="NONE,NOARCHIVE" />
  <title>Index of {{ directory }}</title>
</head>
<body>
  <h1>Index of {{ directory }}</h1>
  <ul>
    {% for f in file_list %}
    <li><a href="{{ f }}">{{ f }}</a></li>
    {% endfor %}
  </ul>
</body>
</html>
```

Antes do Django 1.0.3, havia um bug na visão que fornecia listagem de diretórios. O template que era carregado tinha de ser chamado `static/directory_listing` (sem extensão `.html`). Para compatibilidade retroativa, o Django ainda carrega templates com o nome antigo (sem extensão), mas irá preferir a versão `directory_index.html`, com extensão.

Limitando o uso para `DEBUG=True`

Como `URLconfs` são simples módulos Python, você pode usar lógica Python para fazer com que a exibição de arquivos estáticos seja possível apenas em fase de desenvolvimento. Esse é um truque útil para se certificar de que a exibição de arquivos estáticos não seja usada em modo de produção por engano.

Faça isso inserindo uma instrução `if DEBUG` em volta da inclusão do `django.views.static.serve()`. Aqui está um exemplo completo de `URLconf`:

```
from django.conf.urls.defaults import *
from django.conf import settings

urlpatterns = patterns('',
    (r'^articles/2003/$', 'news.views.special_case_2003'),
    (r'^articles/(?P<year>\d{4})/$', 'news.views.year_archive'),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$', 'news.views.month_archive'),
    (r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d+)/$', 'news.views.
↪article_detail'),
)

if settings.DEBUG:
    urlpatterns += patterns('',
        (r'^site_media/(?P<path>.*)$', 'django.views.static.serve', {'document_root
↪': '/path/to/media'})),
    )
```

Este código é simples. Ele importa as configurações e checa o valor do `DEBUG`. Se for avaliado como `True`, então `site_media` será associada com a view `django.views.static.serve`. Caso contrário então a view não estará disponível.

Obviamente, a questão aqui é lembrar-se de colocar `DEBUG=False` no arquivo de configurações. Mas você já devia fazer isso de qualquer forma.

See also:

O [agregador da comunidade do Django](#) onde nós agregamos conteúdos da comunidade global do Django. Muitos escritores no agregador escrevem este tipo de material, how-to.

Part IV

FAQ do Django

Por que existe este projeto?

O Django surgiu a partir de uma necessidade prática: a World Online, uma operação de jornal online, responsável por construir aplicativos Web intensivamente sob prazos jornalísticos. Na truculenta sala de notícias, com frequência, a World Online precisa construir aplicativos web inteiros em apenas algumas horas, do conceito ao produto público final.

Ao mesmo tempo, os desenvolvedores da World Online têm se mantido perfeccionistas quanto a seguir as melhores práticas de desenvolvimento Web.

No outono de 2003, os desenvolvedores da World Online (Adrian Holovaty e Simon Willison) abandonaram o PHP e começaram a utilizar o Python para desenvolver seus Web Sites. Ao construir sites intensos e ricos em interatividade como o Lawrence.com, eles começaram a sintetizar um framework Web genérico que os permitiam executar o seu trabalho de uma forma melhor e mais rápida. Eles modificaram este framework constantemente, adicionando melhorias por mais de dois anos.

No verão de 2005, a World Online decidiu abrir o código do software resultante, o Django. O Django não seria possível sem todo o apoio de projetos open-source como o [Apache](#), [Python](#), [PostgreSQL](#), e é excitante pra nós sermos capazes de dar um retorno para a comunidade open-source.

O que significa “Django”? E como se pronuncia?

O projeto Django foi nomeado em homenagem a [Django Reinhardt](#), um guitarrista de jazz cigano dos anos 30-50. Até a presente data, ele é considerado um dos melhores guitarristas de todos os tempos.

Dê uma “olhada” em sua música. Você irá gostar!

Django deve ser pronunciado **JANG**-oh. Rima com FANG-oh. O “D” é pronunciado junto ao J, mas com uma entonação leve.

Também existe um [clipe de áudio](#) mostrando como se pronuncia.

O Django é estável?

Sim. A World Online vem utilizando o Django por mais de três anos sem problemas. Temos registros de sites construídos com o Django que já aguentaram picos de tráfego de mais de um milhão de acessos por hora e uma porção de efeitos slashdot. Então, sim, é bem estável.

Django é escalável?

Sim. Comparado com o tempo de desenvolvimento, hardware é um recurso barato, por isso, Django foi desenvolvido para se utilizar de tanto hardware quanto lhe estiver disponível.

O Django utiliza a arquitetura “shared-nothing”, o que significa que você pode adicionar hardware em qualquer nível da arquitetura – servidores de banco de dados, servidores de cache e servidores de aplicação.

O Framework separa claramente as camadas de banco de dados da camada de aplicação. E ainda disponibiliza um simples, porém poderoso, *framework de cache*.

Quem está por trás do Django?

O Django foi originalmente desenvolvido na World Online, o departamento Web de um jornal de Lawrence, Kansas, Estados Unidos. O Django é agora mantido por um time internacional de voluntários; você pode ler tudo sobre eles na *lista de committers*

Que sites utilizam o Django?

O wiki do Django mantém uma crescente *lista de sites desenvolvidos com o Django*. Sinta-se à vontade para adicionar seu site feito com o Django à lista.

O Django parece ser um framework MVC, mas vocês chama o “controller” de “view” e a View de “template”. Por que vocês não utilizam a nomenclatura padrão?

Bem, “nomes padrão” é o tipo de coisa que pode ser debatida.

Na nossa interpretação, a “view” descreve a informação que é mostrada ao usuário. Não é necessariamente *como* a informação *aparece*, mas *qual* informação é mostrada. A view descreve *qual informação você vê*, e não *como você vê*. Há uma diferença sutil.

Então, em nosso caso, uma “view” é a função Python chamada por uma URL em particular, porque essa chamada descreve qual informação será mostrada.

E mais, separar o conteúdo da apresentação é crucial – que é onde entram os templates. No Django, uma “view” descreve qual informação é apresentada, mas é a view que normalmente seleciona um template, que descreve *como* a informação é mostrada.

Onde o “controller” entra, então? No caso do Django, o controller é o próprio framework: a máquina que manda uma requisição ao view apropriado, de acordo com a configuração das URL’s no Django.

Se você é um amante de siglas, você poderia dizer que o Django é um framework “MTV” – isso é, “model”, “template” e “view”. Essa correspondência faz muito mais sentido.

No fim do dia, é claro, toda essa sopa de letrinhas se resume a concluir seu trabalho. E, independente de como as coisas foram nomeadas, o Django consegue fazer o serviço da forma que é mais lógica para nós.

O <Framework X> faz <funcionalidade Y> – Por que Django não faz também?

Nós estamos cientes de que existem outros excelentes frameworks de desenvolvimento web aí fora, e nós não somos avessos a pegar emprestadas idéias que consideraríamos apropriadas. Entretanto, o Django foi desenvolvido precisamente por que nós estamos descontentes com o status quo, então, por favor, saiba que só porque o “<Framework X>” faz algo, não é razão suficiente para adicionar essa funcionalidade ao Django.

Porque vocês escreveram o Django do zero, ao invés de usar outras bibliotecas Python?

Quando o Django foi escrito inicialmente, alguns anos atrás, Adrian e Simon gastaram bastante tempo explorando os vários frameworks Python para a Web disponíveis.

Em nossa opinião, nenhum deles era completamente satisfatório.

Somos esquisitos. Pode até mesmo nos chamar de perfeccionistas. (Com prazos.)

Com o passar do tempo, nós começamos a descobrir outras bibliotecas open-source que faziam coisas que nós já havíamos implementado. Foi confortante ver outras pessoas resolvendo problemas semelhantes de formas semelhantes, mas era tarde demais para integrar código externo: nós já tínhamos escrito, testado e implementado partes de nosso framework em alguns cenários em produção – e nosso código atendia nossas necessidades perfeitamente.

Na maioria dos casos, porém, nós descobrimos que os frameworks e ferramentas existentes tinham sempre algum tipo de falha de design fundamental que nós não gostávamos. Nenhuma ferramenta se encaixava 100% em nossas filosofias.

Como dissemos: Somos estranhos.

Documentamos nossa filosofia na *Página de filosofia sobre o design*.

O Django é um sistema de gerenciamento de conteúdo (CMS)?

Não, o Django não é um CMS, nem nenhum outro tipo de produto finalizado. É um framework web, uma ferramenta que permite que você construa websites.

Por exemplo, não faz muito sentido comparar o Django a produtos como o [Drupal](#), porque o Django é algo que você usa para *criar* coisas como o Drupal.

É claro, a administração automática do Django economiza um bom tempo e é fantástica – mas o site administrativo é um módulo do framework Django. Além disso, mesmo que o Django tenha facilidades especiais para construir aplicações do tipo CMS, isso não significa que ele não seja apropriado para construir aplicações “não CMS” (o que quer que isso signifique!).

Como posso baixar a documentação para lê-la offline?

A documentação do Django está disponível dentro do diretório `docs` de cada pacote lançado do Django. Essas documentações estão no formato ReST (ReStructured Text), e cada arquivo corresponde a uma página no site oficial do Django.

Como a documentação é [mantida sob controle de versão](#), você pode visualizar as mudanças na documentação assim como você visualiza as mudanças de código.

Tecnicamente, as documentações no site do Django são geradas a partir das últimas versões de desenvolvimento desses documentos ReST, assim a documentação no site do Django pode oferecer mais informação que os documentos que vieram com a última versão empacotada do Django.

O Grupo de Localização do Django para o Português também mantêm sua versão da documentação traduzida sob controle de versão.

Onde eu posso encontrar desenvolvedores Django para contratar?

Consulte nossa página de [desenvolvedores disponíveis](#) para uma lista de desenvolvedores Django que ficariam felizes em te ajudar.

Você pode também estar interessado em postar uma oferta de emprego em <http://www.gypsyjobs.com/> . Se você quer encontrar pessoas capacitadas em Django em sua região, tente <http://djangopeople.net/> .

Como eu começo?

1. Baixe o código.
2. Instale o Django (leia o *guia de instalação*).
3. Execute o *tutorial*.
4. Verifique o resto da documentação, e *pergunte* se você estiver com problemas.

Quais são os pré requisitos para executar o Django?

O Django requer o [Python 2.3](#), especificamente qualquer versão do Python da 2.3 até a 2.6. Nenhuma outra biblioteca Python é requerida para o uso básico do Django.

Para um ambiente de desenvolvimento – se você apenas quer experimentar o Django – você não precisa ter um servidor web separado; o Django vem com o seu próprio servidor leve de desenvolvimento. Para um ambiente de produção, o Django segue a especificação [WSGI](#), o que significa que ele pode rodar sobre uma ampla variedade de plataformas de servidores. Veja *Implantando o Django* para algumas alternativas populares. Além disso, a [página wiki de opções de arranjo do servidor](#) contém detalhes para algumas estratégias de implantação.

Se você quer usar o Django com um banco de dados, que é o caso provável, você irá também precisar de um servidor de banco de dados. O [PostgreSQL](#) é recomendado, porque somos fãs do PostgreSQL, e o [MySQL](#), [SQLite 3](#), e [Oracle](#) são suportados também.

Eu perco algo se eu rodar o Django com uma a versão 2.3 do Python em vez de usar a 2.5?

Não no framework. Atualmente, o Django suporta oficialmente quaisquer versões do Python da 2.3 até a 2.6, inclusive. Porém, alguns componentes adicionais podem precisar de uma versão mais recente do Python; o componente `django.contrib.gis`, por exemplo, requer pelo menos o Python 2.4, e aplicações de terceiros para uso com o Django são, é claro, livres para escolher seus próprios requerimentos de versão.

Note porém que no próximo ano e talvez no outro o Django irá começar a remover suporte para versões antigas do Python como parte da migração que culminará com o Django rodando no Python 3.0 (veja a próxima questão para

detalhes). Assim, se você está começando com o Python, é recomendado que você use a última versão da série 2.x (atualmente, o Python 2.6). Isso irá permitir que você tome proveito de numerosas melhorias e otimizações na linguagem Python desde a versão 2.3, e irá ajudar a facilitar o processo de remoção do suporte para versões antigas do Python no caminho para o Python 3.0.

Posso usar o Django com o Python 3.0?

Não agora. O Python 3.0 introduziu um número de mudanças não retro compatíveis na linguagem Python, e apesar dessas mudanças serem algo bom para o futuro do Python, irá demorar um pouco até que a maioria dos softwares Python atualizem e sejam capazes de rodar no Python 3.0. Para bases grandes de código Python, como é o caso do Django, a expectativa é que a transição leve pelo menos um ano ou dois (já que envolve a remoção de suporte para versões mais antigas do Python e por isso deve ser feita gradualmente).

Enquanto isso, versões da série 2.x do Python serão suportadas e receberão correções de bug e atualizações de segurança da equipe de desenvolvimento do Python, assim, continuar a usar a série 2.x do Python durante o período de transição não deve representar risco algum.

O Django executará em hosts compartilhados (como TextDrive ou o Dreamhost)?

Veja nossa página de [hosts amigáveis ao Django](#) .

Eu devo usar a versão oficial ou a versão de desenvolvimento?

Os desenvolvedores do Django melhoram o framework todos os dias e são muito bons em não enviar código quebrado ao repositório. Nós usamos o código do desenvolvimento (do repositório Subversion) diretamente em nossos servidores, então nós o consideramos estável. Com isso em mente, recomendamos que você use a última versão do código em desenvolvimento, porque ele geralmente tem mais funcionalidades e menos bugs que as versões “oficiais”.

FAQ: Usando o Django

Por que acontece um erro ao importar o DJANGO_SETTINGS_MODULE?

Certifique-se de que:

- A variável de ambiente DJANGO_SETTINGS_MODULE está configurada para um módulo Python totalmente qualificado (ex.: “mysite.settings”).
- O módulo está em `sys.path` (import `mysite.settings` deve funcionar).
- O módulo não contém erros de sintaxe (é claro).
- Se você está usando o `mod_python`, mas *não* está usando o manipulador de requisições do Django, você precisa corrigir um bug do `mod_python` relacionado ao uso de `SetEnv`; antes de você importar qualquer coisa do Django você precisa fazer o seguinte:

```
os.environ.update(req.subprocess_env)
```

(onde `req` é o objeto de requisição do `mod_python`).

Eu não gosto da sua linguagem de template. Eu tenho de usá-la?

Pensamos que nossa linguagem de templates é a melhor coisa inventada desde os pedaços de bacon mas reconhecemos que a escolha da linguagem de templates é uma discussão quase religiosa. Não há nada no Django que requer que você use a linguagem de template, então, se você prefere o ZPT, Cheetah, ou o que quer que seja, sinta-se livre para usá-los.

Eu tenho de usar sua camada de modelo/banco de dados?

Não. Assim como o sistema de templates, a camada de modelo/banco de dados está desacoplada do resto do framework.

A única exceção é: se você usar uma biblioteca diferente de banco de dados, você irá perder o sistema de administração gerado automaticamente pelo Django. Esse aplicativo é acoplado à camada de banco de dados do Django.

Como eu uso os campos de imagem e arquivos?

O uso de um *FileField* ou um *ImageField* nos models requer alguns passos:

1. No seu arquivo de configurações, defina o *MEDIA_ROOT* como o caminho completo para o diretório onde você gostaria que o Django salvasse os arquivos enviados. (Para melhor performance, esses arquivos não são guardados em um banco de dados.) Defina o *MEDIA_URL* como a URL pública para esse diretório. Assegure-se de que esse diretório tenha permissão de escrita para o usuário do servidor Web.
2. Adicione o *FileField* ou *ImageField* ao seu model, definindo a opção *upload_to* para instruir o Django para qual subdiretório do *MEDIA_ROOT* ele deve enviar os arquivos.
3. Tudo que será gravado no seu banco de dados é o caminho para o arquivo (relativo ao *MEDIA_ROOT*). Você pode também querer usar o atributo de conveniência *url* fornecido pelo Django. Por exemplo, se o seu *ImageField* é chamado *mug_shot*, você pode obter a URL absoluta para a sua imagem em um template com `{{ object.mug_shot.url }}`.

Como eu torno uma variável disponível em todos os meus templates?

Algumas vezes seus templates precisam todos da mesma coisa. Um exemplo comum seria menus gerados dinamicamente. À primeira vista, parece lógico simplesmente adicionar um dicionário comum ao contexto do template.

A solução correta é usar um *RequestContext*. Detalhes em como fazer isso estão aqui: *Subclassing Context: RequestContext*.

FAQ: Obtendo ajuda

Como eu faço X? Por que Y não funciona? Onde eu posso obter ajuda?

Se este FAQ não contém uma resposta a sua pergunta, você pode tentar a [lista oficial de usuários do Django \(django-users\)](#) ou a [lista da comunidade Django Brasil \(django-brasil\)](#). Sinta-se livre para perguntar quaisquer questões relacionadas a instalação, uso ou depuração do Django.

Se você prefere o IRC, os canais #django e #django-br (este em língua portuguesa) na rede Freenode é uma comunidade ativa de indivíduos que podem ser capazes de ajudá-lo a resolver seu problema.

Por que a minha mensagem não apareceu na django-users ou na django-brasil?

As listas [django-users](#) e [django-brasil](#) tem muitos inscritos. Isso é bom para a comunidade, já que significa que muitas pessoas estão disponíveis para contribuir respondendo as perguntas. Infelizmente, significa também que as listas é um alvo atrativo para spammers.

Para combater o problema do spam, quando você entra em alguma das listas, nós manualmente moderamos a primeira mensagem que você envia para a lista. Isso significa que os spammers são pegos, mas também significa que a sua primeira questão para a lista pode demorar um pouco para ser respondida. Pedimos desculpas por qualquer inconveniente que essa política pode causar.

Ninguém na django-users ou django-brasil respondeu minha questão! O que eu faço?

Tente tornar sua questão mais específica, ou informar um exemplo melhor do seu problema.

Como na maioria das listas de projetos open-source, as pessoas na [django-users](#) e [django-brasil](#) são voluntárias. Se ninguém respondeu sua pergunta, pode ser que ninguém saiba a resposta, pode ser que ninguém tenha entendido a pergunta, ou pode ser que todos os que podem ajudar estão ocupados. Você pode tentar o IRC – visite os canais de IRC #django e #django-br na rede Freenode.

Você pode notar que nós temos uma segunda lista de e-mail chamada [django-developers](#) – mas por favor, não envie perguntas de suporte para essa lista. Essa lista é para discussão dos desenvolvedores do Django em si. Fazer uma pergunta técnica ali pode ser considerado muito inconveniente.

Acho que eu encontrei um bug! O que eu faço?

Instruções detalhadas de como lidar com um bug em potencial podem ser encontradas em nosso *Guia de contribuição do Django*.

Acho que encontrei um problema de segurança. O que eu faço?

Se você pensa ter encontrado um problema de segurança no Django, por favor envie uma mensagem para security@djangoproject.com. Essa é uma lista particular, aberta somente a desenvolvedores mais antigos e altamente confiáveis do Django, e seus arquivos não são públicos.

Devido à natureza sensível de problemas de segurança, pedimos que se você pensa ter encontrado algo, *por favor* não envie uma mensagem para uma das listas públicas do Django. O Django tem uma política para lidar com questões de segurança; enquanto o defeito está em aberto, gostaríamos de minimizar quaisquer danos que poderiam ser infligidos pelo seu conhecimento público.

FAQ: Bancos de dados e modelos

Como eu posso ver as consultas SQL que o Django está executando?

Certifique-se de que a configuração `DEBUG` está definida como `True`. Então, apenas faça isso:

```
>>> from django.db import connection
>>> connection.queries
[{'sql': 'SELECT polls_polls.id,polls_polls.question,polls_polls.pub_date FROM ↵
↵polls_polls',
  'time': '0.002'}]
```

`connection.queries` está disponível apenas se `DEBUG` for `True`. É uma lista de dicionários na ordem de execução das consultas. Cada dicionário tem o seguinte:

```
`sql` -- A declaração SQL
`time` -- O tempo que a declaração levou para executar, em segundos.
```

`connection.queries` inclui todas as declarações SQL – INSERTs, UPDATES, SELECTs, etc. Cada vez que o seu app aciona o banco de dados, a consulta será gravada. Note que os SQL brutos logados em `connection.queries` podem não incluir os parâmetros entre aspas. A colocação de aspas nos parâmatros é executada pelo backend específico de banco de dados, e nem todos os backends fornecem uma forma de obter o SQL depois de colocadas as aspas.

Eu posso usar o Django com um banco de dados pré-existente?

Sim. Veja *Integrando com um banco de dados legado*.

Se eu faço mudanças no model, como eu atualizo o banco de dados?

Se você não se importar em “resetar” os dados, o utilitário `manage.py` do seu projeto tem uma opção para resetar o SQL para uma aplicação em particular:

```
manage.py reset appname
```

Isso apaga quaisquer tabelas relacionadas com o appname e as recria.

Se você se importa sobre a remoção dos dados, você tem de executar as declarações `ALTER TABLE` manualmente no seu banco de dados. Esse é o modo que nós sempre fizemos, porque lidar com dados é uma operação muito sensível e nós procuramos evitar sua automatização. Dito isso, existem trabalhos sendo feitos para adicionar a funcionalidade de atualização automatizada de banco de dados de forma parcial.

Os modelos do Django suportam chaves primárias compostas?

Não. Apenas chaves primárias simples são suportadas.

Mas isso não é um problema na prática, já que nada o impede de adicionar outras restrições (usando a opção do `model unique_together` ou criando a restrição diretamente no seu banco de dados) garantindo a unicidade nesse nível. Chaves primárias simples são necessárias para que coisas como a interface administrativa funcione; por exemplo, você precisa de uma forma simples de ser capaz de especificar um objeto para editar ou remover.

Como eu adiciono opções específicas de banco de dados às minhas declarações `CREATE TABLE`, como especificar a tabela como tipo MyISAM?

Nós tentamos evitar casos especiais no código do Django para acomodar todas as opções específicas de banco de dados, como tipo da tabela, etc. Se você gostaria de usar essas opções, crie um *arquivo de dados SQL inicial* que contenha as declarações `ALTER TABLE` que você quer fazer. Os arquivos de dados iniciais são executados no seu banco de dados após as declarações `CREATE TABLE`.

Por exemplo, se você está usando o MySQL e quer que suas tabelas sejam do tipo MyISAM, crie um arquivo de dados iniciais e coloque algo assim dentro dele:

```
ALTER TABLE myapp_mytable ENGINE=MyISAM;
```

Como explicado na documentação do *arquivo de dados SQL inicial*, esse arquivo SQL pode conter comandos SQL arbitrários, assim você pode fazer quaisquer modificações que quiser nele.

Por que o Django está vazando memória?

O Django não é conhecido por vazar memória. Se você descobrir que seus processos Django estão alocando mais e mais memória, sem nenhum sinal de liberação, verifique se o `DEBUG` está definido como `False`. Se o `DEBUG` for `True`, então o Django salvará uma cópia de cada declaração SQL que ele executou.

(As consultas são salvas em `django.db.connection.queries`. Veja *Como eu posso ver as consultas SQL que o Django está executando?*.)

Para resolver esse problema, defina o `DEBUG` como `False`.

Se você precisa limpar a lista de consultas manualmente em qualquer ponto das suas funções, apenas execute `reset_queries()`, assim:

```
from django import db
db.reset_queries()
```

FAQ: A administração

Não consigo logar. Quando eu insiro um usuário e senha válidos, ele apenas trás a página de login de novo, sem mensagens de erro.

O cookie de login não está configurado corretamente, porque o domínio do cookie enviado pelo Django não casa com o domínio no seu navegador. Verifique essas duas coisas:

- Configure o `SESSION_COOKIE_DOMAIN` na configuração de seu admin para casar com seu domínio. Por exemplo, se você está acessando “<http://www.mysite.com/admin/>” no seu navegador, em “`myproject.settings`” você deverá definir o `SESSION_COOKIE_DOMAIN = 'www.mysite.com'`.
- Alguns navegadores (Firefox?) não gostam de acessar cookies de domínios que não têm pontos. Se você está executando o site administrativo em “localhost” ou outro domínio que não tem um ponto, tente acessar “localhost.localdomain” ou “127.0.0.1” e igualmente configure o `SESSION_COOKIE_DOMAIN`.

Não consigo logar. Quando eu insiro um usuário e senha válidos, ele apenas trás a página de login de novo, com um erro “Por favor, informe um usuário e senha corretos”.

Se você está certo sobre seu usuário e senha, verifique se seu usuário tem o `is_active` e “`is_staff`” definidos como `True`. O site administrativo somente libera o acesso a usuários com esses dois campos em `True`.

Como eu previno o middleware de cache de fazer cache do site administrativo?

Defina o valor de `CACHE_MIDDLEWARE_ANONYMOUS_ONLY` para `True`. Veja a [documentação do cache](#) para mais informações.

Como eu defino automaticamente o valor de um campo para o usuário que editou o objeto por último na administração?

A classe `ModelAdmin` fornece ganchos (hooks) de personalização que permitem que você transforme um objeto quando ele é salvo, usando detalhes obtidos da requisição. Ao extrair o usuário atual da requisição, e personalizar o gancho `ModelAdmin.save_model()`, você pode atualizar um objeto para refletir o usuário que o editou. Veja a [documentação sobre os métodos do `ModelAdmin`](#) para um exemplo.

Como eu restrinjo a edição de objetos na administração a apenas os usuários que os criaram?

A classe `ModelAdmin` também fornece ganchos de personalização que permitem que você controle a visibilidade e editabilidade dos objetos na administração. Usando o mesmo truque de extrair o usuário da requisição, `ModelAdmin.queryset()` e `ModelAdmin.has_change_permission()` podem ser usados para controlar a visibilidade e editabilidade de objetos na administração.

Minhas imagens e CSS do site administrativo funcionam no servidor de desenvolvimento, mas não aparecem ao executar no `mod_python`.

Veja servindo os arquivos da administração <howto-deployment-modpython-serving-the-admin-files na documentação “Como usar o Django com o `mod_python`”.

Meu “`list_filter`” contém um `ManyToManyField`, mas o filtro não é exibido.

O Django não exibirá um filtro para um `ManyToManyField` a menos que existam mais de dois objetos relacionados.

Por exemplo, se o seu `list_filter` inclui `sites`, e se existe apenas um site no seu banco de dados, ele não exibirá um filtro de “Site”. Nesse caso, o filtro por site seria inútil.

Como eu posso personalizar a funcionalidade da interface administrativa?

Você tem algumas opções. Se você quer estender o formulário gerado automaticamente pelo Django, você pode incluir módulos JavaScript na página pelo parâmetro `js` da classe `Admin` do model. Esse parâmetro é uma lista de URLs, como strings, apontando para módulos JavaScript que serão incluídos dentro do formulário da administração via uma tag `<script>`.

Se você quer mais flexibilidade do que apenas ajustar os formulários auto-gerados sinta-se livre para escrever views personalizadas para a administração. A administração é feito em Django puro, e você pode escrever views personalizadas que se liguem ao sistema de autenticação, verifiquem permissões e façam o que for preciso.

Se você quer personalizar a aparência da interface administrativa, leia a próxima questão.

O site administrativo gerado automaticamente é horrível! Como eu posso mudá-lo?

Gostamos dele, mas se você não gosta, você pode mudar a apresentação do site editando a folha de estilos e/ou as imagens assossiadadas. O site é construído usando HTML semântico e está cheio de hooks de CSS, assim, qualquer mudança que você quiser fazer é possível pela edição da folha de estilos. Nós escrevemos um *guia para o CSS usado na administração* para auxiliá-lo.

FAQ: Contribuindo com código

Como eu posso começar a contribuir com o código do Django?

Obrigado por perguntar! Escrevemos um documento inteiro dedicado a essa questão chamado *Contribuindo com o Django*.

Eu enviei uma correção de bug no sistema de tickets semanas atrás. Por que vocês estão ignorando meu patch?

Não se preocupe, não estamos te ignorando!

É importante entender que há uma diferença entre “um ticket está sendo ignorado” e “um ticket que ainda não foi atendido”. O sistema de tickets do Django contém centenas de tickets abertos, de vários graus de impacto na funcionalidade oferecida aos usuários e os desenvolvedores do Django têm de revisá-los e priorizá-los.

Além disso, as pessoas que trabalham com o Django são voluntárias. Como resultado, a quantidade de tempo que temos para trabalhar no framework é limitada e irá variar semanalmente, de acordo com nosso tempo livre. Se estivermos ocupados, não poderemos gastar tanto tempo no Django como gostaríamos.

A melhor maneira de fazer com que seu ticket chegue ao “checkin” é encurtando seu caminho, mesmo para aqueles que podem não estar intimamente familiarizado com a área de código em questão poder entender o problema e verificar a solução:

- Há instruções claras de como reproduzir o bug? Se este possui dependências (como o PIL), um módulo do contrib, ou um banco de dados específico, estas instruções são claras o suficiente para que qualquer um familiarize-se com ele?
- Se houver vários “patches” anexados ao ticket, está claro o que cada um faz, quais podem ser ignorados e quais importam?
- Os patches incluem um teste unitário? Se não, há uma explicação muito clara do por que não tem? Um teste expressa sucintamente qual problema está ocorrendo, e mostra que o patch realmente o corrige.

Se seu patch não teve chance de inclusão no Django, nós não o ignoramos – somente iremos fechar o ticket. Então se seu ticket ainda estiver aberto, não significa que nós estamos ignorando você; somente quer dizer que nós não tivemos tempo de olhá-lo ainda.

Quando e como eu posso lembrar o core team de um patch com qual me importo?

Educadamente, uma mensagem, no momento certo, para a lista de desenvolvimento é um bom caminho para receber atenção. Para determinar o tempo certo, você precisa se manter atento com a agenda. Se você postar sua mensagem quando os desenvolvedores principais estiverem tentando não estourar o prazo de uma funcionalidade, ou em fase de planejamento, você não terá toda a atenção que deseja. Entretanto, se você chamar a atenção com um ticket quando os desenvolvedores principais estão com as atenções voltadas para correção de bugs – bem na véspera de um sprint de correção de bugs, ou perto de um lançamento de uma versão beta, por exemplo – você terá muito mais chance de receber uma resposta produtiva.

Lembretes gentis no IRC também podem funcionar – novamente, num momento estratégico se possível. Durante um sprint de correções de bugs pode ser uma boa hora, por exemplo.

Outra forma de obter atenção é reunir vários tickets relacionados. Quando os desenvolvedores principais sentam para resolver um bug em uma certa área que eles não tenham tocado a algum tempo, eles podem demorar um pouco para lembrar todos os belos detalhes do funcionamento dessa área do código. Se você reúne várias pequenas correções de bug de um certo grupo, você cria um alvo atrativo, como um custo único que pode ser aproveitado por vários tickets.

Por favor, contenha-se em enviar e-mail para um desenvolvedor do core team pessoalmente, ou repetidamente levantar o mesmo problema, insistentemente. Este tipo de comportamento não atrairá nenhuma atenção adicional – certamente não a atenção que você deseja para o seu bug de estimação.

Mas eu tenho lembrado vocês várias vezes e vocês continuam ignorando meu patch!

Sério - nós não ignoramos você. Se seu patch não recebe a chance de ser incluído no Django, nós fechamos o ticket. Nós precisamos priorizar nossos esforços para todos os outros tickets, o que significa que alguns tickets serão atendidos antes de outros.

Um dos critérios que é usado para priorizar um bug é o número de pessoas que serão atingidas por ele. Bugs que possuem potencial para afetar muita gente, geralmente terão prioridade em relação aos outros.

Uma outra razão para que bugs possam ser ignorados por um momento é se o bug é um sintoma de um problema maior. Embora nós podemos gastar tempo escrevendo, testando e aplicando um monte de patches, algumas vezes a solução certa é refazer. Se uma reconstrução ou refatoração de um componente em particular foi proposto ou está a caminho, você pode encontrar bugs que afetam o componente, mas eles não terão muita atenção. Novamente, isto é só uma questão de priorizar recursos escassos. Concentrando na refatoração, nós poderemos fechar todos estes pequenos bugs de uma vez, e esperançosamente prevenir que outros pequenos bugs apareçam no futuro.

Seja qual for o motivo, por favor, mantenha em mente que enquanto você pode se incomodar com um bug regularmente, não significa que todos os usuários do Django tenham o mesmo problema. Diferentes usuários utilizam o Django de diferentes formas, estressando partes diferentes de código em diferentes condições. Quando nós avaliamos as prioridades relativas, nós geralmente tentamos considerar as necessidades de toda comunidade, não somente a gravidade de um determinado usuário. Isto não significa que pensamos que seu problema é insignificante – somente que em um limitado espaço de tempo que temos disponível, nós sempre iremos preferir fazer 10 pessoas felizes ao invés de uma só.

Part V

Referência da API

Os add-ons “django.contrib”

O Django procura seguir a filosofia Python das “baterias incluídas”. Ele vem com uma variedade de extras, ferramentas opcionais que resolvem os problemas comuns ao desenvolvimento web.

O código está em `django/contrib` na distribuição do Django. Este documento apresenta um resumo dos pacotes `contrib`, juntamente com as suas possíveis dependências.

Nota

A maioria desses add-ons – especificamente, os add-ons que incluem models ou template tags – você precisará adicionar o nome do pacote (e.x., `'django.contrib.admin'`) a sua configuração `INSTALLED_APPS` e executar `manage.py syncdb`.

O site admin do Django

Uma das mais poderosas partes do Django é a interface de administração automática. Ela lê os metadados de seu model para fornecer uma interface poderosa e pronta para produção que produtores de conteúdo podem imediatamente usar para começar a adicionar conteúdos ao site. Neste documento, nós discutimos como ativar, usar e personalizar a interface de administração do Django.

Nota

O site admin foi significativamente refatorado desde a versão 0.96. Este documento descreve a versão mais nova do site admin, que permite customizações mais ricas. Se você acompanha o desenvolvimento do Django em si, você pode ter ouvido isso descrito como “newforms-admin.”

Visão geral

Há cinco passos para ativar o site admin do Django:

1. Adicione `django.contrib.admin` no seu `INSTALLED_APPS`.
2. Determine quais models de aplicações devem ser editados pela interface admin.

3. Para cada um destes models, opcionalmente, crie uma classe `ModelAdmin` que encapsula as funcionalidades customizadas do admin e opções para este model particular.
4. Instancie um `AdminSite` e o informe sobre cada um de seus models e classes `ModelAdmin`.
5. Ligue a instância `AdminSite` ao seu `URLconf`.

See also:

Para informações sobre como servir arquivos de mídia (imagens, JavaScript, e CSS) associado ao admin em produção, veja [Servindo arquivos de mídia](#).

Objetos `ModelAdmin`

class `ModelAdmin`

A classe `ModelAdmin` é a representação de um model na interface de administração. Este são armazenados num arquivo chamado `admin.py` na sua aplicação. Vamos dar uma olhada num exemplo muito simples de `ModelAdmin`:

```
from django.contrib import admin
from myproject.myapp.models import Author

class AuthorAdmin(admin.ModelAdmin):
    pass
admin.site.register(Author, AuthorAdmin)
```

Você precisa de um objeto `ModelAdmin` sempre?

No exemplo anterior, a classe `ModelAdmin` não define quaisquer valores personalizados (ainda). Como um resultado, a interface admin padrão será fornecida. Se você estiver feliz com a interface admin padrão, você não precisa definir um objeto `ModelAdmin` sempre – você poder registrar a classe model sem fornecer uma descrição do `ModelAdmin`. O exemplo anterior poderia ser simplificado para:

```
from django.contrib import admin
from myproject.myapp.models import Author

admin.site.register(Author)
```

Opções `ModelAdmin`

O `ModelAdmin` é muito flexível. Ele tem várias opções para lidar com customização da interface. Todas as opções são definidas na subclasse `ModelAdmin`:

```
class AuthorAdmin(admin.ModelAdmin):
    date_hierarchy = 'pub_date'
```

`ModelAdmin.date_hierarchy`

Sete o `date_hierarchy` com o nome de um `DateField` ou `DateTimeField` do seu model, e a lista de dados incluirá uma navegação por data, utilizando o campo informado.

Exemplo:

```
date_hierarchy = 'pub_date'
```

`ModelAdmin.form`

Por padrão um `ModelForm` é dinamicamente criado para seu model. Ele é usado para criar o formulário apresentado em ambas as páginas adicionar/editar. Você pode facilmente fornecer seu próprio `ModelForm` para sobrescrever qualquer comportamento de formulário nas páginas adicionar/editar.

Para um exemplo veja a seção *Adicionando validação personalizada ao admin*.

ModelAdmin.fieldsets

Sete fieldsets para controlar o layout das páginas “adicionar” e “editar” do admin.

fieldsets é uma lista de tuplas duplas, em que cada tupla dupla representa um <fieldset> sobre a página de formulário do admin. (Um <fieldset> é uma “seção” do formulário.)

As tuplas duplas estão no formato (name, field_options), onde name é uma string representando o título do fieldset e field_options é um dicionário com informações sobre o fieldset, incluindo uma lista de campos para serem mostrados nele.

Um exemplo completo, recebido do model `django.contrib.flatpages.FlatPage`:

```
class FlatPageAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': ('url', 'title', 'content', 'sites')
        }),
        ('Advanced options', {
            'classes': ('collapse',),
            'fields': ('enable_comments', 'registration_required', 'template_name')
        }),
    )
```

Isso resulta numa página admin que parece com:

Se o fieldsets não for fornecido, o Django mostrará o padrão de cada campo que não for um AutoField e tenha um `editable=True`, em um fieldset único, na mesma ordem em que os campos foram definidos no model.

O dicionário `field_options` pode ter as seguintes chaves:

- **fields** Uma tupla com nomes de campos que serão mostrados no fieldset. Esta chave é obrigatória.

Exemplo:

```
{
    'fields': ('first_name', 'last_name', 'address', 'city', 'state'),
}
```

Para mostrar vários campos na mesma linha, envolva-os em suas próprias tuplas. Neste exemplo, os campos `first_name` e `last_name` serão mostrados na mesma linha:

```
{
    'fields': (('first_name', 'last_name'), 'address', 'city', 'state'),
}
```

- **classes** Uma lista contendo classes extra de CSS para serem aplicadas ao fieldset.

Exemplo:

```
{
    'classes': ['wide', 'extrapretty'],
}
```

Duas classes úteis definidas por padrão no site admin são `collapse` e `wide`. Os fieldsets com o estilo `collapse` serão inicialmente encolhidos no admin e substituídos por um link “click para expandir”. Os fieldsets com o estilo `wide` ocuparão um espaço horizontal extra.

- **description** Uma string de texto opcional extra para ser mostrado no topo de cada fieldset, abaixo do cabeçalho do fieldset.

Note que este valor *não* tem o HTML escapado quando é mostrado na interface do admin. Isso permite você incluir HTML se assim desejar. Alternativamente você poder usar texto plano e `django.utils.html.escape()` para escapar quaisquer caracteres especiais de HTML.

ModelAdmin.**fields**

Use esta opção como uma alternativa ao `fieldsets` se o layout não importar e se você deseja somente mostrar um sub-conjunto de campos disponíveis no formulário. Por exemplo, você poderia definir uma versão mais simples do formulário do admin para o model `django.contrib.flatpages.FlatPage` desta forma:

```
class FlatPageAdmin(admin.ModelAdmin):
    fields = ('url', 'title', 'content')
```

No exemplo acima, somente os campos ‘url’, ‘title’ e ‘content’ serão mostrados, sequencialmente, no formulário.

Nota

Estas opções `fields` não devem ser confundidas com a chave de dicionário `fields` que fica dentro da opção `fieldsets`, como descrito na seção anterior.

ModelAdmin.**exclude**

Este atributo, se fornecido, deve ser uma lista com nomes de campos a se escluir do formulário.

Por exemplo, vamos considerar o seguinte model:

```
class Author(models.Model):
    name = models.CharField(max_length=100)
    title = models.CharField(max_length=3)
    birth_date = models.DateField(blank=True, null=True)
```

Se você deseja um formulário do model `Author` que inclui somente os campos `name` e `title`, você poderia especificar `fields` ou `exclude` desta forma:

```
class AuthorAdmin(admin.ModelAdmin):
    fields = ('name', 'title')
```

```
class AuthorAdmin(admin.ModelAdmin):
    exclude = ('birth_date',)
```

Assim o model `Author` somente terá três campos, `name`, `title`, e `birth_date`, os formulários resultantes das declarações acima conterão exatamente os mesmos campos.

`ModelAdmin.filter_horizontal`

Use a nifty unobtrusive JavaScript “filter” interface instead of the usability-challenged `<select multiple>` in the admin form. The value is a list of fields that should be displayed as a horizontal filter interface. See `filter_vertical` to use a vertical interface.

`ModelAdmin.filter_vertical`

O mesmo que `filter_horizontal`, mas é um display vertical do filtro.

`ModelAdmin.list_display`

Configure o `list_display` para controlar quais campos são mostrados na listagem do admin.

Exemplo:

```
list_display = ('first_name', 'last_name')
```

Se você não seta `list_display`, o site admin mostrará uma única coluna que mostra o `__unicode__()` representando cada objeto.

Você tem quatro valores possível que podem ser usado no `list_display`:

- Uma campo de model. Por exemplo:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name')
```

- Uma função que aceita um parâmetro para a instância do model. Por exemplo:

```
def upper_case_name(obj):
    return ("%s %s" % (obj.first_name, obj.last_name)).upper()
upper_case_name.short_description = 'Name'

class PersonAdmin(admin.ModelAdmin):
    list_display = (upper_case_name,)
```

- Uma string representando um atributo no `ModelAdmin`. Este se comporta da mesma forma que a função acima. Por exemplo:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('upper_case_name',)

    def upper_case_name(self, obj):
        return ("%s %s" % (obj.first_name, obj.last_name)).upper()
    upper_case_name.short_description = 'Name'
```

- Uma string representando um atributo no model. Este se comporta quase como a função acima, a diferença é que o `self` neste contexto é a instância do model. Aqui temos um exemplo completo:

```
class Person(models.Model):
    name = models.CharField(max_length=50)
    birthday = models.DateField()

    def decade_born_in(self):
        return self.birthday.strftime('%Y')[:3] + "0's"
    decade_born_in.short_description = 'Birth decade'
```



```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'decade_born_in')
```

Uns poucos casos especiais para se tomar nota sobre o `list_display`:

- * Se o campo é um `ForeignKey`, o Django mostrará o `__unicode__()` do objeto relacionado.
- * Campos `ManyToManyField` não são suportados, pois poderiam implicar em execução de consultas SQL em separado pra cada linha da tabela. Se você deseja fazer isso de qualquer forma, dê ao seu model um método personalizado, e adicione o nome desse método no `list_display`. (Veja abaixo para saber mais sobre métodos personalizados no `list_display`.)
- * Se o campo é um `BooleanField` ou `NullBooleanField`, o Django mostrará um belo ícone "on" ou "off" ao invés de `True` ou `False`.
- * Se a string fornecida é um método do model, `ModelAdmin` ou uma função, o Django espacará o HTML da saída por padrão. Se você gostaria de não escapar a saída do método, de ao método um atributo `allow_tags` cujo o valor é `True`.

Aqui temos um exemplo completo::

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def colored_name(self):
        return '<span style="color: #s;">%s %s</span>' % (self.color_code,
↪self.first_name, self.last_name)
        colored_name.allow_tags = True

class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'colored_name')
```

- * Se a string dada é um método do model, `ModelAdmin` ou uma função que retorna `True` ou `False` o Django mostrará um belo ícone "on" ou "off" se você der ao método um atributo `boolean` cujo o valor é `True`.

Aqui temos um exemplo completo do model::

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    birthday = models.DateField()

    def born_in_fifties(self):
        return self.birthday.strftime('%Y')[:3] == '195'
        born_in_fifties.boolean = True

class PersonAdmin(admin.ModelAdmin):
    list_display = ('name', 'born_in_fifties')
```

- * Os métodos `__str__()` e `__unicode__()` não só são válidos no `list_display` como em qualquer outro método de model, então é perfeitamente OK usar isso::

```
list_display = ('__unicode__', 'some_other_field')
```

- * Normalmente, os elementos do `list_display` que não estão dentre os campos do bancos de dados atual não podem ser usado em ordenamentos (pois o Django não sabe como sortearlos a nível de banco de dados).

Entretanto, se um elemento de um `list_display` representa um certo campo do banco de dados, você pode indicar este fato configurando o atributo `admin_order_field` do item.

Por exemplo::

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    color_code = models.CharField(max_length=6)

    def colored_first_name(self):
        return '<span style="color: %s;">%s</span>' % (self.color_code, self.
↪first_name)
    colored_first_name.allow_tags = True
    colored_first_name.admin_order_field = 'first_name'
```

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'colored_first_name')
```

O exemplo acima dirá ao Django para ordenar o campo `first_name` quando tentar sortear pelo `colored_first_name` no admin.

ModelAdmin.list_display_links

Seta um `list_display_links` para controlar quais campos no `list_display` deveriam ser linkados à página de edição de um objeto.

Por padrão, a página de listagem linkará a primeira coluna – o primeiro campo especificado no `list_display` – para a página de edição de cada item. Mas `list_display_links` permite você mudar quais colunas serão linkadas. Configure `list_display_links` com uma lista de tuplas com nomes de campos (no mesmo formato como `list_display`) para linkar.

O `list_display_links` pode especificar um ou mais nomes de campos. Tantos quanto os nomes dos campos que aparecerem no `list_display`, o Django não se preocupa com quantos campos (ou quão poucos) serão linkados. O único requisito é: Se você usar `list_display_links`, você deve definir `list_display`.

Neste exemplo, os campos `first_name` e `last_name` serão linkados na página de listagem:

```
class PersonAdmin(admin.ModelAdmin):
    list_display = ('first_name', 'last_name', 'birthday')
    list_display_links = ('first_name', 'last_name')
```

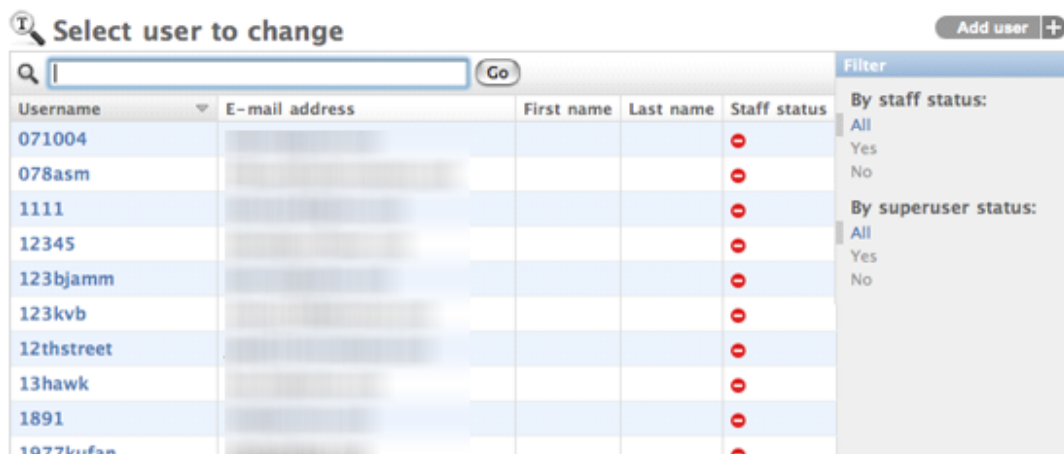
ModelAdmin.list_filter

Configure o `list_filter` para ativar os filtros na barra à direita da listagem do admin. Este deve ser uma lista de nomes de campos, e cada campo especificado devem ser `BooleanField`, `CharField`, `DateField`, `DateTimeField`, `IntegerField` ou `ForeignKey`.

Este exemplo, pêgo do model `django.contrib.auth.models.User`, mostra como ambos `list_display` e `list_filter` funcionam:

```
class UserAdmin(admin.ModelAdmin):
    list_display = ('username', 'email', 'first_name', 'last_name', 'is_staff')
    list_filter = ('is_staff', 'is_superuser')
```

O código acima resulta numa página de listagem do admin que parece com este:



(Este exemplo tem um `search_fields` definido. Veja abaixo.)

`ModelAdmin.list_per_page`

Configure o `list_per_page` para controlar quantos ítems devem aparecer em cada página da paginação da listagem. Por padrão, é setado para 100.

`ModelAdmin.list_select_related`

Configure `list_select_related` para dizer ao Django para usar `select_related()` na lista de objetos recebida na listagem do admin. Isso pode poupar um monte de consultas no banco de dados.

O valor deve ser `True` ou `False`. O padrão é `False`.

Note que o Django usará `select_related()`, indiferente desta configuração, se um dos campos no `list_display` for um `ForeignKey`.

Para mais sobre `select_related()`, veja [a documentação do `select_related\(\)`](#).

`ModelAdmin.inlines`

Veja objetos `InlineModelAdmin` abaixo.

`ModelAdmin.ordering`

Configure `ordering` para especificar como os objetos na listagem do admin devem ser ordenados. Este deve ser uma lista de tuplas no mesmo formato do parâmetro `ordering` do model.

Se este não for fornecido, o Django admin usará o ordenamento padrão do model.

Nota

O Django somente honra o primeiro elemento na lista/tupla; qualquer outro será ignorado.

`ModelAdmin.prepopulated_fields`

Configure `prepopulated_fields` com um mapeamento num dicionário com os campos que devem ser pré-populados a partir de outro campo:

```
class ArticleAdmin(admin.ModelAdmin):
    prepopulated_fields = {"slug": ("title",)}
```

Quando configurado, os campos dados usaram um pouco de JavaScript para popular os campos atribuídos. O principal uso para esta funcionalidade é a de automaticamente gerar um valor para campos `SlugField` a partir de um ou mais campos. O valor gerado é produzido concatenando os valores do campo originário, e então transformando este resultado num slug válido (e.g. substituindo barras por espaços).

O `prepopulated_fields` não aceita campos `DateTimeField`, `ForeignKey`, nem `ManyToManyField`.

`ModelAdmin.radio_fields`

Por padrão, o admin do Django usa uma interface com select-box (<select>) para campos que são `ForeignKey` ou que tem `choices` configurado. Se um campo é apresentado no `radio_fields`, o Django usará uma interface com radio-button ao invés. Assumindo que `group` é um `ForeignKey` no model `Person`:

```
class PersonAdmin(admin.ModelAdmin):
    radio_fields = {"group": admin.VERTICAL}
```

Você tem a escolha de usar `HORIZONTAL` ou `VERTICAL` do módulo `django.contrib.admin`.

Não inclua um campo no `radio_fields` a menos que seja um `ForeignKey` ou tenha `choices` setado.

`ModelAdmin.raw_id_fields`

Por padrão, o admin do Django usa um select-box (<select>) para campos que são `ForeignKey`. Algumas vezes você pode não desejar ficar sujeito a um overhead ao selecionar todos as instâncias relacionadas num drop-down.

O `raw_id_fields` é uma lista de campos que você gostaria de mudar dentro de um widget `Input` tanto para `ForeignKey` quanto `ManyToManyField`:

```
class ArticleAdmin(admin.ModelAdmin):
    raw_id_fields = ("newspaper",)
```

`ModelAdmin.save_as`

Seta `save_as` para habilitar um “save as” no formulário de edição do admin.

Normalmente, objetos tem três opções de salve: “Salvar”, “Salvar e continuar editando” e “Salvar e adicionar outro”. Se `save_as` for `True`, o “Salvar e adicionar outro” será substituído por um botão “Salvar como”.

Nota do tradutor

Se você não estiver utilizando uma versão do Django traduzida, os botões “Salvar e continuar editando”, “Salvar”, “Salvar e adicionar outro” e “Salvar como”, serão, respectivamente, “Save”, “Save and continue editing”, “Save and add another” e “Save as”.

“Salvar como” significa que o objeto será salvo como um novo objeto (com um novo ID), ao invés de um objeto já existente.

Por padrão, `save_as` é setado como `False`.

`ModelAdmin.save_on_top`

Seta `save_on_top` para adicionar os botões de salvar no topo do formulário de edição do admin.

Normalmente, os botões salvar aparecem somente na base do formulário. Se você setar `save_on_top`, os botões aparecerão em ambos os lugares, na base e no topo.

Por padrão, `save_on_top` é setado como `False`.

`ModelAdmin.search_fields`

Configure `search_fields` para habilitar uma caixa de busca na página de listagem do admin. Este deve ser configurado com uma lista de nomes de campos que serão afetados pela busca, sempre que alguém submeter um texto por esta caixa de texto.

Estes campos devem ser algum tipo de campo de texto, como `CharField` ou `TextField`. Você pode também executar uma pesquisa relacionada a `ForeignKey` com a lookup API “siga” a notação:

```
search_fields = ['foreign_key__related_fieldname']
```

Quando alguém faz uma busca pelo admin, o Django divide a consulta em palavras e retorna todos os objetos que contém cada uma das palavras, diferenciando maiúsculas, onde pelo menos uma das palavras deve estar no `search_fields`. Por exemplo, se `search_fields` é setado para `['first_name', 'last_name']` e um usuário procura por `john lennon`, o Django fará o equivalente a esta clausula SQL `WHERE`:

```
WHERE (first_name ILIKE '%john%' OR last_name ILIKE '%john%')
AND (first_name ILIKE '%lennon%' OR last_name ILIKE '%lennon%')
```

Para buscas mais rápidas e/ou mais restritivas, prefixe o nome do campo com um operador:

- ^ Combina com o início do campo. Por exemplo, se `search_fields` é setado para `['^first_name', '^last_name']` e um usuário procurar por john lennon, o Django fará o equivalente a cláusula SQL WHERE:

```
WHERE (first_name ILIKE 'john%' OR last_name ILIKE 'john%')
AND (first_name ILIKE 'lennon%' OR last_name ILIKE 'lennon%')
```

Esta consulta é mais eficiente que a normal `'%john%'`, pois o banco de dados somente precisa checar o início dos dados da coluna, ao invés de procurar através de todo ele. Ainda mais, se a coluna tem um index nela, alguns bancos de dados podem ser capazes de usar o index para esta consulta, mesmo sendo uma consulta LIKE.

- = Combina exatamente, não diferencia maiúsculas. Por exemplo, se `search_fields` for configurado para `['=first_name', '=last_name']` e um usuário procura por john lennon, o Django fará o equivalente a cláusula SQL WHERE:

```
WHERE (first_name ILIKE 'john' OR last_name ILIKE 'john')
AND (first_name ILIKE 'lennon' OR last_name ILIKE 'lennon')
```

Note que a entrada da consulta é dividida nos espaços, então, seguindo este exemplo, atualmente não é possível procurar por todos os dados em que `first_name` é exatamente 'john winston' (contendo espaço).

- @ Executa uma combinação full-text. Essa é como o método de procura padrão, mas usa um index. Atualmente isso só está disponível para MySQL.

`ModelAdmin.change_list_template`

Caminho para um template personalizado que será usado pelos objetos model na visão de “listagem”. Os templates podem sobrescrever ou estender os templates de base do admin como descrito em [Sobrescrevendo Templates do Admin](#).

Se você não especificar este atributo, um template padrão é entregue com o Django que fornece a aparência padrão do admin.

`ModelAdmin.change_form_template`

Caminho para o template personalizado que será usado na visão de criação e edição de um objeto model. Os templates podem sobrescrever ou estender os templates de base do admin como descrito em [Sobrescrevendo Templates do Admin](#).

Se você não especificar este atributo, um template padrão é entregue com o Django que fornece a aparência padrão do admin.

`ModelAdmin.object_history_template`

Caminho para um template que será usado na visão de histórico do objeto model. Os templates podem sobrescrever ou estender os templates de base do admin como descrito em [Sobrescrevendo Templates do Admin](#).

Se você não especificar este atributo, um template padrão é entregue com o Django que fornece a aparência padrão do admin.

`ModelAdmin.delete_confirmation_template`

Caminho para um template personalizado que será usado pela visão responsável por mostrar a confirmação de exclusão de um ou mais objetos. Os templates podem sobrescrever ou estender os templates de base do admin como descrito em [Sobrescrevendo Templates do Admin](#).

Se você não especificar este atributo, um template padrão é entregue com o Django que fornece a aparência padrão do admin.

Métodos do ModelAdmin

`ModelAdmin.save_model(self, request, obj, form, change)`

O método `save_model` recebe o `HttpRequest`, uma instância do `model`, uma instância de `ModelForm` e um valor booleano indicando se está adicionando ou editando um objeto. Aqui você pode fazer quaisquer operações pré ou pós salvamento.

Por exemplo, para atachar `request.user` ao objeto antes de salvá-lo:

```
class ArticleAdmin(admin.ModelAdmin):
    def save_model(self, request, obj, form, change):
        obj.user = request.user
        obj.save()
```

`ModelAdmin.save_formset(self, request, form, formset, change)`

O método `save_formset` recebe o `HttpRequest`, a instância do `ModelForm` pai e um valor booleano que indica se está adicionando ou editando um objeto pai.

Pro exemplo para atachar `request.user` a cada instância de `formset` de mudança:

```
class ArticleAdmin(admin.ModelAdmin):
    def save_formset(self, request, form, formset, change):
        instances = formset.save(commit=False)
        for instance in instances:
            instance.user = request.user
            instance.save()
        formset.save_m2m()
```

Outros métodos

`ModelAdmin.add_view(self, request, form_url='', extra_context=None)`

Um view do Django para a página de inclusão de instância de `model`. Veja a nota abaixo.

`ModelAdmin.change_view(self, request, object_id, extra_context=None)`

Um view do Django para a página de edição de instância de `model`. Veja nota abaixo.

`ModelAdmin.changelist_view(self, request, extra_context=None)`

Um view do Django para página de listagem/ações de instância de `model`. Veja nota abaixo.

`ModelAdmin.delete_view(self, request, object_id, extra_context=None)`

Um view do Django para página de confirmação de exclusão de instância e `model`. Veja nota abaixo.

`ModelAdmin.history_view(self, request, object_id, extra_context=None)`

Um view do Django para a página que mostra o histórico de modificações de uma instância de `model`.

Diferentemente dos métodos tipo hook do `ModelAdmin` detalhados na seção anterior, estes cinco métodos são na verdade, projetados para serem invocados como views do Django pelo URL dispatcher da aplicação `admin` que renderizam páginas com instâncias de `model` para operações CRUD. Como resultado, a completa sobrescrita destes métodos mudará o comportamento da aplicação `admin`.

Uma razão comum para sobrescrever estes métodos é para aumentar o contexto dos dados que são fornecidos para o template que renderiza o view. No exemplo a seguir, o view de edição é sobrescrito de modo que ao template renderizado é fornecido alguns dados extras que caso contrário não estariam disponíveis:

```
class MyModelAdmin(admin.ModelAdmin):

    # Um template para um view bastante personalizada:
    change_form_template = 'admin/myapp/extras/openstreetmap_change_form.html'
```

```
def get_osm_info(self):
    # ...

def change_view(self, request, object_id, extra_context=None):
    my_context = {
        'osm_data': self.get_osm_info(),
    }
    return super(MyModelAdmin, self).change_view(request, object_id,
        extra_context=my_context)
```

Definições de mídia ModelAdmin

Tem vezes em que você gostaria de adicionar um pouco de CSS e/ou JavaScript aos views adicionar/editar. Isso pode ser realizado usando uma classe interna Media no seu ModelAdmin:

```
class ArticleAdmin(admin.ModelAdmin):
    class Media:
        css = {
            "all": ("my_styles.css",)
        }
        js = ("my_code.js",)
```

Tenha em mente que este será prefixado com MEDIA_URL. A mesma regra se aplica nas *regras de definições de mídia em formulários*.

Adicionando validação personalizada ao admin

Adicionando validação personalizada de dados no admin é bem simples. A interface automática de administração reusa o `django.forms`, e a classe `ModelAdmin` tem a mesma habilidade de definir seu próprio formulário:

```
class ArticleAdmin(admin.ModelAdmin):
    form = MyArticleAdminForm
```

O `MyArticleAdminForm` pode ser definido em qualquer lugar dentro de seu caminho de import. Agora com seu form você pode adicionar sua própria validação para qualquer campo:

```
class MyArticleAdminForm(forms.ModelForm):
    class Meta:
        model = Article

    def clean_name(self):
        # faça algo que valide seus dados
        return self.cleaned_data["name"]
```

É importante usar um `ModelForm` aqui, caso contrário algo pode não funcionar. Veja a documentação *forms* em *validação personalizada* e, mais especificamente, o *notas de validação de formulários* para mais informações.

Objetos InlineModelAdmin

A interface admin tem a habilidade de editar models na mesma página de um model pai. Este são chamados de inlines. Suponhamos que você tenha dois models:

```
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author)
    title = models.CharField(max_length=100)
```

Você pode editar os livros escritos por um autor na página do autor. Você pode adicionar inlines ao seu model para especificá-los em um `ModelAdmin.inlines`:

```
class BookInline(admin.TabularInline):
    model = Book

class AuthorAdmin(admin.ModelAdmin):
    inlines = [
        BookInline,
    ]
```

O Django fornece duas subclasses ao `InlineModelAdmin` e elas são:

- `TabularInline`
- `StackedInline`

A diferença entre estas duas é meramente o template utilizado para renderizá-las.

Opções `InlineModelAdmin`

A classe `InlineModelAdmin` é uma subclasse de `ModelAdmin` de modo que ele herda todas as mesmas funcionalidades assim como as suas próprias:

`model`

O model em que o inline é usado. Este é obrigatório.

`fk_name`

O nome da chave estrangeira no model. Na maioria dos casos este será tratado de forma automática, mas `fk_name` deve ser especificada explicitamente se houver mais de uma chave estrangeira para o mesmo model pai.

`formset`

O padrão para `BaseInlineFormSet`. Usar seu próprio formset pode dar a você muitas possibilidades de personalização. Inlines são embutidos a volta de *model formsets*.

`form`

O valor para `form` padrão para `BaseModelForm`. Este é o que é passado através do `formset_factory` quando está criando o formset para este inline.

`extra`

Este controla o número de formulários extra que um formset mostrará além dos formulário iniciais. Veja a *documentação dos formsets* para mais informações.

`max_num`

Este controla o número máximo de formulários que serão mostrados num inline. Este não é diretamente correlacionado ao número de objetos, mas pode ser se o valor for pequeno o suficiente. Veja *Limitando o número de objetos editáveis* para mais informações.

raw_id_fields

Por padrão, o admin do Django usa um select-box (<select>) para campos que são `ForeignKey`. Algumas vezes você pode não desejar ficar sujeito a um overhead ao selecionar todas as instâncias relacionadas num drop-down.

O `raw_id_fields` é uma lista de campos que você gostaria de mudar dentro de um widget `Input` tanto para `ForeignKey` quanto para `ManyToManyField`:

```
class BookInline(admin.TabularInline):
    model = Book
    raw_id_fields = ("pages",)
```

template

O template usado para renderizar o inline na página.

verbose_name

Uma sobrescrita para `verbose_name` encontrado na classe interna `Meta` do model.

verbose_name_plural

Uma sobrescrita para o `verbose_name_plural` encontrado na classe interna `Meta` do model.

Trabalhando com um model com duas ou mais chaves estrangeiras para o mesmo model pai

Algumas vezes é possível haver mais de uma chave estrangeira para o mesmo model. Vejamos esta instância de model:

```
class Friendship(models.Model):
    to_person = models.ForeignKey(Person, related_name="friends")
    from_person = models.ForeignKey(Person, related_name="from_friends")
```

Se você quiser mostrar um inline nas páginas adicionar/editar do admin para `Person`, você precisa explicitamente definir a chave primária, tendo em vista que ele não consegue definir automaticamente:

```
class FriendshipInline(admin.TabularInline):
    model = Friendship
    fk_name = "to_person"

class PersonAdmin(admin.ModelAdmin):
    inlines = [
        FriendshipInline,
    ]
```

Trabalhando com Models Intermediários Many-to-Many

Por padrão, os widgets do admin para relações muito-para-muitos serão mostrados inlines não importando qual model contém a referência atual para o `ManyToManyField`. No entanto, quando você especificar um model intermediário usando o argumento `through` para um `ManyToManyField`, o admin não mostrará o widget por padrão. Isso porque cada instância de model intermediário requer mais informações que poderiam ser mostradas num único widget, e o layout requerido por vários widgets irão variar dependendo do model intermediário.

No entanto, nós ainda queremos ter a possibilidade de editar estas informações inline. Felizmente, isso é fácil de fazer com admin models inline. Suponhamos que temos os seguintes models:

```

class Person(models.Model):
    name = models.CharField(max_length=128)

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

class Membership(models.Model):
    person = models.ForeignKey(Person)
    group = models.ForeignKey(Group)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)

```

O primeiro passo em mostrar este model intermediário no admin é definir uma classe inline para o model Membership:

```

class MembershipInline(admin.TabularInline):
    model = Membership
    extra = 1

```

Este exemplo simples usa o valor padrão `InlineModelAdmin` para o model `Membership`, e limita os formulários adicionais para um. Isso poderia ser personalizado usando qualquer uma das opções disponíveis para classes `InlineModelAdmin`.

Agora crie visões no admin para os models `Person` e `Group`:

```

class PersonAdmin(admin.ModelAdmin):
    inlines = (MembershipInline,)

class GroupAdmin(admin.ModelAdmin):
    inlines = (MembershipInline,)

```

Finalmente, registre seus models `Person` e `Group` no site admin:

```

admin.site.register(Person, PersonAdmin)
admin.site.register(Group, GroupAdmin)

```

Agora seu site admin está configurado para editar objetos `Membership` tanto da página de detalhes do `Person` quanto do `Group`.

Usando relações genéricas como um inline

É possível usar um inline com objetos relacionados genericamente. Vamos dizer que você tenha os seguintes models:

```

class Image(models.Model):
    image = models.ImageField(upload_to="images")
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericForeignKey("content_type", "object_id")

class Product(models.Model):
    name = models.CharField(max_length=100)

```

Se você quer permitir a edição e criação de instâncias `Image` sobre `Product` você pode simplesmente usar `GenericInlineModelAdmin` fornecido pelo `django.contrib.contenttypes.generic`. No seu `admin.py` para esta aplicação de exemplo:

```

from django.contrib import admin
from django.contrib.contenttypes import generic

```

```
from myproject.myapp.models import Image, Product

class ImageInline(generic.GenericTabularInline):
    model = Image

class ProductAdmin(admin.ModelAdmin):
    inlines = [
        ImageInline,
    ]

admin.site.register(Product, ProductAdmin)
```

O `django.contrib.contenttypes.generic` fornece ambos um `GenericTabularInline` e `GenericStackedInline` e se comporta como qualquer outro inline. Veja a [documentação do contenttypes](#) para informações mais específicas.

Sobrescrevendo Templates do Admin

É relativamente fácil sobrescrever a maior parte dos templates que o módulo de admin usa para gerar as várias páginas de um site admin. Você pode sobrescrever somente algumas partes desses templates para uma app específica, ou um model específico.

Configurando os diretórios de template do admin do seu projeto

Os arquivos de template do admin estão localizados no diretório `contrib/admin/templates/admin`.

A fim de sobrescrever um ou mais deles, primeiro crie um diretório admin no diretórios de templates do seu projeto. Este pode ser qualquer um dos diretórios que você especificou no `TEMPLATE_DIRS`.

Dentro deste diretório admin, crie sub-diretórios com o nome de suas apps. Dentro destes sub-diretórios crie sub-diretórios com o nome de seus models. Perceba, que a aplicação admin procura por nomes de diretórios em minúsculo, então esteja certo do nome deles está todo em minúsculo, caso você esteja rodando sua aplicação num sistemas de arquivo case-sensitive.

Para sobrescrever um template do admin para uma app específica, copie e edite o template do diretório `django/contrib/admin/templates/admin`, e salve-o num dos diretórios criados a pouco.

Por exemplo, se nós procurarmos adicionar uma ferramenta na visão de listagem para todos os models de uma aplicação chamada `my_app`, nós copiariíamos `contrib/admin/templates/admin/change_list.html` para o diretório `templates/admin/my_app/` de seu projeto, e fariíamos as mudanças necessárias.

Caso desejássemos adicionar uma ferramenta na visão de listagem somente para um model específico chamado ‘Page’, nós poderíamos copiar aquele mesmo arquivo para o diretório `templates/admin/my_app/page` de seu projeto.

Sobrescrever vs. substituir um template do admin

Por causa do design modular dos templates do admin, geralmente é necessário, mas nem sempre aconselhável, substituir todo um template. É quase sempre melhor sobrescrever somente a seção do template que precisa ser mudada.

Para continuar o exemplo acima, nós queremos adicionar um novo link próximo ao `History` para o model `Page`. Depois de olhar no `change_form.html` nós determinamos que somente precisamos sobrescrever o bloco `object-tools`. Assim, aqui está o novo `change_form.html`:

```
{% extends "admin/change_form.html" %}
{% load i18n %}
{% block object-tools %}
{% if change %}{% if not is_popup %}
```

```

<ul class="object-tools">
  <li><a href="history/" class="historylink">{% trans "History" %}</a></li>
  <li><a href="mylink/" class="historylink">My Link</a></li>
  {% if has_absolute_url %}
    <li><a href="../../../r/{{ content_type_id }}/{{ object_id }}" class=
↪"viewsitelink">
      {% trans "View on site" %}</a>
    </li>
  {% endif %}
</ul>
{% endif %}{% endif %}
{% endblock %}

```

E era isso! Se nós colocarmos este arquivo no diretório `templates/admin/my_app`, nosso link apareceria em todos os formulários de edição dos models.

Templates que podem ser sobrescritos por uma app ou model

Nem todo template em `contrib/admin/templates/admin` pode ser sobrescrito por app ou por model. Os seguintes podem:

- `change_form.html`
- `change_list.html`
- `delete_confirmation.html`
- `object_history.html`

Para a maioria dos templates que não são sobrescritos desta forma, você pode ainda sobrescrevê-los pra o seu projeto inteiro. É só colocar uma nova versão no seu diretório `templates/admin`. Este é particularmente útil para criar páginas de erro 404 e 500.

Note: Alguns dos templates do admin, como o `change_list_request.html` são usados para renderizar inclusion tags personalizadas. Estas podem ser sobrescritas, mas em alguns casos provavelmente é melhor você criar sua própria versão da tag em questão e dá-lhe um nome diferente. Desta forma você pode usá-la seletivamente.

Template raiz e de login

Se você deseja mudar o template index ou de login, seria melhor você criar sua própria instância de `AdminSite` (veja abaixo), e mudar as propriedades `AdminSite.index_template` ou `AdminSite.login_template`.

Objetos AdminSite

Um site de administração do Django é representado por uma instância do `django.contrib.admin.sites.AdminSite`; por padrão, uma instância desta classe é criada como `django.contrib.admin.site` e você pode registrar seus models e instâncias do `ModelAdmin` nele.

Se você gostaria de configurar de setar sua própria interface de administração com comportamentos personalizados, contudo, você é livre para estender o `AdminSite` e sobrescrever ou adicionar qualquer coisa que queira. Então, simplesmente crie uma instância de sua classe estendida do `AdminSite` (da mesma forma como você instancia qualquer outra classe Python), e registre seus models e subclasses de `ModelAdmin` com ele ao invés de usar o padrão.

Atributos do AdminSite

AdminSite.index_template

Caminho para um template personalizado que será usado pela página principal do site de administração. Os templates podem sobrescrever ou estender os templates base do admin como descrito em [Sobrescrevendo Templates do Admin](#).

AdminSite.login_template

Caminho para um templates personalizado que será usado pela página de login do site admin. Os templates podem sobrescrever ou estender os templates base do admin como descrito em [Sobrescrevendo Templates do Admin](#).

Vinculando instâncias do AdminSite ao seu URLconf

O último passo ao configurar o admin do Django é vincular o sua instância AdminSite dentro do seu URLconf. Faça isso apontando uma certa URL para o método AdminSite.urls.

Neste exemplo, nós registramos a instância padrão AdminSite do `django.contrib.admin.site` na URL `/admin`

```
# urls.py
from django.conf.urls.defaults import *
from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',
    ('^admin/(.*)', admin.site.root),
)
```

Acima nós usamos `admin.autodiscover()` para automaticamente carregar os módulos `admin.py` do `INSTALLED_APPS`.

Neste exemplo, nós registramos a instância do AdminSite `myproject.admin.admin_site` na URL `/myadmin/`

```
# urls.py
from django.conf.urls.defaults import *
from myproject.admin import admin_site

urlpatterns = patterns('',
    ('^myadmin/(.*)', admin_site.root),
)
```

Realmente não há necessidade de usar o `autodiscover` quando estiver usando sua própria instância do AdminSite, tendo em vista que você provavelmente já tenha importado todos os seus módulos `admin.py` das aplicações no seu módulo `myproject.admin`.

Note que a expressão regular no URLpattern *deve* agrupar tudo que vier após a raiz da URL – por isso o `(.*)` nestes exemplos.

Vários sites admin no mesmo URLconf

É fácil criar várias instâncias de admin no mesmo site feito em Django. É só criar várias instâncias do AdminSite e colocá-los em URLs diferentes.

Neste exemplo, as URLs `/basic-admin/` e `/advanced-admin/` realçam versões separadas do site admin – usando as instâncias do AdminSite `myproject.admin.basic_site` e `myproject.admin.advanced_site`, respectivamente:

```
# urls.py
from django.conf.urls.defaults import *
from myproject.admin import basic_site, advanced_site

urlpatterns = patterns('',
    ('^basic-admin/(.*)', basic_site.root),
    ('^advanced-admin/(.*)', advanced_site.root),
)
```

django.contrib.auth

See *Autenticação de Usuário no Django*.

Framework de comentários do Django

O Django inclui um simples, e ainda customizável framework de comentários. O framework embutido de comentários pode ser usado para atachar comentário a qualquer model, então você pode usá-lo para comentar entradas de blog, photos, capítulos de livros, ou algo mais.

Note: Se você costumava usar o framework antigo (não documentado) de comentários do Django, precisará atualizá-lo. Veja o *guia de atualização* para instruções.

Guia rápido de início

Para começar a usar a aplicação `comments`, siga estes passos:

1. Instale o framework de comentários adicionando `'django.contrib.comments'` ao *INSTALLED_APPS*.
2. Execute `manage.py syncdb`, e então o Django criará as tabelas do `comments`.
3. Adicione as URLs da aplicação de comentário ao `urls.py` do seu projeto:

```
urlpatterns = patterns('',
    ...
    (r'^comments/', include('django.contrib.comments.urls')),
    ...
)
```

4. Use as *tags de template do comments* abaixo para atachar comentários em seus templates.

Você pode também querer examinar o *Configurações de Comentários*.

Tags de template do comments

Você irá primeiramente interagir com o sistema de comentários através de uma série de tags de templates que deixam você atachar comentários e gerar formulários para seus usuários para postá-los.

Como toda biblioteca de tags de template customizada, você precisará *carregá-las* antes de você usá-las:

```
{% load comments %}
```

Uma vez carregadas você pode usar as tags de templates abaixo.

Especificando para qual objeto o comentário será atachado

Os comentários do Django são todos “atachados” a algum objeto pai. Este pode ser qualquer instância de um modelo do Django. Cada uma das tags abaixo lhe dá diferentes formas de especificar a quais objetos serão atachados:

1. Refere-se ao objeto diretamente – o método mais comum. Na maioria das vezes, você terá algum objeto no contexto do template que você deseja atachar os comentários; você pode simplesmente usar este objeto.

Por exemplo, numa página de entrada de blog que tenha uma variável chamada `entry`, você poderia usar o seguinte, para carregar o número de comentário:

```
{% get_comment_count for entry as comment_count %}.
```

2. Refere-se ao objeto por content-type e id do objeto. Você poderia usar este método se você, por alguma razão, na verdade não tem o acesso direto ao objeto.

No exemplo seguinte, se você souber que ID do objeto é 14 mas não tem acesso ao objeto real, você poderia fazer algo tipo:

```
{% get_comment_count for blog.entry 14 as comment_count %}
```

Acima, `blog.entry` é um label de aplicação e nome de modelo (em minúsculo) da classe modelo.

Mostrando comentários

Para obter uma lista de comentários para algum objeto, use `get_comment_list`:

```
{% get_comment_list for [object] as [varname] %}
```

Por exemplo:

```
{% get_comment_list for event as comment_list %}
{% for comment in comment_list %}
    ...
{% endfor %}
```

Isto retorna uma lista de objetos `Comment`; veja [a documentação do modelo `comment`](#) para detalhes.

Contando comentários

Para contar comentário atachados em um objeto, use `get_comment_count`:

```
{% get_comment_count for [object] as [varname] %}
```

Por exemplo:

```
{% get_comment_count for event as comment_count %}

<p>This event has {{ comment_count }} comments.</p>
```

Mostrando o formulário de comentários

Para mostrar o formulário que os usuários usarão para postar um comentário, você pode usar `render_comment_form` ou `get_comment_form`.

Rápidamente renderizando o formulário de comentário

A forma mais fácil de mostrar um formulário de comentário é usando o `render_comment_form`:

```
{% render_comment_form for [object] %}
```

Por exemplo:

```
{% render_comment_form for event %}
```

Isto irá renderizar comentários usando um template chamado `comments/form.html`, uma versão padrão que é incluída no Django.

Renderizando um formulário customizado de comentário

Se você quiser mais controle sobre a aparência do formulário de comentário, você usa o `get_comment_form` para pegar um *objeto form* que você pode usar num template:

```
{% get_comment_form for [object] as [varname] %}
```

Um formulário completo pode ficar tipo:

```
{% get_comment_form for event as form %}
<form action="{% comment_form_target %}" method="POST">
  {{ form }}
  <tr>
    <td></td>
    <td><input type="submit" name="preview" class="submit-post" value="Preview"></
    <td>
  </tr>
</form>
```

Esteja certo de ler as *notas sobre o formulário de comentário*, abaixo, para alguma consideração especial que você poderá precisar fazer se estiver utilizando esta abordagem.

Pegando o alvo do formulário de comentário

Você pode ter notado que o exemplo acima usa outra tag de template – `comment_form_target` – para na verdade obter o atributo `action` do formulário. Este sempre retornará a URL correta para onde o comentário deverá ser postado;

```
<form action="{% comment_form_target %}" method="POST">
```

Redirecionando depois de uma postagem de comentário

Para especificar a URL que você quer redirecionar depois que um comentário foi postado, você pode incluir um campo invisível no formulário chamado `next`. Por exemplo:

```
<input type="hidden" name="next" value="{% url my_comment_was_posted %}" />
```

Notas sobre o formulário de comentário

O formulário usado pelo sistema de comentário tem uns poucos atributos anti-spam importantes, que você deve conhecer:

- Ele contém um número de campos invisíveis que contém timestamps, informações sobre o objeto ao qual o comentário deve estar atachado, e um “hash de segurança” usado para validar estas informações. Se alguém adultera estes dados – alguns spammers de comentários tentarão – a submissão falhará.

Se você estiver renderizando um formulário de comentário customizado, você precisará garantir a passagem destes valores sem mudanças.

- O timestamp é usado para assegurar que “ataques de resposta” não durem muito tempo. Usuário que esperam muito entre a requisição do formulário e a postagem do comentário terão suas submissões recusadas.
- O formulário de comentário inclui um campo “honeypot” (pote de mel). Isto é uma armadilha: se algum dado é colocado nesse campo, o comentário será considerado spam (spammers frequentemente preenchem todos os campos automaticamente, na tentativa de fazer submissões válidas).

O formulário padrão esconde este campo utilizando um CSS e coloca nele um label adicional com um aviso de campo; Se você usar o formulário de comentário com um template customizado, deve assegurar-se de fazer o mesmo.

Mais informações

Os models embutidos do Comment

class Comment

O model embutido de comentário do Django. Tem os seguintes campos:

content_object

Um atributo `GenericForeignKey` apontando para o objeto ao qual o comment está atachado. Você pode usar isto para obter o objeto relacionado (i.e. `my_comment.content_object`).

Uma vez que este campo é um `GenericForeignKey`, é realmente fácil entender sobre os dois atributos subjacentes, descritos abaixo.

content_type

Uma `ForeignKey` para `ContentType`; este é o tipo de objeto ao qual o coment é atachado.

object_pk

Um `TextField` contendo a chave primária do objeto ao qual o comment está atachado.

site

Uma `ForeignKey` para o `Site` em que o comentário foi postado.

user

Uma `ForeignKey` para o `User` que postou o comentário. Pode ser branco se o comentário foi postado por um usuário não autenticado.

user_name

O nome do usuário que postou o comentário.

user_email

O email do usuário que postou o comentário.

user_url

A URL entrada pela pessoa que postou o comentário.

comment

O conteúdo atual do comentário em si.

submit_date

A data em que o comentário foi postado.

ip_address

O endereço IP do usuário que postou o comentário.

is_public

False se o comentário não foi aprovado para exibição.

is_removed

True se o comentário foi aprovado. Usado para manter o traço de comentários removidos ao invés de somente deletá-los.

Configurações de Comentários

Estas configurações ajustam o comportamento do framework de comentários:

COMMENTS_HIDE_REMOVED

Se `True` (padrão), comentários removidos serão excluídos da lista/contagem de comentários (bem como das tags de template). Por outro lado o autor do template é responsável por alguma mensagem tipo “este comentário foi removido pelo site”.

COMMENT_MAX_LENGTH

O comprimento máximo do campo de comentário, em caracteres. Comentários mais longos serão rejeitados. O padrão é 3000.

COMMENTS_APP

A aplicação (estando inscrita no `INSTALLED_APPS`) responsável por toda “lógica de negócios.” Você pode mudar isto provendo models e formulários customizados, acredito que isso não esteja documentado.

Sinais enviados pela aplicação comments

A aplicação comment envia uma série de *sinais (signals)* para permitir a moderação de comentários em atividades semelhantes. Veja [a introdução aos sinais](#) para mais informações sobre como registrar e receber estes sinais.

comment_will_be_posted

`django.contrib.comments.signals.comment_will_be_posted`

Enviado somente depois que um comentário foi salvo, depois de checada sua sanidade e submetido. Este pode ser usado para modificar o comentário (in place) com detalhes da postagem, ou outras ações.

Se qualquer receptor retorna `False` o comentário será descartado e uma resposta 403 (not allowed) será retornada.

Este sinal é enviado mais ou menos ao mesmo tempo (pouco antes, na verdade) que o sinal `pre_save` do objeto `Comment`.

Argumentos enviados com este sinal:

sender O model comment.

comment A instância de comment que foi postada. Note que ela não deve ter sido salva no banco de dados ainda, então ela não tem uma chave primária, e quaisquer relações pode não funcionar corretamente ainda.

request O `HttpRequest` que foi postado no comentário.

comment_was_posted

`django.contrib.comments.signals.comment_was_posted`

Enviado logo após o comentário ser salvo.

Argumentos enviados com este sinal:

sender O model comment.

comment A instância de comment que foi postada. Note que ela já foi salva, então se você modificá-la, precisará chamar o método `save()` novamente.

request O `HttpRequest` que foi postado no comentário.

comment_was_flagged

`django.contrib.comments.signals.comment_was_flagged`

Enviado após um comentário ter sido “flagged” de alguma forma. Checa o flag para ver se isto foi uma requisição de remoção de um comentário feito por um usuário, um moderador aprovando/removendo um comentário, ou algum outro flag de usuário customizado.

Argumentos enviados com este sinal:

sender O model comment.

comment A instância do comentário que foi postada. Note que ela já está salva, então se você modificá-la, precisará chamar o método `save()` novamente.

flag O `CommentFlag` que foi atachado ao comentário.

created True se este é um novo flag; False se este é um flag duplicado.

request O `HttpRequest` que foi postado no comentário.

Atualizando o sistema de comentários de um Django anterior

Versões anteriores do Django incluem um atrasado, não documentado sistema de comentários. Usuários que usaram engenharia reversa nesse framework precisarão atualizá-lo para usar o novo sistema; este guia explica como.

As principais mudanças são:

- Este novo sistema é documentado.
- Ele usa funcionalidades modernas do Django como o *forms* e *modelforms*.
- Ele tem um único model `Comment` ao invés de ter separados os modelos `FreeComment` e `Comment`.
- Comentários têm campos “email” e “URL”.
- Sem ratings, fotos e karma. Isso só deve afetar o Mundo Online.
- A tag `{% comment_form %}` não existe mais. Em seu lugar, agora tem duas novas funções: `{% get_comment_form %}`, que retorna um formulário para postar um novo comentário, e `{% render_comment_form %}`, que renderiza um dado formulário utilizando o template `comments/form.html`.
- A forma que os comentários são incluídos na URLconf mudaram; você vai precisar substituir:

```
(r'^comments/', include('django.contrib.comments.urls.comments')),
```

com:

```
(r'^comments/', include('django.contrib.comments.urls')),
```

Atualizando dados

Os modelos do sistema de comentários do Django mudou, bem como os nomes das tabelas. Antes de você transferir seus dados para o novo sistema de comentários, esteja certo de que instalou o novo sistema de comentários como explicado em *guia rápido de início*.

Para transferir seus dados para o novo sistema de comentários, você precisará rodar diretamente o seguinte SQL:

```
BEGIN;

INSERT INTO django_comments
    (content_type_id, object_pk, site_id, user_name, user_email, user_url,
     comment, submit_date, ip_address, is_public, is_removed)
SELECT
    content_type_id, object_id, site_id, person_name, '', '', comment,
    submit_date, ip_address, is_public, not approved
FROM comments_freecomment;

INSERT INTO django_comments
    (content_type_id, object_pk, site_id, user_id, user_name, user_email,
     user_url, comment, submit_date, ip_address, is_public, is_removed)
SELECT
    content_type_id, object_id, site_id, user_id, '', '', '', comment,
    submit_date, ip_address, is_public, is_removed
FROM comments_comment;

UPDATE django_comments SET user_name = (
    SELECT username FROM auth_user
    WHERE django_comments.user_id = auth_user.id
) WHERE django_comments.user_id is not NULL;
UPDATE django_comments SET user_email = (
    SELECT email FROM auth_user
    WHERE django_comments.user_id = auth_user.id
) WHERE django_comments.user_id is not NULL;

COMMIT;
```

O Framework contenttypes

O Django inclui uma aplicação `contenttypes` que pode mapear todos os modelos instalados no seu projeto feito com o Django, provendo uma interface genérica de auto-nível para trabalhar com seus models.

Visão geral

O coração da aplicação `contenttypes` é o model `ContentType`, que reside em `django.contrib.contenttypes.models.ContentType`. As instâncias de `ContentType` representam e armazenam informações sobre os modelos instalados no seu projeto, e novas instâncias de `ContentType` são automaticamente criadas quando um novos models são instalados.

As instâncias do `ContentType` possuem métodos para retornar as classes de models que elas representam e para consultas de objetos para estes models. O `ContentType` também tem um *gerenciador customizado* que adiciona métodos par trabalhar com `ContentType` e para obtenção de instâncias de `ContentType` para um model em particular.

Os relacionamentos entre seus models e `ContentType` pode também ser usados para habilitar relacionamentos “genericos” entre uma instância de um de seus models e instâncias de qualquer model que você tenha instalado.

Instalando o framework contenttypes

O framework `contenttypes` vem incluído por padrão no `INSTALLED_APPS` quando é criado pelo `django-admin.py startproject`, mas se você o tiver removido ou se você manualmente setou seu `INSTALLED_APPS`, você pode habilitá-lo adicionando `'django.contrib.contenttypes'` no `INSTALLED_APPS`.

Geralmente é uma boa idéia ter o framework `contenttypes` instalado; várias outras aplicações nativas do Django necessitam dele:

- A aplicação `admin` o usa para logar a história de cada objeto adicionado ou modificado através da interface de administração.
- O *framework de autenticação* do Django o usa para amarrar as permissões de usuários para modelos específicos.
- O sistema de comentários do Django (`django.contrib.comments`) o usa para “atachar” comentário em qualquer model instalado.

O model ContentType

class `models.ContentType`

Cada instância do `ContentType` possui três campos, que juntos, descrevem a unicidade de um model instalado:

app_label

O nome da aplicação da qual o model faz parte. Este é o atributo `app_label` do model, e inclui somente a *última* parte do caminho de import do Python da aplicação; “`django.contrib.contenttypes`”, por exemplo, tem um `app_label` “`contenttypes`”.

model

O nome da classe do model.

name

O nome, legível por humanos, do model. Este é o atributo `verbose_name` do model.

Vamos olhar um exemplo, para entender como isso funciona. Se você já tem a aplicação `contenttypes` instalada, e adicionou a aplicação `sites` no `INSTALLED_APPS` e executou o `manage.py syncdb` para instalá-lo, o model `django.contrib.sites.models.Site` será instalado dentro do seu banco de dados. Imediatamente uma nova instância de `ContentType` será criada com os seguintes valores:

- `app_label` será setado como `'sites'` (a última parte do caminho Python “`django.contrib.sites`”).
- `model` será setada como `'site'`.
- `name` será setada como `'site'`.

Métodos das instâncias do ContentType

class `models.ContentType`

Cada instância `ContentType` possui métodos que permitem você pegar através da instância do `ContentType` o model que ela representa, ou receber objetos daquele model:

`models.ContentType.get_object_for_this_type(**kwargs)`

Pega um conjunto de *argumentos aparentes* válidos para o model que `ContentType` representa, e faz *um get()* *lookup* sobre este model, retornando o objeto correspondente.

`models.ContentType.model_class()`

Retorna a classe de model representada por esta instância de `ContentType`.

Por exemplo, nós podemos procurar no `ContentType` pelo model `User`:

```
>>> from django.contrib.contenttypes.models import ContentType
>>> user_type = ContentType.objects.get(app_label="auth", model="user")
>>> user_type
<ContentType: user>
```

E então usá-lo para pesquisar por um `User` particular, ou para ter acesso a classe model do `User`:

```
>>> user_type.model_class()
<class 'django.contrib.auth.models.User'>
>>> user_type.get_object_for_this_type(username='Guido')
<User: Guido>
```

Juntos, `get_object_for_this_type()` e `model_class()` habilitam dois casos extremamente importantes:

1. Usando estes métodos, você pode escrever código genérico de auto-nível que faz consultas sobre qualquer model instalado – ao invés de importar e usar uma classe de model específica, você passar um `app_label` e `model` dentro de um `ContentType` em tempo de execução, e então trabalhar com a classe model ou receber objetos dela.
2. Você pode relacionar outro model com `ContentType` como uma forma de vinculá-lo a determinadas classes de model, e usar estes métodos para acessar essas classes.

Várias aplicações nativas do Django fazem uso desta última técnica. Por exemplo, o sistema de permissões `<django.contrib.auth.models.Permission>` no framework de autenticação usa um model `Permission` com uma chave estrangeira para `ContentType`; isso permite que o `Permission` represente conceitos como “pode adicionar entrada no blog” ou “pode apagar notícia”.

O ContentTypeManager

`class models.ContentTypeManager`

O `ContentType` também tem um gerenciador personalizado, `ContentTypeManager`, que adiciona os seguintes métodos:

`clear_cache()`

Limpa um cache interno usado por instâncias do `ContentType`. Você provavelmente nunca precisará chamar este método você mesmo; O Django o chamará automaticamente quando for necessário.

`get_for_model(model)`

Pega ambos, um model ou uma instância de um model, e retorna a instância `ContentType` representando aquele model.

O método `get_for_model()` é especialmente usual quando você sabe que precisa trabalhar com um `ContentType` mas não quer ter o trabalho de obter os metadados do model para executar um lookup manual:

```
>>> from django.contrib.auth.models import User
>>> user_type = ContentType.objects.get_for_model(User)
>>> user_type
<ContentType: user>
```

Relações genéricas

Adicionando uma chave estrangeira em um de seus models para `ContentType` permite seu model vincular-se efetivamente em outro model, como no exemplo da classe `Permission` acima. Mas isto é possível ir além e usar `ContentType` para habilitar um relacionamento verdadeiramente genérico (algumas vezes chamado “polimórfico”) entre os models.

Um simples exemplo é um sistema de tags, que pode parecer com isto:

```

from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes import generic

class TaggedItem(models.Model):
    tag = models.SlugField()
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericForeignKey('content_type', 'object_id')

    def __unicode__(self):
        return self.tag

```

Uma `ForeignKey` normal pode somente “apontar para” um outro model, que significa que se o model `TaggedItem` usou uma `ForeignKey` ele teria de escolher somente um model para armazenar as tags. A aplicação `contenttypes` provê um tipo de campo especial – `django.contrib.contenttypes.generic.GenericForeignKey` – que funciona contornando isto e permitindo o relacionamento ser feito com qualquer model. Há três configurações para uma `GenericForeignKey`:

1. Dê a seu model uma `ForeignKey` para `ContentType`.
2. Dê a seu model um campo que consiga armazenar um valor de chave primária dos models que você irá relacionar. (Para maioria dos models, isto significa um `IntegerField` ou `PositiveIntegerField`.)

Este campo deve ser do mesmo tipo da chave primária dos models envolvidos no relacionamento genérico. Por exemplo, se você usa `IntegerField`, você não será capaz de formar uma relação genérica com um model que utiliza um `CharField` como uma chave primária.
3. Dê a seu model uma `GenericForeignKey`, e passe os nomes dos campos descritos acima para ela. Se estes campos são nomeados “`content_type`” e “`object_id`”, você pode omitir isto – estes são nomes padrão dos campos que `GenericForeignKey` irá procurar.

Isto habilitará uma API similar a que é usada por uma `ForeignKey` normal; cada `TaggedItem` terá um campo `content_type` que retorna o objeto ao qual está relacionado, e você pode também atribuir ao campo ou usá-lo quando estiver criando uma `TaggedItem`:

```

>>> from django.contrib.auth.models import User
>>> guido = User.objects.get(username='Guido')
>>> t = TaggedItem(content_object=guido, tag='bdf1')
>>> t.save()
>>> t.content_object
<User: Guido>

```

Devido a forma como o `GenericForeignKey` é implementado, você não pode usar campos estes campos diretamente com filtros (`filter()` e `exclude()`, por exemplo) via API de banco de dados. Eles não são campos normal de objetos. Estes exemplos *não* irão funcionar:

```

# Isto falhará
>>> TaggedItem.objects.filter(content_object=guido)
# Isto também falhará
>>> TaggedItem.objects.get(content_object=guido)

```

Reverso de relações genéricas

Se você sabe quais models serão usados com mais frequência, você pode também adicionar um “reverso” de relações genéricas para habilitar uma API adicional. Por exemplo:

```

class Bookmark(models.Model):
    url = models.URLField()
    tags = generic.GenericRelation(TaggedItem)

```

Cada instância `Bookmark` terá um atributo `tags`, que pode ser usado para receber seus `TaggedItems` associados:

```
>>> b = Bookmark(url='http://www.djangoproject.com/')
>>> b.save()
>>> t1 = TaggedItem(content_object=b, tag='django')
>>> t1.save()
>>> t2 = TaggedItem(content_object=b, tag='python')
>>> t2.save()
>>> b.tags.all()
[<TaggedItem: django>, <TaggedItem: python>]
```

Assim como `django.contrib.contenttypes.generic.GenericForeignKey` aceita os nomes dos campos `content-type` e `object-ID` como argumentos, para construir uma `GenericRelation`; se o model que possui a chave estrangeira genérica está usando nomes diferentes do padrão, para estes campos, você deve passar os nomes dos campos quando estiver configurando uma `GenericRelation`. Por exemplo, se o model `TaggedItem` referido acima usasse campos com nomes `content_type_fk` e `object_primary_key` para criar suas chaves primárias genéricas, então uma `GenericRelation` teria de ser definida como:

```
tags = generic.GenericRelation(TaggedItem, content_type_field='content_type_fk',
↪object_id_field='object_primary_key')
```

É claro, Se você não adicionar o reverso de relacionamento, você pode fazer os mesmos tipo de busca manualmente:

```
>>> b = Bookmark.objects.get(url='http://www.djangoproject.com/')
>>> bookmark_type = ContentType.objects.get_for_model(b)
>>> TaggedItem.objects.filter(content_type__pk=bookmark_type.id,
...                           object_id=b.id)
[<TaggedItem: django>, <TaggedItem: python>]
```

Note que se o model com uma `GenericForeignKey` que você está referenciando usa um valor diferente do padrão para `ct_field` ou `fk_field` (e.g. a aplicação `django.contrib.comments` usa `ct_field="object_pk"`), você precisará passar `content_type_field` e `object_id_field` para `GenericRelation`:

```
comments = generic.GenericRelation(Comment, content_type_field="content_type",
↪object_id_field="object_pk")
```

Note que se você deleta um objeto que tem uma `GenericRelation`, quaisquer objetos que possuem uma `GenericForeignKey` apontando para ele, serão deletados também. No exemplo acima, isto significa que se um objeto `Bookmark` foi deletado, qualquer objeto `TaggedItem` relacionado com ele, será deletado ao mesmo tempo.

Relacionamentos genéricos em formulários e admin

O `django.contrib.contenttypes.generic` provê ambos, `GenericInlineFormSet` e `GenericInlineModelAdmin`. Isso permite o uso de relacionamentos genéricos nos formulários e no admin. Veja a documentação do *model formset* e *admin* para mais informações.

class generic.GenericInlineModelAdmin

A classe `GenericInlineModelAdmin` herda todas as propriedades da classe `InlineModelAdmin`. No entanto, ela adiciona alguns próprios para funcionar com relacionamento genérico.

ct_field

O nome do campo chave estrangeira `ContentType` para o model. O padrão é `content_type`.

ct_fk_field

O nome do campo inteiro que representa o ID do objeto relacionado. O padrão é `object_id`.

Proteção Cross Site Request Forgery

A classe `CsrfMiddleware` provê uma proteção fácil de usar contra [Requisições Cross Site falsas](#). Esse tipo de ataque ocorre quando um website malicioso cria um link ou um botão de formulário que é destinado a executar alguma ação sobre seu site, usando credenciais de um usuário logado que pode ser enganado ao clicar em um link no seu navegador.

A primeira defesa contra ataques CSRF é assegurar que requisições GET são livres de efeitos colaterais. Requisições POST podem então ser protegidas por este middleware, adicionando-o em sua lista de middlewares instalados.

Como usá-lo

Adicione o middleware `'django.contrib.csrf.middleware.CsrfMiddleware'` em sua lista de classes middleware, [MIDDLEWARE_CLASSES](#). Ele necessita processar a resposta depois do `SessionMiddleware`, portanto deve vir antes dele na lista. Ele também deve processar a resposta antes de coisas como compressão, que ocorrerá com a resposta, logo deve vir depois do `GZipMiddleware` na lista.

Como ele funciona

`CsrfMiddleware` faz duas coisas:

1. Modifica requisições de saída adicionando um campo hidden para todo formulário ‘POST’, com o nome ‘`csrfmiddlewaretoken`’ e um valor que é um hash composto pelo ID da sessão mais um secret. Se não houver um ID de sessão configurado, esta modificação da resposta não será feita. Assim há muito pouca perda de performance para essas requisições que não possuem uma sessão.
2. Em todas as requisições POST de entrada que tiverem um cookie de sessão configurado, ele checa se o ‘`csrfmiddlewaretoken`’ está presente e correto. Se não estiver, o usuário receberá um erro 403.

Isso assegura que somente formulários originários de seu website podem ser usados para enviar dados de volta.

Deliberadamente somente são alvos de requisições os POSTS HTTP (e os formulários POST correspondentes). Requisições GET não costumam representar um perigo em potencial (veja [9.1.1 Métodos Seguros, HTTP 1.1, RFC 2616](#)), logo um ataque CSRF com uma requisição GET costuma ser inofensivo.

Requisições POST que não são acompanhadas por um cookie de sessão não estão protegidas, mas elas não precisam ser protegidas, uma vez que o website agressor poderia fazer este tipo de requisição de qualquer maneira.

O Content-Type é checado antes de modificar a resposta, e somente páginas que são servidas como ‘`text/html`’ ou ‘`application/xml+xhtml`’ são modificadas.

Limitações

O `CsrfMiddleware` requer o framework de sessões do Django para funcionar. Se você tem um sistema customizado de autenticação que manualmente configura os cookies, ele não irá ajudá-lo.

Se sua aplicação cria páginas HTML e formulários de alguma forma incomum, (ex: ela envia fragmentos de páginas HTML em chamadas `document.write` de JavaScript), você pode pular o filtro que adiciona o campo hidden no formulário, caso este no qual a submissão do formulário irá sempre falhar. Pode ser ainda possível usar o middleware, desde que você consiga encontrar uma maneira de obter o token CSRF e assegurar-se de que ele está incluído quando os seus formulários forem enviados.

Databrowse

Databrowse é uma aplicação Django que permite você navegar em seus dados.

Assim como a administração do Django cria dinamicamente uma interface espionando os seus modelos, o Databrowse cria dinamicamente um rico e navegável website espionando-os também.

Note

O Databrowse é **muito** novo e está em fase de desenvolvimento. Ele pode mudar substancialmente antes do próximo release do Django.

Como foi dito, ele é fácil de usar e não requer qualquer código novo. Então você pode brincar com ele hoje, com um pouco de investimento de tempo ou código.

Como usar o Databrowse

1. Aponte o Django para os templates padrões do “Databrowse”. Existem duas maneiras de fazer isso:

- Adicione 'django.contrib.databrowse' em seu `INSTALLED_APPS`. Isso funcionará se seu `TEMPLATE_LOADERS` incluir o carregador de templates `app_directories` (que é o caso por padrão). Veja a documentação do *carregor de templates* para saber mais.
- De outra forma, determine o caminho completo no sistema de arquivos para o diretório `django/contrib/databrowse/templates`, e o adicione no `TEMPLATES_DIRS` do arquivo `settings.py`.

2. Registrar um número de modelos com o site Databrowse:

```
from django.contrib import databrowse
from myapp.models import SomeModel, SomeOtherModel

databrowse.site.register(SomeModel)
databrowse.site.register(SomeOtherModel)
```

Note que você pode registrar a *classe* do modelo, não instâncias.

Não importa onde você coloque isto, desde que seja executado em algum ponto. Um bom lugar para isso é seu arquivo *URLconf* (`urls.py`).

3. Mude seu URLconf para importar o módulo *databrowse*:

```
from django.contrib import databrowse
```

...e adicione a seguinte linha em seu URLconf:

```
(r'^databrowse/(.*)', databrowse.site.root),
```

O prefixo não importa – você pode usar `databrowse/` ou `db/` ou o que preferir.

4. Rode o servidor do Django e visite `/databrowse/` em seu navegador.

Requerindo login de usuário

Você pode restringir o acesso para usuário logados com poucas linhas de código extra. Simplesmente adicione o seguinte import em seu URLconf:

```
from django.contrib.auth.decorators import login_required
```

Em seguida modifique o *URLconf* a fim de que o view `databrowse.site.root()` seja afetado pelo decorador `django.contrib.auth.decorators.login_required()`:

```
(r'^databrowse/(.*)', login_required(databrowse.site.root)),
```

Se você ainda não adicionou suporte para logins de usuários à sua *URLconf*, como descrito na documentação de *autenticação de usuário*, então você precisará fazer isso agora com o seguinte mapeamento:

```
(r'^accounts/login/$', 'django.contrib.auth.views.login'),
```

O passo final é criar o formulário de login requerido pelo `django.contrib.auth.views.login()`. A documentação de *autenticação de usuário* provê os detalhes completos e um template de exemplo que pode ser usado com este propósito.

A aplicação Flatpages

Django traz embutido uma aplicação opcional “flatpages”. Ela permite você armazenar conteúdo HTML em um banco de dados e toma conta do gerenciamento para você por meio administração do Django e uma API do Python.

Uma flatpage é um simples objeto com uma URL, título e conteúdo. Utilize-a para páginas únicas, customizadas, como “Quem somos” ou “Política de Privacidade”, que você quer armazenar no banco de dados, porém não quer desenvolver uma aplicação Django exclusiva para isso.

Uma flatpage pode tanto utilizar um template customizado como também um template padrão para todas as flatpages do sistema. Pode estar associada com um ou vários sites. *Please, see the release notes* O campo conteúdo pode opcionalmente ser deixado em branco caso você prefira colocá-lo em um template customizado.

Alguns exemplos de flatpages em sites desenvolvidos com Django:

- <http://www.chicagocrime.org/about/>
- <http://www.everyblock.com/about/>
- <http://www.lawrence.com/about/contact/>

Instalação

Para instalar a aplicação flatpages, siga os passos abaixo:

1. Instale o *framework sites* adicionando `'django.contrib.sites'` em seu *INSTALLED_APPS*, se ele ainda não estiver lá.

Also make sure you’ve correctly set *SITE_ID* to the ID of the site the settings file represents. This will usually be 1 (i.e. `SITE_ID = 1`, but if you’re not using the sites framework to manage multiple sites, it could be the ID of a different site.
2. Adicione `'django.contrib.flatpages'` em seu *INSTALLED_APPS*.
3. Adicione `'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'` em seu *MIDDLEWARE_CLASSES*.
4. Execute o comando `manage.py syncdb`.

Como funciona

`manage.py syncdb` cria duas tabelas em seu banco de dados: `django_flatpage` e `django_flatpage_sites`. `django_flatpage` é uma tabela comum que simplesmente mapeia a URL para um título e um punhado de texto. `django_flatpage_sites` associa uma flatpage a um site.

O `FlatpageFallbackMiddleware` faz todo o trabalho. Toda vez que qualquer aplicação Django chamar um erro 404, este middleware verifica se existe alguma flatpage no banco de dados mapeada para a URL requisitada como um último recurso. Mais especificamente, ele procura por uma flatpage com a URL requisitada e o ID do site que corresponda com a variável *SITE_ID* no arquivo *settings.py*.

Se encontrar uma combinação, ele segue o seguinte algoritmo:

- Se a flatpage tiver um template customizado, ele carrega esse template. Caso contrário carrega o template `flatpages/default.html`.
- Ele passa ao template uma única variável de contexto, `flatpage`, que é um objeto do tipo `flatpage`. Ele utiliza a classe `RequestContext` na renderização do template.

Se não encontrar uma combinação, a requisição continua a ser processada normalmente.

O middleware só é ativado para erros 404 – não para erros 500 ou respostas de qualquer outro código de status.

Note que a ordem de `MIDDLEWARE_CLASSES` importa. Geralmente você pode incluir a classe `FlatpageFallbackMiddleware` no fim da lista, pois ele é um último recurso.

Para saber mais sobre middleware, leia a [documentação do middleware](#).

Tenha certeza de que seu template de erro 404 está funcionando.

Note que a classe `FlatpageFallbackMiddleware` só é chamado caso uma view tenha sucesso ao produzir um erro 404. Se outra view ou classe middleware tentar produzir um erro 404, porém não conseguir e terminar lançando uma exceção (como por exemplo `TemplateDoesNotExist`, caso seu site não possua um template apropriado para respostas HTTP 404), a resposta se tornará um erro HTTP 500 (“Erro Interno de Servidor”) e o `FlatpageFallbackMiddleware` não tentará servir uma flatpage.

Como adicionar, alterar e excluir flatpages

Pela interface de administração

Se você ativou a administração automática do Django, você deverá ver uma seção “Flatpages” na página inicial da administração. Altere as flatpages como você alteraria qualquer outro objeto no sistema.

Usando a API do Python

`class models.FlatPage`

Flatpages são representadas por um *modelo Django*, que fica armazenado em `django/contrib/flatpages/models.py`. Você pode acessar objetos do tipo flatpage pela *API de banco de dados do Django*.

Templates das Flatpages

Por padrão, flatpages são renderizadas pelo template `flatpages/default.html`, mas você pode sobrescrevê-lo por uma flatpage particular.

Criar o template `flatpages/default.html` é sua responsabilidade; em seu diretório de templates, crie uma pasta chamada `flatpages` contendo um arquivo chamado `default.html`.

Para os templates das flatpages é passado apenas uma variável de contexto, `flatpage`, que é o objeto flatpage.

Segue abaixo um exemplo de template para `flatpages/default.html`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/REC-html40/loose.dtd">
<html>
<head>
<title>{{ flatpage.title }}</title>
</head>
<body>
{{ flatpage.content }}
</body>
</html>
```

Uma vez que você já está inserindo código HTML puro na administração para uma flatpage, tanto `flatpage.title` como `flatpage.content` **não** necessitam de *escaping automático de HTML* no template.

django.contrib.formtools

A set of high-level abstractions for Django forms (`django.forms`). Um conjunto de abstrações de auto-nível para os formulários do Django (`django.forms`).

Pré-visualização de formulários

O Django vem acompanhado com uma aplicação opcional, “form preview”, que ajuda a automatizar o seguinte workflow:

“Mostrar um formulário HTML, forçar uma pré-visualização, somente então fazer algo com o enviado.”

Para forçar uma pré-visualização de um envio de formulário, todos devem escrever uma pequena classe Python.

Visão Geral

Dada uma subclasse de `django.forms.Form` que você define, esta aplicação se preocupa com o seguinte fluxo:

1. Mostrar o formulário como uma página HTML.
2. Validar os dados do formulário quando este é enviado via POST. a. Se for válido, mostra uma pré-visualização da página. b. Se não for válido, mostra novamente o formulário com as mensagens de erro.
3. Quando a “confirmação” do formulário é submetida da página de previsão, chame um hook que você define – um método `done()` que captura os dados válidos passados.

O framework fiscaliza a pré-visualização requisitada passando um hash de segredo compartilhado para a página de pré-visualização, por meio de campos hidden. Se alguém ajusta os parâmetros do formulário na página de pré-visualização, o envio do formulário pode falhar no momento da comparação com o hash.

Como usar o FormPreview

1. Aponte o Django para os templates padrão do FormPreview. Existem duas maneiras de fazer isso:
 - Adicione `'django.contrib.formtools.preview'` em seu `INSTALLED_APPS`. Isso funcionará se o seu `TEMPLATE_LOADERS` incluir o carregador de templates (que vem por padrão) `app_directories`. Veja a *documentação do carregador de templates* para saber mais.
 - De outra maneira, determine o caminho completo no sistema de arquivos para o diretório `django/contrib/formtools/templates` e adicione-o no `TEMPLATE_DIRS` no arquivo `settings.py`.
2. Crie uma subclasse `FormPreview` que sobrescreva o método `done()`:

```
from django.contrib.formtools.preview import FormPreview
from myapp.models import SomeModel

class SomeModelFormPreview(FormPreview):

    def done(self, request, cleaned_data):
        # Faça algo com o cleaned_data, e depois redirecione
        # para uma página de "sucesso".
        return HttpResponseRedirect('/form/success')
```

Este método recebe um objeto `HttpRequest` e um dicionário de dados de formulário, depois de terem sido validados e limpos. Podendo retornar um `HttpResponseRedirect`, que é um resultado final de um formulário que foi enviado.

3. Mude seu `URLconf` para apontar para uma instância de sua subclasse `FormPreview`

```
from myapp.preview import SomeModelFormPreview from myapp.models import SomeModel
from django import forms
```

...e adicione a seguinte linha para o modelo apropriado em seu `URLconf`:

```
(r'^post/$', SomeModelFormPreview(SomeModelForm)),
```

onde `SomeModelForm` é uma classe `Form` ou `ModelForm`.

4. Rode o servidor do Django e visite `/post/` no seu navegador.

Classes `FormPreview`

Uma classe `FormPreview` é uma simples classe Python que representa o fluxo da pré-visualização. As classes `FormPreview` devem ser subclasses de `django.contrib.formtools.preview.FormPreview` e sobrescrever o método `done()`. Elas podem ficar em qualquer parte da base de código.

Templates `FormPreview`

Por padrão, o formulário é renderizado por meio do template `formtools/form.html`, e a página de pré-visualização é renderizada por meio do template `formtools.preview.html`. Esses valores podem ser sobrescritos por um formulário de pré-visualização configurando os atributos `preview_template` e `form_template` na subclasse `FormPreview`. Veja `django/contrib/formtools/templates` para conhecer os templates padrões.

Form wizard

Please, see the release notes O Django vem acompanhado uma aplicação opcional “form wizard” que divide *formulários* em várias páginas Web. Ele mantém o estado num hash HTML de campos `<input type="hidden">`, e os dados não são processados no lado do servidor até que a última parte do formulário seja submetida.

Você pode querer usar isto com formulários muito compridos que seriam desagradáveis de se mostrar numa única página. A primeira página pode pedir ao usuário informações centrais, a segunda coisas menos importantes, e assim por diante.

o termo “wizard” (em português “assistente”) neste contexto, é [explicado na Wikipedia](#).

Como ele funciona

Aqui temos o fluxo de funcionamento básico de como um usuário poderia usar um wizard:

1. O usuário visita a primeira página do wizard, preenche o formulário e o submete.
2. O servidor valida os dados. Se eles forem inválidos, o formulário é mostrado novamente, com mensagens de erro. Se for válido, o servidor calcula um hash seguro dos dados e apresenta ao usuário o próximo formulário, poupando os dados validados e o hash em campos `<input type="hidden">`.
3. Repete o passo 1 e 2, para todos os formulários subsequentes do wizard.
4. Um vez que o usuário tenha submetido todos os formulários e todos os dados estejam validados, o wizard processa os dados – salvando-os no banco de dados, enviando um e-mail, ou seja lá o que for que a aplicação precise fazer.

Uso

Esta aplicação manipula tantos mecanismos para você quanto possível. Geralmente, você só tem que fazer estas coisas:

1. Definir um número de classes `django.forms` e `Form` – uma por página do wizard.
2. Criar uma classe `FormWizard` que especifica o que fazer uma vez que todos os seus formulários foram submetidos e validados. Isso também permite você sobrescrever alguns comportamentos do wizard.
3. Criar alguns templates que renderizam formulários. Você pode definir um único template genérico para manipular todos os formulários, ou definir um template específico para cada formulário.
4. Apontar seu `URLconf` para sua classe `FormWizard`.

Definindo classes `Form`

o primeiro passo para criar um “form wizard” é criar as classes `Form`. Estes deveriam ser padronizados com as classes `django.forms` e `Form`, explanados na [documentação do forms](#).

Estas classes podem estar em qualquer lugar de sua base de código, mas por convenção são postas num arquivo chamado `forms.py` dentro da sua aplicação.

Por exemplo, vamos escrever um wizard “contact form”, onde a primeira página coleta o endereço de email do remetente e o assunto, e a segunda página coleta a mensagem em si. Aqui tem um exemplo de como o `forms.py` pode ficar:

```
from django import forms

class ContactForm1(forms.Form):
    subject = forms.CharField(max_length=100)
    sender = forms.EmailField()

class ContactForm2(forms.Form):
    message = forms.CharField(widget=forms.Textarea)
```

Limitação importante: Pelo wizard usar campos HTML hidden para armazenar dados entre as páginas, você não pode incluir um `FileField` em qualquer outra página, a não ser na última.

Criando uma classe `FormWizard`

O próximo passo é criar uma classe `FormWizard`, que deveria ser uma subclasse do `django.contrib.formtools.wizard.FormWizard`.

Assim como suas classes `Form`, esta classe `FormWizard` pode estar em qualquer lugar da sua base de código, mas por convenção é colocado no `forms.py`.

O único requerimento desta subclasse é que ela implementa um método `done()`, que especifica o que deve acontecer quando *todos* os formulários foram submetidos e validados. A este método é passado dois argumentos:

- `request` – um objeto `HttpRequest`
- `form_list` – uma lista de classes `django.forms Form`

Neste exemplo simplista, ao invés de executar qualquer operação de banco de dados, o método simplesmente renderiza um template com os dados validados:

```
from django.shortcuts import render_to_response
from django.contrib.formtools.wizard import FormWizard

class ContactWizard(FormWizard):
    def done(self, request, form_list):
        return render_to_response('done.html', {
```

```
'form_data': [form.cleaned_data for form in form_list],
})
```

Note que este método será chamado via POST, então ele realmente deveria ser um bom cidadão Web e redirecionar depois de processar os dados. Aqui tem outro exemplo:

```
from django.http import HttpResponseRedirect
from django.contrib.formtools.wizard import FormWizard

class ContactWizard(FormWizard):
    def done(self, request, form_list):
        do_something_with_the_form_data(form_list)
        return HttpResponseRedirect('/page-to-redirect-to-when-done/')

```

Veja a seção *Métodos avançados do FormWizard* abaixo para aprender mais sobre os hooks do *FormWizard*.

Criando templates para formulários

Agora, precisamos criar um template que renderize os formulários do wizard. Por padrão, todo formulário usa um template chamado `forms/wizard.html`. (Você pode mudar este nome sobrescrevendo o método `get_template()`, que está documentado abaixo. Este hook também permite você usar um template diferente para cada formulário.)

Este template aguarda o seguinte contexto:

- `step_field` – O nome do campo hidden contendo o passo.
- `step0` – O passo atual (partindo de zero).
- `step` – O passo atual (partindo de um).
- `step_count` – O total de passos.
- `form` – A instância do Form para o passo atual (vazio ou com erros).
- `previous_fields` – Uma string representando todos os campos de dados anteriores, mais os hashes de formulários completados, todos em campos hidden de formulário. Note que você precisará executar isto através do template filter `safe()`, para prevenir o auto escape, por ele é um HTML puro.

Ele também será transmitido a quaisquer objetos no `extra_context`, que é um dicionário que você pode especificar contendo valores extra a serem adicionados ao contexto. Você pode especificá-lo de duas formas:

- Sete o atributo `extra_context` na sua subclasse *FormWizard* para um dicionário.
- Passe `extra_context` como parâmetros extra no URLconf.

Aqui tem um exemplo de template completo:

```
{% extends "base.html" %}

{% block content %}
<p>Step {{ step }} of {{ step_count }}</p>
<form action="." method="post">
<table>
{{ form }}
</table>
<input type="hidden" name="{{ step_field }}" value="{{ step0 }}" />
{{ previous_fields|safe }}
<input type="submit">
</form>
{% endblock %}
```

Note que `previous_fields`, `step_field` e `step0` são todos obrigatórios para o wizard funcionar perfeitamente.

Ligando o wizard num URLconf

Finalmente, dê ao seu novo objeto *FormWizard* uma URL no `urls.py`. O wizard recebe uma lista de seus objetos form como argumentos:

```
from django.conf.urls.defaults import *
from mysite.testapp.forms import ContactForm1, ContactForm2, ContactWizard

urlpatterns = patterns('',
    (r'^contact/$', ContactWizard([ContactForm1, ContactForm2])),
)
```

Métodos avançados do FormWizard

class FormWizard

Além do método `done()`, o *FormWizard* oferece alguns métodos avançados que permitem você customizar o funcionamento do seu wizard.

Alguns destes métodos recebem um argumento `step`, que é um contador, partindo de zero, representando o passo corrente do wizard. (E.g., o primeiro form é 0 e o segundo form é 1.)

FormWizard.prefix_for_step()

Dado um passo, retorna um prefixo de Form para uso. Por padrão, este simplesmente usa o passo em si. Para mais, veja a [documentação do prefixo de formulário](#). Implementação padrão:

```
def prefix_for_step(self, step):
    return str(step)
```

FormWizard.render_hash_failure()

Renderiza um template se a verificação do hash falhar. É raro você precisar sobrescrever este método.

Implementação padrão:

```
def render_hash_failure(self, request, step):
    return self.render(self.get_form(step), request, step,
        context={'wizard_error': 'Desculpe-nos, mas seu formulário expirou.↵
↵Por favor continue preenchendo o formulário desta página.'})
```

FormWizard.security_hash()

Calcula o hash de segurança para um dado objeto `request` e instância de `Form`.

Por padrão, ele usa um hash MD5 dos dados do formulário e sua configuração `SECRET_KEY`. É raro alguém precisar sobrescrever isso.

Exemplo:

```
def security_hash(self, request, form):
    return my_hash_function(request, form)
```

FormWizard.parse_params()

Um hook para salvar o estado do objeto `request` e `args / kwargs` que foram capturados da URL pelo seu URLconf.

Por padrão, isso não faz nada.

Exemplo:

```
def parse_params(self, request, *args, **kwargs):
    self.my_state = args[0]
```

FormWizard.get_template()

Retorna o nome do template que deveria ser usado para um certo passo.

Por padrão, ele retorna `'forms/wizard.html'`, indiferente do passo.

Exemplo:

```
def get_template(self, step):
    return 'myapp/wizard_%s.html' % step
```

Se o `get_template()` retorna uma lista de strings, então o wizard usará a função do sistema de template `select_template()`, *explicada na documentação de template*. Isto significa que o sistema usará o primeiro template que existir no sistema de arquivo. Por exemplo:

```
def get_template(self, step):
    return ['myapp/wizard_%s.html' % step, 'myapp/wizard.html']
```

`FormWizard.render_template()`

Renderiza o template para o dado passo, retornando um objeto `HttpResponse`.

Sobrescreva este método se você deseja adicionar um contexto customizado, retornar um tipo MIME diferente, etc. Se você somente precisa sobrescrever o nome do template, use o método `get_template()`.

O template será renderizado com o contexto documentado na seção acima “criando templates para formulários”.

`FormWizard.process_step()`

Um hook para modificar o estado interno do wizard, dado uma totalidade de objetos `Form` validados. O `Form` está garantido a ter dados limpos e válidos.

Este método *não* deve modificar qualquer dado. Ao invés disso, ele pode querer setar um `self.extra_context` ou dinamicamente alterar o `self.form_list`, baseado nos formulários submetidos anteriormente.

Note que este método é chamado toda vez que uma página é renderizada para *todos* os passos submetidos.

A assinatura da função:

```
def process_step(self, request, form, step):
    # ...
```

django.contrib.humanize

Um conjunto de filtros de templates do Django úteis para adicionar um “toque humano” aos dados.

Para ativar estes filtros, adicione `'django.contrib.humanize'` no seu `INSTALLED_APPS`. Uma vez que feito isso, use `{% load humanize %}` em um template, e você terá acesso a estes filtros:

apnumber

Para números 1-9, retorna o número como se fala. Do contrário, retorna o próprio número. Este segue o estilo Associated Press.

Exemplos:

- 1 torna-se 'one'.
- 2 torna-se 'two'.
- 10 torna-se 10.

Você pode passar um inteiro ou mesmo uma representação em string de um inteiro.

intcomma

Converte um inteiro em uma string contendo vírgulas a cada três dígitos.

Exemplos:

- 4500 torna-se '4,500'.
- 45000 torna-se '45,000'.
- 450000 torna-se '450,000'.
- 4500000 torna-se '4,500,000'.

Você pode passar um inteiro ou mesmo uma representação em string de um inteiro.

intword

Converte um inteiro grande em uma representação textual agradável. Funciona melhor com números acima de 1 milhão.

Exemplos:

- 1000000 torna-se '1.0 million'.
- 1200000 torna-se '1.2 million'.
- 1200000000 torna-se '1.2 billion'.

Valores acima de 1000000000000000 (um quadrilhão) são suportados.

Você pode passar um inteiro ou mesmo uma representação em string de um inteiro.

ordinal

Converte um inteiro em seu ordinal como uma string.

Exemplos:

- 1 torna-se '1st'.
- 2 torna-se '2nd'.
- 3 torna-se '3rd'.

Você pode passar um inteiro ou mesmo uma representação em string de um inteiro.

naturalday

Please, see the release notes Para datas que são o dia atual ou dentro de um dia, retorna “today”, “tomorrow” ou “yesterday”, como for apropriado. Do contrário, formata a data usando o formato passado.

Argumento: String de formato da data como descrito na tag *now*.

Exemplos (quando ‘today’ é 17 Feb 2007):

- 16 Feb 2007 torna-se yesterday.
- 17 Feb 2007 torna-se today.
- 18 Feb 2007 torna-se tomorrow.
- Any other day is formatted according to given argument or the *DATE_FORMAT* setting if no argument is given.
- Qualquer outro dia é formatado de acordo com o argumento fornecido ou pela configuração *DATE_FORMAT*, se não houver argumentos informados.

The “local flavor” add-ons

Following its “batteries included” philosophy, Django comes with assorted pieces of code that are useful for particular countries or cultures. These are called the “local flavor” add-ons and live in the `django.contrib.localflavor` package.

Inside that package, country- or culture-specific code is organized into subpackages, named using [ISO 3166 country codes](#).

Most of the `localflavor` add-ons are localized form components deriving from the `forms` framework – for example, a `USStateField` that knows how to validate U.S. state abbreviations, and a `FISocialSecurityNumber` that knows how to validate Finnish social security numbers.

To use one of these localized components, just import the relevant subpackage. For example, here’s how you can create a form with a field representing a French telephone number:

```
from django import forms
from django.contrib.localflavor.fr.forms import FRPhoneNumberField

class MyForm(forms.Form):
    my_french_phone_no = FRPhoneNumberField()
```

Supported countries

Countries currently supported by `localflavor` are:

- *Argentina*
- *Australia*
- *Austria*
- *Brazil*
- *Canada*
- *Chile*
- *Finland*
- *France*
- *Germany*
- *Iceland*
- *India*
- *Italy*
- *Japan*
- *Mexico*
- *The Netherlands*
- *Norway*
- *Peru*
- *Poland*
- *Romania*
- *Slovakia*
- *South Africa*
- *Spain*

- *Switzerland*
- *United Kingdom*
- *United States of America*

The `django.contrib.localflavor` package also includes a `generic` subpackage, containing useful code that is not specific to one particular country or culture. Currently, it defines date and datetime input fields based on those from *forms*, but with non-US default formats. Here’s an example of how to use them:

```
from django import forms
from django.contrib.localflavor import generic

class MyForm(forms.Form):
    my_date_field = generic.forms.DateField()
```

Adding flavors

We’d love to add more of these to Django, so please [create a ticket](#) with any code you’d like to contribute. One thing we ask is that you please use Unicode objects (`u'mystring'`) for strings, rather than setting the encoding in the file. See any of the existing flavors for examples.

Argentina (ar)

class `ar.forms.ARPostalCodeField`

A form field that validates input as either a classic four-digit Argentinian postal code or a [CPA](#).

class `ar.forms.ARDNIField`

A form field that validates input as a Documento Nacional de Identidad (DNI) number.

class `ar.forms.ARCUITField`

A form field that validates input as a Codigo Unico de Identificacion Tributaria (CUIT) number.

class `ar.forms.ARProvinceSelect`

A `Select` widget that uses a list of Argentina’s provinces and autonomous cities as its choices.

Australia (au)

class `au.forms.AUPostCodeField`

A form field that validates input as an Australian postcode.

class `au.forms.AUPhoneNumberField`

A form field that validates input as an Australian phone number. Valid numbers have ten digits.

class `au.forms.AUStateSelect`

A `Select` widget that uses a list of Australian states/territories as its choices.

Austria (at)

class `at.forms.ATZipCodeField`

A form field that validates its input as an Austrian zip code.

class `at.forms.ATStateSelect`

A `Select` widget that uses a list of Austrian states as its choices.

class `at.forms.ATSocialSecurityNumberField`

A form field that validates its input as an Austrian social security number.

Brazil (br)

class `br.forms.BRPhoneNumberField`

A form field that validates input as a Brazilian phone number, with the format XX-XXXX-XXXX.

class `br.forms.BRZipCodeField`

A form field that validates input as a Brazilian zip code, with the format XXXXX-XXX.

class `br.forms.BRStateSelect`

A `Select` widget that uses a list of Brazilian states/territories as its choices.

Canada (ca)

class `ca.forms.CAPhoneNumberField`

A form field that validates input as a Canadian phone number, with the format XXX-XXX-XXXX.

class `ca.forms.CAPostalCodeField`

A form field that validates input as a Canadian postal code, with the format XXX XXX.

class `ca.forms.CAProvinceField`

A form field that validates input as a Canadian province name or abbreviation.

class `ca.forms.CASocialInsuranceNumberField`

A form field that validates input as a Canadian Social Insurance Number (SIN). A valid number must have the format XXX-XXX-XXX and pass a [Luhn mod-10 checksum](#).

class `ca.forms.CAProvinceSelect`

A `Select` widget that uses a list of Canadian provinces and territories as its choices.

Chile (cl)

class `cl.forms.CLRutField`

A form field that validates input as a Chilean national identification number ('Rol Unico Tributario' or RUT). The valid format is XX.XXX.XXX-X.

class `cl.forms.CLRegionSelect`

A `Select` widget that uses a list of Chilean regions (Regiones) as its choices.

Finland (fi)

class `fi.forms.FISocialSecurityNumber`

A form field that validates input as a Finnish social security number.

class `fi.forms.FIZipCodeField`

A form field that validates input as a Finnish zip code. Valid codes consist of five digits.

class `fi.forms.FIMunicipalitySelect`

A `Select` widget that uses a list of Finnish municipalities as its choices.

France (fr)

class `fr.forms.FRPhoneNumberField`

A form field that validates input as a French local phone number. The correct format is 0X XX XX XX XX. 0X.XX.XX.XX.XX and 0XXXXXXXXX validate but are corrected to 0X XX XX XX XX.

class `fr.forms.FRZipCodeField`

A form field that validates input as a French zip code. Valid codes consist of five digits.

class `fr.forms.FRDepartmentSelect`

A `Select` widget that uses a list of French departments as its choices.

Germany (de)

class `de.forms.DEIdentityCardNumberField`

A form field that validates input as a German identity card number ([Personalausweis](#)). Valid numbers have the format XXXXXXXXXXX-XXXXXX-XXXXXX-X, with no group consisting entirely of zeroes.

class `de.forms.DEZipCodeField`

A form field that validates input as a German zip code. Valid codes consist of five digits.

class `de.forms.DEStateSelect`

A `Select` widget that uses a list of German states as its choices.

The Netherlands (nl)

class `nl.forms.NLPhoneNumberField`

A form field that validates input as a Dutch telephone number.

class `nl.forms.NLSofiNumberField`

A form field that validates input as a Dutch social security number (SofI/BSN).

class `nl.forms.NLZipCodeField`

A form field that validates input as a Dutch zip code.

class `nl.forms.NLProvinceSelect`

A `Select` widget that uses a list of Dutch provinces as its list of choices.

Iceland (is_)

class `is_.forms.ISIdNumberField`

A form field that validates input as an Icelandic identification number (kennitala). The format is XXXXXX-XXXX.

class `is_.forms.ISPhoneNumberField`

A form field that validates input as an Icelandic phone number (seven digits with an optional hyphen or space after the first three digits).

class `is_.forms.ISPostalCodeSelect`

A `Select` widget that uses a list of Icelandic postal codes as its choices.

India (in_)

class `in.forms.INStateField`

A form field that validates input as an Indian state/territory name or abbreviation. Input is normalized to the standard two-letter vehicle registration abbreviation for the given state or territory.

class `in.forms.INZipCodeField`

A form field that validates input as an Indian zip code, with the format XXXXXX.

class `in.forms.INStateSelect`

A `Select` widget that uses a list of Indian states/territories as its choices.

Italy (it)

class `it.forms.ITSocialSecurityNumberField`

A form field that validates input as an Italian social security number ([codice fiscale](#)).

class `it.forms.ITVatNumberField`

A form field that validates Italian VAT numbers (partita IVA).

class `it.forms.ITZipCodeField`

A form field that validates input as an Italian zip code. Valid codes must have five digits.

class `it.forms.ITProvinceSelect`

A `Select` widget that uses a list of Italian provinces as its choices.

class `it.forms.ITRegionSelect`

A `Select` widget that uses a list of Italian regions as its choices.

Japan (jp)

class `jp.forms.JPPostalCodeField`

A form field that validates input as a Japanese postcode. It accepts seven digits, with or without a hyphen.

class `jp.forms.JPPrefectureSelect`

A `Select` widget that uses a list of Japanese prefectures as its choices.

Mexico (mx)

class `mx.forms.MXStateSelect`

A `Select` widget that uses a list of Mexican states as its choices.

Norway (no)

class `no.forms.NOSocialSecurityNumber`

A form field that validates input as a Norwegian social security number (`personnummer`).

class `no.forms.NOZipCodeField`

A form field that validates input as a Norwegian zip code. Valid codes have four digits.

class `no.forms.NOMunicipalitySelect`

A `Select` widget that uses a list of Norwegian municipalities (`fylker`) as its choices.

Peru (pe)

class `pt.forms.PEDNIField`

A form field that validates input as a DNI (Peruvian national identity) number.

class `pt.forms.PERUCField`

A form field that validates input as an RUC (Registro Unico de Contribuyentes) number. Valid RUC numbers have 11 digits.

class `pt.forms.PEDepartmentSelect`

A `Select` widget that uses a list of Peruvian Departments as its choices.

Poland (pl)

class `pl.forms.PLNationalIdentificationNumberField`

A form field that validates input as a Polish national identification number (`PESEL`).

class `pl.forms.PLNationalBusinessRegisterField`

A form field that validates input as a Polish National Official Business Register Number (`REGON`), having either seven or nine digits. The checksum algorithm used for REGONs is documented at <http://wipos.p.lodz.pl/zylla/ut/nip-rego.html>.

class `pl.forms.PLPostalCodeField`

A form field that validates input as a Polish postal code. The valid format is XX-XXX, where X is a digit.

class `pl.forms.PLTaxNumberField`

A form field that validates input as a Polish Tax Number (NIP). Valid formats are XXX-XXX-XX-XX or XX-XX-XXX-XXX. The checksum algorithm used for NIPs is documented at <http://wipos.p.lodz.pl/zylla/ut/nip-rego.html>.

class `pl.forms.PLAdministrativeUnitSelect`

A `Select` widget that uses a list of Polish administrative units as its choices.

class `pl.forms.PLVoivodeshipSelect`

A `Select` widget that uses a list of Polish voivodeships (administrative provinces) as its choices.

Romania (ro)

class `ro.forms.ROCIFField`

A form field that validates Romanian fiscal identification codes (CIF). The return value strips the leading RO, if given.

class `ro.forms.ROCNPField`

A form field that validates Romanian personal numeric codes (CNP).

class `ro.forms.ROCountyField`

A form field that validates its input as a Romanian county (judet) name or abbreviation. It normalizes the input to the standard vehicle registration abbreviation for the given county. This field will only accept names written with diacritics; consider using `ROCountySelect` as an alternative.

class `ro.forms.ROCountySelect`

A `Select` widget that uses a list of Romanian counties (judete) as its choices.

class `ro.forms.ROIBANField`

A form field that validates its input as a Romanian International Bank Account Number (IBAN). The valid format is ROXX-XXXX-XXXX-XXXX-XXXX-XXXX, with or without hyphens.

class `ro.forms.ROPhoneNumberField`

A form field that validates Romanian phone numbers, short special numbers excluded.

class `ro.forms.ROPostalCodeField`

A form field that validates Romanian postal codes.

Slovakia (sk)

class `sk.forms.SKPostalCodeField`

A form field that validates input as a Slovak postal code. Valid formats are XXXXXX or XXX XX, where X is a digit.

class `sk.forms.SKDistrictSelect`

A `Select` widget that uses a list of Slovak districts as its choices.

class `sk.forms.SKRegionSelect`

A `Select` widget that uses a list of Slovak regions as its choices.

South Africa (za)

class `za.forms.ZAIDField`

A form field that validates input as a South African ID number. Validation uses the Luhn checksum and a simplistic (i.e., not entirely accurate) check for birth date.

class `za.forms.ZAPostCodeField`

A form field that validates input as a South African postcode. Valid postcodes must have four digits.

Spain (es)

class `es.forms.ESIdentityCardNumberField`

A form field that validates input as a Spanish NIF/NIE/CIF (Fiscal Identification Number) code.

class `es.forms.ESCCCFfield`

A form field that validates input as a Spanish bank account number (Codigo Cuenta Cliente or CCC). A valid CCC number has the format EEEE-OOOO-CC-AAAAAAAAAA, where the E, O, C and A digits denote the entity, office, checksum and account, respectively. The first checksum digit validates the entity and office. The second checksum digit validates the account. It is also valid to use a space as a delimiter, or to use no delimiter.

class `es.forms.ESPhoneNumberField`

A form field that validates input as a Spanish phone number. Valid numbers have nine digits, the first of which is 6, 8 or 9.

class `es.forms.ESPostalCodeField`

A form field that validates input as a Spanish postal code. Valid codes have five digits, the first two being in the range 01 to 52, representing the province.

class `es.forms.ESProvinceSelect`

A `Select` widget that uses a list of Spanish provinces as its choices.

class `es.forms.ESRegionSelect`

A `Select` widget that uses a list of Spanish regions as its choices.

Switzerland (ch)

class `ch.forms.CHIdentityCardNumberField`

A form field that validates input as a Swiss identity card number. A valid number must confirm to the X1234567<0 or 1234567890 format and have the correct checksums – see <http://adi.kousz.ch/artikel/IDCHE.htm>.

class `ch.forms.CHPhoneNumberField`

A form field that validates input as a Swiss phone number. The correct format is 0XX XXX XX XX. 0XX.XXX.XX.XX and 0XXXXXXXXX validate but are corrected to 0XX XXX XX XX.

class `ch.forms.CHZipCodeField`

A form field that validates input as a Swiss zip code. Valid codes consist of four digits.

class `ch.forms.CHStateSelect`

A `Select` widget that uses a list of Swiss states as its choices.

United Kingdom (uk)

class `uk.forms.UKPostcodeField`

A form field that validates input as a UK postcode. The regular expression used is sourced from the schema for British Standard BS7666 address types at <http://www.govtalk.gov.uk/gdsc/schemas/bs7666-v2-0.xsd>.

class `uk.forms.UKCountySelect`

A `Select` widget that uses a list of UK counties/regions as its choices.

class `uk.forms.UKNationSelect`

A `Select` widget that uses a list of UK nations as its choices.

United States of America (us)

class `us.forms.USPhoneNumberField`

A form field that validates input as a U.S. phone number.

class `us.forms.USSocialSecurityNumberField`

A form field that validates input as a U.S. Social Security Number (SSN). A valid SSN must obey the following rules:

- Format of XXX-XX-XXXX
- No group of digits consisting entirely of zeroes
- Leading group of digits cannot be 666
- Number not in promotional block 987-65-4320 through 987-65-4329
- Number not one known to be invalid due to widespread promotional use or distribution (e.g., the Woolworth's number or the 1962 promotional number)

class `us.forms.USStateField`

A form field that validates input as a U.S. state name or abbreviation. It normalizes the input to the standard two-letter postal service abbreviation for the given state.

class `us.forms.USZipCodeField`

A form field that validates input as a U.S. ZIP code. Valid formats are XXXXX or XXXXX-XXXX.

class `us.forms.USStateSelect`

A form `Select` widget that uses a list of U.S. states/territories as its choices.

class `us.models.PhoneNumberField`

A `CharField` that checks that the value is a valid U.S.A.-style phone number (in the format XXX-XXX-XXXX).

class `us.models.USStateField`

A model field that forms represent as a `forms.USStateField` field and stores the two-letter U.S. state abbreviation in the database.

A aplicação de redirecionamento

O Django vem acompanhado de uma aplicação de redirecionamento opcional. Ela permite você armazenar redirecionadores no banco de dados e manipula o redirecionamento pra você.

Instalação

Para instalá-lo, siga os seguintes passos:

1. Adicione `'django.contrib.redirects'` em suas `INSTALLED_APPS`.
2. Adicione `'django.contrib.redirects.middleware.RedirectFallbackMiddleware'` nas suas `MIDDLEWARE_CLASSES`.
3. Execute o comando `manage.py syncdb`.

Como ele funciona

O `manage.py syncdb` cria uma tabela `django_redirect` no seu banco de dados. Essa é uma tabela de aparência simples com os campos `site_id`, `old_path` e `new_path`.

O `RedirectFallbackMiddleware` faz todo trabalho. Toda vez que uma aplicação Django gera um erro 404, esse middleware checa o banco de dados do redirecionador para a URL acessada como um último recurso. Especialmente, ele procura por um redirecionador que tenha um `old_path` com um ID do site que corresponde ao `SITE_ID` no arquivo `settings.py`.

- Se ele encontra uma correspondência, e `new_path` não está vazio, ele redireciona para o `new_path`.
- Se ele encontra uma correspondência, e `new_path` está vazio, ele envia um cabeçalho HTTP 410 (“Gone”) e uma resposta vazia (content-less).

- Se ele não encontra uma correspondência, a requisição continua sendo processada normalmente.

O middleware somente opera sobre 404s – não para 500s ou respostas de qualquer outro código de status.

Note que a ordem do `MIDDLEWARE_CLASSES` importa. Geralmente, você pode colocar o `RedirectFallbackMiddleware` no final da lista, porque ele é um último recurso.

Para saber mais sobre o middleware, leia a [documentação do middleware](#).

Como adicionar, mudar e deletar redirecionadores

Via interface de administração

Se você tem ativada a interface de administração automática do Django, você pode ver uma seção “Redirects” na página principal do admin. Edite os redirecionadores como você edita qualquer outro objeto do sistema.

Via API do Python

`class models.Redirect`

Os redirecionadores são representados por um *modelo Django* padrão, que se encontram em `django/contrib/redirects/models.py`. Você pode acessar os objetos de redirecionamento utilizando a *API de banco de dados do Django*.

O framework sitemap

Django comes with a high-level sitemap-generating framework that makes creating *sitemap* XML files easy.

O Django vem com um framework de alto-nível gerador de sitemaps que torna a criação de um *sitemap* em XML fácil.

Visão geral

Um sitemap é um arquivo XML sobre o seu site Web que diz aos indexadores dos motores de busca, frequentemente, como suas páginas mudam e quão “importante” é certas páginas em relação a outras. Esta informação ajuda o buscadores a indexar seu site.

O framework sitemap do Django automatiza a criação destes arquivos XML deixando vocês expressar esta informação em código Python.

Ele funciona parecido com o *framework syndication*. Para criar um sitemap, é só escrever uma classe *Sitemap* e criar uma URL no seu *URLconf*.

Instalação

Para instalar a aplicação sitemap, siga estes passos:

1. Adicione `'django.contrib.sitemaps'` ao seu *INSTALLED_APPS*.
2. Esteja certo de que tenha o `'django.template.loaders.app_directories.load_template_source'` no seu *TEMPLATE_LOADERS*. Ele é setado por padrão, assim você só precisa mudar isso se você tiver alterado essa configuração.
3. Assegure-se de ter instalado o *framework sites*.

(Nota: A aplicação sitemap não instala qualquer tabela no banco de dados. A única razão dela precisar estar no *INSTALLED_APPS* é para o template loader `load_template_source()` poder encontrar os templates padrões.)

Inicialização

Para ativar a geração do sitemap no seu site Django, adiciona esta linha ao seu *URLconf*:

```
(r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': sitemaps}))
```

Isto diz ao Django para fazer o sitemap quando um cliente acessa `/sitemap.xml`.

O nome do arquivo do sitemap não é importante, mas a localização é. Os buscadores somente indexam links no seu sitemap para o nível atual e inferiores. Por instância, se `sitemap.xml` reside no seu diretório root, ele poderá referenciar qualquer URL no seu site. Entretanto, se seu sitemap fica em `/content/sitemap.xml`, ele somente referencia as URLs a partir de `/content/`.

O view `sitemap` recebe um argumento extra, e obrigatório: `{ 'sitemaps': sitemaps }`. O `sitemaps` deve ser um dicionário que mapeia um label curto da seção (e.g. `blog` ou `news`) para sua classe *Sitemap* (e.g. `BlogSitemap` ou `NewsSitemap`). Ele pode também mapear para uma *instância* da classe *Sitemap* (e.g., `BlogSitemap(some_var)`).

Classes do sitemap

Uma classe *Sitemap* é uma simples classe Python que representa uma “seção” de entradas no sitemap. Por exemplo, uma classe *Sitemap* representaria todas as entradas do seu weblog, enquanto outra representaria todos os eventos do seu calendário de eventos.

Num caso simplista, todas essas seções seriam agrupadas num arquivo `sitemap.xml`, mas somente é possível usar o framework para gerar um sitemap index que representa arquivos individuais, um por seção. (Veja, *Criando um sitemap index* abaixo.)

As classes *Sitemap* devem estender `django.contrib.sitemaps.Sitemap`. Elas podem estar em qualquer lugar de sua base de código.

Um exemplo simples

Vamos assumir que você tem um sistema de blog, com um model `Entry`, e você deseja que seu sitemap inclua todos os links de suas entradas de no blog. Aqui tem como sua classe sitemap pode parecer:

```
from django.contrib.sitemaps import Sitemap
from mysite.blog.models import Entry

class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Entry.objects.filter(is_draft=False)

    def lastmod(self, obj):
        return obj.pub_date
```

Note:

- *changefreq* e *priority* são atributos de classes correspondentes aos elementos `<changefreq>` e `<priority>`, respectivamente. Eles podem ser chamados como funções, como o *lastmod* foi no último exemplo.
- *items()* é simplesmente um método que retorna uma lista de objetos. Os objetos retornados serão passados para qualquer método chamável correspondendo a uma propriedade do sitemap (*location*, *lastmod*, *changefreq*, e *priority*).
- *lastmod* deve retornar um objeto Python `datetime`.

- Não há o método `location` neste exemplo, mas você pode fornecê-la a fim de especificar a URL para o objeto. Por padrão, `location()` chama o método `get_absolute_url()` de cada objeto e retorna o resultado.

Referência da classe Sitemap

class Sitemap

Uma classe `Sitemap` pode definir os seguintes métodos/atributos:

items

Obrigatório. Um método que retorna uma lista de objetos. O framework não se preocupa com o *tipo* de objetos que eles são; tudo que importa é que estes objetos são passados para os métodos `location()`, `lastmod()`, `changefreq()` e `priority()`.

location

Opcional. Ambos método ou atributo.

Se ele for um método, ele deve retornar a URL absoluta para um dado objeto como retornado pelo `items()`.

Se ele for um atributo, seu valor deve ser uma string representando uma URL absoluta para ser usada por *todos* os objetos retornados pelo `items()`.

Em ambos os casos, a “URL absoluta” significa uma URL que não inclui o protocolo ou domínio. Exemplos:

- Bom: `'/foo/bar/'`
- Ruim: `'example.com/foo/bar/'`
- Ruim: `'http://example.com/foo/bar/'`

Se o `location` não é fornecido, o framework chamará o método `get_absolute_url()` de cada objeto retornado pelo `items()`.

lastmod

Opcional. Ambos, método ou atributo.

Se for um método, deve receber um argumento – um objeto retornado pelo `items()` – e retornar a data/hora da última modificação deste objeto, como um objeto Python `datetime.datetime`.

Se for um atributo, seu valor deve ser um objeto Python `datetime.datetime` representando a data/hora da última modificação para *todo* objeto retornado por `items()`.

changefreq

Opcional. Ambos, método ou atributo.

Se for um método, ele deve receber um argumento – um objeto retornado por `items()` – e retornar a frequência de mudança deste objeto, como uma string Python.

Se for um atributo, seu valor deve ser uma string representando a frequência de mudança para *todo* objeto retornado por `items()`.

Os valores possíveis para `changefreq`, se você usar um método ou atributo, são:

- `'always'`
- `'hourly'`
- `'daily'`
- `'weekly'`
- `'monthly'`
- `'yearly'`

- 'never'

priority()

Opcional. Ambos, método ou atributo.

Se for um método, deve ser receber um argumento – um objeto retornado por `items()` – e retorna a prioridade deste objeto, como uma string ou float.

Se for um atributo, seu valor deve ser ou uma string ou uma float representando a prioridade de *todo* objeto retornado por `items()`.

Exemplo de valores para `:attr~Sitemap.priority`: 0.4, 1.0. O valor de prioridade padrão de uma página é 0.5. Veja a [documentação do sitemaps.org](#) para saber mais.

Shortcuts

O framework sitemap fornece algumas classes convenientes para casos comuns:

class FlatPageSitemap

A classe `django.contrib.sitemaps.FlatPageSitemap` se parece com a `flatpages` definida para o `SITE_ID` atual (Veja a [documentação do sites](#)) e cria uma entrada no sitemap. Estas entradas inclui somente o atributo `location` – não `lastmod`, `changefreq` ou `priority`.

class GenericSitemap

A classe `django.contrib.sitemaps.GenericSitemap` funciona como qualquer *generic views* que você já tenha. Para usá-la, cria uma instância, passando o mesmo `info_dict` que você passa para uma generic view. O único requisito é que o dicionário tenha uma `queryset` de entradas. Ele pode também ter uma entrada `date_field` que especifica um campo de data para objetos recebidos de um `queryset`. Esse será usado para o atributo `lastmod` no sitemap gerado. Você pode também passar os argumentos nomeados `priority` e `changefreq` para o construtor do `GenericSitemap` para especificar estes atributos para todas as URLs.

Exemplo

Aqui tem um exemplo de um `URLconf` usando ambos:

```
from django.conf.urls.defaults import *
from django.contrib.sitemaps import FlatPageSitemap, GenericSitemap
from mysite.blog.models import Entry

info_dict = {
    'queryset': Entry.objects.all(),
    'date_field': 'pub_date',
}

sitemaps = {
    'flatpages': FlatPageSitemap,
    'blog': GenericSitemap(info_dict, priority=0.6),
}

urlpatterns = patterns('',
    # Algum generic view usando info_dict
    # ...

    # o sitemap
    (r'^sitemap.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps': ↵
↵sitemaps})
)
```

Criando um sitemap index

O framework sitemap também tem a habilidade de criar um sitemap index que referencia arquivos de sitemap individuais, um para cada seção definida no seu dicionário sitemaps. As únicas diferenças no uso são:

- você usa dois views no seu URLconf: `django.contrib.sitemaps.views.index()` e `django.contrib.sitemaps.views.sitemap()`.
- O view `django.contrib.sitemaps.views.sitemap()` deve receber um argumento nomeado `section`.

Aqui tem como linhas relevantes do URLconf poderiam parecer para o exemplo acima:

```
(r'^sitemap.xml$', 'django.contrib.sitemaps.views.index', {'sitemaps': sitemaps}),
(r'^sitemap-(?P<section>.+)\.xml$', 'django.contrib.sitemaps.views.sitemap', {
    ↪ 'sitemaps': sitemaps}),
```

Isto gerará automaticamente um arquivo `sitemap.xml` que referencia ambos `sitemap-flatpages.xml` e `sitemap-blog.xml`. As classes `Sitemap` e o dicionário `sitemaps` não mudam.

Você deve criar um arquivo `index` se um de seus sitemaps tem mais de 50,000 URIs. Neste caso, o Django automaticamente paginará o sitemap, e o `index` irá reletir isso.

Pingando o Google

Você deve desejar “pingar” o Google quando seu site mudar, para permitir que ele saiba que tem que rein-dexar seu site. O framework sitemaps provê uma função só para fazer isso: `django.contrib.sitemaps.ping_google()`.

`ping_google()`

A função `ping_google()` recebe um argumento, `sitemap_url`, que deve ser uma URL absoluta do sitemap do seu site. (e.g., `'/sitemap.xml'`). Se este argumento não for fornecido, o `ping_google()` tentará adivinhar seu sitemap, executando uma engenharia reversa sobre seu URLconf.

O `ping_google()` lança uma exceção `django.contrib.sitemaps.SitemapNotFound` se ele não conseguir determinar a URL do sitemap.

Register with Google first!

O comando `ping_google()` somente funciona se você tiver registrado seu site com o [Google Webmaster Tools](#).

Uma forma usual para chamar `ping_google()` é de um método `save()` de um model:

```
from django.contrib.sitemaps import ping_google

class Entry(models.Model):
    # ...
    def save(self, force_insert=False, force_update=False):
        super(Entry, self).save(force_insert, force_update)
        try:
            ping_google()
        except Exception:
            # 'Exceção' vazia, pois nós podemos receber uma variedade de
            # exceções relacionadas ao HTTP.
            pass
```

Uma solução mais eficiente, entretanto, seria chamar `ping_google()` de um script cron, ou alguma outra tarefa agendada. A função faz uma requisição HTTP, para os servidores do Google, então você precisaria introduzir uma sobrecarga de transferência de dados para toda chamada do `save()`.

Pingando o Google via *manage.py*

Please, see the release notes Uma vez que a aplicação sitemap esteja adicionada ao seu projeto, você pode também pingar os servidores do Google através da interface de linha de comando *manage.py*:

```
python manage.py ping_google [/sitemap.xml]
```

The “sites” framework

Django comes with an optional “sites” framework. It’s a hook for associating objects and functionality to particular Web sites, and it’s a holding place for the domain names and “verbose” names of your Django-powered sites.

Use it if your single Django installation powers more than one site and you need to differentiate between those sites in some way.

The whole sites framework is based on a simple model:

```
class django.contrib.sites.models.Site
```

This model has `domain` and `name` fields. The `SITE_ID` setting specifies the database ID of the *Site* object associated with that particular settings file.

How you use this is up to you, but Django uses it in a couple of ways automatically via simple conventions.

Example usage

Why would you use sites? It’s best explained through examples.

Associating content with multiple sites

The Django-powered sites [LJWorld.com](#) and [Lawrence.com](#) are operated by the same news organization – the Lawrence Journal-World newspaper in Lawrence, Kansas. LJWorld.com focuses on news, while Lawrence.com focuses on local entertainment. But sometimes editors want to publish an article on *both* sites.

The brain-dead way of solving the problem would be to require site producers to publish the same story twice: once for LJWorld.com and again for Lawrence.com. But that’s inefficient for site producers, and it’s redundant to store multiple copies of the same story in the database.

The better solution is simple: Both sites use the same article database, and an article is associated with one or more sites. In Django model terminology, that’s represented by a *ManyToManyField* in the *Article* model:

```
from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    sites = models.ManyToManyField(Site)
```

This accomplishes several things quite nicely:

- It lets the site producers edit all content – on both sites – in a single interface (the Django admin).
- It means the same story doesn’t have to be published twice in the database; it only has a single record in the database.
- It lets the site developers use the same Django view code for both sites. The view code that displays a given story just checks to make sure the requested story is on the current site. It looks something like this:

```

from django.conf import settings

def article_detail(request, article_id):
    try:
        a = Article.objects.get(id=article_id, sites__id__exact=settings.SITE_
↪ID)
    except Article.DoesNotExist:
        raise Http404
    # ...

```

Associating content with a single site

Similarly, you can associate a model to the *Site* model in a many-to-one relationship, using `ForeignKey`.

For example, if an article is only allowed on a single site, you'd use a model like this:

```

from django.db import models
from django.contrib.sites.models import Site

class Article(models.Model):
    headline = models.CharField(max_length=200)
    # ...
    site = models.ForeignKey(Site)

```

This has the same benefits as described in the last section.

Hooking into the current site from views

On a lower level, you can use the sites framework in your Django views to do particular things based on the site in which the view is being called. For example:

```

from django.conf import settings

def my_view(request):
    if settings.SITE_ID == 3:
        # Do something.
    else:
        # Do something else.

```

Of course, it's ugly to hard-code the site IDs like that. This sort of hard-coding is best for hackish fixes that you need done quickly. A slightly cleaner way of accomplishing the same thing is to check the current site's domain:

```

from django.conf import settings
from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get(id=settings.SITE_ID)
    if current_site.domain == 'foo.com':
        # Do something
    else:
        # Do something else.

```

The idiom of retrieving the *Site* object for the value of `settings.SITE_ID` is quite common, so the *Site* model's manager has a `get_current()` method. This example is equivalent to the previous one:

```

from django.contrib.sites.models import Site

def my_view(request):
    current_site = Site.objects.get_current()
    if current_site.domain == 'foo.com':

```

```
    # Do something
else:
    # Do something else.
```

Getting the current domain for display

LJWorld.com and Lawrence.com both have e-mail alert functionality, which lets readers sign up to get notifications when news happens. It’s pretty basic: A reader signs up on a Web form, and he immediately gets an e-mail saying, “Thanks for your subscription.”

It’d be inefficient and redundant to implement this signup-processing code twice, so the sites use the same code behind the scenes. But the “thank you for signing up” notice needs to be different for each site. By using *Site* objects, we can abstract the “thank you” notice to use the values of the current site’s name and domain.

Here’s an example of what the form-handling view looks like:

```
from django.contrib.sites.models import Site
from django.core.mail import send_mail

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...

    current_site = Site.objects.get_current()
    send_mail('Thanks for subscribing to %s alerts' % current_site.name,
            'Thanks for your subscription. We appreciate it.\n\n-The %s team.' %
↪current_site.name,
            'editor@%s' % current_site.domain,
            [user.email])

    # ...
```

On Lawrence.com, this e-mail has the subject line “Thanks for subscribing to lawrence.com alerts.” On LJWorld.com, the e-mail has the subject “Thanks for subscribing to LJWorld.com alerts.” Same goes for the e-mail’s message body.

Note that an even more flexible (but more heavyweight) way of doing this would be to use Django’s template system. Assuming Lawrence.com and LJWorld.com have different template directories (*TEMPLATE_DIRS*), you could simply farm out to the template system like so:

```
from django.core.mail import send_mail
from django.template import loader, Context

def register_for_newsletter(request):
    # Check form values, etc., and subscribe the user.
    # ...

    subject = loader.get_template('alerts/subject.txt').render(Context({}))
    message = loader.get_template('alerts/message.txt').render(Context({}))
    send_mail(subject, message, 'editor@ljworld.com', [user.email])

    # ...
```

In this case, you’d have to create `subject.txt` and `message.txt` template files for both the LJWorld.com and Lawrence.com template directories. That gives you more flexibility, but it’s also more complex.

It’s a good idea to exploit the *Site* objects as much as possible, to remove unneeded complexity and redundancy.

Getting the current domain for full URLs

Django’s `get_absolute_url()` convention is nice for getting your objects’ URL without the domain name, but in some cases you might want to display the full URL – with `http://` and the domain and everything – for an object. To do this, you can use the sites framework. A simple example:

```
>>> from django.contrib.sites.models import Site
>>> obj = MyModel.objects.get(id=3)
>>> obj.get_absolute_url()
'/mymodel/objects/3/'
>>> Site.objects.get_current().domain
'example.com'
>>> 'http://%s%s' % (Site.objects.get_current().domain, obj.get_absolute_url())
'http://example.com/mymodel/objects/3/'
```

Caching the current Site object

Please, see the release notes As the current site is stored in the database, each call to `Site.objects.get_current()` could result in a database query. But Django is a little cleverer than that: on the first request, the current site is cached, and any subsequent call returns the cached data instead of hitting the database.

If for any reason you want to force a database query, you can tell Django to clear the cache using `Site.objects.clear_cache()`:

```
# First call; current site fetched from database.
current_site = Site.objects.get_current()
# ...

# Second call; current site fetched from cache.
current_site = Site.objects.get_current()
# ...

# Force a database query for the third call.
Site.objects.clear_cache()
current_site = Site.objects.get_current()
```

The CurrentSiteManager

class `django.contrib.sites.managers.CurrentSiteManager`

If *Sites* play a key role in your application, consider using the helpful *CurrentSiteManager* in your model(s). It’s a model *manager* that automatically filters its queries to include only objects associated with the current *Site*.

Use *CurrentSiteManager* by adding it to your model explicitly. For example:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    site = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager()
```

With this model, `Photo.objects.all()` will return all `Photo` objects in the database, but `Photo.on_site.all()` will return only the `Photo` objects associated with the current site, according to the `SITE_ID` setting.

Put another way, these two statements are equivalent:

```
Photo.objects.filter(site=settings.SITE_ID)
Photo.on_site.all()
```

How did `CurrentSiteManager` know which field of `Photo` was the `Site`? It defaults to looking for a field called `Site`. If your model has a `ForeignKey` or `ManyToManyField` called something *other* than `Site`, you need to explicitly pass that as the parameter to `CurrentSiteManager`. The following model, which has a field called `publish_on`, demonstrates this:

```
from django.db import models
from django.contrib.sites.models import Site
from django.contrib.sites.managers import CurrentSiteManager

class Photo(models.Model):
    photo = models.FileField(upload_to='/home/photos')
    photographer_name = models.CharField(max_length=100)
    pub_date = models.DateField()
    publish_on = models.ForeignKey(Site)
    objects = models.Manager()
    on_site = CurrentSiteManager('publish_on')
```

If you attempt to use `CurrentSiteManager` and pass a field name that doesn't exist, Django will raise a `ValueError`.

Finally, note that you'll probably want to keep a normal (non-site-specific) `Manager` on your model, even if you use `CurrentSiteManager`. As explained in the [manager documentation](#), if you define a manager manually, then Django won't create the automatic `objects = models.Manager()` manager for you. Also, note that certain parts of Django – namely, the Django admin site and generic views – use whichever manager is defined *first* in the model, so if you want your admin site to have access to all objects (not just site-specific ones), put `objects = models.Manager()` in your model, before you define `CurrentSiteManager`.

How Django uses the sites framework

Although it's not required that you use the sites framework, it's strongly encouraged, because Django takes advantage of it in a few places. Even if your Django installation is powering only a single site, you should take the two seconds to create the site object with your domain and name, and point to its ID in your `SITE_ID` setting.

Here's how Django uses the sites framework:

- In the [redirects framework](#), each redirect object is associated with a particular site. When Django searches for a redirect, it takes into account the current `SITE_ID`.
- In the [comments framework](#), each comment is associated with a particular site. When a comment is posted, its `Site` is set to the current `SITE_ID`, and when comments are listed via the appropriate template tag, only the comments for the current site are displayed.
- In the [flatpages framework](#), each flatpage is associated with a particular site. When a flatpage is created, you specify its `Site`, and the `FlatpageFallbackMiddleware` checks the current `SITE_ID` in retrieving flatpages to display.
- In the [syndication framework](#), the templates for title and description automatically have access to a variable `{{ site }}`, which is the `Site` object representing the current site. Also, the hook for providing item URLs will use the domain from the current `Site` object if you don't specify a fully-qualified domain.
- In the [authentication framework](#), the `django.contrib.auth.views.login()` view passes the current `Site` name to the template as `{{ site_name }}`.

- The shortcut view (`django.views.defaults.shortcut()`) uses the domain of the current *Site* object when calculating an object's URL.
- In the admin framework, the “view on site” link uses the current *Site* to work out the domain for the site that it will redirect to.

RequestSite objects

Please, see the release notes Some *django.contrib* applications take advantage of the sites framework but are architected in a way that doesn't *require* the sites framework to be installed in your database. (Some people don't want to, or just aren't *able* to install the extra database table that the sites framework requires.) For those cases, the framework provides a *RequestSite* class, which can be used as a fallback when the database-backed sites framework is not available.

A *RequestSite* object has a similar interface to a normal *Site* object, except its `__init__()` method takes an *HttpRequest* object. It's able to deduce the domain and name by looking at the request's domain. It has `save()` and `delete()` methods to match the interface of *Site*, but the methods raise `NotImplementedError`.

O framework syndication feed

O Django vem com um framework de alto nível para geração de feeds que criam feeds em *RSS* e *Atom* facilmente.

Para criar qualquer feed, tudo que você tem de fazer é escrever uma pequena classe Python. Você pode criar quantos feeds você quiser.

O Django também vem com uma API de baixo nível para geração de feeds. Use-a se você deseja gerar feeds fora do contexto Web, ou de alguma outra forma de mais baixo nível.

O framework de alto nível

Visão geral

O framework de alto nível é um view vinculada a URL `/feeds/` por padrão. O Django usa o restante da URL (tudo que vier depois de `/feeds/`) para determinar qual feed mostrar.

Para criar um feed, é só escrever uma classe *Feed* e apontá-la para seu *URLconf*.

Inicialização

Para ativar os feeds no seu site Django, adicione esta linha ao seu *URLconf*:

```
(r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed', {'feed_dict': ↵
↵ feeds})),
```

Isso diz ao Django para usar o framework RSS para manipular todas as URLs que começam com `"feeds/"`. (Você pode mudar o prefixo `"feeds/"` para fechar com suas próprias necessidades.)

Esta linha do *URLconf* tem um argumento extra: `{'feed_dict': feeds}`. Use este argumento extra para passar ao framework os feeds que devem ser publicados sob aquela URL.

Especificamente, `feed_dict` deve ser um dicionário que mapeia um slug de um feed (uma label curta da URL) para sua classe *Feed*.

Você pode definir o `feed_dict` no próprio *URLconf*. Aqui temos um exemplo completo do *URLconf*:

```
from django.conf.urls.defaults import *
from myproject.feeds import LatestEntries, LatestEntriesByCategory

feeds = {
    'latest': LatestEntries,
    'categories': LatestEntriesByCategory,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
     {'feed_dict': feeds}),
    # ...
)
```

O exemplo acima registra dois feeds:

- O feed representado pelo `LatestEntries` ficará em `feeds/latest/`.
- O feed representado pelo `LatestEntriesByCategory` ficará em `feeds/categories/`.

Uma vez configurado, você só precisa definir as classes *Feed* elas mesmas.

Classes Feed

Uma classe *Feed* é uma simples classe Python que representa um feed. Um feed pode ser uma simples (e.g., um feed de “notícias do site”, ou um feed básico mostrando as últimas entradas de um blog) ou algo mais complexo (e.g., um feed mostrando todas as entradas do blog numa categoria particular, onde a categoria é variável).

As classes *Feed* devem estender `django.contrib.syndication.feeds.Feed`. Elas podem estar em qualquer parte de sua base de código.

Um exemplo simples

Este exemplo simples, extraído do chicagocrime.org, descreve um feed dos últimos cinco itens mais novos:

```
from django.contrib.syndication.feeds import Feed
from chicagocrime.models import NewsItem

class LatestEntries(Feed):
    title = "Chicagocrime.org site news"
    link = "/siteneews/"
    description = "Updates on changes and additions to chicagocrime.org."

    def items(self):
        return NewsItem.objects.order_by('-pub_date')[:5]
```

Note:

- A classe estende `django.contrib.syndication.feeds.Feed`.
- `title`, `link` e `description` correspondem aos elementos padrão do RSS `<title>`, `<link>` e `<description>`, respectivamente.
- O `items()` é, simplesmente, um método que retorna a lista de objetos que deveriam ser incluídos no feed como elementos `<item>`. Embora este exemplo retorne objetos `NewsItem` usando o *mapeador objeto-relacional* do Django, o `items()` não tem como retornar instâncias de model. Apesar de você ter algumas funcionalidades “for free” usando os models do Django, o `items()` pode retornar qualquer tipo de objeto que você quiser.
- Se você estiver criando um feed Atom, ao invés de um feed RSS, configure o atributo `subtitle` ao invés do atributo `description`. Veja *Publicando feeds Atom e RSS no tandem*, depois, só para exemplificar.

Uma coisa que resta fazer. Num feed RSS, cada `<item>` tem um `<title>`, `<link>` e `<description>`. Nós precisamos dizer ao framework quais dados por nesses elementos.

- Para especificar os conteúdos de `<title>` e `<description>`, crie *templates Django* chamados `feeds/latest_title.html` e `feeds/latest_description.html`, onde `latest` é o slug especificado no URLconf para o dado feed. Note que a extensão `.html` é obrigatória. O sistema RSS renderiza este template para cada item, passando-o duas variáveis de contexto:

- `{{ obj }}` – O objeto atual (um dos objetos que você retornou em `items()`).
- `{{ site }}` – Um objeto `django.contrib.sites.models.Site` representando o site atual. Este é útil para `{{ site.domain }}` ou `{{ site.name }}`. Se você *não* tem o framework Django sites instalado, isso será setado com um objeto `django.contrib.sites.models.RequestSite`. Veja a [seção RequestSite da documentação do framework sites](#) para mais.

Se você não criar um template para ambos título ou descrição, o framework usará o template `"{{ obj }}"` por padrão – isto é, a representação normal do objeto. Você pode também mudar os nomes deste dois templates especificando `title_template` e `description_template` como atributos de sua própria classe `Feed`.

- Para especificar os conteúdos do `<link>`, você tem duas opções. Para cada item em `items()`, O Django primeiro tenta chamar o método `item_link()` na classe `Feed`, passando-o um único parametro, `item`, que 'o objeto em si. Se este método não existe, o Django tenta executar um método `get_absolute_url()` neste objeto. Ambos `get_absolute_url()` e `item_link()` devem retornar a URL do item como uma string normal do Python. Como no `get_absolute_url()`, o resultado do `item_link()` será incluído diretamente na URL, então você é responsável por fazer todo o necessário para escapar a URL e convertê-la para ASCII dentro do método.
- Para o exemplo LatestEntries acima, nós poderíamos ter um template de feed muito mais simples:

- `latest_title.html`:

```
{{ obj.title }}
```

- `latest_description.html`:

```
{{ obj.description }}
```

Um exemplo complexo

O framework também suporta feeds mais complexos, via parâmetros.

Por exemplo, [chicagocrime.org](#) oferece um RSS feed de crimes recentes para cada batida policial em Chicago. Seria bobagem criar uma classe `Feed` para cada batida policial; o que poderia violar o *princípio DRY* e juntamente a lógica de programação dos dados. Ao invés, o framework deixa você fazer feeds genéricos que mostram itens baseados em informações da URL do feed.

No [chicagocrime.org](#), os feeds de batidas policiais são acessíveis por URLs como estas:

- `/rss/beats/0613/` – Retorna crimes recentes para batida 0613.
- `/rss/beats/1424/` – Retorna crimes recentes para batida 1424.

O slug aqui é "beats" ``. O framework exerga os fragmentos extras na URL depois do slug -- ``0613 e 1424 – e dá a você um hook para dizer-lhe o que esses fragmentos significam, e como eles podem influenciar em quais itens serão publicados no feed.

Um exemplo deixará tudo mais claro. Aqui temos um código para esses feeds de batidas em específico:

```
from django.contrib.syndication.feeds import FeedDoesNotExist
from django.core.exceptions import ObjectDoesNotExist

class BeatFeed(Feed):
    def get_object(self, bits):
```



```

# No caso de "/rss/beats/0613/foo/bar/baz/", ou outra coisa do tipo,
# checa se fragmentos possui somente um membro.
if len(bits) != 1:
    raise ObjectDoesNotExist
return Beat.objects.get(beat__exact=bits[0])

def title(self, obj):
    return "Chicagocrime.org: Crimes for beat %s" % obj.beat

def link(self, obj):
    if not obj:
        raise FeedDoesNotExist
    return obj.get_absolute_url()

def description(self, obj):
    return "Crimes recently reported in police beat %s" % obj.beat

def items(self, obj):
    return Crime.objects.filter(beat__id__exact=obj.id).order_by('-crime_date
↪')[:30]

```

Aqui temos um algoritmo básico que o framework RSS segue, dada esta classe e uma requisição para uma URL `/rss/beats/0613/`:

- O framework pega a URL `/rss/beats/0613/` e identifica onde há um fragmento extra na URL depois do slug. Ele separa esta string remanescente por um caracter barra ("`/`") e chama o método da classe `Feed.get_object()`, passando-o os fragmentos. Neste caso, fragmentos são `['0613']`. Para uma requisição como `/rss/beats/0613/foo/bar/`, os fragmentos poderiam ser `['0613', 'foo', 'bar']`.
- O `get_object()` é responsável por receber a certa batida, de um fragmento. Neste caso, ele usa a API de banco de dados do Django para receber a batida. Note que `get_object()` deve lançar um `django.core.exceptions.ObjectDoesNotExist` se os parâmetros dados forem inválidos. Não há `try/except` numa chamada `Beat.objects.get()`, pois não é necessário; esta função lança um `Beat.DoesNotExist` em caso de falha, e `Beat.DoesNotExist` é uma subclasse de `ObjectDoesNotExist`. Lançando `ObjectDoesNotExist` no `get_object()` diz ao Django para produzir um erro 404 para aquela requisição. `get_object()` pode manipular a url `/rss/beats/`. O método `get_object()` também tem a oportunidade de lidar com a url `/rss/beats/`. Neste caso, `bits` será uma lista vazia. No nosso exemplo, `len(bits) != 1` e uma exceção `ObjectDoesNotExist` serão lançados, então `/rss/beats/` gerará uma página 404. Mas você pode lidar com este caso da forma como quiser. Por exemplo, você poderia gerar um feed combinado para todas as batidas.
- Para gerar o `<title>`, `<link>` e `<description>` do feed, O Django usa os métodos `title()`, `link()` e `description()`. No exemplo anterior, eles foram uma string simples num atributo da classe, mas este exemplo ilustra que eles podem ser tanto strings quanto métodos. Para cada um `title`, `link` e `description`, o Django segue este algoritmo:
 - Primeiro, ele tenta chamar o método, passando o argumento `obj`, onde `obj` é o objeto retornado por `get_object()`.
 - Caso falhe, ele tenta chamar o método sem argumentos.
 - Caso falhe, ele usa o atributo da classe.

Dentro do método `link()`, nós lidamos com a possibilidade de que o `obj` pode ser `None`, isso pode ocorrer quando a URL não foi completamente especificada. Em alguns casos, você pode querer fazer algo a mais, o que poderia significar que você precisará chegar a existência do `obj` nos outros métodos também. (O método `link()` é chamada muito antes do processo de geração do feed, então é um bom lugar para liberá-lo logo.)

- Finalmente, no que `items()` nesse exemplo também recebe o argumento `obj`. O algoritmo para o `items` é o mesmo descrito no passo anterior – primeiro, ele tenta o `items(obj)`, depois `items()`, e finalmente o atributo da classe `items` (que deve ser uma lista).

The `ExampleFeed` class below gives full documentation on methods and attributes of *Feed* classes.

Especificando o tipo de feed

Por padrão, os feeds produzidos neste framework usa RSS 2.0.

Para mudar isso, adicione um atributo `feed_type` na sua classe *Feed*, tipo:

```
from django.utils.feedgenerator import Atom1Feed

class MyFeed(Feed):
    feed_type = Atom1Feed
```

Note que é para setar `feed_type` no objeto da classe, não numa instância.

Atualmente os tipos de feed disponíveis são:

- `django.utils.feedgenerator.Rss201rev2Feed` (RSS 2.01. Padrão.)
- `django.utils.feedgenerator.RssUserland091Feed` (RSS 0.91.)
- `django.utils.feedgenerator.Atom1Feed` (Atom 1.0.)

Compartimentos

Para especificar compartimento, como estes usados na criação de feeds de podcasts, use os hooks `item_enclosure_url`, `item_enclosure_length` e `item_enclosure_mime_type`. Veja a classe `ExampleFeed` abaixo para exemplos de uso.

Language

Os feeds criados pelo framework syndication automaticamente incluem a tag `<language>` (RSS 2.0) ou atributo `xml:lang` (Atom). Este vem diretamente de sua configuração `LANGUAGE_CODE`.

URLs

O método/atributo `link` pode retornar tanto uma URL absoluta (e.g. `"/blog/"`) quanto uma URL completa com o domínio e protocolo (e.g. `"http://www.example.com/blog/"`). Se `link` não retorna o domínio, o framework inserirá o domínio do site atual, de acordo com a sua configuração `SITE_ID`.

Os feeds Atom requerem um `<link rel="self">` que define a localização do feed atual. O framework syndication popula isso automaticamente, usando o domínio do site atual de acordo com a configuração `SITE_ID`.

Publicando feeds Atom e Rss no tandem

Alguns desenvolvedores disponibilizam ambas versões Atom e RSS de seus feeds. Isso é fácil de fazer com o Django: É só criar uma subclasse para sua classe *Feed* e setar o `feed_type` para algo diferente. Então atualizar seu `URLconf` para adicionar as versões extras.

Aqui temos um exemplo completo:

```
from django.contrib.syndication.feeds import Feed
from chicanocrime.models import NewsItem
from django.utils.feedgenerator import Atom1Feed

class RssSiteNewsFeed(Feed):
    title = "Chicanocrime.org site news"
    link = "/siteneWS/"
    description = "Updates on changes and additions to chicanocrime.org."
```

```
def items(self):
    return NewsItem.objects.order_by('-pub_date')[:5]

class AtomSiteNewsFeed(RssSiteNewsFeed):
    feed_type = Atom1Feed
    subtitle = RssSiteNewsFeed.description
```

Note: Neste exemplo, o feed RSS usa um `description` enquanto o feed Atom usa um `subtitle`. Isso porque os feeds Atom não fornece um “description” a nível de feed, mas eles*fornece* um “subtitle.”

Se você fornecer um `description` na sua classe `Feed`, o Django *não* colocará automaticamente isso dentro do elemento `subtitle`, pois um `subtitle` e um `description` não são necessariamente a mesma coisa. Ao invés disso, você deve definir um atributo `subtitle`.

No exemplo acima, nós simplesmente setamos o `subutitle` do feed Atom para o `description` do feed RSS, pois é bastante curto já.

E o `URLconf` acompanhante:

```
from django.conf.urls.defaults import *
from myproject.feeds import RssSiteNewsFeed, AtomSiteNewsFeed

feeds = {
    'rss': RssSiteNewsFeed,
    'atom': AtomSiteNewsFeed,
}

urlpatterns = patterns('',
    # ...
    (r'^feeds/(?P<url>.*)/$', 'django.contrib.syndication.views.feed',
     {'feed_dict': feeds}),
    # ...
)
```

Referência da classe Feed

`class django.contrib.syndication.feeds.Feed`

Este exemplo ilustra todos os possíveis atributos e métodos para uma classe `Feed`:

```
from django.contrib.syndication.feeds import Feed
from django.utils import feedgenerator

class ExampleFeed(Feed):

    # FEED TYPE -- Opcional. Este deve ser uma classe que estende
    # django.utils.feedgenerator.SyndicationFeed. Este designa qual tipo
    # de feed deve ser: RSS 2.0, Atom 1.0, etc.
    # Se você não especificar feed_type, seu feed será RSS 2.0.
    # Este deve ser uma classe, não uma instância de classe.

    feed_type = feedgenerator.Rss201rev2Feed

    # TEMPLATE NAMES -- Opcional. Estes devem ser strings representando
    # nome dos templates do Django que o sistema deve usar para renderizar o
    # title e description de seus itens do feed. Ambos são opcionais.
    # Se você não especificar um, ou ambos, o Django usará o template
    # 'feeds/SLUG_title.html' e 'feeds/SLUG_description.html', onde SLUG
    # é o slug que você especificar na URL.
```

```

title_template = None
description_template = None

# TITLE -- Pelo menos um dos três é obrigatório. O framework procura por
# eles nessa ordem.

def title(self, obj):
    """
    Recebe o objeto retornado por get_object() e retorna o title do feed
    como uma string normal do Python.
    """

def title(self):
    """
    Retorna o title do feed como uma string normal do Python.
    """

title = 'foo' # Hard-coded title.

# LINK -- Pelo menos um dos três é obrigatório. O framework procura por
# eles nessa ordem.

def link(self, obj):
    """
    Recebe o objeto retornado por get_object() e retorna o link do feed
    como uma string normal do Python.
    """

def link(self):
    """
    Retorna o link do feed como uma string normal do Python.
    """

link = '/foo/bar/' # Hard-coded link.

# GUID -- Pelo menos um dos três é opcional. O framework procura por
# eles nesta ordem. Esta propriedade é somente usada em feeds Atom
# (onde é o ID do elemento no nível do feed). Se não for fornecido, o
# link do feed é usado como um ID.

def feed_guid(self, obj):
    """
    Recebe um objeto retornado por get_object() e retorna o ID único
    global para o feed como uma string normal do Python.
    """

def feed_guid(self):
    """
    Retorna o ID único global do feed como uma string normal do Python.
    """

feed_guid = '/foo/bar/1234' # Hard-coded guid.

# DESCRIPTION -- Pelo menos um dos três é obrigatório. O framework
# procura por eles nesta ordem.

def description(self, obj):
    """
    Recebe o objeto retornado pelo get_object() e retorna o description
    do feed como uma string normal do Python.
    """

```

```
def description(self):
    """
    Retorna o description do feed como uma string normal do Python.
    """

    description = 'Foo bar baz.' # Hard-coded description.

    # AUTHOR NAME -- Pelo menos um dos três é opcional. O framework procura
    # por eles nesta ordem.

    def author_name(self, obj):
        """
        Recebe um objeto retornado por get_object() e retorna o nome do
        autor do feed como uma string normal do Python.
        """

    def author_name(self):
        """
        Retorna o nome do autor do feed como uma string normal do Python.
        """

    author_name = 'Sally Smith' # Hard-coded author name.

    # AUTHOR E-MAIL --Pelo menos um dos três é opcional. O framework procura
    # por eles nesta ordem.

    def author_email(self, obj):
        """
        Recebe um objeto retornado por get_object() e retorna o e-mail do
        autor como uma string normal do Python.
        """

    def author_email(self):
        """
        Retorna o e-mail do autor como uma string normal do Python.
        """

    author_email = 'test@example.com' # Hard-coded author e-mail.

    # AUTHOR LINK --Pelo menos um dos três é opcional. O framework
    # procura por eles nessa ordem. Em cada caso, a URL deve incluir
    # o "http://" e o nome do domínio.

    def author_link(self, obj):
        """
        Recebe o objeto retornado por get_object() e retorna a URL do autor
        do feed como uma string normal do Python.
        """

    def author_link(self):
        """
        Retorna a URL do autor do feed como uma string normal do Python.
        """

    author_link = 'http://www.example.com/' # Hard-coded author URL.

    # CATEGORIES -- Pelo menos um dos três é opcional. O framework
    # procura por eles nessa ordem. Em cada caso, o método/atributo
    # deve retornar um objeto iterável que retorna strings.

    def categories(self, obj):
        """
        Recebe o objeto retornado por get_object() e retorna as categorias
```

```

do feed como iteráveis sobre strings.
"""

def categories(self):
    """
    Retorna as categorias do feed como iteráveis sobre strings.
    """

categories = ("python", "django") # Hard-coded list of categories.

# COPYRIGHT NOTICE -- Pelo menos um dos três é opcional. O framework
# procura por eles nessa ordem..

def copyright(self, obj):
    """
    Recebe o objeto retornado por get_object() e retorna o aviso de
    copyright do feed como uma string normal do Python.
    """

def copyright(self):
    """
    Retorna o aviso de copyright do feed como uma string normal do
    Python.
    """

copyright = 'Copyright (c) 2007, Sally Smith' # Hard-coded copyright notice.

# TTL -- Pelo menos um dos três é opcional. O framework procura por eles
# nessa ordem. Ignorado por feeds Atom.

def ttl(self, obj):
    """
    Recebe o objeto retornado por get_object() e retorna o TTL (Tempo de
    vida) do feed como uma string normal do Python.
    """

def ttl(self):
    """
    Retorna o TTL do feed como uma string normal do Python.
    """

ttl = 600 # Hard-coded Time To Live.

# ITEMS -- Pelo menos um dos três é obrigatório. O framework procura
# por eles nessa ordem.

def items(self, obj):
    """
    Recebe o objeto retornado por get_object() e retorna uma lista de
    itens para publicar nesse feed.
    """

def items(self):
    """
    Retorna uma lista de itens para publicar neste feed.
    """

items = ('Item 1', 'Item 2') # Hard-coded items.

# GET_OBJECT -- Esse é obrigatório para feeds que publicam diferentes
# dados para diferentes parâmetros de URL. (Veja "Um exemplo complexo"
# acima.)

```

```
def get_object(self, bits):
    """
    Recebe uma lista de strings recolhidos de uma URL e retorna um
    objeto representado por este feed. Lança
    django.core.exceptions.ObjectDoesNotExist sobre erro.
    """

    # ITEM LINK -- Pelo menos um dos três é obrigatório. O framework procura
    # por eles nessa ordem.

    # Primeiro, o framework tenta os dois métodos abaixo, na ordem.
    # Caso falhem, ele cai de volta para no método get_absolute_url()
    # de cada item retornado por items().

    def item_link(self, item):
        """
        Recebe um item, como retornado por items(), e retorna a URL do item.
        """

    def item_link(self):
        """
        Retorna a URL para todo item no feed.
        """

    # ITEM_GUID -- O seguinte método é opcional. Se não fornecido, o link
    # do item é usado por padrão.

    def item_guid(self, obj):
        """
        Recebe um item, como retornado por items(), e retorna o ID do item.
        """

    # ITEM_AUTHOR_NAME -- Pelo menos um dos três é opcional. O framework
    # procura por eles nessa ordem.

    def item_author_name(self, item):
        """
        Recebe um item, como retornado por items(), e retorna o nome do
        autor do feed como uma string normal do Python.
        """

    def item_author_name(self):
        """
        Retorna o nome do autor do feed para todo item do feed.
        """

    item_author_name = 'Sally Smith' # Hard-coded author name.

    # ITEM_AUTHOR E-MAIL --Pelo menos um dos três é opcional. O
    # framework procura por eles nessa ordem.
    #
    # Se você especifica isso, você pode especificar item_author_name.

    def item_author_email(self, obj):
        """
        Recebe um item, como retornado por items(), e retorna o e-mail do
        autor do feed como uma string normal do Python.
        """

    def item_author_email(self):
        """
        Retorna o e-mail do autor para todo item no feed.
        """
```

```

item_author_email = 'test@example.com' # Hard-coded author e-mail.

# ITEM AUTHOR LINK --Pelo menos um dos três é opcional. O
# framework procura por eles nessa ordem. Em cada caso, a URL deve
# incluir o "http://" e nome de domínio.
#
# Se você especificar isso, você deve especificar o item_author_name.

def item_author_link(self, obj):
    """
    Recebe um item, como retornado por items(), e retorna a URL do autor
    do item como uma string normal do Python.
    """

def item_author_link(self):
    """
    Retorna a URL do autor para todo item do feed.
    """

item_author_link = 'http://www.example.com/' # Hard-coded author URL.

# ITEM ENCLOSURE URL -- Pelo menos um dos três é obrigatório se você
# estiver publicando compartimentos. O framework procura por eles nessa
# ordem..

def item_enclosure_url(self, item):
    """
    Recebe um item, como retornado por items(), e retorna o
    URL de compartimento do item.
    """

def item_enclosure_url(self):
    """
    Retorna a URL do compartimento para todo item do feed.
    """

item_enclosure_url = "/foo/bar.mp3" # Hard-coded enclosure link.

# ITEM ENCLOSURE LENGTH -- Pelo menos um dos três é obrigatório se você
# estiver publicando compartimentos. O framework procura por eles nessa
# ordem. Em cada caso, o valor retornado deve ser um inteiro, ou uma
# representação em string de um inteiro, em bytes.

def item_enclosure_length(self, item):
    """
    Recebe um item, como retornado por items(), e retorna o comprimento
    do compartimento do item.
    """

def item_enclosure_length(self):
    """
    Retorna o comprimento do compartimento do item para todo item no
    feed.
    """

item_enclosure_length = 32000 # Hard-coded enclosure length.

# ITEM ENCLOSURE MIME TYPE -- Pelo menos um dos três é obrigatório se
# você estiver publicando compartimentos. O framework procura por eles
# nessa ordem.

def item_enclosure_mime_type(self, item):

```



```
"""
    Recebe um item, como retornado por items(), e retorna o tipo MIME do
    compartimento do item.
"""

def item_enclosure_mime_type(self):
    """
    Returns the enclosure MIME type for every item in the feed.
    """

    item_enclosure_mime_type = "audio/mpeg" # Hard-coded enclosure MIME type.

    # ITEM PUBDATE -- É opcional para usar um dos três. Este é um hook
    # que especifica como obter a data de um dado item.
    # Em cada caso, o método/atributo deve retornar um objeto
    # datetime.datetime do Python.

    def item_pubdate(self, item):
        """
        Recebe um item, como retornado por items(), e retorna o pubdate do
        item.
        """

    def item_pubdate(self):
        """
        Retorna o pubdate para todo item do feed.
        """

    item_pubdate = datetime.datetime(2005, 5, 3) # Hard-coded pubdate.

    # ITEM CATEGORIES -- É opcional usar um dos três. Este é um hook que
    # especifica como obter a lista de categorias para um dado item.
    # Em cada caso, o método/atributo deve retorna um objeto iterável que
    # retorna strings.

    def item_categories(self, item):
        """
        Recebe um item, como retornado por items(), e retorna as categorias
        do item.
        """

    def item_categories(self):
        """
        Retorna as categorias para cada item no feed.
        """

    item_categories = ("python", "django") # Hard-coded categories.

    # ITEM COPYRIGHT NOTICE (somente aplicável a feeds Atom) -- Pelo menos
    # um dos três é opcional. O framework procura por eles nessa ordem.

    def item_copyright(self, obj):
        """
        Recebe um item, como retornado por items(), e retorna o aviso de
        copyright do item como uma string normal do Python.
        """

    def item_copyright(self):
        """
        Retorna o aviso de copyright para todo item no feed.
        """

    item_copyright = 'Copyright (c) 2007, Sally Smith' # Hard-coded copyright_
↪notice.
```

O framework de baixo nível

Por trás das cenas, o framework RSS de alto nível usa o framework de baixo nível para gerar o XML do feed. Este framework fica num único módulo: [django/feeds/feedgenerator.py](#).

Você usa este framework você mesmo, para geração de feed de baixo nível. Você pode também criar um gerador de feed personalizado estendendo-o para usar com a opção `feed_type` do `Feed`.

Classes `SyndicationFeed`

O módulo `feedgenerator` contém uma classe base:

```
class django.utils.feedgenerator.SyndicationFeed
```

e várias subclasses:

```
class django.utils.feedgenerator.RssUserland091Feed
```

```
class django.utils.feedgenerator.Rss201rev2Feed
```

```
class django.utils.feedgenerator.Atom1Feed
```

Cada uma dessas três classes sabe como renderizar um certo tipo de feed como XML. Elas compartilham essa interface:

```
SyndicationFeed.__init__(**kwargs)
```

Inicia o feed com o dicionário de metadado fornecido, que é aplicado ao feed todo. Os argumentos nomeados obrigatórios são:

- `title`
- `link`
- `description`

Também existe um conjunto de outros argumentos opções:

- `language`
- `author_email`
- `author_name`
- `author_link`
- `subtitle`
- `categories`
- `feed_url`
- `feed_copyright`
- `feed_guid`
- `ttl`

Qualquer argumento extra que você passar para o `__init__` será armazenado no `self.feed` para uso com *geradores de feed personalizados*.

Todos os parâmetros devem ser objetos Unicode, exceto `categories`, que deve ser uma sequência de objetos Unicode.

```
SyndicationFeed.add_item(**kwargs)
```

Adiciona um item ao feed com os dados parâmetros.

Os Argumentos requeridos são:

- title
- link
- description

O argumentos opcionais são:

- author_email
- author_name
- author_link
- pubdate
- comments
- unique_id
- enclosure
- categories
- item_copyright
- ttl

Argumentos extra serão armazenados para *geradores de feed personalizados*.

Todos os parametros, se fornecidos, devem ser objetos Unicode, exceto:

- pubdate deve ser um objeto `datetime` do Python.
- enclosure deve ser uma instância do `feedgenerator.Enclosure`.
- categories deve ser uma sequência de objetos Unicode.

`SyndicationFeed.write(outfile, encoding)`

Mostra o feed na codificação fornecida para o outfile, que é um objeto tipo arquivo.

`SyndicationFeed.writeString(encoding)`

Retorna o feed como uma string numa dada codificação.

Por exemplo, para criar um Atom 1.0 e imprimi-lo na saída padrão:

```
>>> from django.utils import feedgenerator
>>> f = feedgenerator.Atom1Feed(
...     title=u"My Weblog",
...     link=u"http://www.example.com/",
...     description=u"In which I write about what I ate today.",
...     language=u"en")
>>> f.add_item(title=u"Hot dog today",
...             link=u"http://www.example.com/entries/1/",
...             description=u"<p>Today I had a Vienna Beef hot dog. It was pink, plump and_
↳perfect.</p>")
>>> print f.writeString('UTF-8')
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en">
...
</feed>
```

Geradores de feed personalizados

Se você precisa produzir um formato de feed personalizado, você tem algumas opções.

Se o formato do feed é totalmente personalizado, você poderia estender o `SyndicationFeed` e substituir completamente os métodos `write()` e `writeString()`.

No entanto, se o formato do feed é um spin-off de RSS ou Atom (i.e. [GeoRSS](#), [formato iTunes podcast](#) da Apple, etc.) você tem uma escolha melhor. Esses tipos de feeds tipicamente adicionam elementos extra e/ou atributos ao formato subjacente, e há um conjunto de métodos que `SyndicationFeed` chama para obter estes atributos extra. Deste modo, você pode estender a classe geradora de feed apropriada (`Atom1Feed` ou `rss201rev2Feed`) e estender estes callbacks. São eles:

`SyndicationFeed.root_attributes(self,)` Retorna um dict de atributos para adicionar ao elemento raiz do feed. (`feed/channel`).

`SyndicationFeed.add_root_elements(self, handler)` Callback para adicionar elementos dentro do elemento raiz do feed (`feed/channel`). `handler` é um [XMLGenerator](#) da biblioteca SAX built-in do Python; você poderia chamar métodos sobre ela para adicionar o documento XML no processo.

`SyndicationFeed.item_attributes(self, item)` Retorna um dict de atributos para adicionar a cada item (`item/entry`). O argumento, `item`, é um dicionário com todos os dados passados ao `SyndicationFeed.add_item()`.

`SyndicationFeed.add_item_elements(self, handler, item)` Callback para adicionar elementos a cada item (`item/entry`). `handler` e `item` são como mostrado acima.

Warning: Se você sobrescrever qualquer um destes métodos, esteja seguro de chamar os métodos da super-classe de modo que eles adicionem os elementos obrigatórios para cada formato de feed.

Por exemplo, você pode começar a implementando um gerador de feed iTunes RSS desta forma:

```
class iTunesFeed(Rss201rev2Feed):
    def root_attributes(self):
        attrs = super(iTunesFeed, self).root_attributes()
        attrs['xmlns:itunes'] = 'http://www.itunes.com/dtlds/podcast-1.0.dtd'
        return attrs

    def add_root_elements(self, handler):
        super(iTunesFeed, self).add_root_elements(handler)
        handler.addQuickElement('itunes:explicit', 'clean')
```

Obviamente há um monte de trabalho a ser feito para uma classe de feed personalizada completa, mas o exemplo acima deve demonstrar a idéia básica.

django.contrib.webdesign

O pacote `django.contrib.webdesign`, parte dos *add-ons* “[django.contrib](#)”, provê ajudantes que são particularmente úteis à *web designers* (o oposto de desenvolvedores).

Atualmente, o pacote contém uma única template tag. Se você tem idéias sobre funcionalidades para o Django se tornar mais amigável a *web designers*, por favor [sugira-as](#).

Template tags

Para usar essas template tags, adicione `'django.contrib.webdesign'` às suas `INSTALLED_APPS`. Uma vez feito, use `{% load webdesign %}` em um template para ganhar acesso as template tags.

lorem

Exibe um randômico texto latino “lorem ipsum”. Isso é útil para mostrar dados de exemplo em templates.

Uso:

```
{% lorem [count] [method] [random] %}
```

A tag `{% lorem %}` pode ser usada com zero, um, dois ou três argumentos. Os argumentos são:

Argumento	Descrição
<code>count</code>	Um número (ou variável) contendo o número de parágrafos ou palavras para gerar (o padrão é 1).
<code>method</code>	Use <code>w</code> para palavras, <code>p</code> para parágrafos HTML ou <code>b</code> para blocos de texto puro (o padrão é <code>b</code>).
<code>random</code>	A palavra <code>random</code> , se dada, não usa o parágrafo comum (“Lorem ipsum dolor sit amet...”) ao gerar o texto.

Exemplos:

- `{% lorem %}` exibirá o parágrafo “lorem ipsum” padrão.
- `{% lorem 3 p %}` exibirá o parágrafo “lorem ipsum” padrão e dois parágrafos randômicos, entre tags HTML `<p>`.
- `{% lorem 2 w random %}` exibirá 2 palavras latinas randômicas.

admin

A interface administrativa automática do Django. Para mais informações, veja [Tutorial 2](#) e a [documentação do admin](#).

Precisa que os pacotes contrib [auth](#) e [contenttypes](#) estejam instalados.

auth

O framework de autenticação do Django.

Veja [Autenticação de Usuário no Django](#).

comments

O aplicativo de comentários foi reescrito. Veja [Atualizando o sistema de comentários de um Django anterior](#) para informações sobre como atualizar seu código. Um simples e flexível sistema de comentários. Veja [Framework de comentários do Django](#).

contenttypes

Um framework leve para a conexão de “tipos” de conteúdo, onde cada model Django instalado é um tipo de conteúdo diferente.

Veja a [documentação do contenttypes](#).

csrf

Um middleware para prevenir Cross Site Request Forgeries.

Veja a [documentação csrf](#).

flatpages

Um framework para gerenciar simples conteúdo HTML “planos” no banco de dados.

Veja a *documentação flatpages*.

Precisa que o pacote contrib *sites* seja instalado também.

formtools

Um conjunto de abstrações de alto nível para os forms Django (`django.forms`).

django.contrib.formtools.preview

Uma abstração para o seguinte fluxo:

“Exibe um formulário HTML, força um preview, então faça algo com a submissão.”

Veja a *documentação do form preview*.

django.contrib.formtools.wizard

Divide um form em várias páginas.

Veja a *documentação do assistente de formulários*.

humanize

Um conjunto de template filters úteis para adicionar um “toque humano” aos dados.

Veja a *documentação do humanize*.

localflavor

Uma coleção de vários snippets que são úteis somente para um país ou cultura. Por exemplo, `django.contrib.localflavor.us.forms` contém um `USZipCodeField` que você pode usar para validar zip codes dos Estado Unidos.

Veja a *documentação localflavor*.

markup

Uma coleção de template filters que implementam as linguagens de marcação mais comuns:

- `textile` – implementa *Textile*
- `markdown` – implementa *Markdown*
- `restructuredtext` – implementa *ReST (ReStructured Text)*

Em cada caso, o filtro espera por uma marcação formatada como string e retorna uma string representando o texto processado. Por exemplo, o filtro `textile` converte texto escrito no formato Textile em HTML.

Para ativar esses filtros, adicione `'django.contrib.markup'` às suas `INSTALLED_APPS`. Uma vez adicionado, use `{% load markup %}` em um template, e você terá acesso à esses filtros. Para mais documentação, leia o código fonte em `django/contrib/markup/templatetags/markup.py`.

redirects

Um framework para gerenciar redirecionamentos.

Veja a *documentação de redirects*.

sessions

Um framework para armazenar dados em sessões anônimas.

Veja a *documentação de sessões*.

sites

Um framework leve que permite que você gerencie múltiplos web sites no mesmo banco de dados e instalação do Django. Ele te dá ganchos para associar objetos a um ou mais sites.

Veja a *documentação de sites*.

sitemaps

Um framework para gerar arquivos XML no formato de sitemap do Google.

Veja a *documentação de sitemaps*.

syndication

Um framework para gerar feed, em RSS e Atom, facilmente.

Veja a *documentação do syndication*.

webdesign

Ajudantes que são particularmente úteis à web *designers*, e não desenvolvedores web.

Veja a *documentação dos ajudantes de web design*.

Outros add-ons

Se você tem uma idéia de funcionalidade para incluir no `contrib`, nos conte! Codifique-a e poste na [django-users mailing list](#).

Notes about supported databases

Django attempts to support as many features as possible on all database backends. However, not all database backends are alike, and we've had to make design decisions on which features to support and which assumptions we can make safely.

This file describes some of the features that might be relevant to Django usage. Of course, it is not intended as a replacement for server-specific documentation or reference manuals.

PostgreSQL notes

Transaction handling

By default, Django starts a transaction when a database connection is first used and commits the result at the end of the request/response handling. The PostgreSQL backends normally operate the same as any other Django backend in this respect.

MySQL notes

Django expects the database to support transactions, referential integrity, and Unicode (UTF-8 encoding). Fortunately, [MySQL](#) has all these features as available as far back as 3.23. While it may be possible to use 3.23 or 4.0, you'll probably have less trouble if you use 4.1 or 5.0.

MySQL 4.1

[MySQL 4.1](#) has greatly improved support for character sets. It is possible to set different default character sets on the database, table, and column. Previous versions have only a server-wide character set setting. It's also the first version where the character set can be changed on the fly. 4.1 also has support for views, but Django currently doesn't use views.

MySQL 5.0

MySQL 5.0 adds the `information_schema` database, which contains detailed data on all database schema. Django's `inspectdb` feature uses this `information_schema` if it's available. 5.0 also has support for stored procedures, but Django currently doesn't use stored procedures.

Storage engines

MySQL has several [storage engines](#) (previously called table types). You can change the default storage engine in the server configuration.

The default engine is [MyISAM](#)¹. The main drawback of MyISAM is that it doesn't currently support transactions or foreign keys. On the plus side, it's currently the only engine that supports full-text indexing and searching.

The [InnoDB](#) engine is fully transactional and supports foreign key references.

The [BDB](#) engine, like InnoDB, is also fully transactional and supports foreign key references. However, its use seems to be deprecated.

Other [storage engines](#), including [SolidDB](#) and [Falcon](#), are on the horizon. For now, InnoDB is probably your best choice.

MySQLdb

[MySQLdb](#) is the Python interface to MySQL. Version 1.2.1p2 or later is required for full MySQL support in Django.

Note: If you see `ImportError: cannot import name ImmutableSet` when trying to use Django, your MySQLdb installation may contain an outdated `sets.py` file that conflicts with the built-in module of the same name from Python 2.4 and later. To fix this, verify that you have installed MySQLdb version 1.2.1p2 or newer, then delete the `sets.py` file in the MySQLdb directory that was left by an earlier version.

Creating your database

You can [create your database](#) using the command-line tools and this SQL:

```
CREATE DATABASE <dbname> CHARACTER SET utf8;
```

This ensures all tables and columns will use UTF-8 by default.

Collation settings

The collation setting for a column controls the order in which data is sorted as well as what strings compare as equal. It can be set on a database-wide level and also per-table and per-column. This is [documented thoroughly](#) in the MySQL documentation. In all cases, you set the collation by directly manipulating the database tables; Django doesn't provide a way to set this on the model definition.

By default, with a UTF-8 database, MySQL will use the `utf8_general_ci_swedish` collation. This results in all string equality comparisons being done in a *case-insensitive* manner. That is, "Fred" and "freD" are considered equal at the database level. If you have a unique constraint on a field, it would be illegal to try to insert both "aa" and "AA" into the same column, since they compare as equal (and, hence, non-unique) with the default collation.

¹ Unless this was changed by the packager of your MySQL package. We've had reports that the Windows Community Server installer sets up InnoDB as the default storage engine, for example.

In many cases, this default will not be a problem. However, if you really want case-sensitive comparisons on a particular column or table, you would change the column or table to use the `utf8_bin` collation. The main thing to be aware of in this case is that if you are using MySQLdb 1.2.2, the database backend in Django will then return bytestrings (instead of unicode strings) for any character fields it returns receive from the database. This is a strong variation from Django’s normal practice of *always* returning unicode strings. It is up to you, the developer, to handle the fact that you will receive bytestrings if you configure your table(s) to use `utf8_bin` collation. Django itself should work smoothly with such columns, but if your code must be prepared to call `django.utils.encoding.smart_unicode()` at times if it really wants to work with consistent data – Django will not do this for you (the database backend layer and the model population layer are separated internally so the database layer doesn’t know it needs to make this conversion in this one particular case).

If you’re using MySQLdb 1.2.1p2, Django’s standard `CharField` class will return unicode strings even with `utf8_bin` collation. However, `TextField` fields will be returned as an `array.array` instance (from Python’s standard `array` module). There isn’t a lot Django can do about that, since, again, the information needed to make the necessary conversions isn’t available when the data is read in from the database. This problem was [fixed in MySQLdb 1.2.2](#), so if you want to use `TextField` with `utf8_bin` collation, upgrading to version 1.2.2 and then dealing with the bytestrings (which shouldn’t be too difficult) is the recommended solution.

Should you decide to use `utf8_bin` collation for some of your tables with MySQLdb 1.2.1p2, you should still use `utf8_collation_ci_swedish` (the default) collation for the `django.contrib.sessions.models.Session` table (usually called `django_session`) and the `django.contrib.admin.models.LogEntry` table (usually called `django_admin_log`). Those are the two standard tables that use `TextField` internally.

Connecting to the database

Refer to the [settings documentation](#).

Connection settings are used in this order:

1. `DATABASE_OPTIONS`.
2. `DATABASE_NAME`, `DATABASE_USER`, `DATABASE_PASSWORD`, `DATABASE_HOST`, `DATABASE_PORT`
3. MySQL option files.

In other words, if you set the name of the database in `DATABASE_OPTIONS`, this will take precedence over `DATABASE_NAME`, which would override anything in a [MySQL option file](#).

Here’s a sample configuration which uses a MySQL option file:

```
# settings.py
DATABASE_ENGINE = "mysql"
DATABASE_OPTIONS = {
    'read_default_file': '/path/to/my.cnf',
}

# my.cnf
[client]
database = DATABASE_NAME
user = DATABASE_USER
password = DATABASE_PASSWORD
default-character-set = utf8
```

Several other MySQLdb connection options may be useful, such as `ssl`, `use_unicode`, `init_command`, and `sql_mode`. Consult the [MySQLdb documentation](#) for more details.

Creating your tables

When Django generates the schema, it doesn’t specify a storage engine, so tables will be created with whatever default storage engine your database server is configured for. The easiest solution is to set your database server’s

default storage engine to the desired engine.

If you're using a hosting service and can't change your server's default storage engine, you have a couple of options.

- After the tables are created, execute an `ALTER TABLE` statement to convert a table to a new storage engine (such as InnoDB):

```
ALTER TABLE <tablename> ENGINE=INNODB;
```

This can be tedious if you have a lot of tables.

- Another option is to use the `init_command` option for `MySQLdb` prior to creating your tables:

```
DATABASE_OPTIONS = {  
    "init_command": "SET storage_engine=INNODB",  
}
```

This sets the default storage engine upon connecting to the database. After your tables have been created, you should remove this option.

- Another method for changing the storage engine is described in [AlterModelOnSyncDB](#).

Notes on specific fields

Boolean fields

Since MySQL doesn't have a direct `BOOLEAN` column type, Django uses a `TINYINT` column with values of 1 and 0 to store values for the `BooleanField` model field. Refer to the documentation of that field for more details, but usually this won't be something that will matter unless you're printing out the field values and are expecting to see `True` and `False`.

Character fields

Any fields that are stored with `VARCHAR` column types have their `max_length` restricted to 255 characters if you are using `unique=True` for the field. This affects `CharField`, `SlugField` and `CommaSeparatedIntegerField`.

Furthermore, if you are using a version of MySQL prior to 5.0.3, all of those column types have a maximum length restriction of 255 characters, regardless of whether `unique=True` is specified or not.

SQLite notes

`SQLite` provides an excellent development alternative for applications that are predominantly read-only or require a smaller installation footprint. As with all database servers, though, there are some differences that are specific to `SQLite` that you should be aware of.

String matching for non-ASCII strings

`SQLite` doesn't support case-insensitive matching for non-ASCII strings. Some possible workarounds for this are [documented at `sqlite.org`](#), but they are not utilised by the default `SQLite` backend in Django. Therefore, if you are using the `iexact` lookup type in your queryset filters, be aware that it will not work as expected for non-ASCII strings.

SQLite 3.3.6 or newer strongly recommended

Versions of SQLite 3.3.5 and older contains a bug when [handling](#) `ORDER BY` parameters. This can cause problems when you use the `select` parameter for the `extra()` `QuerySet` method. The bug can be identified by the error message `OperationalError: ORDER BY terms must not be non-integer constants`.

SQLite 3.3.6 was released in April 2006, so most current binary distributions for different platforms include newer version of SQLite usable from Python through either the `pysqlite2` or the `sqlite3` modules.

However, some platform/Python version combinations include older versions of SQLite (e.g. the official binary distribution of Python 2.5 for Windows, 2.5.4 as of this writing, includes SQLite 3.3.4).

Version 3.5.9

The Ubuntu “Intrepid Ibex” (8.10) SQLite 3.5.9-3 package contains a bug that causes problems with the evaluation of query expressions. If you are using Ubuntu “Intrepid Ibex”, you will need to update the package to version 3.5.9-3ubuntu1 or newer (recommended) or find an alternate source for SQLite packages, or install SQLite from source.

At one time, Debian Lenny shipped with the same malfunctioning SQLite 3.5.9-3 package. However the Debian project has subsequently issued updated versions of the SQLite package that correct these bugs. If you find you are getting unexpected results under Debian, ensure you have updated your SQLite package to 3.5.9-5 or later.

The problem does not appear to exist with other versions of SQLite packaged with other operating systems.

Version 3.6.2

SQLite version 3.6.2 (released August 30, 2008) introduced a bug into `SELECT DISTINCT` handling that is triggered by, amongst other things, Django’s `DateQuerySet` (returned by the `dates()` method on a `queryset`).

You should avoid using this version of SQLite with Django. Either upgrade to 3.6.3 (released September 22, 2008) or later, or downgrade to an earlier version of SQLite.

“Database is locked” errors

SQLite is meant to be a lightweight database, and thus can’t support a high level of concurrency. `OperationalError: database is locked` errors indicate that your application is experiencing more concurrency than `sqlite` can handle in default configuration. This error means that one thread or process has an exclusive lock on the database connection and another thread timed out waiting for the lock to be released.

Python’s SQLite wrapper has a default timeout value that determines how long the second thread is allowed to wait on the lock before it times out and raises the `OperationalError: database is locked` error.

If you’re getting this error, you can solve it by:

- Switching to another database backend. At a certain point SQLite becomes too “lite” for real-world applications, and these sorts of concurrency errors indicate you’ve reached that point.
- Rewriting your code to reduce concurrency and ensure that database transactions are short-lived.
- Increase the default timeout value by setting the `timeout` database option option:

```
DATABASE_OPTIONS = {
    # ...
    "timeout": 20,
    # ...
}
```

This will simply make SQLite wait a bit longer before throwing “database is locked” errors; it won’t really do anything to solve them.

Oracle notes

Django supports [Oracle Database Server](#) versions 9i and higher. Oracle version 10g or later is required to use Django's `regex` and `iregex` query operators. You will also need at least version 4.3.1 of the `cx_Oracle` Python driver.

Note that due to a Unicode-corruption bug in `cx_Oracle` 5.0, that version of the driver should **not** be used with Django; `cx_Oracle` 5.0.1 resolved this issue, so if you'd like to use a more recent `cx_Oracle`, use version 5.0.1.

In order for the `python manage.py syncdb` command to work, your Oracle database user must have privileges to run the following commands:

- CREATE TABLE
- CREATE SEQUENCE
- CREATE PROCEDURE
- CREATE TRIGGER

To run Django's test suite, the user needs these *additional* privileges:

- CREATE USER
- DROP USER
- CREATE TABLESPACE
- DROP TABLESPACE
- CONNECT WITH ADMIN OPTION
- RESOURCE WITH ADMIN OPTION

Connecting to the database

Your Django `settings.py` file should look something like this for Oracle:

```
DATABASE_ENGINE = 'oracle'
DATABASE_NAME = 'xe'
DATABASE_USER = 'a_user'
DATABASE_PASSWORD = 'a_password'
DATABASE_HOST = ''
DATABASE_PORT = ''
```

If you don't use a `tnsnames.ora` file or a similar naming method that recognizes the SID ("xe" in this example), then fill in both `DATABASE_HOST` and `DATABASE_PORT` like so:

```
DATABASE_ENGINE = 'oracle'
DATABASE_NAME = 'xe'
DATABASE_USER = 'a_user'
DATABASE_PASSWORD = 'a_password'
DATABASE_HOST = 'dbprod01ned.mycompany.com'
DATABASE_PORT = '1540'
```

You should supply both `DATABASE_HOST` and `DATABASE_PORT`, or leave both as empty strings.

Tablespace options

A common paradigm for optimizing performance in Oracle-based systems is the use of [tablespaces](#) to organize disk layout. The Oracle backend supports this use case by adding `db_tablespace` options to the `Meta` and `Field` classes. (When you use a backend that lacks support for tablespaces, Django ignores these options.)

A tablespace can be specified for the table(s) generated by a model by supplying the `db_tablespace` option inside the model's class `Meta`. Additionally, you can pass the `db_tablespace` option to a `Field` constructor to specify an alternate tablespace for the `Field`'s column index. If no index would be created for the column, the `db_tablespace` option is ignored:

```
class TablespaceExample(models.Model):
    name = models.CharField(max_length=30, db_index=True, db_tablespace="indexes")
    data = models.CharField(max_length=255, db_index=True)
    edges = models.ManyToManyField(to="self", db_tablespace="indexes")

    class Meta:
        db_tablespace = "tables"
```

In this example, the tables generated by the `TablespaceExample` model (i.e., the model table and the many-to-many table) would be stored in the `tables` tablespace. The index for the `name` field and the indexes on the many-to-many table would be stored in the `indexes` tablespace. The `data` field would also generate an index, but no tablespace for it is specified, so it would be stored in the model tablespace `tables` by default. *Please, see the release notes* Use the `DEFAULT_TABLESPACE` and `DEFAULT_INDEX_TABLESPACE` settings to specify default values for the `db_tablespace` options. These are useful for setting a tablespace for the built-in Django apps and other applications whose code you cannot control.

Django does not create the tablespaces for you. Please refer to [Oracle's documentation](#) for details on creating and managing tablespaces.

Naming issues

Oracle imposes a name length limit of 30 characters. To accommodate this, the backend truncates database identifiers to fit, replacing the final four characters of the truncated name with a repeatable MD5 hash value.

NULL and empty strings

Django generally prefers to use the empty string (`''`) rather than `NULL`, but Oracle treats both identically. To get around this, the Oracle backend coerces the `null=True` option on fields that have the empty string as a possible value. When fetching from the database, it is assumed that a `NULL` value in one of these fields really means the empty string, and the data is silently converted to reflect this assumption.

TextField limitations

The Oracle backend stores `TextFields` as `NCLOB` columns. Oracle imposes some limitations on the usage of such LOB columns in general:

- LOB columns may not be used as primary keys.
- LOB columns may not be used in indexes.
- LOB columns may not be used in a `SELECT DISTINCT` list. This means that attempting to use the `QuerySet.distinct` method on a model that includes `TextField` columns will result in an error when run against Oracle. A workaround to this is to keep `TextField` columns out of any models that you foresee performing `distinct()` queries on, and to include the `TextField` in a related model instead.

Using a 3rd-party database backend

In addition to the officially supported databases, there are backends provided by 3rd parties that allow you to use other databases with Django:

- [Sybase SQL Anywhere](#)
- [IBM DB2](#)

- [Microsoft SQL Server 2005](#)
- [Firebird](#)
- [ODBC](#)

The Django versions and ORM features supported by these unofficial backends vary considerably. Queries regarding the specific capabilities of these unofficial backends, along with any support queries, should be directed to the support channels provided by each 3rd party project.

django-admin.py e manage.py

`django-admin.py` é o utilitário de linha de comando do Django para tarefas administrativas.

Este documento contempla tudo que ele pode fazer.

Além disso, `manage.py` é automaticamente criado em cada projeto Django. O `manage.py` é um wrapper simples em volta do `django-admin.py` que se preocupa com duas coisas por você antes de delegar tarefas ao `django-admin.py`:

- Ele coloca o pacote do seu projeto no `sys.path`.
- Ele define a variável de ambiente `DJANGO_SETTINGS_MODULE` que então aponta para o arquivo `settings.py` do seu projeto.

O script `django-admin.py` deve estar no *path* do seu sistema se você instalou o Django via o utilitário `setup.py`. Se não estiver no seu *path*, você pode encontrá-lo dentro da sua instalação Python em `site-packages/django/bin`. Considere criar um link simbólico para ele no *path* do seu sistema, como `/usr/local/bin`.

Para usuários Windows, que não possuem a funcionalidade de link simbólico disponível, você pode copiar o `django-admin.py` para um local existente em seu *path* ou editar as configurações de PATH (sob Configurações – Painel de Controle – Sistema – Avançado – Ambiente. . .) para apontar para o local instalado.

Geralmente, quando se trabalha com um único projeto Django, é mais fácil usar o `manage.py`. Usar o `django-admin.py` com `DJANGO_SETTINGS_MODULE`, ou a opção de linha de comando `--settings`, se você precisar trocar entre vários arquivos `settings` do Django.

Os exemplos de linha de comando de todo este documento usa o `django-admin.py` para ser consistente, mas qualquer exemplo pode usar o `manage.py` da mesma forma.

Uso

```
django-admin.py <subcommand> [options]
manage.py <subcommand> [options]
```

`subcommand` deve ser um dos subcomandos listados neste documento. `options`, que é opcional, deve ser zero ou mais das opções disponíveis para um dado subcomando.

Obtendo ajuda em tempo de execução

--help

Execute `django-admin.py help` para mostrar uma lista de todos os subcomandos disponíveis. Execute `django-admin.py help <subcommand>` para mostrar uma descrição do subcomando em questão e uma lista de suas opções disponíveis.

App names

Muitos subcomandos recebem uma lista de “app names”. Um “app name” é o nome básico do pacote contendo seus models. Por exemplo, se sua `INSTALLED_APPS` contém a string `'mysite.blog'`, o *app name* é `blog`.

Determinando a versão

--version

Execute `django-admin.py --version` para mostra a versão atual do Django.

Exemplos de saídas:

```
0.95
0.96
0.97-pre-SVN-6069
```

Mostrando saída de debug

--verbosity <amount>

Use `--verbosity` para especificar o montante de notificações e informações de debug que `django-admin.py` deve imprimir no console.

- 0 significa nenhuma saída.
- 1 significa saída normal (padrão).
- 2 significa saída prolixa.

Subcomandos disponíveis

cleanup

Please, see the release notes Pode ser executado como um cronjob ou diretamente para limpar dados antigos do banco de dados (somente sessões expiradas até o momento).

compilemessages

Antes do 1.0 este era o comando `“bin/compile-messages.py”`. Compila arquivos `.po` criados com `makemessages` para arquivos `.mo` para uso com o suporte embutido ao gettext. Veja *Internacionalização*.

–locale

Use as opções `--locale` ou `-l` para especificar o locale a ser processado. Se você não especificá-lo, todos os locales são processados.

Exemplo de uso:

```
django-admin.py compilemessages --locale=pt_BR
```

createcachetable

django-admin.py createcachetable <tablename>

Cria uma tabela de cache com o nome `tablename` para uso com o *backend* de cache em banco de dados.

createsuperuser

django-admin.py createsuperuser

Please, see the release notes Cria uma conta de superusuário (um usuário que tem todas as permissões). Isso é útil se você precisa criar uma conta inicial de superusuário, mas não o fez durante o `syncdb`, ou se você precisa programaticamente gerar contas de superusuário para seu(s) site(s).

Quando executar interativamente, este comando pedirá uma senha para a nova conta de super usuário. Quando executá-lo de forma não interativa, nenhuma senha será definida, e a conta de superusuário não será hábil a logar-se até que uma senha seja manualmente definida para ele.

--username

--email

O nome de usuário e endereço de e-mail para a nova conta podem ser fornecidos usando os argumentos `--username` e `--email` na linha de comando. Se algum deles não forem fornecidos, `createsuperuser` pedirá por eles quando estiver rodando interativamente.

Este comando estará disponível somente se o *sistema de autenticação* do Django (`django.contrib.auth`) estiver instalado.

dbshell

django-admin.py dbshell

Executa o cliente de linha de comando para o *engine* de banco de dados especificado em sua configuração `DATABASE_ENGINE`, com os parâmetros de conexão especificados em `DATABASE_USER`, `DATABASE_PASSWORD`, etc.

- Para PostgreSQL, rodará o cliente de linha de comando `psql`.
- Para MySQL, rodará o cliente de linha de comando `mysql`.
- Para SQLite, rodará o cliente de linha de comando `sqlite3`.

Este comando assume que os programas estão no seu `PATH`, de modo que uma simples chamada pelo nome do programa (`psql`, `mysql`, `sqlite3`) encontrará o programa no lugar certo. Não há formas de especificar a localização do programa manualmente.

diffsettings

django-admin.py diffsettings

Mostra diferenças entre as configurações atuais e as padrões do Django.

Configurações que não aparecem nas padrões são seguidas por "###". Por exemplo, a configuração padrão não define `ROOT_URLCONF`, então o `ROOT_URLCONF` é seguido de "###" na saída do `diffsettings`.

Note que as configurações padrões do Django estão em `django/conf/global_settings.py`, se você estiver curioso para ver a lista completa delas.

dumpdata

django-admin.py dumpdata <appname appname ...>

Mostra na saída padrão todos os dados no banco de dados associado às aplicações listadas.

Se nenhum nome de aplicação é fornecido, todas as aplicações instaladas serão extraídas.

A saída de `dumpdata` pode ser usada como entrada para `loaddata`.

Note que `dumpdata` usa o *manager* padrão do model para selecionar os registros a serem exportados. Se você estiver usando um *manager personalizado* como manager padrão e ele filtra algumas entradas disponíveis, nem todos os objetos serão exportados.

--exclude

Please, see the release notes Exclui uma aplicação específica das aplicações cujo conteúdo é mostrado. Por exemplo, para especificadamente excluir a aplicação *auth* da saída, você deve chamar:

```
django-admin.py dumpdata --exclude=auth
```

Se você deseja excluir várias aplicações, use várias diretivas `--exclude`:

```
django-admin.py dumpdata --exclude=auth --exclude=contenttypes
```

--format <fmt>

Por padrão, `dumpdata` formatará sua saída como JSON, mas você pode usar a opção `--format` para especificar outro formato. Os formatos atualmente suportados estão listados em *Formatos de serialização*.

--indent <num>

Por padrão, `dumpdata` exportará todos os dados em uma única linha. Isso não é fácil para humanos lerem, então você pode usar a opção `--indent` para imprimir uma bela saída com alguns espaços de indentação.

flush

Retorna o banco de dados ao estado em que ele estava imediatamente após a execução do `syncdb`. Isto significa que todos os dados serão removidos do banco de dados, quaisquer manipuladores de sincronização posterior, serão re-executados, e o fixture `initial_data` será reinstalado.

--noinput

Use a opção `--noinput` para suprimir todo prompt para o usuário, como as mensagens de confirmação "Você tem certeza?". Isso é útil se o `django-admin.py` é executado como um script autônomo.

inspectdb

Faz introspecção nas tabelas do banco de dados apontado pela configuração `DATABASE_NAME` e mostra um módulo de model do Django (um arquivo `models.py`) na saída padrão.

Use isso se você tem um banco de dados legado com o qual você gostaria de usar o Django. O script irá inspecionar o banco de dados e criar um model para cada tabela dentro dele.

Como você pode esperar, os models criados terão um atributo para todos os campos da tabela. Note que `inspectdb` tem uns poucos casos especiais em sua saída de nomes de campos:

- Se `inspectdb` não mapear um tipo de coluna para o tipo de campo do model, ele usará um `TextField` e inserirá um comentário Python `'This field type is a guess'` (traduzindo, 'Este tipo de campo é uma adivinhação'.) próximo ao campo gerado no model.
- Se o nome da coluna do banco de dados é uma palavra reservada do Python (tipo `'pass'`, `'class'` ou `'for'`), o `inspectdb` adicionará `'_field'` ao nome do atributo. Por exemplo, se uma tabela tem uma coluna `'for'`, o model gerado terá um campo `'for_field'`, com o atributo `db_column` setado para `'for'`. O `inspectdb` inserirá o comentário Python `'Field renamed because it was a Python reserved word'` (traduzindo, 'Campo renomeado porque é uma palavra reservada do Python') próximo ao campo.

Esta funcionalidade é para ser um atalho, não um gerador de models definitivo. Depois que você rodá-la, você precisará mexer nos models para personalizá-lo. Em particular, você precisará reorganizar a ordem dos models, desta forma os models que são referenciados por outros, funcionarão apropriadamente.

Chaves primárias são automaticamente introspectadas no PostgreSQL, MySQL e SQLite, nestes casos o Django coloca `primary_key=True` onde for necessário.

O `inspectdb` trabalha com PostgreSQL, MySQL e SQLite. Detecção de chaves estrangeiras somente funcionam no PostgreSQL e com certos tipos de tabelas do MySQL.

loaddata <fixture fixture ...>

Procura e carrega os conteúdos de fixtures para dentro do banco de dados.

Um *fixture* é uma coleção de arquivos que contém os conteúdos serializados do banco de dados. Cada fixture tem um nome único, e os arquivos que compreendem aos fixtures podem estar distribuídos em vários diretórios, em várias aplicações.

O Django procurará em três lugares por fixtures:

1. No diretório `fixtures` de cada aplicação instalada
2. Em qualquer diretório encontrado na configuração `FIXTURE_DIRS`
3. No caminho literal nomeado para o fixture

O Django carregará qualquer e todos os fixtures que encontrar, nos lugares que combinarem com os nomes de fixtures fornecidos.

Se o fixture nomeado tem uma extensão de arquivo, somente fixtures daquele tipo serão carregados. Por exemplo:

```
django-admin.py loaddata mydata.json
```

poderia somente carregar fixtures JSON chamados `mydata`. A extensão do fixture deve corresponder ao nome registrado do serializador (e.g., `json` ou `xml`).

Se você omitir a extensão, o Django procurará todos os tipos de fixtures disponíveis até encontrar uma combinação. Por exemplo:

```
django-admin.py loaddata mydata
```

poderia procurar por qualquer fixture de qualquer tipo chamado `mydata`. Se um diretório de fixture contendo `mydata.json`, cujo a fixture poderia ser carregada como uma fixture JSON. Entretanto, se duas fixtures com o mesmo nome mas tipos diferentes forem descobertas (por exemplo, se `mydata.json` e `mydata.xml` fossem encontradas em algum diretório de fixtures), a instalação do fixtures será abortada, e qualquer dado instalado na chamada do `loaddata` será removido do banco de dados.

Os fixtures que são nomeados podem incluir componentes de diretório. Estes diretórios serão incluídos no caminho de busca. Por exemplo:

```
django-admin.py loaddata foo/bar/mydata.json
```

poderia procurar `<appname>/fixtures/foo/bar/mydata.json` para cada aplicação instalada, `<dirname>/foo/bar/mydata.json` para cada diretório no `FIXTURE_DIRS`, e o caminho literal `foo/bar/mydata.json`.

Quando arquivos fixture são processados, os dados são salvos no banco de dados como estão. O método `save` do `model` definido e o sinal `pre_save` não são chamados.

Note que a ordem em que os arquivos de fixtures são processados é indefinida. Entretanto, todos os dados são instalados como uma transação única, então, dados de uma fixture podem referenciar dados de outra fixture. Se o backend de banco de dados suporta constraints a nível de linha, estes constraints serão checados no final da transação.

O comando `dumbdata` pode ser usado para gerar entrada para `loaddata`.

MySQL e Fixtures

Infelizmente, o MySQL não é capaz de suportar completamente todas as funcionalidades dos fixtures do Django. Se você usa tabelas MyISAM, o MySQL não suporta transações ou constraints, então você não pode fazer um rollback se vários arquivos de transação são encontrados, ou fixtures com dados de validação. Se você usa tabelas InnoDB, você não poderá ter qualquer referência nos seus arquivos de dados - o MySQL não provê um mecanismo para diferir checagens de constraints de linhas até que uma transação seja comitada.

`--verbosity`

Use `--verbosity` para especificar o quanto de notificações e informações de debug o `django-admin.py` deverá imprimir no console.

- 0 significa nenhuma saída.
- 1 significa saída normal (padrão).
- 2 significa saída prolixa.

Exemplo de uso:

```
django-admin.py loaddata --verbosity=2
```

makemessages

Antes da versão 1.0 este era o comando `bin/make-messages.py`. Executa sobre toda árvore de código do diretório atual e extrai todas as *strings* marcadas para tradução. Ele cria (ou atualiza) um arquivo de mensagem em `conf/locale` (na árvore do django) ou um diretório local (para o projeto e aplicação). Depois de fazer as mudanças nos arquivos de mensagens você precisará compilá-los com `compilemessages` para usar com o suporte ao gettext embutido. Veja [a documentação i18n](#) para detalhes.

`--all`

Use as opções `--all` ou `-a` para atualizar os arquivos de mensagens para todas os idiomas disponíveis.

Exemplo de uso:

```
django-admin.py makemessages --all
```

`--extension`

Use as opções `--extension` ou `-e` para especificar uma lista de extensões de arquivos para examinar (padrão: `".html"`).

Exemplo de uso:

```
django-admin.py makemessages --locale=de --extension xhtml
```

Separe as várias extensões com vírgulas ou usando -e ou --extension várias vezes:

```
django-admin.py makemessages --locale=de --extension=html,txt --extension xml
```

--locale

Use as opções --locale ou -l para especificar o locale a ser processado.

Exemplo de uso:

```
django-admin.py makemessages --locale=pt_BR
```

--domain

Use as opções --domain ou -d para mudar o domínio dos arquivos de mensagens. Atualmente são suportados:

- django para todo arquivo *.py e *.html (padrão)
- djangojs para arquivos *.js

--verbosity

Use --verbosity para especificar o quanto de notificações e informações de debug o django-admin.py deverá imprimir no console.

- 0 significa nenhuma saída.
- 1 significa saída normal (padrão).
- 2 significa saída prolixa.

Exemplo de uso:

```
django-admin.py makemessages --verbosity=2
```

reset <appname appname ...>

Executa o equivalente a sqlreset para uma app específica.

--noinput

Use a opção --noinput para suprimir todo prompt ao usuário, como mensagens de confirmação “Você tem certeza?”. Isso é útil se o django-admin.py for executado de forma autônoma, por um script.

runfcgi [options]

Inicia um conjunto de processos FastCGI adequados para uso com qualquer servidor Web que suporta o protocolo FastCGI. Veja a [documentação de implantação no FastCGI](#) para detalhes. Requer o módulo FastCGI do Python `flup`.

runserver

django-admin.py runserver [port or ipaddr:port]

Inicia um servidor Web leve de desenvolvimento na máquina local. Por padrão, o servidor roda na porta 8000 no endereço IP 127.0.0.1. Você pode passar um endereço IP e uma porta explicitamente.

Se você executar este script como um usuário com privilégios normais (recomendado), você pode não ter permissão para poder iniciar o servidor numa porta de número baixo. Portas de números baixos são reservadas ao super-usuário (root).

NÃO USE ESTE SERVIDOR EM AMBIENTE DE PRODUÇÃO. Ele não passou por auditorias de segurança ou testes de desempenho. (E isso vai ficar como está. Nosso negócio é fazer um framework Web, não servidores Web, portanto, improvisar este servidor para torná-lo usável em ambiente de produção está fora do escopo do Django.)

O servidor de desenvolvimento recarrega o código Python a cada request, se necessário. Você não precisa reiniciá-lo para que mudanças no código tenham efeito.

Quando iniciar o servidor, a cada vez que você mudar o código Python enquanto ele estiver rodando, o servidor validará todos os seus models instalados. (Veja o comando `validate` abaixo.) Se o validador encontrar erros, ele os imprimirá na saída padrão, mas não parará o servidor.

Você pode rodar quantos servidores você quiser, desde que estejam em portas separadas. É só executar `django-admin.py runserver` mais de uma vez.

Note que o endereço IP padrão, 127.0.0.1, não é acessível para outros computadores de sua rede. Para ter seu servidor de desenvolvimento visível por outros na rede, use seu próprio endereço IP (e.g. 192.168.2.1) ou 0.0.0.0.

--adminmedia

Use a opção `--adminmedia` para dizer ao Django onde encontrar os vários arquivos CSS e JavaScript da interface de administração do Django. Normalmente, o servidor de desenvolvimento serve estes arquivos da árvore de código do Django magicamente, mas você pode querer usar isso, caso tenha feito quaisquer mudanças nos arquivos para seu próprio site.

Exemplo de uso:

```
django-admin.py runserver --adminmedia=/tmp/new-admin-style/
```

--noreload

Use a opção `--noreload` para desabilitar o uso do auto-reloader. Isso significa que quaisquer mudanças no código Python, feitas enquanto o servidor estiver rodando, não terá efeito se o módulo em particular já foi carregado na memória.

Exemplo de uso:

```
django-admin.py runserver --noreload
```

Exemplos de uso com diferentes portas e endereços

Porta 8000 no endereço IP 127.0.0.1:

```
django-admin.py runserver
```

Porta 8000 no endereço IP 1.2.3.4:

```
django-admin.py runserver 1.2.3.4:8000
```

Porta 7000 no endereço IP 127.0.0.1:

```
django-admin.py runserver 7000
```

Porta 7000 no endereço IP 1.2.3.4:

```
django-admin.py runserver 1.2.3.4:7000
```

Servindo arquivos estáticos com o servidor de desenvolvimento

Por padrão, o servidor de desenvolvimento não serve quaisquer arquivos estáticos para o site (como arquivos CSS, imagens, coisas sob o `MEDIA_URL`, etc). Se você deseja configurar o Django para servir arquivos de mídia estáticos, leia [Como servir arquivos estáticos](#).

shell

Começa o interpretador Python interativo.

O Django usará o [IPython](#), se ele estiver instalado. Se você tem o IPython instalado e quer forçar o uso do interpretador Python “plano”, use a opção `--plain`, desta forma:

```
django-admin.py shell --plain
```

sql <appname appname ...>

Imprime a consulta SQL CREATE TABLE para as *app name* fornecidas.

sqlall <appname appname ...>

Imprime as consultas SQL CREATE TABLE e initial-data, para as *app name* fornecidas.

Leia a descrição de `sqlcustom` para uma explicação de como especificar dados iniciais.

sqlclear <appname appname ...>

Imprime a consulta SQL DROP TABLE para as *app name* fornecidas.

sqlcustom <appname appname ...>

Imprime a consulta SQL para as *app name* fornecidas.

Para cada model em cada app especificada, este comando procura pelo arquivo `<appname>/sql/<modelname>.sql`, onde `<appname>` é o nome dado da aplicação `<modelname>` é o nome do model em minúsculo. Por exemplo, se você tem uma app `news` que inclui um model `Story`, o `sqlcustom` tentará ler o arquivo `news/sql/story.sql` e irá adicionar à saída deste comando.

Cada arquivo SQL, se fornecido, é esperado que contenha SQL válido. Os arquivos SQL são entubados diretamente dentro do banco de dados depois que todas as consultas de criação de tabelas foram executadas. Use este hook SQL para fazer quaisquer modificações em tabelas, ou inserir qualquer função SQL dentro do banco de dados.

Note que a ordem em que os arquivos SQL são processados é indefinida.

sqlflush

Imprime a instrução SQL que será executada pelo comando [*flush*](#).

sqlindexes <appname appname ...>

Imprime a instrução SQL CREATE INDEX SQL para as *app name* fornecidas.

sqlreset <appname appname ...>

Imprime o SQL DROP TABLE, seguido do SQL CREATE TABLE, para as *app name* fornecidas.

sqlsequencereset <appname appname ...>

Imprime as instruções SQL para *resetar* as seqüências das *app name* fornecidas.

Seqüências são índices usados por alguns bancos de dados para rastrear o próximo número disponível automaticamente, em campos incrementais.

Use este comando para gerar SQL que consertará casos em que uma seqüência está fora de sincronia com seus dados de campos automaticamente incrementados.

startapp <appname>

Cria uma estrutura de diretório de aplicação Django com o nome fornecido, no diretório corrente.

startproject <projectname>

Cria uma estrutura de diretório de projeto Django com o nome fornecido, no diretório corrente.

Este comando é desabilitado quando a opção `--settings` para o `django-admin.py` é usado, ou quando a variável de ambiente `DJANGO_SETTINGS_MODULE` estiver definida. Para reabilitá-lo nestas situações, deve-se omitir a opção `--settings` ou anular o `DJANGO_SETTINGS_MODULE`.

syncdb

Cria as tabelas do banco de dados para todas as aplicações em `INSTALLED_APPS` cujo as tabelas ainda não foram criadas.

Use este comando quando você tiver adicionado novas aplicações ao seu projeto e quer instalá-las no banco de dados. Isso inclui quaisquer aplicações entregues com o Django, que podem estar no `INSTALLED_APPS` por padrão. Quando você iniciar um novo projeto, execute este comando para instalar as aplicações padrão.

syncdb não altera tabelas existentes

`syncdb` somente criará tabelas para os models que não foram instalados ainda. Ele *nunca* emite uma consulta `ALTER TABLE` para combinar mudanças feitas na classe do model depois da instalação. Mudanças de models e esquemas de banco de dados normalmente envolvem alguma forma de ambiguidade e, em alguns casos, o Django não adivinhará quais as mudanças corretas fazer. Há um risco de que dados críticos poderiam ser perdidos no processo.

Se você tem feito mudanças em models e deseja alterar as tabelas do banco de dados, use o comando `sql` para mostrar a nova estrutura do SQL e compare-a com o seu esquema de tabelas atual, para fazer as mudanças.

Se você estiver instalando a aplicação `django.contrib.auth`, o `syncdb` dará a você a opção de criar um superusuário imediatamente.

O `syncdb` também procurará por, e instalará, qualquer fixture chamada `initial_data` com uma extensão apropriada (e.g. `json` ou `xml`). Veja a documentação para `loaddata` para detalhes sobre a especificação de arquivos de dados de fixture.

--verbosity

Use `--verbosity` para especificar o quanto de notificações e informações de debug o `django-admin.py` deverá imprimir no console.

- 0 significa nenhuma saída.
- 1 significa saída normal (padrão).
- 2 significa saída prolixa.

Exemplo de uso:

```
django-admin.py syncdb --verbosity=2
```

--noinput

Use a opção `--noinput` para suprimir todo prompt ao usuário, como mensagens de confirmação “Você tem certeza?”. Isso é útil se o `django-admin.py` for executado de forma autônoma, por um script.

test

Executa os testes para todos os models instalados. Veja *Testando aplicações Django* para mais informações.

--noinput

Use a opção `--noinput` para suprimir todo prompt ao usuário, como mensagens de confirmação “Você tem certeza?”. Isso é útil se o `django-admin.py` for executado de forma autônoma, por um script.

--verbosity

Use `--verbosity` para especificar o quanto de notificações e informações de debug o `django-admin.py` deverá imprimir no console.

- 0 significa nenhuma saída.
- 1 significa saída normal (padrão).
- 2 significa saída prolixa.

Exemplo de uso:

```
django-admin.py test --verbosity=2
```

testserver <fixture fixture ...>

Please, see the release notes Executa um servidor de desenvolvimento Django (como no `runserver`) usando dados de fixture(s) fornecida(s).

Por exemplo, este comando:

```
django-admin.py testserver mydata.json
```

...executaria os seguintes passos:

1. Cria um banco de dados de teste, como descrito em *Testando aplicações Django*.
2. Popula o banco de dados de teste com os dados das fixtures. (Para saber mais sobre fixtures, veja a documentação para `loaddata` acima.)

3. Executa o servidor de desenvolvimento do Django (como no `runserver`), apontando para este novo banco de dados criado, ao invés do seu banco de dados de produção.

Isso é útil de diversas formas:

- Quando você estiver escrevendo *unit tests* de como seus views agem com certos dados de fixture, você pode usar `testserver` para interagir com os views no navegador Web, manualmente.
- Vamos dizer que você está desenvolvendo sua aplicação Django e tem uma cópia antiga do banco de dados com a qual você gostaria de interagir. Você pode extrair seu banco de dados para uma fixture (usando o comando `dumpdata`, explicado acima), e então usar `testserver` para rodar sua aplicação Web com estes dados. Com este arranjo, você tem a flexibilidade de interagir com seus dados de qualquer forma, sabendo que qualquer dado alterado está tendo efeito somente no seu banco de dados de teste.

Note que este servidor *não* detecta automaticamente mudanças no seu código Python (como `runserver` faz). Ele, no entanto, detecta mudanças nos seus templates.

--addrport [número da porta ou ipaddr:port]

Use `--addrport` para especificar uma porta diferente, ou endereço de IP e porta, no padrão `127.0.0.1:8000`. Este valor segue exatamente o mesmo formato e exerce a mesma função do argumento para o subcomando `runserver`.

Exemplos:

Para rodar o servidor de test na porta 7000 com o `fixture1` e `fixture2`:

```
django-admin.py testserver --addrport 7000 fixture1 fixture2
django-admin.py testserver fixture1 fixture2 --addrport 7000
```

(As regras acima são equivalente. Nós incluímos ambas para demonstrar que não importa se as opções vem antes ou depois dos argumentos de fixture.)

Para rodar sobre `1.2.3.4:7000` um `fixture test`:

```
django-admin.py testserver --addrport 1.2.3.4:7000 test
```

--verbosity

Use `--verbosity` para especificar o quanto de notificações e informações de debug o `django-admin.py` deverá imprimir no console.

- 0 significa nenhuma saída.
- 1 significa saída normal (padrão).
- 2 significa saída prolixa.

Exemplo de uso:

```
django-admin.py testserver --verbosity=2
```

validate

Valida todos os models instalados (de acordo com a configuração `INSTALLED_APPS`) e imprime erros de validação na saída padrão.

Opções padrão

Embora alguns subcomandos possam permitir suas próprias opções personalizadas, todo subcomando permite as seguintes opções:

–pythonpath

Exemplo de uso:

```
django-admin.py syncdb --pythonpath='/home/djangoprojects/myproject'
```

Adiciona o caminho do sistema de arquivos para o [caminho de busca](#) do Python. Se este não for fornecido, o `django-admin.py` usará a variável de ambiente `PYTHONPATH`.

Note que esta opção é desnecessária para o `manage.py`, pois ele se preocupa em definir o caminho do Python por você.

–settings

Exemplo de uso:

```
django-admin.py syncdb --settings=mysite.settings
```

Explicitamente especifica o módulo `settings` a se usar. O módulo `settings` deve estar na sintaxe de pacotes Python, e.g. `mysite.settings`. Se este não for fornecido, `django-admin.py` usará a variável de ambiente `DJANGO_SETTINGS_MODULE`.

Note que esta opção é desnecessária no `manage.py`, pois ele usa o `settings.py` do projeto atual por padrão.

–traceback

Exemplo de uso:

```
django-admin.py syncdb --traceback
```

Por padrão, `django-admin.py` mostrará uma mensagem de erro simples sempre que um erro ocorrer. Se você especificar `--traceback`, o `django-admin.py` mostrará uma pilha completa sempre que uma exceção for lançada.

Sutilezas extra

Sintaxe colorida

Os comandos `django-admin.py` / `manage.py` que mostram SQL na saída padrão usarão uma saída de código colorida se seu terminal suporta saída ANSI-colored. Ele não usa os códigos de cor se você estiver tunelando comandos para um outro programa.

Bash completion

Se você usa o terminal Bash, considere instalar o script de Bash completion do Django, que reside em `extras/django_bash_completion` na distribuição do Django. Ele habilita o auto completar com tab dos comandos `django-admin.py` e `manage.py`, então você pode, por instância...

- Escrever `django-admin.py`.

- Precionar [TAB] para ver todas as opções disponíveis.
- Escrever `sql`, então [TAB], para ver todas as opções disponíveis cujo nome começa com `sql`.

Veja *Escrevendo `commando` customizados para o `django-admin`* para saber como adicionar ações personalizadas.

Referência de manipulação de arquivos

O objeto `File`

`class File (file_object)`

atributos e métodos de `File`

O `File` do Django tem os seguintes atributos e métodos:

`File.name`

O nome do arquivo incluindo o caminho relativo de `MEDIA_ROOT`.

`File.path`

O caminho absoluto do arquivo no sistema de arquivos

Sistemas de armazenamento de arquivos personalizados podem não gravar arquivos localmente; arquivos gravados nesses sistemas terão o `path` com valor `None`.

`File.url`

A URL onde o arquivo pode ser obtido. Isso é frequentemente útil em *templates*; por exemplo, um snippet de um template para exibir um `Car` (veja abaixo) poderia ser assim:

```
<img src='{{ car.photo.url }}' alt='{{ car.name }}' />
```

`File.size`

O tamanho do arquivo em bytes.

`File.open (mode=None)`

Abre ou reabre o arquivo (que por definição também faz um `File.seek(0)`). O argumento `mode` segue os mesmos valores do `open()` padrão do Python.

Ao reabrir um arquivo, o `mode` irá sobrescrever o modo com quem o arquivo foi originalmente aberto; `None` significa reabrir no modo original.

`File.read (num_bytes=None)`

Lê o conteúdo do arquivo. O argumento opcional `size` é o número de bytes a serem lidos; se não for especificado, o arquivo será lido por completo.

`File.__iter__()`

Itera sobre o arquivo retornando uma linha por vez.

`File`.**chunks** (*chunk_size=None*)

Itera sobre o arquivo retornando “pedaços” de um dado tamanho. O padrão para `chunk_size` é 64 KB.

Isto é especialmente útil com arquivos muito grandes, desde que os permitem serem enviados por stream para o disco, evitando o armazenamento de todo o arquivo na memória.

`File`.**multiple_chunks** (*chunk_size=None*)

Retorna `True` se o arquivo é grande o suficiente para exigir múltiplos pedaços ou se todo o seu conteúdo pode ser acessado no `chunk_size` fornecido.

`File`.**write** (*content*)

Escreve a string especificada no conteúdo do arquivo. Dependendo do sistema de armazenamento nos bastidores, esse conteúdo pode não estar completamente escrito até `close()` ser chamado no arquivo.

`File`.**close** ()

Fecha o arquivo.

Atributos adicionais de `ImageField`

`File`.**width**

Tamanho da imagem.

`File`.**height**

Altura da image.

Métodos adicionais de arquivos anexados a objetos

Qualquer `File` que esteja associado com um objeto (como o `Car.photo`, no exemplo acima) irá ter alguns métodos extras:

`File`.**save** (*name, content, save=True*)

Salva um novo arquivo com o nome de arquivo e conteúdo fornecidos. Não irá substituir o arquivo original, mas criar um novo arquivo e atualizar o objeto para apontar pra ele. Se `save` é `True`, o método `save()` do modelo irá ser chamado assim que o arquivo é salvo. Assim, essas duas linhas:

```
>>> car.photo.save('myphoto.jpg', contents, save=False)
>>> car.save()
```

são o mesmo que essa única linha:

```
>>> car.photo.save('myphoto.jpg', contents, save=True)
```

Note que o argumento `content` deve ser uma instância da classe `File` ou uma subclasse de `File`.

`File`.**delete** (*save=True*)

Remove o arquivo da instância do modelo e deleta o arquivo em si. O argumento `save` funciona como acima.

API de armazenamento de arquivos.

`Storage.exists(name)`

`True` se um determinado arquivo já existe com `name`.

`Storage.path(name)`

O caminho do sistema de arquivos de onde o arquivo pode ser aberto usando a função padrão do Python, `open()`. Para sistemas de armazenamento que não são acessíveis do sistema de arquivo local, esta função irá gerar um erro `NotImplementedError`.

`Storage.size(name)`

Retorna o tamanho total, em bytes, do arquivo referenciado por `name`.

`Storage.url(name)`

Retorna a URL onde o conteúdo do arquivo referenciado por `name` pode ser acessado.

`Storage.open(name, mode='rb')`

Abre o arquivo dado por `name`. Note que apesar do arquivo retornado ser mesmo um objeto `File`, ele pode na verdade ser alguma subclasse. No caso de ser um sistema de armazenamento remoto, isto significa que a leitura/escrita pode ser lenta, então fique atento.

`Storage.save(name, content)`

Salva um novo arquivo usando o sistema de armazenamento, preferencialmente com o nome especificado. Se já existe um arquivo com este nome `name`, o sistema pode modificar o nome para ter um nome único. O nome real do arquivo armazenado será retornado.

O argumento `content` pode ser uma instancia de `django.core.files.File` ou de uma subclasse de `File`.

`Storage.delete(name)`

Deleta o arquivo referenciado por `name`. Este método não irá lançar uma exceção caso o arquivo não exista.

Referência detalhada da API de formulário. Para um material introdutório, veja *Trabalhando com formulários*.

A API de formulários

Sobre este documento

Este documento cobre os detalhes arenosos da API de formulário do Django. Você deve ler a *introdução ao trabalho com formulários* primeiro.

Bound e unbound formulários

A `Form` instance is either **bound** to a set of data, or **unbound**. Uma instância `Form` **bound** para um conjunto de dados, ou **unbound**.

- If it's **bound** to a set of data, it's capable of validating that data and rendering the form as HTML with the data displayed in the HTML.
- If it's **unbound**, it cannot do validation (because there's no data to validate!), but it can still render the blank form as HTML.

To create an unbound `Form` instance, simply instantiate the class:

```
>>> f = ContactForm()
```

To bind data to a form, pass the data as a dictionary as the first parameter to your `Form` class constructor:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
```

In this dictionary, the keys are the field names, which correspond to the attributes in your `Form` class. The values are the data you're trying to validate. These will usually be strings, but there's no requirement that they be strings; the type of data you pass depends on the *Field*, as we'll see in a moment.

Form.is_bound

If you need to distinguish between bound and unbound form instances at runtime, check the value of the form's `is_bound` attribute:

```
>>> f = ContactForm()
>>> f.is_bound
False
>>> f = ContactForm({'subject': 'hello'})
>>> f.is_bound
True
```

Note that passing an empty dictionary creates a *bound* form with empty data:

```
>>> f = ContactForm({})
>>> f.is_bound
True
```

If you have a bound `Form` instance and want to change the data somehow, or if you want to bind an unbound `Form` instance to some data, create another `Form` instance. There is no way to change data in a `Form` instance. Once a `Form` instance has been created, you should consider its data immutable, whether it has data or not.

Usando formulários para validar dados

Form.is_valid()

A primeira tarefa de um objeto `Form` é validar dados. Com uma instância de preenchida de um formulário, chame o método `is_valid()` para executar a validação e retornar um booleano designando se ele foi validado:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
```

Vamos tentar com algum dado inválido. Neste caso, `subject` está em branco (um erro, pois todos os campos são obrigatórios por padrão) e `sender` é um endereço de email inválido:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid e-mail address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
```

Form.errors

Acesse o atributo `errors` para pegar um dicionário com as mensagens de erro:

```
>>> f.errors
{'sender': [u'Enter a valid e-mail address.'], 'subject': [u'This field is_
↪required.']}
```

Neste dicionário, as chaves são nomes de campos, e os valores são listas de strings Unicode representando as mensagens de erro. As mensagens de erro são armazenadas na lista porque um campo pode ter mais de uma mensagem de erro.

Você pode acessar `errors` sem ter de chamar `is_valid()` primeiro. Os dados do formulário estarão validados na primeira vez que você chamar `Form.is_valid()` ou acessar o `errors`.

As rotinas de validação serão chamadas somente uma vez, indiferente de quantas vezes você acessar `errors` ou chamar `is_valid()`. Isso significa que se a validação tem efeitos de sentido único, estes efeitos somente serão disparados uma vez.

Comportamento de formulário vazios

É sem sentido validar um formulário sem dados, mas, para registro, aqui está o que acontece com formulários vazios:

```
>>> f = ContactForm()
>>> f.is_valid()
False
>>> f.errors
{}
```

Valores iniciais dinâmicos

Form.initial

Use `initial` para declarar valores iniciais de campos de formulários em tempo de execução. Por exemplo, você pode querer preencher um campo `username` com o nome de usuário da sessão atual.

Para realizar isso, use o argumento `initial` de um `Form`. Este argumento, se fornecido, deve ser um dicionário mapeando os nomes dos campos com valores iniciais. Somente inclua os campos que você estiver especificando um valor inicial; não é necessário incluir todos os campos do seu formulário. Por exemplo:

```
>>> f = ContactForm(initial={'subject': 'Hi there!'})
```

Estes valores são somente mostrados para formulários vazios, e eles não são usados como valores de recuo se um valor particular não é fornecido.

Note que se um `Field` define `initial` e você inclui `initial` quando instância o `Form`, então mais tarde `initial` terá prioridade. Neste exemplo, `initial` é fornecido a nível de campo e a nível de instância de formulário, e mais tarde terá prioridade:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='class')
...     url = forms.URLField()
...     comment = forms.CharField()
>>> f = CommentForm(initial={'name': 'instance'}, auto_id=False)
>>> print f
<tr><th>Name:</th><td><input type="text" name="name" value="instance" /></td></tr>
<tr><th>Url:</th><td><input type="text" name="url" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

Acessando dados “limpos”

Cada `Field` numa classe `Form` é responsável não somente por validar dados, mas também por limpá-los – normalizando-os para um formato consistente. Essa é funcionalidade legal, pois ela permite dados de um campo particular ser inserido de diversas formas, sempre resultando numa saída consistente.

Por exemplo, `DateTimeField` normaliza a entrada num objeto `datetime.date` do Python. Indiferentemente de você passar a ele uma string no formato `'1994-07-15'`, um objeto `datetime.date` ou um número de outros formatos, `DateTimeField` sempre será normalizado para um objeto `datetime.date` assim que ele for validado.

Uma vez que você tenha criado uma instância do `Form` com um conjunto de dados e validado-os, você pode acessar os dados limpos via atributo `cleaned_data` do objeto `Form`:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'cc_myself': True, 'message': u'Hi there', 'sender': u'foo@example.com', 'subject': u'hello'}
```

The `cleaned_data` attribute was called `clean_data` in earlier releases. Note que qualquer campo baseado em texto – como um `CharField` ou `EmailField` – sempre limpa a entrada numa string Unicode. Nós iremos cobrir as implicações da codificação mais tarde nesse documento.

Se seus dados *não* validam, sua instância do Form não terá um atributo `cleaned_data`:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid e-mail address',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> f.is_valid()
False
>>> f.cleaned_data
Traceback (most recent call last):
...
AttributeError: 'ContactForm' object has no attribute 'cleaned_data'
```

`cleaned_data` sempre conterá *somente* uma chave para campos definidos no Form, mesmo se você passar um dados extra quando você definir o Form. Neste exemplo, nós passamos um grupo de campos extras ao construtor do `ContactForm`, mas o `cleaned_data` contém somente os campos do formulário:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True,
...         'extra_field_1': 'foo',
...         'extra_field_2': 'bar',
...         'extra_field_3': 'baz'}
>>> f = ContactForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data # Doesn't contain extra_field_1, etc.
{'cc_myself': True, 'message': u'Hi there', 'sender': u'foo@example.com', 'subject': u'hello'}
```

`cleaned_data` incluirá uma chave e um valor para *todos* os campos definidos no Form, mesmo se os dados não incluem um valor por campo, pois isso não é obrigatório. Neste exemplo, o dicionário de dados não inclui um valor para o campo `nick_name`, mas `cleaned_data` o inclui, com um valor vazio:

```
>>> class OptionalPersonForm(Form):
...     first_name = CharField()
...     last_name = CharField()
...     nick_name = CharField(required=False)
>>> data = {'first_name': u'John', 'last_name': u'Lennon'}
>>> f = OptionalPersonForm(data)
>>> f.is_valid()
True
>>> f.cleaned_data
{'nick_name': u'', 'first_name': u'John', 'last_name': u'Lennon'}
```

Neste exemplo acima, o valor `cleaned_data` para `nick_name` é setado para uma string vazia, porque o `nick_name` é `CharField`, e `CharFields` tratam valores vazios como uma string vazia. Cada tipo de campo sabe como seus valores “brancos” são – e.g., para `DateField`, ele é `None` ao invés de uma string vazia. Para detalhes completos sobre cada comportamento de campo neste caso, veja a nota “Valor vazio” para cada campos na seção “classes `Field` embutidas” abaixo.

Você pode escrever código para executar validação para campos de formulário específicos (baseados nos seus nomes) ou para um formulário como um todo (considerando combinações de vários campos). Mais informações sobre isso está em *Validação de campos e formulários*.

Mostrando os formulários como HTML

A segunda tarefa de um objeto `Form` é renderizar a si mesmo como HTML. Para fazê-lo, simplesmente chame `print`:

```
>>> f = ContactForm()
>>> print f
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject"
↳ type="text" name="subject" maxlength="100" /></td></tr>
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name=
↳ "message" id="id_message" /></td></tr>
<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name=
↳ "sender" id="id_sender" /></td></tr>
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox
↳ " name="cc_myself" id="id_cc_myself" /></td></tr>
```

Se o formulário está preenchido com dados, a saída HTML incluirá esses dados apropriadamente. Por exemplo, se um campo é representado por um `<input type="text">`, o dados estará no atributo `value`. Se um campo é representado por um `<input type="checkbox">`, quando este HTML incluir `checked="checked"` se apropriado:

```
>>> data = {'subject': 'hello',
...         'message': 'Hi there',
...         'sender': 'foo@example.com',
...         'cc_myself': True}
>>> f = ContactForm(data)
>>> print f
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject"
↳ type="text" name="subject" maxlength="100" value="hello" /></td></tr>
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name=
↳ "message" id="id_message" value="Hi there" /></td></tr>
<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name=
↳ "sender" id="id_sender" value="foo@example.com" /></td></tr>
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox
↳ " name="cc_myself" id="id_cc_myself" checked="checked" /></td></tr>
```

Esta saída padrão é um tabela HTML com duas colunas, com um `<tr>` para cada campo: Note o seguinte:

- Para flexibilidade, a saída *não* inclui as tags `<table>` e `</table>`, nem inclui as tags `<form>` `</form>` ou um `<input type="submit">`. É seu trabalho fazer isso.
- Cada tipo de campo tem uma representação padrão do HTML. `CharField` e `EmailField` são representado por um `<input type="text">`. O `BooleanField` é representado por um `<input type="checkbox">`. Note estes são simplesmente padrões sensíveis; você pode especificar qual HTML usar para um dado campo usando widgets, que nós explanaremos logo.
- O nome de cada tag HTML é dado diretamente por seu atributo `name` na classe `ContactForm`.
- O texto da label de cada campo – e.g. `'Subject:'`, `'Message:'` e `'Cc myself:'` é gerado a partir do nome do campo convertendo todos os underscores em espaços e capitalizando a primeira letra. Novamente, perceba que estes são meramente padrões sensíveis; você pode também especificar labels manualmente.

- Cada texto de label é envolvido por uma tag HTML `<label>`, que aponta para o campo de formulário apropriado, via `id` do campo. Seu `id`, por sua vez, é gerado prefixado por um `'id_'` no nome do campo. Os atributo `id` e tag `<label>` são incluídos na saída por padrão, seguindo as melhores práticas, mas você pode mudar este comportamento.

Embora a saída `<table>` seja o estilo de saída padrão quando você imprime um formulário, outros estilos de saídas estão disponíveis. Cada estilo está disponível como um método no objeto `form`, e cada método de renderização retorna um objeto Unicode.

`as_p()`

`Form.as_p()` renderiza o formulário como uma série de tags `<p>`, com cada `<p>` contendo um campo:

```
>>> f = ContactForm()
>>> f.as_p()
u'<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text"
↳name="subject" maxlength="100" /></p>\n<p><label for="id_message">Message:</
↳label> <input type="text" name="message" id="id_message" /></p>\n<p><label for=
↳"id_sender">Sender:</label> <input type="text" name="sender" id="id_sender" /></
↳p>\n<p><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name=
↳"cc_myself" id="id_cc_myself" /></p>'
>>> print f.as_p()
<p><label for="id_subject">Subject:</label> <input id="id_subject" type="text"
↳name="subject" maxlength="100" /></p>
<p><label for="id_message">Message:</label> <input type="text" name="message" id=
↳"id_message" /></p>
<p><label for="id_sender">Sender:</label> <input type="text" name="sender" id="id_
↳sender" /></p>
<p><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_
↳myself" id="id_cc_myself" /></p>
```

`as_ul()`

`Form.as_ul()` renderiza o formulário como uma série de tags ``, com cada `` contendo um campo. Ele *não* inclui as tags `` ou ``, então você pode especificar quaisquer atributos sobre o ``:

```
>>> f = ContactForm()
>>> f.as_ul()
u'<li><label for="id_subject">Subject:</label> <input id="id_subject" type="text"
↳name="subject" maxlength="100" /></li>\n<li><label for="id_message">Message:</
↳label> <input type="text" name="message" id="id_message" /></li>\n<li><label for=
↳"id_sender">Sender:</label> <input type="text" name="sender" id="id_sender" /></
↳li>\n<li><label for="id_cc_myself">Cc myself:</label> <input type="checkbox"
↳name="cc_myself" id="id_cc_myself" /></li>'
>>> print f.as_ul()
<li><label for="id_subject">Subject:</label> <input id="id_subject" type="text"
↳name="subject" maxlength="100" /></li>
<li><label for="id_message">Message:</label> <input type="text" name="message" id=
↳"id_message" /></li>
<li><label for="id_sender">Sender:</label> <input type="text" name="sender" id="id_
↳sender" /></li>
<li><label for="id_cc_myself">Cc myself:</label> <input type="checkbox" name="cc_
↳myself" id="id_cc_myself" /></li>
```

`as_table()`

Finalmente, `Form.as_table()` mostrar o formulário como uma `<table>` HTML. Isso é exatamente o mesmo que o `print`. De fato, quando você executa um `print` um objeto `form`, ele chama seu método `as_table()` por trás das cenas:

```
>>> f = ContactForm()
>>> f.as_table()
u'<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject"
↳ type="text" name="subject" maxlength="100" /></td></tr>\n<tr><th><label for="id_
↳ message">Message:</label></th><td><input type="text" name="message" id="id_
↳ message" /></td></tr>\n<tr><th><label for="id_sender">Sender:</label></th><td>
↳ <input type="text" name="sender" id="id_sender" /></td></tr>\n<tr><th><label for=
↳ "id_cc_myself">Cc myself:</label></th><td><input type="checkbox" name="cc_myself
↳ " id="id_cc_myself" /></td></tr>'
>>> print f.as_table()
<tr><th><label for="id_subject">Subject:</label></th><td><input id="id_subject"
↳ type="text" name="subject" maxlength="100" /></td></tr>
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name=
↳ "message" id="id_message" /></td></tr>
<tr><th><label for="id_sender">Sender:</label></th><td><input type="text" name=
↳ "sender" id="id_sender" /></td></tr>
<tr><th><label for="id_cc_myself">Cc myself:</label></th><td><input type="checkbox
↳ " name="cc_myself" id="id_cc_myself" /></td></tr>
```

Configurando tags HTML <label>

Uma tag HTML <label> designa que um texto está associado a um elemento do formulário. Este pequeno reforço faz o formulário mais usável e acessível para dispositivos de assistência. É sempre uma boa idéia usar as tags <label>.

Por padrão, os métodos de renderização incluem atributos `id` aos elementos do formulário e envolvem as labels com tags <label>. O atributo `id` terá seu valor gerado prefixando os nomes dos campos com um `'id_'`. Este comportamento é configurável, contudo, se você deseja mudar a convenção do `id` ou remover os atributos `id` do HTML e as tags <label> inteiramente.

use o argumento `auto_id` no construtor do Form para controlar o comportamento das labels e `id`. Este argumento deve ser `True`, `False` ou uma string.

Se o `auto_id` for `False`, então o formulário mostrado não incluirá tags <label> nem atributos `id`:

```
>>> f = ContactForm(auto_id=False)
>>> print f.as_table()
<tr><th>Subject:</th><td><input type="text" name="subject" maxlength="100" /></td>
↳ </tr>
<tr><th>Message:</th><td><input type="text" name="message" /></td></tr>
<tr><th>Sender:</th><td><input type="text" name="sender" /></td></tr>
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_myself" /></td></tr>
>>> print f.as_ul()
<li>Subject: <input type="text" name="subject" maxlength="100" /></li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="text" name="sender" /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
>>> print f.as_p()
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" /></p>
<p>Sender: <input type="text" name="sender" /></p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

Se o `auto_id` for `True`, então o formulário mostrado **incluirá** tags <label> e simplesmente usará o nome dos campos como seu `id` para cada campo do formulário:

```
>>> f = ContactForm(auto_id=True)
>>> print f.as_table()
<tr><th><label for="subject">Subject:</label></th><td><input id="subject" type=
↳ "text" name="subject" maxlength="100" /></td></tr>
<tr><th><label for="message">Message:</label></th><td><input type="text" name=
↳ "message" id="message" /></td></tr>
```



```

<tr><th><label for="sender">Sender:</label></th><td><input type="text" name="sender"
↳ id="sender" /></td></tr>
<tr><th><label for="cc_myself">Cc myself:</label></th><td><input type="checkbox"
↳ name="cc_myself" id="cc_myself" /></td></tr>
>>> print f.as_ul()
<li><label for="subject">Subject:</label> <input id="subject" type="text" name=
↳ "subject" maxlength="100" /></li>
<li><label for="message">Message:</label> <input type="text" name="message" id=
↳ "message" /></li>
<li><label for="sender">Sender:</label> <input type="text" name="sender" id="sender"
↳ /></li>
<li><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_
↳ myself" id="cc_myself" /></li>
>>> print f.as_p()
<p><label for="subject">Subject:</label> <input id="subject" type="text" name=
↳ "subject" maxlength="100" /></p>
<p><label for="message">Message:</label> <input type="text" name="message" id=
↳ "message" /></p>
<p><label for="sender">Sender:</label> <input type="text" name="sender" id="sender"
↳ /></p>
<p><label for="cc_myself">Cc myself:</label> <input type="checkbox" name="cc_myself"
↳ id="cc_myself" /></p>

```

Se `auto_id` for uma string contendo o caracter de formatação '%s', então o formulário mostrado incluirá as tags `<label>`, e será gerado um atributo `id` baseado na string passada. Por exemplo, para uma string de formatação `'field_%s'`, um nome de campo `subject` terá o valor do `id` igual a `'field_subject'`. Continuando nosso exemplo:

```

>>> f = ContactForm(auto_id='id_for_%s')
>>> print f.as_table()
<tr><th><label for="id_for_subject">Subject:</label></th><td><input id="id_for_
↳ subject" type="text" name="subject" maxlength="100" /></td></tr>
<tr><th><label for="id_for_message">Message:</label></th><td><input type="text"
↳ name="message" id="id_for_message" /></td></tr>
<tr><th><label for="id_for_sender">Sender:</label></th><td><input type="text" name=
↳ "sender" id="id_for_sender" /></td></tr>
<tr><th><label for="id_for_cc_myself">Cc myself:</label></th><td><input type=
↳ "checkbox" name="cc_myself" id="id_for_cc_myself" /></td></tr>
>>> print f.as_ul()
<li><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type=
↳ "text" name="subject" maxlength="100" /></li>
<li><label for="id_for_message">Message:</label> <input type="text" name="message"
↳ id="id_for_message" /></li>
<li><label for="id_for_sender">Sender:</label> <input type="text" name="sender" id=
↳ "id_for_sender" /></li>
<li><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name=
↳ "cc_myself" id="id_for_cc_myself" /></li>
>>> print f.as_p()
<p><label for="id_for_subject">Subject:</label> <input id="id_for_subject" type=
↳ "text" name="subject" maxlength="100" /></p>
<p><label for="id_for_message">Message:</label> <input type="text" name="message"
↳ id="id_for_message" /></p>
<p><label for="id_for_sender">Sender:</label> <input type="text" name="sender" id=
↳ "id_for_sender" /></p>
<p><label for="id_for_cc_myself">Cc myself:</label> <input type="checkbox" name=
↳ "cc_myself" id="id_for_cc_myself" /></p>

```

Se o `auto_id` for setado para qualquer outro valor verdadeiro – como uma string que não inclui `%s` – então a biblioteca agirá como se o `auto_id` fosse `True`.

Por padrão, `auto_id` é setado como uma string `'id_%s'`.

Normalmente, dois pontos (:) serão adicionados após qualquer nome de label quando um formulário for ren-

derizado. É possível mudar os dois pontos por outro caracter, ou omiti-lo inteiramente, usando o parâmetro `label_suffix`:

```
>>> f = ContactForm(auto_id='id_for_%s', label_suffix='')
>>> print f.as_ul()
<li><label for="id_for_subject">Subject</label> <input id="id_for_subject" type=
↪"text" name="subject" maxlength="100" /></li>
<li><label for="id_for_message">Message</label> <input type="text" name="message"
↪id="id_for_message" /></li>
<li><label for="id_for_sender">Sender</label> <input type="text" name="sender" id=
↪"id_for_sender" /></li>
<li><label for="id_for_cc_myself">Cc myself</label> <input type="checkbox" name=
↪"cc_myself" id="id_for_cc_myself" /></li>
>>> f = ContactForm(auto_id='id_for_%s', label_suffix=' ->')
>>> print f.as_ul()
<li><label for="id_for_subject">Subject -></label> <input id="id_for_subject" type=
↪"text" name="subject" maxlength="100" /></li>
<li><label for="id_for_message">Message -></label> <input type="text" name="message
↪" id="id_for_message" /></li>
<li><label for="id_for_sender">Sender -></label> <input type="text" name="sender"
↪id="id_for_sender" /></li>
<li><label for="id_for_cc_myself">Cc myself -></label> <input type="checkbox" name=
↪"cc_myself" id="id_for_cc_myself" /></li>
```

Note que o sufixo da label é adicionado somente se o último caracter da label não é um caracter de pontuação (., !, ? or :)

Notas sobre o campo de ordenação

Nos atalhos `as_p()`, `as_ul()` e `as_table()`, os campos são mostrados na ordem em que você definir na sua classe form. Por exemplo, no exemplo `ContactForm`, os campos são definidos nessa ordem: `subject`, `message`, `sender`, `cc_myself`. Para re-ordenar a saída HTML, é só mudar a ordem em que esses campos foram listados na classe.

Como os erros são mostrados

If you render a bound Form object, the act of rendering will automatically run the form's validation if it hasn't already happened, and the HTML output will include the validation errors as a `<ul class="errorlist">` near the field. The particular positioning of the error messages depends on the output method you're using:

```
>>> data = {'subject': '',
...         'message': 'Hi there',
...         'sender': 'invalid e-mail address',
...         'cc_myself': True}
>>> f = ContactForm(data, auto_id=False)
>>> print f.as_table()
<tr><th>Subject:</th><td><ul class="errorlist"><li>This field is required.</li></
↪ul><input type="text" name="subject" maxlength="100" /></td></tr>
<tr><th>Message:</th><td><input type="text" name="message" value="Hi there" /></td>
↪</tr>
<tr><th>Sender:</th><td><ul class="errorlist"><li>Enter a valid e-mail address.</
↪li></ul><input type="text" name="sender" value="invalid e-mail address" /></td></
↪tr>
<tr><th>Cc myself:</th><td><input checked="checked" type="checkbox" name="cc_myself
↪" /></td></tr>
>>> print f.as_ul()
<li><ul class="errorlist"><li>This field is required.</li></ul>Subject: <input
↪type="text" name="subject" maxlength="100" /></li>
<li>Message: <input type="text" name="message" value="Hi there" /></li>
<li><ul class="errorlist"><li>Enter a valid e-mail address.</li></ul>Sender:
↪<input type="text" name="sender" value="invalid e-mail address" /></li>
```

```
<li>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></li>
>>> print f.as_p()
<p><ul class="errorlist"><li>This field is required.</li></ul></p>
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" value="Hi there" /></p>
<p><ul class="errorlist"><li>Enter a valid e-mail address.</li></ul></p>
<p>Sender: <input type="text" name="sender" value="invalid e-mail address" /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></p>
```

Personalização do formato da lista de erros

Por padrão, formulários usam `django.forms.util.ErrorList` para formatar os erros de validação. Se você gostaria de usar uma classe alternativa para mostrar os erros, você pode passá-la na hora da construção:

```
>>> from django.forms.util import ErrorList
>>> class DivErrorList(ErrorList):
...     def __unicode__(self):
...         return self.as_divs()
...     def as_divs(self):
...         if not self: return u''
...         return u'<div class="errorlist">%s</div>' % ''.join([u'<div class=
↪"error">%s</div>' % e for e in self])
>>> f = ContactForm(data, auto_id=False, error_class=DivErrorList)
>>> f.as_p()
<div class="errorlist"><div class="error">This field is required.</div></div>
<p>Subject: <input type="text" name="subject" maxlength="100" /></p>
<p>Message: <input type="text" name="message" value="Hi there" /></p>
<div class="errorlist"><div class="error">Enter a valid e-mail address.</div></div>
<p>Sender: <input type="text" name="sender" value="invalid e-mail address" /></p>
<p>Cc myself: <input checked="checked" type="checkbox" name="cc_myself" /></p>
```

Saída mais granular

Os métodos `as_p()`, `as_ul()` e `as_table()` são simplesmente atalhos para desenvolvedores preguiçosos – eles não são a única forma de um objeto form ser mostrado.

Para mostrar o HTML para um único campo no seu formulário, use a sintaxe de acesso de dicionário, usando o nome do campo como chave, e imprimindo o objeto resultante:

```
>>> f = ContactForm()
>>> print f['subject']
<input id="id_subject" type="text" name="subject" maxlength="100" />
>>> print f['message']
<input type="text" name="message" id="id_message" />
>>> print f['sender']
<input type="text" name="sender" id="id_sender" />
>>> print f['cc_myself']
<input type="checkbox" name="cc_myself" id="id_cc_myself" />
```

Chame `str()` ou `unicode()` sobre um campo para obter seu HTML renderizado como uma string ou objeto Unicode, respectivamente:

```
>>> str(f['subject'])
'<input id="id_subject" type="text" name="subject" maxlength="100" />'
>>> unicode(f['subject'])
u'<input id="id_subject" type="text" name="subject" maxlength="100" />'
```

Objetos de formulário definem um método personalizado `__iter__()`, que permite você iterar sobre seus campos:

```
>>> f = ContactForm()
>>> for field in f: print field
<input id="id_subject" type="text" name="subject" maxlength="100" />
<input type="text" name="message" id="id_message" />
<input type="text" name="sender" id="id_sender" />
<input type="checkbox" name="cc_myself" id="id_cc_myself" />
```

A saída de um campo específico honra a configuração `auto_id` de objetos de formulários:

```
>>> f = ContactForm(auto_id=False)
>>> print f['message']
<input type="text" name="message" />
>>> f = ContactForm(auto_id='id_%s')
>>> print f['message']
<input type="text" name="message" id="id_message" />
```

Para uma lista de erros dos campos, acesse o atributo `errors` dos campos. Este é um objeto tipo lista que é mostrado como um HTML `<ul class="errorlist">` quando impresso:

```
>>> data = {'subject': 'hi', 'message': '', 'sender': '', 'cc_myself': ''}
>>> f = ContactForm(data, auto_id=False)
>>> print f['message']
<input type="text" name="message" />
>>> f['message'].errors
[u'This field is required.']
>>> print f['message'].errors
<ul class="errorlist"><li>This field is required.</li></ul>
>>> f['subject'].errors
[]
>>> print f['subject'].errors

>>> str(f['subject'].errors)
''
```

Vinculando arquivos enviados a um formulário

Please, see the release notes Lidar com formulários que possuem campos `FileField` e `ImageField` é um pouco mais complicado que um formulário normal.

Primeiramente, a fim de fazer o upload de arquivos, você precisará estar certo de que seu elemento `<form>` define corretamente o atributo `enctype` como `"multipart/form-data"`:

```
<form enctype="multipart/form-data" method="post" action="/foo/">
```

Depois, quando você usar o formulário, você precisa tratar os dados do arquivo. Dados de arquivos são manipulados separadamente dos dados normais de formulário, então quando seu formulário contiver um `FileField` e `ImageField`, você precisará especificar um segundo argumento quando for vincular o formulário. Então se nós extendemos nosso `ContactForm` para incluir um `ImageField` chamado `mugshot`, nós precisamos vincular os dados do arquivo que contém a imagem `mugshot`:

```
# Bound form with an image field
>>> from django.core.files.uploadedfile import SimpleUploadedFile
>>> data = {'subject': 'hello',
...        'message': 'Hi there',
...        'sender': 'foo@example.com',
...        'cc_myself': True}
>>> file_data = {'mugshot': SimpleUploadedFile('face.jpg', <file data>)}
>>> f = ContactFormWithMugshot(data, file_data)
```

Na prática, você normalmente especificará `request.FILES` como a fonte de dados do arquivo (da mesma forma do `request.POST` como a fonte de dados de formulários):

```
# Bound form with an image field, data from the request
>>> f = ContactFormWithMugshot(request.POST, request.FILES)
```

Constructing an unbound form is the same as always – just omit both form data *and* file data:

```
# Unbound form with a image field
>>> f = ContactFormWithMugshot()
```

Testando formulários multipart

Se você está escrevendo views ou templates reusáveis, você pode não saber, antes do tempo, se seu formulário é um formulário multipart ou não. O método `is_multipart()` diz a você se o formulário requer codificação multipart para sua submissão:

```
>>> f = ContactFormWithMugshot()
>>> f.is_multipart()
True
```

Aqui temos um exemplo de como você pode usar isso num template:

```
{% if form.is_multipart %}
    <form enctype="multipart/form-data" method="post" action="/foo/">
{% else %}
    <form method="post" action="/foo/">
{% endif %}
{{ form }}
</form>
```

Extendendo formulários

Se você tem várias classes `Form` que compartilham campos, você pode usar herança para eliminar a redundância.

Quando você estende uma classe `Form` personalizada, à subclasse resultante será incluso todos os campos da(s) classe(s) pai, seguido pelos campos que você definiu na subclasse.

Neste exemplo, o `ContactFormWithPriority` contém todos os campos do `ContactForm`, mais um campo adicional, `priority`. Os campos do `ContactForm` são ordenados primeiro:

```
>>> class ContactFormWithPriority(ContactForm):
...     priority = forms.CharField()
>>> f = ContactFormWithPriority(auto_id=False)
>>> print f.as_ul()
<li>Subject: <input type="text" name="subject" maxlength="100" /></li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="text" name="sender" /></li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
<li>Priority: <input type="text" name="priority" /></li>
```

É possível estender vários formulários, tratando formulários como “mix-ins”. Neste exemplo, `BeatleForm` estende ambos `PersonForm` e `InstrumentForm` (nessa ordem), e sua lista de campos inclui os campos das classes pai:

```
>>> class PersonForm(Form):
...     first_name = CharField()
...     last_name = CharField()
>>> class InstrumentForm(Form):
...     instrument = CharField()
>>> class BeatleForm(PersonForm, InstrumentForm):
...     haircut_type = CharField()
```

```
>>> b = BeatleForm(auto_id=False)
>>> print b.as_ul()
<li>First name: <input type="text" name="first_name" /></li>
<li>Last name: <input type="text" name="last_name" /></li>
<li>Instrument: <input type="text" name="instrument" /></li>
<li>Haircut type: <input type="text" name="haircut_type" /></li>
```

Prefixes for forms

Form.`prefix`

Você pode colocar vários formulários Django dentro de uma tag `<form>`. Para dar a cada Form seu próprio namespace, use o argumento `prefix`:

```
>>> mother = PersonForm(prefix="mother")
>>> father = PersonForm(prefix="father")
>>> print mother.as_ul()
<li><label for="id_mother-first_name">First name:</label> <input type="text" name=
↪ "mother-first_name" id="id_mother-first_name" /></li>
<li><label for="id_mother-last_name">Last name:</label> <input type="text" name=
↪ "mother-last_name" id="id_mother-last_name" /></li>
>>> print father.as_ul()
<li><label for="id_father-first_name">First name:</label> <input type="text" name=
↪ "father-first_name" id="id_father-first_name" /></li>
<li><label for="id_father-last_name">Last name:</label> <input type="text" name=
↪ "father-last_name" id="id_father-last_name" /></li>
```

Form fields

`class Field(**kwargs)`

When you create a Form class, the most important part is defining the fields of the form. Each field has custom validation logic, along with a few other hooks.

Field.`clean`(*value*)

Although the primary way you'll use Field classes is in Form classes, you can also instantiate them and use them directly to get a better idea of how they work. Each Field instance has a `clean()` method, which takes a single argument and either raises a `django.forms.ValidationError` exception or returns the clean value:

```
>>> from django import forms
>>> f = forms.EmailField()
>>> f.clean('foo@example.com')
u'foo@example.com'
>>> f.clean(u'foo@example.com')
u'foo@example.com'
>>> f.clean('invalid e-mail address')
Traceback (most recent call last):
...
ValidationError: [u'Enter a valid e-mail address.']
```

Core field arguments

Each Field class constructor takes at least these arguments. Some Field classes take additional, field-specific arguments, but the following should *always* be accepted:

required

Field.required

By default, each `Field` class assumes the value is required, so if you pass an empty value – either `None` or the empty string `""` – then `clean()` will raise a `ValidationError` exception:

```
>>> f = forms.CharField()
>>> f.clean('foo')
u'foo'
>>> f.clean('')
Traceback (most recent call last):
...
ValidationError: [u'This field is required.']
>>> f.clean(None)
Traceback (most recent call last):
...
ValidationError: [u'This field is required.']
>>> f.clean(' ')
u' '
>>> f.clean(0)
u'0'
>>> f.clean(True)
u'True'
>>> f.clean(False)
u'False'
```

To specify that a field is *not* required, pass `required=False` to the `Field` constructor:

```
>>> f = forms.CharField(required=False)
>>> f.clean('foo')
u'foo'
>>> f.clean('')
u''
>>> f.clean(None)
u''
>>> f.clean(0)
u'0'
>>> f.clean(True)
u'True'
>>> f.clean(False)
u'False'
```

If a `Field` has `required=False` and you pass `clean()` an empty value, then `clean()` will return a *normalized* empty value rather than raising `ValidationError`. For `CharField`, this will be a Unicode empty string. For other `Field` classes, it might be `None`. (This varies from field to field.)

label

Field.label

The `label` argument lets you specify the “human-friendly” label for this field. This is used when the `Field` is displayed in a `Form`.

As explained in “Outputting forms as HTML” above, the default label for a `Field` is generated from the field name by converting all underscores to spaces and upper-casing the first letter. Specify `label` if that default behavior doesn’t result in an adequate label.

Here’s a full example `Form` that implements `label` for two of its fields. We’ve specified `auto_id=False` to simplify the output:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(label='Your name')
```

```
...     url = forms.URLField(label='Your Web site', required=False)
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print f
<tr><th>Your name:</th><td><input type="text" name="name" /></td></tr>
<tr><th>Your Web site:</th><td><input type="text" name="url" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

initial

Field.initial

The `initial` argument lets you specify the initial value to use when rendering this `Field` in an unbound `Form`.

To specify dynamic initial data, see the `Form.initial` parameter.

The use-case for this is when you want to display an “empty” form in which a field is initialized to a particular value. For example:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> f = CommentForm(auto_id=False)
>>> print f
<tr><th>Name:</th><td><input type="text" name="name" value="Your name" /></td></tr>
<tr><th>Url:</th><td><input type="text" name="url" value="http://" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

You may be thinking, why not just pass a dictionary of the initial values as data when displaying the form? Well, if you do that, you’ll trigger validation, and the HTML output will include any validation errors:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField()
...     url = forms.URLField()
...     comment = forms.CharField()
>>> default_data = {'name': 'Your name', 'url': 'http://'}
>>> f = CommentForm(default_data, auto_id=False)
>>> print f
<tr><th>Name:</th><td><input type="text" name="name" value="Your name" /></td></tr>
<tr><th>Url:</th><td><ul class="errorlist"><li>Enter a valid URL.</li></ul><input
↪ type="text" name="url" value="http://" /></td></tr>
<tr><th>Comment:</th><td><ul class="errorlist"><li>This field is required.</li></
↪ ul><input type="text" name="comment" /></td></tr>
```

This is why initial values are only displayed for unbound forms. For bound forms, the HTML output will use the bound data.

Also note that initial values are *not* used as “fallback” data in validation if a particular field’s value is not given. initial values are *only* intended for initial form display:

```
>>> class CommentForm(forms.Form):
...     name = forms.CharField(initial='Your name')
...     url = forms.URLField(initial='http://')
...     comment = forms.CharField()
>>> data = {'name': '', 'url': '', 'comment': 'Foo'}
>>> f = CommentForm(data)
>>> f.is_valid()
False
# The form does *not* fall back to using the initial values.
>>> f.errors
{'url': [u'This field is required.'], 'name': [u'This field is required.']}
```


Instead of a constant, you can also pass any callable:

```
>>> import datetime
>>> class DateForm(forms.Form):
...     day = forms.DateField(initial=datetime.date.today)
>>> print DateForm()
<tr><th>Day:</th><td><input type="text" name="day" value="12/23/2008" /></td></tr>
```

The callable will be evaluated only when the unbound form is displayed, not when it is defined.

widget

Field.widget

The `widget` argument lets you specify a `Widget` class to use when rendering this `Field`. See [Widgets](#) for more information.

help_text

Field.help_text

The `help_text` argument lets you specify descriptive text for this `Field`. If you provide `help_text`, it will be displayed next to the `Field` when the `Field` is rendered by one of the convenience Form methods (e.g., `as_ul()`).

Here's a full example Form that implements `help_text` for two of its fields. We've specified `auto_id=False` to simplify the output:

```
>>> class HelpTextContactForm(forms.Form):
...     subject = forms.CharField(max_length=100, help_text='100 characters max.')
...     message = forms.CharField()
...     sender = forms.EmailField(help_text='A valid e-mail address, please.')
...     cc_myself = forms.BooleanField(required=False)
>>> f = HelpTextContactForm(auto_id=False)
>>> print f.as_table()
<tr><th>Subject:</th><td><input type="text" name="subject" maxlength="100" /><br />
↪100 characters max.</td></tr>
<tr><th>Message:</th><td><input type="text" name="message" /></td></tr>
<tr><th>Sender:</th><td><input type="text" name="sender" /><br />A valid e-mail_
↪address, please.</td></tr>
<tr><th>Cc myself:</th><td><input type="checkbox" name="cc_myself" /></td></tr>
>>> print f.as_ul()
<li>Subject: <input type="text" name="subject" maxlength="100" /> 100 characters_
↪max.</li>
<li>Message: <input type="text" name="message" /></li>
<li>Sender: <input type="text" name="sender" /> A valid e-mail address, please.</
↪li>
<li>Cc myself: <input type="checkbox" name="cc_myself" /></li>
>>> print f.as_p()
<p>Subject: <input type="text" name="subject" maxlength="100" /> 100 characters_
↪max.</p>
<p>Message: <input type="text" name="message" /></p>
<p>Sender: <input type="text" name="sender" /> A valid e-mail address, please.</p>
<p>Cc myself: <input type="checkbox" name="cc_myself" /></p>
```

error_messages

Please, see the release notes

Field.error_messages

The `error_messages` argument lets you override the default messages that the field will raise. Pass in a dictionary with keys matching the error messages you want to override. For example, here is the default error message:

```
>>> generic = forms.CharField()
>>> generic.clean('')
Traceback (most recent call last):
...
ValidationError: [u'This field is required.']
```

And here is a custom error message:

```
>>> name = forms.CharField(error_messages={'required': 'Please enter your name'})
>>> name.clean('')
Traceback (most recent call last):
...
ValidationError: [u'Please enter your name']
```

In the *built-in Field classes* section below, each `Field` defines the error message keys it uses.

Built-in Field classes

Naturally, the `forms` library comes with a set of `Field` classes that represent common validation needs. This section documents each built-in field.

For each field, we describe the default widget used if you don't specify `widget`. We also specify the value returned when you provide an empty value (see the section on `required` above to understand what that means).

BooleanField

class BooleanField (***kwargs*)

- Default widget: `CheckboxInput`
- Empty value: `False`
- Normalizes to: A Python `True` or `False` value.
- Validates that the value is `True` (e.g. the check box is checked) if the field has `required=True`.
- Error message keys: `required`

The empty value for a `CheckboxInput` (and hence the standard `BooleanField`) has changed to return `False` instead of `None` in the Django 1.0.

Note: Since all `Field` subclasses have `required=True` by default, the validation condition here is important. If you want to include a boolean in your form that can be either `True` or `False` (e.g. a checked or unchecked checkbox), you must remember to pass in `required=False` when creating the `BooleanField`.

CharField

class CharField (***kwargs*)

- Default widget: `TextInput`
- Empty value: `' '` (an empty string)
- Normalizes to: A Unicode object.
- Validates `max_length` or `min_length`, if they are provided. Otherwise, all inputs are valid.
- Error message keys: `required`, `max_length`, `min_length`

Has two optional arguments for validation:

`CharField.max_length`

`CharField.min_length`

If provided, these arguments ensure that the string is at most or at least the given length.

ChoiceField

class ChoiceField (***kwargs*)

- Default widget: `Select`
- Empty value: `' '` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value exists in the list of choices.
- Error message keys: `required`, `invalid_choice`

Takes one extra required argument:

`ChoiceField.choices`

An iterable (e.g., a list or tuple) of 2-tuples to use as choices for this field.

TypedChoiceField

class TypedChoiceField (***kwargs*)

Just like a *ChoiceField*, except *TypedChoiceField* takes an extra `coerce` argument.

- Default widget: `Select`
- Empty value: Whatever you’ve given as `empty_value`
- Normalizes to: the value returned by the `coerce` argument.
- Validates that the given value exists in the list of choices.
- Error message keys: `required`, `invalid_choice`

Takes extra arguments:

`TypedChoiceField.coerce`

A function that takes one argument and returns a coerced value. Examples include the built-in `int`, `float`, `bool` and other types. Defaults to an identity function.

`TypedChoiceField.empty_value`

The value to use to represent “empty.” Defaults to the empty string; `None` is another common choice here.

DateField

class DateField (***kwargs*)

- Default widget: `TextInput`
- Empty value: `None`
- Normalizes to: A Python `datetime.date` object.
- Validates that the given value is either a `datetime.date`, `datetime.datetime` or string formatted in a particular date format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

DateField.input_formats

A list of formats used to attempt to convert a string to a valid `datetime.date` object.

If no `input_formats` argument is provided, the default input formats are:

```
'%Y-%m-%d', '%m/%d/%Y', '%m/%d/%y', # '2006-10-25', '10/25/2006', '10/25/06'
'%b %d %Y', '%b %d, %Y',             # 'Oct 25 2006', 'Oct 25, 2006'
'%d %b %Y', '%d %b, %Y',             # '25 Oct 2006', '25 Oct, 2006'
'%B %d %Y', '%B %d, %Y',             # 'October 25 2006', 'October 25, 2006'
'%d %B %Y', '%d %B, %Y',             # '25 October 2006', '25 October, 2006'
```

DateTimeField

```
class DateTimeField(**kwargs)
```

- Default widget: `DateTimeInput`
- Empty value: `None`
- Normalizes to: A Python `datetime.datetime` object.
- Validates that the given value is either a `datetime.datetime`, `datetime.date` or string formatted in a particular datetime format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

DateTimeField.input_formats

A list of formats used to attempt to convert a string to a valid `datetime.datetime` object.

If no `input_formats` argument is provided, the default input formats are:

```
'%Y-%m-%d %H:%M:%S', # '2006-10-25 14:30:59'
'%Y-%m-%d %H:%M',    # '2006-10-25 14:30'
'%Y-%m-%d',          # '2006-10-25'
'%m/%d/%Y %H:%M:%S', # '10/25/2006 14:30:59'
'%m/%d/%Y %H:%M',    # '10/25/2006 14:30'
'%m/%d/%Y',          # '10/25/2006'
'%m/%d/%y %H:%M:%S', # '10/25/06 14:30:59'
'%m/%d/%y %H:%M',    # '10/25/06 14:30'
'%m/%d/%y',          # '10/25/06'
```

The `DateTimeField` used to use a `TextInput` widget by default. This has now changed.

DecimalField

Please, see the release notes

```
class DecimalField(**kwargs)
```

- Default widget: `TextInput`
- Empty value: `None`
- Normalizes to: A Python decimal.
- Validates that the given value is a decimal. Leading and trailing whitespace is ignored.
- Error message keys: `required`, `invalid`, `max_value`, `min_value`, `max_digits`, `max_decimal_places`, `max_whole_digits`

Takes four optional arguments:

```
DecimalField.max_value
```

`DecimalField.min_value`

These attributes define the limits for the fields value.

`DecimalField.max_digits`

The maximum number of digits (those before the decimal point plus those after the decimal point, with leading zeros stripped) permitted in the value.

`DecimalField.decimal_places`

The maximum number of decimal places permitted.

`EmailField`

class `EmailField` (***kwargs*)

- Default widget: `TextInput`
- Empty value: `' '` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value is a valid e-mail address, using a moderately complex regular expression.
- Error message keys: `required`, `invalid`

Has two optional arguments for validation, `max_length` and `min_length`. If provided, these arguments ensure that the string is at most or at least the given length.

`FileField`

Please, see the release notes

class `FileField` (***kwargs*)

- Default widget: `FileInput`
- Empty value: `None`
- Normalizes to: An `UploadedFile` object that wraps the file content and file name into a single object.
- Validates that non-empty file data has been bound to the form.
- Error message keys: `required`, `invalid`, `missing`, `empty`

To learn more about the `UploadedFile` object, see the [file uploads documentation](#).

When you use a `FileField` in a form, you must also remember to *bind the file data to the form*.

`FilePathField`

Please, see the release notes

class `FilePathField` (***kwargs*)

- Default widget: `Select`
- Empty value: `None`
- Normalizes to: A unicode object
- Validates that the selected choice exists in the list of choices.
- Error message keys: `required`, `invalid_choice`

The field allows choosing from files inside a certain directory. It takes three extra arguments; only `path` is required:

FilePathField.path

The absolute path to the directory whose contents you want listed. This directory must exist.

FilePathField.recursive

If `False` (the default) only the direct contents of `path` will be offered as choices. If `True`, the directory will be descended into recursively and all descendants will be listed as choices.

FilePathField.match

A regular expression pattern; only files with names matching this expression will be allowed as choices.

FloatField

- Default widget: `TextInput`
- Empty value: `None`
- Normalizes to: A Python float.
- Validates that the given value is an float. Leading and trailing whitespace is allowed, as in Python's `float()` function.
- Error message keys: `required`, `invalid`, `max_value`, `min_value`

Takes two optional arguments for validation, `max_value` and `min_value`. These control the range of values permitted in the field.

ImageField

Please, see the release notes

class ImageField (***kwargs*)

- Default widget: `FileInput`
- Empty value: `None`
- Normalizes to: An `UploadedFile` object that wraps the file content and file name into a single object.
- Validates that file data has been bound to the form, and that the file is of an image format understood by PIL.
- Error message keys: `required`, `invalid`, `missing`, `empty`, `invalid_image`

Using an `ImageField` requires that the [Python Imaging Library](#) is installed.

When you use an `ImageField` on a form, you must also remember to *bind the file data to the form*.

IntegerField

class IntegerField (***kwargs*)

- Default widget: `TextInput`
- Empty value: `None`
- Normalizes to: A Python integer or long integer.
- Validates that the given value is an integer. Leading and trailing whitespace is allowed, as in Python's `int()` function.
- Error message keys: `required`, `invalid`, `max_value`, `min_value`

Takes two optional arguments for validation:

IntegerField.max_value

`IntegerField.min_value`

These control the range of values permitted in the field.

IPAddressField

class `IPAddressField` (***kwargs*)

- Default widget: `TextInput`
- Empty value: `' '` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value is a valid IPv4 address, using a regular expression.
- Error message keys: `required`, `invalid`

MultipleChoiceField

class `MultipleChoiceField` (***kwargs*)

- Default widget: `SelectMultiple`
- Empty value: `[]` (an empty list)
- Normalizes to: A list of Unicode objects.
- Validates that every value in the given list of values exists in the list of choices.
- Error message keys: `required`, `invalid_choice`, `invalid_list`

Takes one extra argument, `choices`, as for `ChoiceField`.

NullBooleanField

class `NullBooleanField` (***kwargs*)

- Default widget: `NullBooleanSelect`
- Empty value: `None`
- Normalizes to: A Python `True`, `False` or `None` value.
- Validates nothing (i.e., it never raises a `ValidationError`).

RegexField

class `RegexField` (***kwargs*)

- Default widget: `TextInput`
- Empty value: `' '` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value matches against a certain regular expression.
- Error message keys: `required`, `invalid`

Takes one required argument:

`RegexField.regex`

A regular expression specified either as a string or a compiled regular expression object.

Also takes `max_length` and `min_length`, which work just as they do for `CharField`.

The optional argument `error_message` is also accepted for backwards compatibility. The preferred way to provide an error message is to use the `error_messages` argument, passing a dictionary with 'invalid' as a key and the error message as the value.

TimeField

class TimeField (***kwargs*)

- Default widget: `TextInput`
- Empty value: `None`
- Normalizes to: A Python `datetime.time` object.
- Validates that the given value is either a `datetime.time` or string formatted in a particular time format.
- Error message keys: `required`, `invalid`

Takes one optional argument:

TimeField.input_formats

A list of formats used to attempt to convert a string to a valid `datetime.time` object.

If no `input_formats` argument is provided, the default input formats are:

```
'%H:%M:%S',      # '14:30:59'
'%H:%M',         # '14:30'
```

URLField

class URLField (***kwargs*)

- Default widget: `TextInput`
- Empty value: `' '` (an empty string)
- Normalizes to: A Unicode object.
- Validates that the given value is a valid URL.
- Error message keys: `required`, `invalid`, `invalid_link`

Takes the following optional arguments:

URLField.max_length

URLField.min_length

Same as `CharField.max_length` and `CharField.min_length`.

URLField.verify_exists

If `True`, the validator will attempt to load the given URL, raising `ValidationError` if the page gives a 404. Defaults to `False`.

URLField.validator_user_agent

String used as the user-agent used when checking for a URL's existence. Defaults to the value of the `URL_VALIDATOR_USER_AGENT` setting.

Slightly complex built-in Field classes

The following are not yet documented.

class ComboField (***kwargs*)


```
class MultiValueField(**kwargs)
```

```
class SplitDateTimeField(**kwargs)
```

Fields which handle relationships

For representing relationships between models, two fields are provided which can derive their choices from a `QuerySet`:

```
class ModelChoiceField(**kwargs)
```

```
class ModelMultipleChoiceField(**kwargs)
```

These fields place one or more model objects into the `cleaned_data` dictionary of forms in which they're used. Both of these fields have an additional required argument:

`ModelChoiceField.queryset`

A `QuerySet` of model objects from which the choices for the field will be derived, and which will be used to validate the user's selection.

ModelChoiceField

Allows the selection of a single model object, suitable for representing a foreign key.

The `__unicode__` method of the model will be called to generate string representations of the objects for use in the field's choices; to provide customized representations, subclass `ModelChoiceField` and override `label_from_instance`. This method will receive a model object, and should return a string suitable for representing it. For example:

```
class MyModelChoiceField(ModelChoiceField):
    def label_from_instance(self, obj):
        return "My Object #%i" % obj.id
```

`ModelChoiceField.empty_label`

By default the `<select>` widget used by `ModelChoiceField` will have an empty choice at the top of the list. You can change the text of this label (which is `"-----"` by default) with the `empty_label` attribute, or you can disable the empty label entirely by setting `empty_label` to `None`:

```
# A custom empty label
field1 = forms.ModelChoiceField(queryset=..., empty_label="(Nothing)")

# No empty label
field2 = forms.ModelChoiceField(queryset=..., empty_label=None)
```

Note that if a `ModelChoiceField` is required and has a default initial value, no empty choice is created (regardless of the value of `empty_label`).

ModelMultipleChoiceField

Allows the selection of one or more model objects, suitable for representing a many-to-many relation. As with `ModelChoiceField`, you can use `label_from_instance` to customize the object representations.

Creating custom fields

If the built-in `Field` classes don't meet your needs, you can easily create custom `Field` classes. To do this, just create a subclass of `django.forms.Field`. Its only requirements are that it implement a `clean()` method and that its `__init__()` method accept the core arguments mentioned above (`required`, `label`, `initial`, `widget`, `help_text`).

Widgets

Um widget é uma representação do Django de um elemento input do HTML. O widget manipula a renderização do HTML, e a extração de dados de um dicionário GET/POST que correspondem ao widget.

O Django fornece uma representação de todos os widget básicos do HTML, mais alguns grupos de widgets comumente usados:

class TextInput

Campo de texto: `<input type='text' ...>`

class PasswordInput

Campo de senha: `<input type='password' ...>`

class HiddenInput

Campo invisível: `<input type='hidden' ...>`

class MultipleHiddenInput

Múltiplos widgets `<input type='hidden' ...>`.

class FileInput

Campo de upload de arquivo: `<input type='file' ...>`

class DateTimeInput

Please, see the release notes Campo de Data/hora como uma caixa de texto: `<input type='text' ...>`

class Textarea

Area de texto: `<textarea>...</textarea>`

class CheckboxInput

Checkbox: `<input type='checkbox' ...>`

class Select

Campo select: `<select><option ...>...</select>`

Requer que seu campo forneça um choices.

class NullBooleanSelect

Um campo select com as opções 'Unknown', 'Yes' e 'No'

class SelectMultiple

Um select que permite seleção múltipla: `<select multiple='multiple'>...</select>`

Requer que seu campo forneça um choices.

class RadioSelect

Uma lista de botões radio:

```
<ul>
  <li><input type='radio' ...></li>
  ...
</ul>
```

Requer que seu campo forneça um choices.

class CheckboxSelectMultiple

Uma lista de checkboxes:

```
<ul>
  <li><input type='checkbox' ...></li>
  ...
</ul>
```

class MultiWidget

Invólucro em torno de vários outros widgets

class SplitDateTimeWidget

Invólucro em torno de dois widgets `TextInput`: um para data, e um para a hora.

Especificando widgets

Form.widget

Sempre que você especificar um campo em um formulário, o Django irá usar um widget padrão que seja apropriado ao tipo de dado que será mostrado. Para encontrar que widget é usado em que campo, veja a documentação das classes `Field` embutidas.

No entanto, se você quiser usar um widget diferente para um campo, você pode - simplesmente usar o argumento `'widget'` na definição de campo. Por exemplo:

```
from django import forms

class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField(widget=forms.Textarea)
```

Isso poderia especificar um formulário com um comentário que usa um widget `Textarea`, ao invés do widget padrão `TextInput`.

Customizando instâncias de widget

Quando o Django renderiza um widget como HTML, ele somente renderiza o HTML mínimo - o Django não adicionar uma definição class, ou qualquer outro atributo específico de widget. Isto significa que todos os widgets `'TextInput'` parecerão os mesmos em sua página web.

If you want to make one widget look different to another, you need to specify additional attributes for each widget. When you specify a widget, you can provide a list of attributes that will be added to the rendered HTML for the widget.

Se você quiser fazer um widget parecer diferente de outro, você precisa especificar atributos adicionais para cada widget. Quando você especificar um widget, você pode prover uma lista de atributos que serão adicionados ao HTML renderizado do widget.

Por exemplo, pegue este simples formulário:

```
class CommentForm(forms.Form):
    name = forms.CharField()
    url = forms.URLField()
    comment = forms.CharField()
```

Este formulário incluirá três widgets `TextInput` padrão, com renderização padrão - sem classe CSS, sem atributos extras. Isto significa que as caixas de entrada fornecidas para cada widget serão renderizadas iguais:

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" /></td></tr>
<tr><th>Url:</th><td><input type="text" name="url" /></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" /></td></tr>
```

Numa página web real, você provavelmente não vai querer todos os campos iguais. Você pode querer um campo maior para o comentário, e você pode querer que o widget `'name'` tenha alguma classe CSS especial. Para fazer isto, você usa o argumento `attrs` quando estiver criando o widget:

Widget.attrs

Por exemplo:

```
class CommentForm(forms.Form):
    name = forms.CharField(
        widget=forms.TextInput(attrs={'class':'special'}))
    url = forms.URLField()
    comment = forms.CharField(
        widget=forms.TextInput(attrs={'size':'40'}))
```

O Django irá então incluir os atributos extras na saída renderizada:

```
>>> f = CommentForm(auto_id=False)
>>> f.as_table()
<tr><th>Name:</th><td><input type="text" name="name" class="special"/></td></tr>
<tr><th>Url:</th><td><input type="text" name="url"/></td></tr>
<tr><th>Comment:</th><td><input type="text" name="comment" size="40"/></td></tr>
```

Validação de campos e formulários

A validação de formulários acontece quando os dados são limpos. Se você quer customizar este processo, há vários lugares que você pode mudar, cada um serve para um diferente propósito. Três tipos de métodos de limpeza são executados durante o processamento do formulário. Estes são normalmente executados quando você chama o método `is_valid()` do formulário. Há outras coisas que conseguem disparar a limpeza e validação (acessando os atributos `errors` ou chamando `full_clean()` diretamente), mas normalmente eles não são necessários.

Geralmente, qualquer método de limpeza pode lançar `ValidationError` se houver um problema com os dados processados, passando a mensagem de erro relevante para o construtor `ValidationError`. Se nenhum `ValidationError` é lançado, o método deve retornar os dados limpos (normalizados) como um objeto Python.

Se você detectar vários erros durante o método de limpeza e deseja sinalizar todos eles para o submissor do formulário, é possível passar uma lista dos erros para o construtor do `ValidationError`.

Os três tipos de métodos de limpeza são:

- O método `clean()` em uma subclasse `Field`. Este é responsável por limpar os dados de uma forma genérica para esse tipo de campo. Por exemplo, um `FloatField` será transformado em um dado Python do tipo `float` ou lançará o `ValidationError`. Este método retorna o dado limpo, que é então inserido dentro do dicionário `cleaned_data` do formulário.
- O método `clean_<fieldname>()` em uma subclasse de formulário – onde `<fieldname>` é substituído com o nome do atributo campo do formulário.
- O método `clean_<fieldname>()` em uma subclasse de formulário – onde o `<fieldname>` é substituído com o nome do atributo do campo do formulário. Este método faz qualquer limpeza que seja específica para um atributo, independente do tipo de campo que ele for. A este método não é passado parâmetros. Você precisará olhar o valor do campo em `self.cleaned_data` e lembrar que ele será um objeto Python neste ponto, não a string original enviada pelo formulário (ele estará no `cleaned_data` porque o método `clean()` dos campos em geral, acima, já validaram os dados uma vez).

Por exemplo, se você procura validar o conteúdo de um `CharField` chamado `serialnumber` como único, `clean_serialnumber()` seria o lugar certo para fazer isto. Você não precisa de um campo específico (ele é só um `CharField`), mas você deseja a validação de um campo de formulário específico e, possivelmente, limpeza/normalização dos dados.

Assim como o método `clean()` de campo geral, acima, este método deveria retornar o dado normalizado, independentemente do fato de ter mudado alguma coisa ou não.

- O método `clean()` de subclasse do `Form`. Este método pode executar qualquer validação que requer acesso a múltiplos campos de um formulário de uma vez. Aqui é onde você pode colocar coisas para checar se um campo A é fornecido, campo B deve conter um endereço de e-mail válido e, algo mais. O dado que este método retorna é o atributo final `cleaned_data` para o formulário, então não esqueça de retornar uma lista completa com os dados validados se você sobrescrever este método (por padrão, `Form.clean()` retorna somente `self.cleaned_data`).

Note que quaisquer erros lançados por sua sobrescrita `Form.clean()` não será associado como qualquer campo em particular. Eles irão dentro de um “campo” especial (chamado, `__all__`), que você pode acessar via o método `non_field_errors()` se você precisar. Se você quer atachar os erros a um campo específico do formulário, você precisará acessar o atributo `_errors` do formulário, que é *descrito mais tarde*.

Estes métodos são executados na ordem mostrada acima, um campos por vez. Isto é, para campo no formulário (na ordem em que foram declarados, na definição do formulário), o método `Field.clean()` (ou sua sobrescrita) é executado, e então `clean_<fieldname>()`. Finalmente, um dos dois método são executados para cada ampo, o método `Form.clean()`, ou sua sobrescrita, é executado.

Exemplo de cada um destes métodos são mostrados abaixo.

Como mencionando, qualquer um desses métodos podem lançar um `ValidationError`. Para qualquer campo, se o método `Field.clean()` lançar um `ValidationError`, qualquer método de limpeza de campo específico não é chamado. Entretanto, os métodos de limpeza para todos os campos remanescentes ainda serão executados.

O método `clean()` para a classe ou subclasse `Form` é sempre executado. Se este método lançar um `ValidationError`, o `cleaned_data` será um dicionário vazio.

O parágrafo anterior significa que se você estiver sobrescrevendo `Form.clean()`, você deve iterar sobre `self.cleaned_data.items()`, possivelmente considerando o atributo dicionário `_errors` sobre o formulário. Desta forma, você já saberá quais campos passaram na validação individual.

Sublasses de Form e modificando erros de campos

As vezes, num método `clean()` de um formulário, você pode querer adicionar uma mensagem de erro para um campo em particular. Isso nem sempre é apropriado e a situação mais comum é lançar um `ValidationError` num `Form.clean()`, que é ativado em um erro ao longo do formulário e disponibilizado através do método `Form.non_field_errors()`.

Quando você realmente precisa atachar um erro para um campo em particular, você deve armazenar (ou alterar) uma chave no atributo `Form._errors`. Este atributo é uma instância da classe `django.forms.util.ErrorDict`. Essencialmente, porém, é só um dicionário. Cada valor no dicionário é uma instância `django.forms.util.ErrorList`, que é uma lista que sabe como se mostrar de diferentes formas. então você pode tratar `_erros` como um dicionário, mapeado com os nomes dos campos.

Se você quiser adicionar um novo erro a campo em particular, você deve checar se a chave já existe em `self._errors` ou não. Se não, crie um nova entrada para a dada chave, mantendo um instância vazia do `ErrorList`. Em ambos os casos, você pode então atachar sua mensagem de erro a lista para o nome do campo em questão e ela será exibida quando o formulário for mostrado.

Há um exemplo de como modificar o `self._errors` na próxima seção.

O que está no nome?

Você pode estar pensando por que este atributo é chamado `_erros` e não `errors`. Na prática normal do Python é para prefixar um nome com um underscore quando este não for de uso externo. Neste caso, você está estendendo a classe `Form`, então você está essencialmente escrevendo um novo núcleo. Na verdade, é dado a permissão a você para acessar alguns atributos internos do `Form`.

É claro, que qualquer código fora de seu formulário nunca deve acessar o `_erros` diretamente. Os dados estão disponíveis para código externo através da propriedade `errors`, que popula `_errors` antes de retorná-lo).

Outra razão é puramente histórica: o atributo tem sido chamado `_errors` desde os primeiros dias do módulo `forms` e mudá-lo agora (particularmente desde que `errors` fora usado como o nome da propriedade somente leitura) seria inconveniente por inúmeras razões. Você pode usar a explicação que lhe for mais confortável. O resultado é o mesmo.

Usando validação na prática

A seção anterior explicou como a validação nos formulários geralmente funciona. Uma vez que pode ser mais fácil de por as coisas no lugar ao ver cada característica em uso, aqui está uma série de pequenos exemplos que usam cada uma das funcionalidades anteriormente citadas.

Limpeza padrão de campos do Form

Vamos em primeiro lugar criar um campo de formulário customizado que valida sua entrada como uma string contendo endereços de e-mail separados por vírgula, com pelo menos um endereço. Nós manteremos ele simples e assumimos que a validação de e-mail está contida na função chamada `is_valid_email()`. A classe completa parece com esta:

```
from django import forms

class MultiEmailField(forms.Field):
    def clean(self, value):
        """
        Checa que o campo contém um ou mais email separados por vírgula
        e normaliza os dados para uma lista de strings de email.
        """
        if not value:
            raise forms.ValidationError('Enter at least one e-mail address.')
        emails = value.split(',')
        for email in emails:
            if not is_valid_email(email):
                raise forms.ValidationError('%s is not a valid e-mail address.' %
↪email)

        # Sempre retorna o dado limpo.
        return emails
```

Todo formulário que usar este campo terpa este método `clean()` executado antes que algo mais seja feito com o os dados do campo. Esta é a validação que é específica para este tipo de campo, independentemente da forma como é subsequentemente usado.

Vamos criar um simples `ContactForm` para demonstrar como este campo poderia ser utilizado:

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    recipients = MultiEmailField()
    cc_myself = forms.BooleanField(required=False)
```

Simplesmente use o `MultiEmailField` como qualquer outro campo. Quando o método `is_valid()` é chamado no formulário, o método `MultiEmailField` será executado como parte do processo de validação.

Validando um atributo campo específico

Continuando sobre o exemplo anterior, supomos que em seu `ContactForm`, nós queremos estar certos de que o campo `recipients` sempre conterá o endereço `"fred@example.com"`. Este é uma validação que é específica para o nosso formulário, então nós não queremos colocá-la dentro da classe geral `MultiEmailField`. Ao invés, nós escreveremos um método que opera sobre o campo `recipients`, desta forma:

```
class ContactForm(forms.Form):
    # Tudo como antes.
    ...

    def clean_recipients(self):
```

```
data = self.cleaned_data['recipients']
if "fred@example.com" not in data:
    raise forms.ValidationError("You have forgotten about Fred!")

# Sempre retorna o dado validado, você tendo mudado ele ou não.
return data
```

Limpendo e validando campos que dependem uns dos outros

Suponhamos adicionar outra exigência ao nosso formulário de contato: se o campo `cc_myself` é `True`, o `subject` deve conter a palavra "help". Nós estamos executando a validação em mais de um campo ao mesmo tempo, então o método `clean()` do formulário é um bom lugar para se fazer isto. Observe que nós estamos falando sobre o método `clean()` de um formulário, considerando que antes nós estávamos escrevendo um método `clean()` de um campo. É importante manter o domínio de forma clara quando estamos trabalhando com validações. Campos são um ponto único de dados, formulários são uma coleção de campos.

Agora, o método `clean()` do formulário é chamado, todos os métodos `clean` individuais dos campos serão executados (as duas seções anteriores), então o `self.cleaned_data` será populado com qualquer dado que tenha sobrevivido até então. Você também precisa lembrar de permitir o fato de que os campos que você está esperando validar podem não ter sobrevivido a checagem inicial dos campos.

Há duas formas de reportar quaisquer erros de um passo. Provavelmente o método mais comum é mostrar o erro no topo do formulário. Para criar esse erro, você pode lançar um `ValidationError` no método `clean()`. Por exemplo:

```
class ContactForm(forms.Form):
    # Tudo como antes.
    ...

    def clean(self):
        cleaned_data = self.cleaned_data
        cc_myself = cleaned_data.get("cc_myself")
        subject = cleaned_data.get("subject")

        if cc_myself and subject:
            # Somente faça algo se ambos os campos forem válidos.
            if "help" not in subject:
                raise forms.ValidationError("Did not send for 'help' in "
                                           "the subject despite CC'ing yourself.")

        # Sempre retorne a coleção completa de dados válidos.
        return cleaned_data
```

Neste código, se o erro de validação é lançado, o formulário mostrará uma mensagem de erro no seu topo (normalmente) descrevendo o problema.

A segunda abordagem pode envolver atribuição de uma mensagem de erro para um ou mais campos. Neste caso, vamos atribuir uma mensagem de erro para ambas linhas "subject" e "cc_myself" na exibição do formulário. Seja cuidadoso quando utilizar esta prática, pois pode conduzir a uma saída confusa de formulário. Nós estamos mostrando o que é possível aqui, e deixando você e seus designers trabalharem no que efetivamente é a sua situação em particular. Nosso novo código (substituindo o exemplo anterior) é este:

```
from django.forms.util import ErrorList

class ContactForm(forms.Form):
    # Tudo como antes.
    ...

    def clean(self):
        cleaned_data = self.cleaned_data
        cc_myself = cleaned_data.get("cc_myself")
```

```
subject = cleaned_data.get("subject")

if cc_myself and subject and "help" not in subject:
    # Nós sabemos estes não estão em self._errors agora (veja a
    # discussão abaixo).
    msg = u"Must put 'help' in subject when cc'ing yourself."
    self._errors["cc_myself"] = ErrorList([msg])
    self._errors["subject"] = ErrorList([msg])

    # Estes campos não são válidos. Remova-os dos dados válidos.
    del cleaned_data["cc_myself"]
    del cleaned_data["subject"]

# Sempre retorne a coleção completa de dados válidos.
return cleaned_data
```

Como você pode ver, esta abordagem requer um pouco mais de esforço, não suportando um esforço extra de design para criar uma visualização sensata do formulário. Os detalhes são dignos de nota, no entanto. Primeiramente, como mencionamos que você pode precisar checar se o nome do campo já existe no dicionário `_errors`. Neste caso, desde que sabemos que os campos existem no `self.cleaned_data`, eles devem ser válidos quando validados individualmente, então haverá chaves não correspondentes em `_errors`.

Depois, uma vez que nós tenhamos decidido que os dados combinados dos dois campos não são válidos, nós devemos lembrar de removê-los do `cleaned_data`.

De fato, o Django irá, no momento, limpar completamente o dicionário `cleaned_data` se houver qualquer erro no formulário. No entanto, este comportamento pode ser mudado no futuro, então não é uma má ideia, depois você mesmo limpá-lo, em primeiro lugar.

Generic views (Visões genéricas)

Escrever aplicações Web pode ser monótono, pois nós repetimos certos padrões repetidamente. No Django, os mais comuns dentre esses padrões são abstraídos dentro de “generic views” que permitem você rapidamente fornecer views comuns de um objeto sem precisar escrever qualquer código Python.

Uma introdução geral d generic views pode ser encontrado em *topic guide*.

Esta referência contém detalhes das generic views embutidas do Django, junto com uma lista de todos os argumentos que uma generic view espera. Lembre-se que argumentos podem vir tanto de uma URL quanto de informações adicionais num dicionário `extra_context`.

A maioria dos generic views requer a chave `queryset`, que é uma instância do `QuerySet`; veja *Fazendo consultas* para mais informações sobre objetos `QuerySet`.

“Simple” generic views

O módulo `django.views.generic.simple` contém views simples para manipular alguns casos comuns: renderizar um template quando nenhuma view lógica é necessária, e emitir redirecionamentos.

`django.views.generic.simple.direct_to_template`

Descrição:

Renderiza um dado template, passando-o uma variável de template `{{ params }}`, que é um dicionário de parametros capturados numa URL.

Argumentos obrigatórios:

- `template`: O nome completo de um template a se usar.

Argumentos opcionais:

- `extra_context`: Um dicionário de valores para adicionar ao contexto do template. Por padrão, este é um dicionário vazio. Se um valor no dicionário for chamável, o generic view o chamará assim que estiver renderizando o template.
- `mimetype`: O tipo MIME a ser usado no documento resultante. Por padrão é o valor da configuração `DEFAULT_CONTEXT_TYPE`.

Exemplo:

Dados os seguintes padrões de URL:

```
urlpatterns = patterns('django.views.generic.simple',
    (r'^foo/$', 'direct_to_template', {'template': 'foo_index.html'}),
    (r'^foo/(?P<id>\d+)/$', 'direct_to_template', {'template': 'foo_detail.html'}),
)
```

... uma requisição para `/foo/` renderizaria o template `foo_index.html`, e uma requisição para `/foo/15/` renderizaria o `foo_detail.html` com uma variável de contexto `{{ params.id }}` que está setado como 15.

`django.views.generic.simple.redirect_to`

Descrição:

Redireciona para uma dada URL.

A dada URL pode conter strings na forma de dicionário, que serão intercaladas contra os parametros capturados na URL. Pois a interpolação de chaves é *sempre* feita (mesmo se nenhum argumento for passado), qualquer caracter "%" na URL deve ser escrito como "%" assim o Python os converterá para um único sinal de percentual na saída.

Se uma dada URL é None, o Django retornará um `HttpResponseGone` (410).

Argumentos obrigatórios:

- `url`: A URL para onde redirecionar, como uma string. Ou None para lançar um erro HTTP 410 (Gone).

Exemplo:

Este exemplo redireciona de `/foo/<id>/` para `/bar/<id>/`:

```
urlpatterns = patterns('django.views.generic.simple',
    (r'^foo/(?P<id>\d+)/$', 'redirect_to', {'url': '/bar/%(id)s/'}),
)
```

Este exemplo retorna um erro HTTP 410 para a requisição `/bar/`:

```
urlpatterns = patterns('django.views.generic.simple',
    (r'^bar/$', 'redirect_to', {'url': None}),
)
```

Este exemplo mostra como caracteres "%" devem ser escritos na URL afim de evitar confusão com os marcadores de strings do Python. Se a strings de redirecionamento é escrita como `"%7Ejacob/"` (com somente um %), uma exceção poderia ser lançada:

```
urlpatterns = patterns('django.views.generic.simple',
    (r'^bar/$', 'redirect_to', {'url': '%7Ejacob.'}),
)
```

Date-based generic views

Date-based generic views (in the module `django.views.generic.date_based`) are views for displaying drilldown pages for date-based data.

`django.views.generic.date_based.archive_index`

Descrição:

A top-level index page showing the “latest” objects, by date. Objects with a date in the *future* are not included unless you set `allow_future` to `True`.

Argumentos obrigatórios:

- `queryset`: A `QuerySet` of objects for which the archive serves.
- `date_field`: The name of the `DateField` or `DateTimeField` in the `QuerySet`’s model that the date-based archive should use to determine the objects on the page.

Optional arguments:

- `num_latest`: The number of latest objects to send to the template context. By default, it’s 15.
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it’s `django.template.loader`.
- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `allow_empty`: A boolean specifying whether to display the page if no objects are available. If this is `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `True`.
- `context_processors`: A list of template-context processors to apply to the view’s template.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `allow_future`: A boolean specifying whether to include “future” objects on this page, where “future” means objects in which the field specified in `date_field` is greater than the current date/time. By default, this is `False`.

Please, see the release notes

- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'latest'`.

Template name:

If `template_name` isn’t specified, this view will use the template `<app_label>/<model_name>_archive.html` by default, where:

- `<model_name>` is your model’s name in all lowercase. For a model `StaffMember`, that’d be `staffmember`.
- `<app_label>` is the right-most part of the full Python path to your model’s app. For example, if your model lives in `apps/blog/models.py`, that’d be `blog`.

Template context:

In addition to `extra_context`, the template’s context will be:

- `date_list`: A list of `datetime.date` objects representing all years that have objects available according to `queryset`. These are ordered in reverse. This is equivalent to `queryset.dates(date_field, 'year')[::-1]`.

The behaviour depending on `template_object_name` is new in this version.

- `latest`: The `num_latest` objects in the system, ordered descending by `date_field`. For example, if `num_latest` is 10, then `latest` will be a list of the latest 10 objects in `queryset`.

This variable's name depends on the `template_object_name` parameter, which is `'latest'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo`.

`django.views.generic.date_based.archive_year`

Descrição:

A yearly archive page showing all available months in a given year. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Argumentos obrigatórios:

- `year`: The four-digit year for which the archive serves.
- `queryset`: A `QuerySet` of objects for which the archive serves.
- `date_field`: The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the date-based archive should use to determine the objects on the page.

Optional arguments:

- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `allow_empty`: A boolean specifying whether to display the page if no objects are available. If this is `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `False`.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`. The view will append `'_list'` to the value of this parameter in determining the variable's name.
- `make_object_list`: A boolean specifying whether to retrieve the full list of objects for this year and pass those to the template. If `True`, this list of objects will be made available to the template as `object_list`. (The name `object_list` may be different; see the docs for `object_list` in the “Template context” section below.) By default, this is `False`.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `allow_future`: A boolean specifying whether to include “future” objects on this page, where “future” means objects in which the field specified in `date_field` is greater than the current date/time. By default, this is `False`.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_archive_year.html` by default.

Template context:

In addition to `extra_context`, the template's context will be:

- `date_list`: A list of `datetime.date` objects representing all months that have objects available in the given year, according to `queryset`, in ascending order.
- `year`: The given year, as a four-character string.
- `object_list`: If the `make_object_list` parameter is `True`, this will be set to a list of objects available for the given year, ordered by the date field. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

If `make_object_list` is `False`, `object_list` will be passed to the template as an empty list.

`django.views.generic.date_based.archive_month`

Descrição:

A monthly archive page showing all objects in a given month. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Argumentos obrigatórios:

- `year`: The four-digit year for which the archive serves (a string).
- `month`: The month for which the archive serves, formatted according to the `month_format` argument.
- `queryset`: A `QuerySet` of objects for which the archive serves.
- `date_field`: The name of the `DateTimeField` or `DateField` in the `QuerySet`'s model that the date-based archive should use to determine the objects on the page.

Optional arguments:

- `month_format`: A format string that regulates what format the `month` parameter uses. This should be in the syntax accepted by Python's `time.strftime`. (See the [strftime docs](#).) It's set to `"%b"` by default, which is a three-letter month abbreviation. To change it to use numbers, use `"%m"`.
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `allow_empty`: A boolean specifying whether to display the page if no objects are available. If this is `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `False`.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`. The view will append `'_list'` to the value of this parameter in determining the variable's name.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `allow_future`: A boolean specifying whether to include “future” objects on this page, where “future” means objects in which the field specified in `date_field` is greater than the current date/time. By default, this is `False`.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_archive_month.html` by default.

Template context:

In addition to `extra_context`, the template's context will be:

- `month`: A `datetime.date` object representing the given month.
- `next_month`: A `datetime.date` object representing the first day of the next month. If the next month is in the future, this will be `None`.
- `previous_month`: A `datetime.date` object representing the first day of the previous month. Unlike `next_month`, this will never be `None`.
- `object_list`: A list of objects available for the given month. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

`django.views.generic.date_based.archive_week`

Descrição:

A weekly archive page showing all objects in a given week. Objects with a date in the *future* are not displayed unless you set `allow_future` to `True`.

Argumentos obrigatórios:

- `year`: The four-digit year for which the archive serves (a string).
- `week`: The week of the year for which the archive serves (a string). Weeks start with Sunday.
- `queryset`: A `QuerySet` of objects for which the archive serves.
- `date_field`: The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the date-based archive should use to determine the objects on the page.

Optional arguments:

- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `allow_empty`: A boolean specifying whether to display the page if no objects are available. If this is `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `True`.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`. The view will append `'_list'` to the value of this parameter in determining the variable's name.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `allow_future`: A boolean specifying whether to include “future” objects on this page, where “future” means objects in which the field specified in `date_field` is greater than the current date/time. By default, this is `False`.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_archive_week.html` by default.

Template context:

In addition to `extra_context`, the template's context will be:

- `week`: A `datetime.date` object representing the first day of the given week.
- `object_list`: A list of objects available for the given week. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

`django.views.generic.date_based.archive_day`

Descrição:

A day archive page showing all objects in a given day. Days in the future throw a 404 error, regardless of whether any objects exist for future days, unless you set `allow_future` to `True`.

Argumentos obrigatórios:

- `year`: The four-digit year for which the archive serves (a string).
- `month`: The month for which the archive serves, formatted according to the `month_format` argument.
- `day`: The day for which the archive serves, formatted according to the `day_format` argument.
- `queryset`: A `QuerySet` of objects for which the archive serves.
- `date_field`: The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the date-based archive should use to determine the objects on the page.

Optional arguments:

- `month_format`: A format string that regulates what format the `month` parameter uses. This should be in the syntax accepted by Python's `time.strftime`. (See the [strftime docs](#).) It's set to `"%b"` by default, which is a three-letter month abbreviation. To change it to use numbers, use `"%m"`.
- `day_format`: Like `month_format`, but for the `day` parameter. It defaults to `"%d"` (day of the month as a decimal number, 01-31).
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `allow_empty`: A boolean specifying whether to display the page if no objects are available. If this is `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `False`.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`. The view will append `'_list'` to the value of this parameter in determining the variable's name.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `allow_future`: A boolean specifying whether to include “future” objects on this page, where “future” means objects in which the field specified in `date_field` is greater than the current date/time. By default, this is `False`.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_archive_day.html` by default.

Template context:

In addition to `extra_context`, the template's context will be:

- `day`: A `datetime.date` object representing the given day.
- `next_day`: A `datetime.date` object representing the next day. If the next day is in the future, this will be `None`.
- `previous_day`: A `datetime.date` object representing the given day. Unlike `next_day`, this will never be `None`.
- `object_list`: A list of objects available for the given day. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.

`django.views.generic.date_based.archive_today`**Descrição:**

A day archive page showing all objects for *today*. This is exactly the same as `archive_day`, except the `year/month/day` arguments are not used, and today's date is used instead.

`django.views.generic.date_based.object_detail`**Descrição:**

A page representing an individual object. If the object has a date value in the future, the view will throw a 404 error by default, unless you set `allow_future` to `True`.

Argumentos obrigatórios:

- `year`: The object's four-digit year (a string).
- `month`: The object's month, formatted according to the `month_format` argument.
- `day`: The object's day, formatted according to the `day_format` argument.
- `queryset`: A `QuerySet` that contains the object.
- `date_field`: The name of the `DateField` or `DateTimeField` in the `QuerySet`'s model that the generic view should use to look up the object according to `year`, `month` and `day`.
- Either `object_id` or (`slug` and `slug_field`) is required.

If you provide `object_id`, it should be the value of the primary-key field for the object being displayed on this page.

Otherwise, `slug` should be the slug of the given object, and `slug_field` should be the name of the slug field in the `QuerySet`'s model. By default, `slug_field` is `'slug'`.

Optional arguments:

- `month_format`: A format string that regulates what format the `month` parameter uses. This should be in the syntax accepted by Python's `time.strftime`. (See the [strftime docs](#).) It's set to `"%b"` by default, which is a three-letter month abbreviation. To change it to use numbers, use `"%m"`.
- `day_format`: Like `month_format`, but for the `day` parameter. It defaults to `"%d"` (day of the month as a decimal number, 01-31).
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_name_field`: The name of a field on the object whose value is the template name to use. This lets you store template names in the data. In other words, if your object has a field `'the_template'` that contains a string `'foo.html'`, and you set `template_name_field` to `'the_template'`, then the generic view for this object will use the template `'foo.html'`.

It's a bit of a brain-bender, but it's useful in some cases.

- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.
- `allow_future`: A boolean specifying whether to include “future” objects on this page, where “future” means objects in which the field specified in `date_field` is greater than the current date/time. By default, this is `False`.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_detail.html` by default.

Template context:

In addition to `extra_context`, the template's context will be:

- `object`: The object. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo`.

List/detail generic views

The list-detail generic-view framework (in the `django.views.generic.list_detail` module) is similar to the date-based one, except the former simply has two views: a list of objects and an individual object page.

`django.views.generic.list_detail.object_list`

Descrição:

A page representing a list of objects.

Argumentos obrigatórios:

- `queryset`: A `QuerySet` that represents the objects.

Optional arguments:

- `paginate_by`: An integer specifying how many objects should be displayed per page. If this is given, the view will paginate objects with `paginate_by` objects per page. The view will expect either a page query string parameter (via GET) or a page variable specified in the URLconf. See [Notes on pagination](#) below.
- `page`: The current page number, as an integer, or the string `'last'`. This is 1-based. See [Notes on pagination](#) below.
- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.

- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `allow_empty`: A boolean specifying whether to display the page if no objects are available. If this is `False` and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is `True`.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`. The view will append `'_list'` to the value of this parameter in determining the variable's name.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_list.html` by default.

Template context: The `paginator` and `page_obj` context variables are new. In addition to `extra_context`, the template's context will be:

- `object_list`: The list of objects. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo_list`.
- `is_paginated`: A boolean representing whether the results are paginated. Specifically, this is set to `False` if the number of available objects is less than or equal to `paginate_by`.

If the results are paginated, the context will contain these extra variables:

- `paginator`: An instance of `django.core.paginator.Paginator`.
- `page_obj`: An instance of `django.core.paginator.Page`.

Notes on pagination

If `paginate_by` is specified, Django will paginate the results. You can specify the page number in the URL in one of two ways:

- Use the `page` parameter in the URLconf. For example, this is what your URLconf might look like:

```
(r'^objects/page(?:P<page>[0-9]+)/$', 'object_list', dict(info_dict))
```

- Pass the page number via the `page` query-string parameter. For example, a URL would look like this:

```
/objects/?page=3
```

- To loop over all the available page numbers, use the `page_range` variable. You can iterate over the list provided by `page_range` to create a link to every page of results.

These values and lists are 1-based, not 0-based, so the first page would be represented as page 1.

For more on pagination, read the [pagination documentation](#). Please, see the [release notes](#) As a special case, you are also permitted to use `last` as a value for `page`:

```
/objects/?page=last
```

This allows you to access the final page of results without first having to determine how many pages there are.

Note that `page` *must* be either a valid page number or the value `last`; any other value for `page` will result in a 404 error.

`django.views.generic.list_detail.object_detail`

A page representing an individual object.

Descrição:

A page representing an individual object.

Argumentos obrigatórios:

- `queryset`: A `QuerySet` that contains the object.
- Either `object_id` or (`slug` and `slug_field`) is required.

If you provide `object_id`, it should be the value of the primary-key field for the object being displayed on this page.

Otherwise, `slug` should be the slug of the given object, and `slug_field` should be the name of the slug field in the `QuerySet`'s model. By default, `slug_field` is `'slug'`.

Optional arguments:

- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_name_field`: The name of a field on the object whose value is the template name to use. This lets you store template names in the data. In other words, if your object has a field `'the_template'` that contains a string `'foo.html'`, and you set `template_name_field` to `'the_template'`, then the generic view for this object will use the template `'foo.html'`.
It's a bit of a brain-bender, but it's useful in some cases.
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`.
- `mimetype`: The MIME type to use for the resulting document. Defaults to the value of the `DEFAULT_CONTENT_TYPE` setting.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_detail.html` by default.

Template context:

In addition to `extra_context`, the template's context will be:

- `object`: The object. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo`.

Create/update/delete generic views

The `django.views.generic.create_update` module contains a set of functions for creating, editing and deleting objects. *Please, see the [release notes](#)* `django.views.generic.create_update.create_object` and `django.views.generic.create_update.update_object` now use the new *forms library* to build and display the form.

`django.views.generic.create_update.create_object`

Descrição:

A page that displays a form for creating an object, redisplaying the form with validation errors (if there are any) and saving the object.

Argumentos obrigatórios:

- Either `form_class` or `model` is required.

If you provide `form_class`, it should be a `django.forms.ModelForm` subclass. Use this argument when you need to customize the model's form. See the [ModelForm docs](#) for more information.

Otherwise, `model` should be a Django model class and the form used will be a standard `ModelForm` for `model`.

Optional arguments:

- `post_save_redirect`: A URL to which the view will redirect after saving the object. By default, it's `object.get_absolute_url()`.

`post_save_redirect` may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use `post_save_redirect="/polls/%(slug)s/"`.

- `login_required`: A boolean that designates whether a user must be logged in, in order to see the page and save changes. This hooks into the Django [authentication system](#). By default, this is `False`.

If this is `True`, and a non-logged-in user attempts to visit this page or save the form, Django will redirect the request to `/accounts/login/`.

- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `context_processors`: A list of template-context processors to apply to the view's template.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_form.html` by default.

Template context:

In addition to `extra_context`, the template's context will be:

- `form`: A `django.forms.ModelForm` instance representing the form for creating the object. This lets you refer to form fields easily in the template system.

For example, if the model has two fields, `name` and `address`:

```
<form action="" method="post">
<p>{{ form.name.label_tag }} {{ form.name }}</p>
<p>{{ form.address.label_tag }} {{ form.address }}</p>
</form>
```

See the [forms documentation](#) for more information about using Form objects in templates.

`django.views.generic.create_update.update_object`

Descrição:

A page that displays a form for editing an existing object, redisplaying the form with validation errors (if there are any) and saving changes to the object. This uses a form automatically generated from the object's model class.

Argumentos obrigatórios:

- Either `form_class` or `model` is required.

If you provide `form_class`, it should be a `django.forms.ModelForm` subclass. Use this argument when you need to customize the model's form. See the [ModelForm docs](#) for more information.

Otherwise, `model` should be a Django model class and the form used will be a standard `ModelForm` for `model`.

- Either `object_id` or (`slug` and `slug_field`) is required.

If you provide `object_id`, it should be the value of the primary-key field for the object being displayed on this page.

Otherwise, `slug` should be the slug of the given object, and `slug_field` should be the name of the slug field in the `QuerySet`'s model. By default, `slug_field` is `'slug'`.

Optional arguments:

- `post_save_redirect`: A URL to which the view will redirect after saving the object. By default, it's `object.get_absolute_url()`.

`post_save_redirect` may contain dictionary string formatting, which will be interpolated against the object's field attributes. For example, you could use `post_save_redirect="/polls/%(slug)s/"`.

- `login_required`: A boolean that designates whether a user must be logged in, in order to see the page and save changes. This hooks into the Django [authentication system](#). By default, this is `False`.

If this is `True`, and a non-logged-in user attempts to visit this page or save the form, Django will redirect the request to `/accounts/login/`.

- `template_name`: The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- `template_loader`: The template loader to use when loading the template. By default, it's `django.template.loader`.
- `extra_context`: A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- `context_processors`: A list of template-context processors to apply to the view's template.
- `template_object_name`: Designates the name of the template variable to use in the template context. By default, this is `'object'`.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_form.html` by default.

Template context:

In addition to `extra_context`, the template's context will be:

- `form`: A `django.forms.ModelForm` instance representing the form for editing the object. This lets you refer to form fields easily in the template system.

For example, if the model has two fields, `name` and `address`:

```
<form action="" method="post">
<p>{{ form.name.label_tag }} {{ form.name }}</p>
<p>{{ form.address.label_tag }} {{ form.address }}</p>
</form>
```

See the [forms documentation](#) for more information about using Form objects in templates.

- **object:** The original object being edited. This variable's name depends on the `template_object_name` parameter, which is 'object' by default. If `template_object_name` is 'foo', this variable's name will be `foo`.

`django.views.generic.create_update.delete_object`

Descrição:

A view that displays a confirmation page and deletes an existing object. The given object will only be deleted if the request method is POST. If this view is fetched via GET, it will display a confirmation page that should contain a form that POSTs to the same URL.

Argumentos obrigatórios:

- **model:** The Django model class of the object that the form will create.
- Either `object_id` or (`slug` and `slug_field`) is required.

If you provide `object_id`, it should be the value of the primary-key field for the object being displayed on this page.

Otherwise, `slug` should be the slug of the given object, and `slug_field` should be the name of the slug field in the QuerySet's model. By default, `slug_field` is 'slug'.

- **post_delete_redirect:** A URL to which the view will redirect after deleting the object.

Optional arguments:

- **login_required:** A boolean that designates whether a user must be logged in, in order to see the page and save changes. This hooks into the Django [authentication system](#). By default, this is `False`.

If this is `True`, and a non-logged-in user attempts to visit this page or save the form, Django will redirect the request to `/accounts/login/`.

- **template_name:** The full name of a template to use in rendering the page. This lets you override the default template name (see below).
- **template_loader:** The template loader to use when loading the template. By default, it's `django.template.loader`.
- **extra_context:** A dictionary of values to add to the template context. By default, this is an empty dictionary. If a value in the dictionary is callable, the generic view will call it just before rendering the template.
- **context_processors:** A list of template-context processors to apply to the view's template.
- **template_object_name:** Designates the name of the template variable to use in the template context. By default, this is 'object'.

Template name:

If `template_name` isn't specified, this view will use the template `<app_label>/<model_name>_confirm_delete.html` by default.

Template context:

In addition to `extra_context`, the template's context will be:

- `object`: The original object that's about to be deleted. This variable's name depends on the `template_object_name` parameter, which is `'object'` by default. If `template_object_name` is `'foo'`, this variable's name will be `foo`.

Referência dos middlewares embutidos

Este documento detalha todos os componentes middleware que acompanham o Django. Para informações de como usá-los e de como escrever o seu próprio middleware, veja o *guia de uso de middleware*.

Middlewares disponíveis

Middleware de cache

```
class django.middleware.cache.UpdateCacheMiddleware
```

```
class django.middleware.cache.FetchFromCacheMiddleware
```

Habilita o cache em todo o site. Se este estiver habilitado, cada página construída pelo Django será cacheada durante o tempo definido em `CACHE_MIDDLEWARE_SECONDS`. Veja a *documentação do cache*.

Middleware “Common”

```
class django.middleware.common.CommonMiddleware
```

Adiciona algumas conveniências para perfeccionistas:

- Proíbe o acesso de “user agents” especificados em `DISALLOWED_USER_AGENTS`, o qual deve ser uma lista de strings.
- Executa uma reescrita de URL baseada nas configurações `APPEND_SLASH` e `PREPEND_WWW`.

Se o `APPEND_SLASH` for `True` e a URL inicial não terminar com uma barra, e ela não for encontrada no URLconf, então uma nova URL é formada, adicionando uma barra no final. Se esta nova URL é encontrada no URLconf, o Django redireciona a requisição para esta nova URL. Por outro lado, a URL inicial é processada habitualmente.

Por exemplo, `foo.com/bar` será redirecionado para `foo.com/bar/` se você não tiver um padrão de URL válido para `foo.com/bar` mas *tem* um padrão válido para `foo.com/bar/`. O comportamento do `APPEND_SLASH` mudou ligeiramente nesta versão. Ele não é utilizado para verificar se o padrão confere na URLconf. Se o `PREPEND_WWW` é `True`, as URLs que faltam um “www.” na frente serão redirecionadas para a mesma URL com o “www.” na frente.

Estes dois tipos de opções são destinadas a normalizar URLs. A filosofia é que cada URL deve existir em um, e somente um, lugar. Tecnicamente uma URL `foo.com/bar` é diferente de `foo.com/bar/`

– um indexador de motor de busca tratará elas como URLs separadas – então esta é a melhor prática para normalizar URLs.

- Manipula ETags baseado na configuração `USE_ETAGS`. Se o `USE_ETAGS` é definido como `True`, o Django calculará um ETag para cada requisição usando um hash MD5 do conteúdo da página, e irá se responsabilizar por enviar repostas `Not Modified` (não modificado), se apropriado.

Middleware view metadata

`class django.middleware.doc.XViewMiddleware`

Envia cabeçalhos HTTP X-View personalizados para requisições HEAD oriundas de endereços IP definidos na configuração `INTERNAL_IPS`. Isso é utilizado pelo sistema de documentação automática do Django.

Middleware GZIP

`class django.middleware.gzip.GZipMiddleware`

Comprime conteúdos para navegadores que entendem compressão gzip (todos os navegadores modernos).

É sugerido colocá-lo em primeiro na lista de middlewares, desta forma a compressão do conteúdo de resposta será a última coisa a ser feita. Não serão comprimidos conteúdos menores que 200 bytes, quando o código de resposta for diferente de 200, quando arquivos JavaScript (para compatibilidade com IE), ou quando as respostas possuem o cabeçalho `Content-Encoding` já especificado.

Middleware GET condicional

`class django.middleware.http.ConditionalGetMiddleware`

Manipula operações condicionais do GET. Se a resposta tem um cabeçalho ETag ou Last-Modified, e a requisição possui `If-None-Match` ou `If-Modified-Since`, a resposta é substituída por um `HttpNotModified`.

Também define os cabeçalhos de resposta `Date` e `Content-Length`.

Middleware para proxy reverso

`class django.middleware.http.SetRemoteAddrFromForwardedFor`

Define o `request.META['REMOTE_ADDR']` baseado no `request.META['HTTP_X_FORWARDED_FOR']`, se o último estiver definido. Isso é útil se você estiver por trás de um proxy reverso que faz com que cada `REMOTE_ADDR` seja definido como `127.0.0.1`.

Nota importante: Isso NÃO valida o `HTTP_X_FORWARDED_FOR`. Se você não está atrás de um proxy reverso que define o `HTTP_X_FORWARDED_FOR` automaticamente, não utilize este middleware. Ninguém pode burlar o valor do `HTTP_X_FORWARDED_FOR`, e por causa disso defina `REMOTE_ADDR` baseado no `HTTP_X_FORWARDED_FOR`, o que significa que ninguém pode “falsificar” seus endereços de IP. Somente use isto quando você confia absolutamente no valor do `HTTP_X_FORWARDED_FOR`.

Middleware Locale

`class django.middleware.locale.LocaleMiddleware`

Habilita a seleção de idioma baseado em dados vindos da requisição. Ele personaliza o conteúdo para cada usuário. Veja a *documentação de internacionalização*.

Middleware Session

`class django.contrib.sessions.middleware.SessionMiddleware`

Habilita o suporte a sessões. Veja a *documentação de sessões*.

Middleware Authentication

`class django.contrib.auth.middleware.AuthenticationMiddleware`

Adiciona o atributo `user`, representando o usuário atualmente logado, para todo objeto `HttpRequest` que chega. Veja *Autenticação em requisições Web*.

Middleware de proteção CSRF

`class django.contrib.csrf.middleware.CsrfMiddleware`

Please, see the release notes Adiciona proteção contra Cross Site Request Forgeries adicionando campos invisíveis de formulários aos formulários POST e checando suas requisições pelos valores corretos. Veja a *documentação da proteção de Cross Site Request Forgeries*.

Middleware Transaction

`class django.middleware.transaction.TransactionMiddleware`

Vincula commit e rollback às fases request/response. Se uma função view é executada com sucesso, um commit é feito. Se ela falhar com uma exceção, um rollback é realizado.

A ordem deste middleware na pilha é importante: módulos middleware rodando fora dele rodam com commit-on-save - o comportamento padrão do Django. Módulos middleware rodando dentro dele (vêm depois na pilha) estarão abaixo do mesmo controle de transações como nas funções view.

Veja a *documentação de gerenciamento de transações*.

Referência da API de modelos. Para um material introdutório, veja *Models*.

Referência de campos do Model

Este documento contém todos os detalhes sobre todas as *opções dos campos* e *tipos de campos* que o Django oferece.

See also:

Se os campos embutidos não atendem a sua necessidade, você pode facilmente *escrever seus próprios campos de model*.

Note: Tecnicamente, estes models são definidos em `django.db.models.fields`, mas por conveniência eles são importados dentro do `django.db.models`; a convenção padrão é usar `from django.db import models` e referir-se aos campos como `models.<Foo>Field`.

Opções dos campos

O seguintes argumentos estão disponíveis para todos os tipos de campos. Todos são opcionais.

`null`

`Field.null`

Se `True`, o Django irá gravar valores vazios como `NULL` no banco de dados. O padrão é `False`.

Repare que strings vazias sempre serão gravadas como strings vazias, não como `NULL`. Use o `null=True` apenas para campos que não sejam texto, como inteiros, booleanos e datas. Para ambos os tipos de campos, você irá precisar configurar o `blank=True` se você quer permitir valores vazios nos formulários, como o parâmetro `null` afeta somente o banco de dados (veja *blank*, abaixo).

Evite o uso de `null` em campos de texto, como *CharField* e *TextField* a menos que você tenha uma ótima razão. Se um campo baseado em texto tem `null=True`, isso significa que ele tem dois valores possíveis para

“sem dados”: `NULL`, e a string vazia. Na maioria dos casos, é redundante ter dois valores possíveis para “sem dados;” a convenção do Django é usar a string vazia, não `NULL`.

Note: Ao usar o Oracle como backend, a opção `null=True` será convertida para campos de texto que podem ser vazios, e o valor `NULL` será gravado para indicar a string vazia.

`blank`

`Field.blank`

Se `True`, o campo pode ser vazio. o padrão é `False`.

Note que isso é diferente de `null`. `null` é puramente relacionado ao banco de dados, e `blank` é relacionado com validação. Se um campo tem `blank=True`, a validação na administração do Django irá permitir a entrada de um valor vazio. Se um campo tem `blank=False`, o campo será obrigatório.

`choices`

`Field.choices`

Um iterável(e.g., uma lista ou tupla) de tupla duplas para usar como escolhas para esse campo.

Se fornecido, a administração do Django irá usar uma caixa de seleção no lugar de um campo de texto padrão e irá limitar as escolhas as opções dadas.

Uma lista de opções parece com isto:

```
YEAR_IN_SCHOOL_CHOICES = (
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
)
```

O primeiro elemeno de cada tupla é o verdadeiro valor a ser gravado. O segundo elemento é o nome legível por humanos para essa opção.

A lista de escolhas pode ser definida como uma parte de sua classe de modelo:

```
class Foo(models.Model):
    GENDER_CHOICES = (
        ('M', 'Male'),
        ('F', 'Female'),
    )
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
```

ou fora da sua classe de modelo:

```
GENDER_CHOICES = (
    ('M', 'Male'),
    ('F', 'Female'),
)
class Foo(models.Model):
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
```

Você pode também coletar suas escolhas disponíveis em grupos nomeados que podem ser usados com o proposito de organização:

```
MEDIA_CHOICES = (
    ('Audio', (
        ('vinyl', 'Vinyl'),
        ('cd', 'CD'),
    )),
    ('Video', (
        ('vhs', 'VHS Tape'),
        ('dvd', 'DVD'),
    )),
    ('unknown', 'Unknown'),
)
```

O primeiro elemento em cada tupla é o nome a ser aplicado no grupo. O segundo elemento é um iterável de tuplas duplas, com cada tupla dupla contendo um valor e um nome, legível para humanos, para uma opção. Opções agrupadas podem ser combinadas com opções não agrupadas em uma única lista (como a opção *unknown* deste exemplo).

Para cada campo de modelo que tem o *choices* configurado, o Django adicionará um método para obter o valor legível para o valor atual do campo. Veja *get_FOO_display()* na documentação da API de banco de dados.

Finalmente, veja que as opções podem ser qualquer objeto iterável – não necessariamente uma lista ou tupla. Isso permite que você construa suas opções dinamicamente. Mas se você se pegar hackeando o *choices* para ser dinâmico, você provavelmente deveria estar usando uma tabela do banco de dados com uma *ForeignKey*. *choices* deve ser usados para dados que não mudem muito, ou que não mudem nunca.

db_column

Field.db_column

O nome da coluna do banco de dados a ser usada com esse campo. Se não for informada, o Django irá usar o nome do campo.

Se o nome da sua coluna no banco de dados é uma palavra chave SQL reservada, ou contém caracteres que não são permitidos em nomes de variáveis no Python – notadamente, o hífen tudo bem. O Django escapa os nomes de colunas e tabelas por trás dos panos.

db_index

Field.db_index

Se “True”, *django-admin.py sqlindexes <sqlindexes>* irá gerar uma declaração `CREATE INDEX` para esse campo.

db_tablespace

Field.db_tablespace

Please, see the release notes O nome do tablespace no banco de dados a ser usado para esse o índice desse campo, se esse campo é indexado. O padrão é a configuração `DEFAULT_INDEX_TABLESPACE` do projeto, se configurado, ou o *db_tablespace* do modelo, se houver. Se o backend não suporta tablespaces, essa opção é ignorada.

default

Field.default

O valor padrão para o campo. Pode ser também um objeto chamável. Se for um chamável, será chamado a cada vez que um novo objeto for criado.

editable

Field.editable

Se `False`, o campo não será editável na administração ou por formulários gerados automaticamente a partir de `models`. O padrão é `True`.

help_text

Field.help_text

“Ajuda” extra a ser exibida sobre o campo na administração de formulários do objeto. É útil para documentação, mesmo que seu objeto não tenha um formulário administrativo.

Note que esse valor *não* é escapado pelo processamento de HTML ao ser exibido na interface administrativa. Isso permite que você inclua HTML no `help_text` se você desejar. Por exemplo:

```
help_text="Por favor use o seguinte formato: <em>DD/MM/YYYY</em>."
```

Como alternativa você pode utilizar texto plano e `django.utils.html.escape()` para escapar quaisquer caracter especial do HTML.

primary_key

Field.primary_key

Se `True`, esse campo será a chave primária para o modelo.

Se você não especificar `primary_key=True` para nenhum campo no seu modelo, o Django irá adicionar automaticamente um `IntegerField` para ser a chave primária, então você não precisa setar `primary_key=True` sobre algum de seus campos, a menos que você queira sobrescrever o comportamento de chave primária padrão. Para mais, veja *[Campos de chave primária automáticos](#)*.

`primary_key=True` implica em `blank=False`, `null=False` (`<Field.null>`) e `unique=True`. So-mente uma chave primária é permitida por objeto.

unique

Field.unique

Se `True`, esse campo deve ser único na tabela.

Isso é garantido no banco de dados e no formulário de administração do Django. Se você tentar salvar um modelo com um valor duplicado em um campo `unique`, uma exceção `django.db.IntegrityError` será lançada pelo método `save()` do modelo.

Esta opção é válida para todos os tipos de campos exceto `:class:ManyToManyField` e `FileField`.

unique_for_date

Field.unique_for_date

Configure isso para o nome de um campo `DateField` ou `DateTimeField` para exigir que esse campo seja único para o valor da data.

Por exemplo, se você tem um campo `title` que tem `unique_for_date="pub_date"`, o Django não irá permitir a criação de dois registros com o mesmo `title` e `pub_date`.

Isso é reforçado no nível do formulário de administração do Django, mas não no banco de dados.

`unique_for_month`

`Field.unique_for_month`

Semelhante ao `unique_for_date`, mas requer que o campo seja único com respeito ao mês.

`unique_for_year`

`Field.unique_for_year`

Semelhante a `unique_for_date` e `unique_for_month`.

`verbose_name`

`Field.verbose_name`

Um nome, legível para humanos, para o campo. Se o nome prolixo não for dado, o Django criará um automaticamente, usando o nome do campo, convertendo os underscores em espaços. Veja *Nomes de campos por extenso*.

Tipos de campos

`AutoField`

`class AutoField(**options)`

Um `IntegerField` que incrementa seu valor automaticamente de acordo com os IDs disponíveis. Você normalmente não precisa usá-lo diretamente; um campo de chave primária será automaticamente adicionado ao seu modelo se você não especificar um. Veja *Campos de chave primária automáticos*.

`BooleanField`

`class BooleanField(**options)`

Um campo Verdadeiro/Falso.

O site administrativo representa-o como uma checkbox.

MySQL users..

Um campo booleano é armazenado no MySQL como uma coluna `TINYINT` com o valor sendo 0 ou 1 (a maioria dos bancos de dados possuem a propriedade do tipo `BOOLEAN` ao invés). Então, para o MySQL, somente, quando um `BooleanField` é recebido de um banco de dados e armazenado em um atributo de model, ele terá os valores 1 ou 0, ao invés de `True` ou `False`. Normalmente, isto não deveria ser um problema, desde que o Python garanta que `0 == True` e `0 == False` sejam ambos verdadeiros. Somente tenha cuidado se você estiver escrevendo algo como `obj is True` quando `obj` é um valor vindo de um atributo booleano de um model. Se esse model foi construído usando o backend `mysql`, o teste “`is`” falhará. Dê preferência para um teste de igualdade (usando “`==`”) em casos como este.

`CharField`

`class CharField(max_length=None[, **options])`

Um campo string, para campos texto de tamanhos pequeno e médio.

Para grandes quantidades de texto, use `django.db.models.TextField`.

A administração representa-o como um `<input type="text">` (um input de uma única linha).

`CharField` tem um argumento extra obrigatório:

`CharField.max_length`

O tamanho máximo (em caracteres) do campo. O `max_length` é verificado a nível de banco de dados e na validação do Django.

Note: Se você estiver escrevendo uma aplicação que deve ser portátil para vários backends de banco de dados, você deve estar alerta, pois há restrições para o `max_length` em alguns backends. Leia as [notas sobre backend de bancos de dados](#).

MySQL users

Se você estiver usando este campo com MySQLdb 1.2.2 e a colação `utf8_bin` (que *não* é a padrão), há alguns problemas que deve ter em mente. Leia as [notas do banco de dados MySQL](#) para detalhes.

CommaSeparatedIntegerField

`class CommaSeparatedIntegerField (max_length=None[, **options])`

Um campo de inteiros separados por vírgulas. Como no `:class 'CharField'`, o argumento `max_length` é obrigatório e atente para as portabilidades mencionadas.

DateField

`class DateField ([auto_now=False, auto_now_add=False, **options])`

Uma data, representada por uma instância `datetime.date` do Python. Tem alguns poucos argumentos extra, opcionais:

`DateField.auto_now`

Configura o campo automaticamente para a hora em que o objeto é salvo. Útil para campos de hora da “última modificação”. Note que *sempre* é usada a data atual, não é apenas um valor padrão que você pode sobrescrever.

`DateField.auto_now_add`

Configura automaticamente o campo para a data em que ele foi primeiramente criado. Útil para a criação de carimbos de data. Note que *sempre* é usada a data atual, não é apenas um valor padrão que você pode sobrescrever.

A administração representa isso como um `<input type="text">` com um calendário JavaScript, e um atalho para “Hoje.” O calendário JavaScript sempre irá iniciar a semana no Domingo.

DateTimeField

`class DateTimeField ([auto_now=False, auto_now_add=False, **options])`

A date and time, represented in Python by a `datetime.datetime` instance. Um campo de data e hora. Tem as mesmas opções extras de `DateField`.

A administração representa esse campo como dois campos `<input type="text">`, com atalhos JavaScript.

DecimalField

Please, see the release notes

class `DecimalField` (`max_digits=None`, `decimal_places=None`[, `**options`])

Um número decimal de precisão fixa, representado no Python por uma instância de `Decimal`. Possui dois argumentos **obrigatórios**:

`DecimalField.max_digits`

O número máximo de dígitos permitidos no número.

`DecimalField.decimal_places`

O número de casas decimais para salvar com o número.

Por exemplo, para gravar números até 999 com uma precisão de 2 casas decimais, você deve usar:

```
models.DecimalField(..., max_digits=5, decimal_places=2)
```

E para guardar números de até aproximadamente um bilhão, com uma precisão de 10 casas decimais:

```
models.DecimalField(..., max_digits=19, decimal_places=10)
```

A administração representa isso como um `<input type="text">` (um input de uma única linha).

EmailField

class `EmailField` ([`max_length=75`, `**options`])

Um `CharField` que verifica se o valor é um e-mail válido.

FileField

class `FileField` (`upload_to=None`[, `max_length=100`, `**options`])

Um campo para upload de arquivo. Possui um argumento **obrigatório**

Note: O argumentos `primary_key` e `unique` não são suportados, e lançarão um `TypeError` se usados.

Has one **required** argument:

`FileField.upload_to`

Um caminho no seu sistema de arquivos local que será adicionado a sua configuração `MEDIA_ROOT` para determinar o valor do atributo `url`.

Esse caminho pode conter *formatação strftime*, que será substituído pela data/hora do upload do arquivo (para que os arquivos enviados não encham o diretório fornecido). *Please, see the release notes* Este pode também ser um chamável, como uma função, que será chamada para obter o caminho do upload, incluindo o nome do arquivo. Este chamável deve ser preparado para receber dois argumentos, e retornar um caminho no estilo Unix (com barras) para serem passados ao longo do sistema de armazenamento. Os dois argumentos que serão passados são:

Argumento	Descrição
<code>instance</code>	Uma instância do model onde o <code>FileField</code> é definido. Mais especificamente, esta é a instância particular onde o arquivo atual está atachado. Na maioria dos casos, este objeto não terá sido salvo no banco de dados ainda, então se ele usa o padrão <code>AutoField</code> , <i>ele pode não ter ainda um valor para seu campo de chave primária</i> .
<code>filename</code>	O nome que foi originalmente dado para o arquivo. Este pode ou não pode ser levado em conta durante a definição do caminho de destino final.

Também há um argumento opcional:

`FileField.storage`

Please, see the release notes Opcional. Um objeto storage, que manipula o armazenamento e recebimento de seus arquivos. Veja *Gerenciando arquivos* para mais detalhes de como prover este objeto.

A administração representa esse campo como um `<input type="file">` (um widget de upload de arquivo).

O uso de um `FileField` ou um `ImageField` (veja abaixo) em um modelo requer alguns passos adicionais:

1. No seu arquivo de configurações, você precisa definir `MEDIA_ROOT` como o caminho completo para o diretório onde você gostaria que o Django guardasse os arquivos enviados. (Por questões de performance, esses arquivos não são guardados no banco de dados). Defina o `MEDIA_URL` como a URL pública desse diretório. Assegure-se de que esse diretório tenha permissão de escrita pela conta do usuário que executa o servidor web.
2. Adiciona o `FileField` ou `ImageField` ao seu modelo, definindo a opção `upload_to` para dizer ao Django para qual subdiretório do `MEDIA_ROOT` ele deve enviar os arquivos.
3. Tudo o que será gravado no seu banco de dados é o caminho para o arquivo (relativo ao `MEDIA_ROOT`). Você muito provavelmente usará a função `url` fornecida pelo Django. Por exemplo, se seu `ImageField` é chamado de `mug_shot`, você pode obter a URL absoluta para sua imagem em um template com `{{ object.get_mug_shot.url }}`.

Por exemplo, digamos que o seu `MEDIA_ROOT` esteja configurado para `"/home/media"`, e o `upload_to` está configurado para `"photos/%Y/%m/%d"`. A parte `"%Y/%m/%d"` do `upload_to` é a *formatação strftime*; `"%Y"` é o ano com quatro dígitos, `"%m"` é o mês com dois dígitos e `"%d"` é o dia com dois dígitos. Se você enviar um arquivo em 15 de janeiro de 2007, ele será salvo no diretório `/home/media/photos/2007/01/15`.

Se você quer obter o nome do arquivo no disco, ou a URL que se refere àquele arquivo, ou o tamanho do arquivo, você pode usar os métodos `name`, `url` e `size`. Veja *Gerenciando arquivos*.

Observe que independente da forma que você lida com arquivos enviados, você deve prestar muita atenção para onde você os está enviado e para quais tipos eles são, para evitar furos de segurança. *Valide todos os arquivos enviados* para assegurar-se que eles são o que você pensa que eles são. Por exemplo, se você permite que alguém envie arquivos sem validação, para um diretório que esteja dentro do document root do seu servidor web, então alguém poderia enviar um script CGI ou PHP e executar esse script visitando essa URL em seu site URL. Não permita isso. O argumento `max_length` foi adicionado nesta versão. Por padrão, instâncias de `FileField` são criadas como colunas `varchar(100)` no seu banco de dados. Assim como nos outros campos, você pode mudar o tamanho máximo usando o argumento `max_length`.

FilePathField

```
class FilePathField (path=None[, match=None, recursive=False, max_length=100, **options])
```

Um campo `CharField` cujas escolhas são limitadas a nomes de arquivos em um certo diretório no sistema de arquivos. Tem três argumentos especiais, sendo o primeiro **obrigatório**:

`FilePathField.path`

Obrigatório. O caminho absoluto para um diretório no sistema de arquivos a partir de onde o `FilePathField` deve obter suas opções. Exemplo: `"/home/images"`.

FilePathField.match

Opcional. Uma expressão regular, como uma string, que o *FilePathField* usará para filtrar os nomes dos arquivos. Perceba que a regex será aplicada apenas ao nome do arquivo, não ao caminho completo. Exemplo: "foo.*\.txt\$", que irá casar um arquivo chamado foo23.txt mas não bar.txt ou foo23.gif.

FilePathField.recursive

Opcional. True ou "False". O padrão é False. Especifica se todos os subdiretórios do *path* devem ser incluídos.

E é claro que todos esses argumentos podem ser usados juntos.

A única pegadinha em potencial é que o *match* aplica-se ao nome básico do arquivo, não ao caminho completo. Assim, esse exemplo:

```
FilePathField(path="/home/images", match="foo.*", recursive=True)
```

...irá casar /home/images/foo.gif mas não /home/images/foo/bar.gif porque o argumento *match* aplicase aos nomes básicos dos arquivos (foo.gif e bar.gif). O argumento *max_length* foi adicionado nesta versão. Por padrão, instâncias de *FilePathField* são criadas como colunas *varchar*(100) no seu banco de dados. Assim como nos outros campos, você pode mudar o tamanho máximo usando o argumento *max_length*.

FloatField

```
class FloatField(**options)
```

Please, see the release notes Um número de ponto flutuante, representado no Python como uma instância de um float.

A administração representa-o como um <input type="text"> (um input de uma única linha).

ImageField

```
class ImageField(upload_to=None[, height_field=None, width_field=None, max_length=100, **options])
```

Semelhante ao *FileField*, mas verifica se o objeto enviado é uma imagem válida. Tem dois argumentos extras opcionais:

ImageField.height_field

Nome de um campo de model que será auto-populado com a altura da imagem toda vez que a instância do model for salva.

ImageField.width_field

Nome de um campo de model que será auto-populado com a largura da imagem toda vez que a instância do model for salva.

Além dos métodos especiais que estão disponíveis para *FileField*, um class:*ImageField* também tem os atributos *File.height* e *File.width*. Veja *Gerenciando arquivos*.

Requer a *Python Imaging Library*. O argumento *attr:~CharField.max_length* foi adicionado nesta versão. Por padrão, instâncias de *ImageField* são criadas como colunas *varchar*(100) no seu banco de dados. Assim como nos outros campos, você pode mudar o tamanho máximo usando o argumento *max_length*.

IntegerField

```
class IntegerField(**options)
```

Um inteiro. A administração representa-o como um <input type="text"> (um input de uma única linha).

IPAddressField

```
class IPAddressField(**options)
```

Um endereço IP, em formato texto (ex: “192.0.2.30”). A administração representa-o como um `<input type="text">` (um input de uma única linha).

NullBooleanField

```
class NullBooleanField(**options)
```

Como um *BooleanField*, mas permite NULL como uma das opções. Use-o no lugar de um *BooleanField* com `null=True`. A administração representa-o como uma caixa `<select>` com as escolhas “Unknown”, “Yes” e “No”.

PositiveIntegerField

```
class PositiveIntegerField(**options)
```

Como um *IntegerField*, mas deve ser positivo.

PositiveSmallIntegerField

```
class PositiveSmallIntegerField(**options)
```

Como um *PositiveIntegerField*, mas somente permite valores até um certo ponto (de acordo com o tipo de banco de dados utilizado).

SlugField

```
class SlugField(max_length=50, **options)
```

Slug é um termo jornalístico. Um slug é um apelido curto para algo, contendo somente letras, números, under-scores ou hífens. Eles normalmente são usados em URLs.

Semelhantemente ao *CharField*, você pode especificar o *max_length* (leia a nota sobre portabilidade de banco de dados e *max_length* nesta seção também). Se o *max_length* não for especificado, o Django irá usar um tamanho padrão de 50.

Implica em *Field.db_index*.

É muitas vezes mais útil para pre-popular automaticamente um campo baseado no seu valor. Você pode fazer isso automaticamente no admin usando *prepopulated_fields*.

SmallIntegerField

```
class SmallIntegerField(**options)
```

Como um *IntegerField*, mas somente permite valores até um certo ponto (dependendo do banco de dados).

TextField

```
class TextField(**options)
```

Um campo de texto longo. A administração o representa como uma `<textarea>` (um input de múltiplas linhas).

MySQL users

Se você estiver usando este campo com MySQLdb 1.2.1p2 e a colação `utf8_bin` (que *não* é a padrão), há alguns problemas que deve ter em mente. Leia as [notas do banco de dados MySQL](#) para detalhes.

TimeField

```
class TimeField([auto_now=False, auto_now_add=False, **options])
```

Uma hora. Representado em Python por uma instância `datetime.time`. Aceita as mesmas opções de auto preenchimento dos campos `DateField`.

A administração o representa como uma `<input type="text">` com alguns atalhos JavaScript.

URLField

```
class URLField([verify_exists=True, max_length=200, **options])
```

Um `CharField` para uma URL. Possui um argumento extra opcional:

`URLField.verify_exists`

Se a opção for `True` (padrão), a existência da URL será verificada (i.e., será verificado se a URL carrega e não retorna uma resposta 404). It should be noted that when using the single-threaded development server, validating a url being serverd by the same server will hang. This should not be a problem for multithreaded servers.

A administração representa-o como um `<input type="text">` (um input de uma única linha).

Como toda subclasse de `CharField`, `URLField` recebe um argumento opcional, `max_length`. Se você não especificar o `max_length`, o padrão de 200 é usado.

XMLField

```
class XMLField(schema_path=None[, **options])
```

Um `TextField` que verifica se o valor é um XML válido de acordo com um esquema fornecido. Possui um argumento obrigatório:

`schema_path`

O caminho no sistema de arquivos para um esquema [RelaxNG](#) que será usado para a validação do campo.

Campos de relacionamento

O Django também define um conjunto de campos que representam relacionamentos.

ForeignKey

```
class ForeignKey(othermodel[, **options])
```

Uma reção muitos-para-um. Requer um argumento posicional: a classe com a qual o model está relacionada. Para criar um relacionamento recursivo – um objeto que tem um relacionamento muitos-para-um consigo mesmo – use `models.ForeignKey('self')`. Se você precisa criar um relacionamento em um model que não não foi definido ainda, você pode usar o nome do model, ao invés do objeto em si:


```
class Car(models.Model):
    manufacturer = models.ForeignKey('Manufacturer')
    # ...

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

Please, see the release notes Para referir-se a models definidos em outra aplicação, você pode explicitamente especificar um model com a label completa da aplicação. Por exemplo, se o model `Manufacturer` acima é definido em outra aplicação chamada `production`, você precisará usar:

```
class Car(models.Model):
    manufacturer = models.ForeignKey('production.Manufacturer')
```

Este tipo de referência pode ser útil para resolver dependências de import circular entre duas aplicações.

Representação no banco de dados

Por trás das cenas, o Django adiciona `"_id"` ao nome do campo para criar suas nomes de colunas. No exemplo acima, a tabela para o model `Car` terá uma coluna `manufacturer_id`. (Você pode mudar isto sendo explícito, especificando `db_column`) No entanto, seu código nunca deve lidar com o nome da coluna do banco de dados, a menos que você esteja escrevendo SQL customizado. Você sempre trabalhará com os nomes de campos de seu objeto model.

Argumentos

A `ForeignKey` aceita um conjunto extra de argumentos – todos opcionais – que definem detalhes de como a relação funciona.

`ForeignKey.limit_choices_to`

Um dicionário de argumentos e valores aparente (veja *Fazendo consultas*) que limita as escolhas disponíveis para o admin sobre este objeto. Use isto com funções do módulo do Python `datetime` para limitar as escolhas dos objetos por data. Por exemplo:

```
limit_choices_to = {'pub_date__lte': datetime.now}
```

permite somente a escolha de objetos relatados com uma `pub_date` anterior a data/hora atual.

Ao invés de um dicionário este pode ser também um objeto `Q` (um objeto com um método `get_sql()`) para consultas mais complexas.

O `limit_choices_to` não tem efeito sobre os `FormSets` inline que são criados para mostra objetos relacionados no admin.

`ForeignKey.related_name`

O nome para ser usado pelo relacionamento do objeto relacionado para este. Veja a *documentação de objetos relacionados* para uma explicação completa e exemplos. Note que você deve setar este valor quando estiver definindo relacionamentos em *models abstratos*; e quando você utilizar *alguma sintaxe especial* disponível.

`ForeignKey.to_field`

O campo para qual o objeto relacionado aponta. Por padrão, o Django usa a chave primária dos objetos relacionados.

ManyToManyField

```
class ManyToManyField(othermodel[, **options])
```

Um relacionamento muitos-para-muitos. Requer um argumento posicional: a classe com a qual o model se relaciona. Isso funciona exatamente da mesma forma como um *ForeignKey*, incluindo todas as opções em relação a relacionamentos *recursivos* e *lazy*.

Representação no banco de dados.

Por trás das cenas, o Django cria uma tabela de ligação intermediária para representar o relacionamento muitos-para-muitos. Por padrão, o nome desta tabela é gerado usando os nomes das duas tabelas juntos. Desde que alguns bancos de dados não suportam nomes de tabelas acima de um certo comprimento, estes nomes serão automaticamente truncados para 64 caracteres e um hash de unicidade será utilizado. Isto significa que pode ver nomes de tabelas como `author_books_9cdf4`; isto é perfeitamente normal. Você pode manualmente provê o nome da tabela de ligação usando a opção *db_table*.

Argumentos

O *ManyToManyField* aceita um conjunto extra de argumentos – todos opcionais – que controlam como o relacionamento funciona.

`ManyToManyField.related_name`

O mesmo que *ForeignKey.related_name*.

`ManyToManyField.limit_choices_to`

O mesmo que *ForeignKey.limit_choices_to*.

O `limit_choices_to` não surte efeito quando usado em um *ManyToManyField* com uma tabela intermediária personalizada usando o parâmetro *through*.

`ManyToManyField.symmetrical`

Somente usado na definição de *ManyToManyFields* no *self*. Considere o seguinte model:

```
class Person(models.Model):
    friends = models.ManyToManyField("self")
```

Quando o Django processa este model, ele identifica que ele tem um *ManyToManyField* para si mesmo, e como um resultado, ele não adiciona um atributo `person_set` a classe *Person*. Em vez disso, o *ManyToManyField* é assumido como simétrico – que quer dizer, se eu sou seu amigo, então você é meu amigo.

Se você não quer simetria em relacionamentos muitos-para-muitos com *self*, mude *symmetrical* para *False*. Isto forçará o Django a adicionar um descrito para o reverso do relacionamento, permitindo relacionamentos *ManyToManyField* serem não-simétricos.

`ManyToManyField.through`

O Django gerará automaticamente uma tabela para gerenciar o relacionamento muitos-para-muitos. Entretanto, se você quer especificar manualmente a tabela intermediária, você pode usar a opção *through* para especificar o model do Django que representa a tabela intermediária que você deseja usar.

O uso mais comum para esta opção é quando você quer associar *dados extra com um relacionamento muitos-para-muitos*.

`ManyToManyField.db_table`

O nome da tabela a ser criada para o armazenamento dos dados muitos-para-muitos. Se este não for fornecido, o Django assumirá um nome padrão baseado nos nomes das duas tabelas que serão juntadas.

OneToOneField

```
class OneToOneField(othermodel[, parent_link=False, **options])
```

Um relacionamento um-para-um. Conceitualmente, este é similar a um *ForeignKey* com *unique=True*, mas o lado “reverso” da relação retornará diretamente um único objeto.

Este é mais útil como a chave primária de um model que “estende” outro model de alguma forma; a *Herança com Multi-Tabelas* é implementada adicionando uma relação implícita um-para-um do model filho para o model pai, por exemplo.

Um argumento posicional é obrigatório: a classe para a qual o model será relacionado. Isso funciona exatamente da mesma forma como se fosse um *ForeignKey*, incluindo todas as opções em relação a relacionamentos *recursivos* e *lazy*. Adicionalmente, *OneToOneField* aceita todos os argumentos extra aceitos pelo *ForeignKey*, mais um argumento extra:

`OneToOneField.parent_link`

Quando `True` e usado em um model que herda de outro model (concreto), indica que este campo deve ser usado como o link para a classe pai, ao invés de um *OneToOneField* extra que normalmente seria implicitamente criado pela subclasse.

Referência de objetos relacionados

This document describes extra methods available on managers when used in a one-to-many or many-to-many related context. This happens in two cases:

- The “other side” of a *ForeignKey* relation. That is:

```
class Reporter(models.Model):
    ...

class Article(models.Model):
    reporter = models.ForeignKey(Reporter)
```

In the above example, the methods below will be available on the manager `reporter.article_set`.

- Both sides of a *ManyToManyField* relation:

```
class Topping(models.Model):
    ...

class Pizza(models.Model):
    toppings = models.ManyToManyField(Topping)
```

In this example, the methods below will be available both on `topping.pizza_set` and on `pizza.toppings`.

`QuerySet.add(obj1[, obj2, ...])`

Adiciona objeto model especificado para o conjunto de objetos relacionados.

Exemplo:

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.add(e) # Associa Entry e com Blog b.
```

`QuerySet.create(**kwargs)`

Cria um novo objeto, o salva e o coloca no conjunto de objetos relacionados. Retorna o novo objeto criado:

```
>>> b = Blog.objects.get(id=1)
>>> e = b.entry_set.create(
...     headline='Hello',
...     body_text='Hi',
...     pub_date=datetime.date(2005, 1, 1)
... )

# Não necessita chamar e.save() neste ponto -- ele já está salvo.
```

Isto é equivalente (mas muito mais simples):

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry(
....     blog=b,
....     headline='Hello',
....     body_text='Hi',
....     pub_date=datetime.date(2005, 1, 1)
.... )
>>> e.save(force_insert=True)
```

Note que não há necessidade de especificar argumentos nomeados para o model que define a relação. No exemplo acima, nós não passamos o parâmetro `blog` para `create()`. O Django descobre que o novo campo `blog` do objeto `Entry` deve ser definido como `b`.

`QuerySet.remove(obj1[, obj2, ...])`

Remove o model especificado do conjunto de objetos relacionados:

```
>>> b = Blog.objects.get(id=1)
>>> e = Entry.objects.get(id=234)
>>> b.entry_set.remove(e) # Disassocia Entry e de Blog b.
```

A fim de prevenir inconsistências no banco de dados, este método somente existe em objetos `ForeignKey` onde `null=True`. Se o campo relacionado não puder ser `None` (`NULL`), então o objeto não pode ser removido de uma relação sem ser adicionado em outra. No exemplo acima, removendo `e` de `b.entry_set()` é equivalente a fazer `e.blog = None`, e por isso o `ForeignKey` `blog` não tem `null=True`, isto é inválido.

`QuerySet.clear()`

Remove todos os objetos de um conjunto de objetos relacionados:

```
>>> b = Blog.objects.get(id=1)
>>> b.entry_set.clear()
```

Note que isto não deleta os objetos relacionados – somente os desassocia.

Assim como `remove()`, `clear()` é somente disponível em `ForeignKeys` onde `null=True`.

Meta opções do model

Este documento explica todas as *opções de metadados* <meta-options> possíveis que você pode dar a seu model em sua class `Meta`.

Opções disponíveis do Meta

abstract

`Options.abstract`

Se `True`, este model será um *uma classe abstrata básica*.

db_table

`Options.db_table`

O nome da tabela que será usado pelo model no banco de dados:

```
db_table = 'music_album'
```

Nomes de tabelas

Para poupar o seu tempo, o Django automaticamente deriva o nome das tabelas de banco de dados usando os nomes das classes models e da aplicação que o contém. O nome de uma tabela de um model é construído juntando o “nome da app” do model – o nome usado em `manage.py startapp` – ao nome da classe do model, com um underscore entre eles.

Por exemplo, se você tem uma app `bookstore` (criada pelo `manage.py startapp bookstore`), um modelo definido como `class Book` terá uma tabela com o nome `bookstore_book`.

Para sobrescrever o nome da tabela, use o parametro `db_table` na class `Meta`.

Se o nome da sua tabela é uma palavra reservada do SQL, ou contém caracteres que não são permitidos em nomes de variáveis do Python – notadamente, o hífen – está OK. O Django colocará crases os nomes das colunas e tabelas sem você saber.

`db_tablespace`

`Options.db_tablespace`

Please, see the release notes O nome do tablespace do banco dados utilizado para o model. Se o backend não suporta tablespaces, esta opção é ignorada.

`get_latest_by`

`Options.get_latest_by`

O nome de um `DateTimeField` ou `DateTimeField` no model. Isso especifica o campo padrão do seu model que será utilizado pelo método `latest` do `Manager`.

Exemplo:

```
get_latest_by = "order_date"
```

Veja a documentação para `latest()` para mais.

`order_with_respect_to`

`Options.order_with_respect_to`

Marca este objeto como “ordenável” em relação ao campo dado. Isso é quase sempre utilizado com objetos relacionados para permiti-los ser ordenados a respeito de um objeto pai. Por exemplo, se um objeto `Answer` está relacionado a um objeto `Question`, e uma questão tem mais de uma resposta, e a ordem das respostas importa, você poderia fazer isso:

```
class Answer(models.Model):
    question = models.ForeignKey(Question)
    # ...

    class Meta:
        order_with_respect_to = 'question'
```

`ordering`

`Options.ordering`

O ordenamento padrão de um objeto, para usar quando obtem-se listas de objetos:

```
ordering = ['-order_date']
```

Isso é uma tupla ou lista de strings. Cada string é o nome de um campo com um prefixo opcional “-”, que indica ordem descendente. Campos sem um “-” na frente serão ordenados ascendentemente. Use a string “?” para ordenar aleatoriamente.

Note: Indiferentemente de quantos sejam os campos em `ordering`, o admin usará somente o primeiro campo.

Por exemplo, para ordenar por um campo `pub_date`, use isto:

```
ordering = ['pub_date']
```

Para ordenar por `pub_date` decendentemente, use isto:

```
ordering = ['-pub_date']
```

Para ordenar por `pub_date` decendentemente, e por `author` ascendentemente, use isto:

```
ordering = ['-pub_date', 'author']
```

permissions

`Options.permissions`

Para adicionar permissões extras na tabela de permissões quando criar este objeto. Permissões para adicionar, deletar e editar são automaticamente criadas para cada objeto que use o admin. Este exemplo especifica uma permissão extra `can_deliver_pizzas`:

```
permissions = (("can_deliver_pizzas", "Can deliver pizzas"),)
```

Isso é uma lista ou tupla de tuplas duplas no formato (código_da_permissão, nome_da_permissão_legível_por_humanos).

unique_together

`Options.unique_together`

Seta os nomes de campos, que juntos, devem ser únicos:

```
unique_together = (("driver", "restaurant"),)
```

Isso é uma lista de listas de campos que devem ser únicos quando estão juntos. ela é usada no Django admin e é executada a nível de banco de dados (i.e., a regra `UNIQUE` apropriada é incluída na consulta `CREATE TABLE`). *Please, see the release notes* Por conveniência, `unique_together` pode ser uma lista singular quando lida com campos sozinhos:

```
unique_together = ("driver", "restaurant")
```

verbose_name

`Options.verbose_name`

Um nome legível-por-humanos para o objeto, no singular:

```
verbose_name = "pizza"
```

Se este não for fornecido, o Django usará uma versão do nome da classe: `CamelCase` se torna `camel case`.

`verbose_name_plural`

`Options.verbose_name_plural`

O nome no plural para o objeto:

```
verbose_name_plural = "stories"
```

Se este não for fornecido, o Django usará `verbose_name` + "s".

Referência das instâncias de Model

Esse documento descreve os detalhes da API `Model`. Isso foi feito baseado no material apresentado nos guias *model* e *consultas ao banco de dados*, então provavelmente você vai querer ler e entender esses documentos antes de ler este aqui.

Durante essa referência, usaremos os *models de exemplo do weblog* apresentado no *guia de consultas ao banco de dados*.

Criando objetos

Para criar uma nova instância de um model, somente instancie-o assim como qualquer outra classe Python:

```
class Model (**kwargs)
```

Os argumentos são simplesmente os nomes dos campos que você definiu no seu model. Perceba que instanciando um model você não alterará o banco de dados; para isso, você precisa do `save()`.

Salvando objetos

Para salvar um objeto no banco de dados, chame `save()`:

```
Model.save([force_insert=False, force_update=False])
```

É claro, há algumas legendas; veja as seções abaixo. *Please, see the release notes* A assinatura do método `save()` tem mudado nas versões mais novas (`force_insert` e `force_update` foram adicionados). Se você está sobrescrevendo estes métodos, esteja certo do uso correto da assinatura.

Auto-incrementando chaves primárias

Se um model possui um `AutoField` – uma chave primária que auto-incrementa – então o valor auto-increment será calculado e salvo como um atributo no seu objeto, na primeira vez que você chamar o método `save()`:

```
>>> b2 = Blog(name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b2.id      # Retorna None, porque b nao tem um ID ainda.
>>> b2.save()
>>> b2.id      # Retorna o ID de seu novo objeto.
```

Não há meios de dizer qual será o valor do ID antes de você chamar o método `save()`, porque este valor é calculado pelo banco de dados, não pelo Django.

(Por conveniencia, cada modelo possui um `AutoField` chamado `id`, por padrão, a menos que você explicitamente especifique `primary_key=True` em um campo. Veja a documentação para `AutoField` para saber mais detalhes.

A propriedade pk

Please, see the release notes

Model .pk

Independentemente de saber se você vai definir uma chave primária, ou deixar o Django fazer isso por você, cada modelo irá ter uma propriedade chamada `pk`. Ela se comporta como um atributo normal dentro do modelo, mas na verdade é um alias para qualquer atributo que seja a chave primária de seu modelo. Você pode acessar ou setar esse valor, como em qualquer outro atributo, e ele irá atualizar o campo correto do modelo.

Especificando explicitamente valores para auto-primary-key

Se um modelo tem um `AutoField` mas você quer definir um novo ID de objeto explicitamente quando salva, então defina-o explicitamente antes de salvar, em vez de confiar na auto-atribuição do ID:

```
>>> b3 = Blog(id=3, name='Cheddar Talk', tagline='Thoughts on cheese.')
>>> b3.id      # Retorna 3.
>>> b3.save()
>>> b3.id      # Retorna 3.
```

Se você atribui manualmente, valores para auto-primary-key, esteja certo de não usar um valor de primary-key já existente! Se você criar um novo objeto com um valor de chave primária explícito já existente no banco de dados, o Django irá assumir que você está alternado um dado existente ao invés de criar um novo.

Como mostrado no exemplo acima, blog 'Cheddar Talk', neste exemplo o dado será atualizado no banco de dados:

```
b4 = Blog(id=3, name='Not Cheddar', tagline='Anything but cheese.')
b4.save()  # Sobrescreve o blog já existente com ID=3!
```

Veja *Como o Django sabe quando fazer UPDATE vs. INSERT*, abaixo, para entender por que isso acontece.

Especificar explicitamente os valores de auto-primary-key é mais usual para salvar objetos em massa, quando você está certo de que não há colisões de chaves primárias.

O que acontece quando você salva?

Quando você salva um objeto, o Django executa os seguintes passos:

1. **Emita um sinal pre-save.** O `signal django.db.models.signals.pre_save` é enviado, permitindo qualquer função ouvir o sinal para executar alguma ação personalizada.
2. **Pre-processa os dados.** Cada campo no objeto é consultado para realização de qualquer modificação dos dados que o campo possa precisar.

A maioria dos campos não fazem pre-processamento – o dado do campo é mantido como está. O pre-processamento é utilizado somente em campos que possuem um comportamento especial. Por exemplo, se seu modelo tem uma `DateTimeField` com um `auto_now=True`, o pre-save irá alterar o dado deste campo, assegurando que o campo contém a data corrente. (Nossa documentação ainda não inclui a lista de todos os campos com este “comportamento especial.”)

3. **Prepara os dados para o banco de dados.** Cada campo é consultado para prover seu valor corrente em um formato que pode ser escrito no banco de dados. Most fields require *no* data preparation. Simple data types, such as integers and strings, are ‘ready to write’ as a Python object. However, more complex data types often require some modification.

Por exemplo, `DateTimeFields` usam um objeto `datetime` do Python para armazenar o dado. O banco de dados não sabe armazenar objetos `datetime`, então o valor do campo deve ser convertido em uma string compilada no formato de data ISO, para inserção no banco de dados.

4. **Insere o dado no banco de dados.** O dado pre-processado, preparado, é então composto em um chamada SQL para inserção no banco de dados.
5. **Emite um sinal post-save.** O sinal `django.db.models.post_save` é enviado, permitindo qualquer função ouvir o sinal para executar alguma ação personalizada.

Como o Django sabe quando fazer UPDATE vs. INSERT

Você pode ter notado que os objetos de banco de dados do Django, utilizam o mesmo método `save()` para criar e atualizar objetos. O Django abstrai a necessidade de uso do `INSERT` ou `UPDATE`. Especificamente, quando você executa `save()`, o Django segue este algoritmo:

- Se o atributo chave primária do objeto está setado de forma a ser considerado *verdadeiro* (ex. um valor que não seja `None` ou uma string vazia), o Django executa um `SELECT` pra averiguar se existe algum dado com esta chave primária.
- Se o dado com a chave primária não existe, o Django já existe, o Django executa uma consulta `UPDATE`.
- Se o atributo chave primária do objeto não está setado, ou se está, mas ela não existe no banco de dados, então o Django executa uma consulta `INSERT`.

O ponto aqui é que você deve ter cuidado para não especificar um valor de chave primária explicitamente ao salvar novos objetos, se você não tem como garantir que o valor não foi utilizado. Para mais sobre esta nuancia, veja *Especificando explicitamente valores para auto-primary-key* acima e *Forçando um INSERT ou UPDATE* abaixo.

Forçando um INSERT ou UPDATE

Please, see the release notes Em algumas raras circunstancias, é necessário forçar o método `save()` a executar um SQL `INSERT` e não retroceder para fazer um `UPDATE`. Ou vice-versa: atualizar, se possível, mas não inserir uma nova linha. Nestes casos você pode passar os parametros `force_insert=True` ou `force_update=True` para o método `save()`. Utilizar ambos os parametros é um erro, tendo em vista que você não irá utilizar ambos, `INSERT` e `UPDATE` ao mesmo tempo.

Deve ser muito raro você precisar usar estes parâmetros. O Django irá quase sempre fazer a coisa certa e tentar contornar isso irá conduzi-los a erros difíceis de rastrear. Esta funcionalidade é somente para usuários avança-dos.

Deletando objetos

`Model.delete()`

Emite um SQL `DELETE` para um objeto. Este somente deleta o objeto no banco de dados; a instância do Python persistirá, e conterà dados em seus campos.

Outros métodos da instância de um model

Alguns métodos de objetos têm propostas especiais.

`__str__`

`Model.__str__()`

`__str__()` é um “método mágico” do Python que define o que deve ser retornado se você chamar `str()` sobre o objeto. O Django usa `str(obj)` (ou a função relacionada, `unicode(obj)` – veja abaixo) em vários lugares, mais notavelmente como um valor mostrado para renderizar um objeto no site de administração do Django e como valores inseridos em um template quando é mostrado um objeto. Por isso, que você deve sempre retornar uma string, legível para humanos, dos métodos `__str__`. Embora ele não seja obrigatório, é fortemente encorajado (veja a descrição de `__unicode__`, abaixo, antes de por métodos `__str__` pra todo lado).

Por exemplo:

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __str__(self):
        # Note o uso do django.utils.encoding.smart_str() aqui, por que
        # first_name e last_name serão strings unicode.
        return smart_str('%s %s' % (self.first_name, self.last_name))
```

__unicode__

Model.__unicode__()

O método `__unicode__()` é chamado quando você executa `unicode()` sobre um objeto. Visto que os back-ends de banco de dados do Django retornarão strings Unicode para os atributos de seu model, você naturalmente irá querer escrever um método `__unicode__()` para seu model. O exemplo da seção anterior poderia ser escrito de maneira mais simples, como:

```
class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)

    def __unicode__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

Se você define um método `__unicode__()` em seu modelo e não um método `__str__()`, o Django irá automaticamente prover um método `__str__()` que chama o método `__unicode__()` e então converte o resultado corretamente para strings codificadas em UTF-8. Isto é uma prática de desenvolvimento recomendada: definir somente `__unicode__()` e deixar o Django se preocupar com a conversão das strings quando necessário.

get_absolute_url

Model.get_absolute_url()

Definir um método `get_absolute_url()` pra dizer ao Django como calcular a URL para um objeto. Por exemplo:

```
def get_absolute_url(self):
    return "/people/%i/" % self.id
```

O Django usa isto em sua interface de administração. Se um objeto define `get_absolute_url()`, a página de edição do objeto terá um link “Veja no site” que aponta diretamente para a visão pública do objeto, de acordo com o `get_absolute_url()`.

Também, em outros lugares a mais, como no *framework de feeds*, o uso do `get_absolute_url()` como uma conveniência para recompensar quem tenha definido o método.

É uma boa prática usar o `get_absolute_url()` em templates, ao invés de escrever a mão as URLs de seus objetos. Por exemplo, este código de template não é legal:

```
<a href="/people/{{ object.id }}/">{{ object.name }}</a>
```

Mas esse código é bem melhor:

```
<a href="{{ object.get_absolute_url }}">{{ object.name }}</a>
```

Note: A string que você retorna do `get_absolute_url()` deve conter somente caracteres ASCII (exigidos pela especificação URI, RFC 2396) que tenham sido codificados para URL, se necessário. Um template escrito usando `get_absolute_url()` deve ser capaz de utilizar o resultado diretamente sem a necessidade de qualquer outro tratamento. Você pode desejar usar a função `django.utils.encoding.iri_to_uri()` para ajudá-lo, se você estiver usando muitas strings unicode.

O decorador `permalink`

O problema com a solução que descrevemos acima, utilizando o `get_absolute_url()`, é que ela viola ligeiramente os princípios DRY: a URL para este objeto é definida em ambos arquivo `URLconf` e no modelo.

Você ainda pode dissociar seus modelos do `URLconf` utilizando o decorador `permalink`:

`permalink()`

Este decorador é passado para a função `view`, uma lista de parâmetros posicionais e (opcionalmente) um dicionário de parâmetros nomeados. O Django então constroi o caminho completo da URL, usando o `URLconf`, substituindo os parâmetros que você forneceu dentro da URL. Por exemplo, se seu `URLconf` contém uma linha como esta:

```
(r'^people/(\d+)/$', 'people.views.details'),
```

...seu model pode ter um método `get_absolute_url` que parece com isto:

```
from django.db import models

@models.permalink
def get_absolute_url(self):
    return ('people.views.details', [str(self.id)])
```

Similarmente, se você tem uma entrada no `URLconf` que pareça isto:

```
(r'/archive/(?P<year>\d{4})/(?P<month>\d{1,2})/(?P<day>\d{1,2})/$', archive_view)
```

...você poderia referenciá-la usando o `permalink()` como a seguir:

```
@models.permalink
def get_absolute_url(self):
    return ('archive_view', (), {
        'year': self.created.year,
        'month': self.created.month,
        'day': self.created.day})
```

Observe que nós especificamos uma sequência vazia para o segundo parâmetro neste caso, porque nós somente queremos passar parâmetros nomeados, ao invés de posicionais.

Desta forma, você está amarrando a URL absoluta do model com a `view` que é usada para mostrá-lo, sem repetir as informações de URL. Você ainda pode usar o método `get_absolute_url` nos templates, como antes.

Em alguns casos, como no caso de uso de generic views, ou no re-uso de views customizadas para múltiplos modelos, especificar a função de `view` pode confundir o reverse URL matcher (pois múltiplos padrões podem apontar para uma mesma `view`.)

Para este problema, o Django utiliza **padrões de URL nomeados**. Usando padrões de URL nomeados, é possível ter um nome para um padrão, e então referenciar este nome ao invés da função `view`. Um padrão de URL nomeado é definida através da substituição da tupla padrão por uma chamada da função `url`:

```
from django.conf.urls.defaults import *

url(r'^people/(\d+)/$',
    'django.views.generic.list_detail.object_detail',
    name='people_view'),
```

...e então usando este nome para fazer a reversão URL ao invés do nome da view:

```
from django.db.models import permalink

def get_absolute_url(self):
    return ('people_view', [str(self.id)])
get_absolute_url = permalink(get_absolute_url)
```

Mais detalhes sobre padrões de URL nomeadas estão em *documentação do URL dispatch*.

Métodos extra de instancia

Além de `save()`, `delete()`, um objeto de modelo pode ter qualquer, ou todos os seguintes métodos:

`Model.get_FOO_display()`

Para todo campo que possui `choices`, o objeto terá um método `get_FOO_display()`, onde `FOO` é o nome do campo. Este método retorna o valor “legível para humanos” do campo. Por exemplo, no seguinte model:

```
GENDER_CHOICES = (
    ('M', 'Male'),
    ('F', 'Female'),
)

class Person(models.Model):
    name = models.CharField(max_length=20)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES)
```

...cada instancia de `Person` terá um método `get_gender_display()`. Exemplo:

```
>>> p = Person(name='John', gender='M')
>>> p.save()
>>> p.gender
'M'
>>> p.get_gender_display()
'Male'
```

`Model.get_next_by_FOO(**kwargs)`

`Model.get_previous_by_FOO(**kwargs)`

Para todo `DateField` e `DateTimeField` que não possui `null=True`, o objeto terá os métodos `get_next_by_FOO()` e `get_previous_by_FOO()` para o mesmo campo. Este retorna o próximo objeto ou o anterior correspondente ao campo de data, lançando a exceção `DoesNotExist` se for o caso.

Ambos os métodos aceitam argumentos nomeados opcionalmente, que deve ser no formato descrito em *Field lookups*.

Note que no caso de valores de datas idênticos, estes métodos irão usar o ID como solução de checagem. Isto garante que nenhum dado será pulado ou duplicado.

Referência da API QuerySet

Este documento descreve os detalhes da API do `QuerySet`. Ela se baseia no material apresentado nos guias *model* e *query de banco de dados*, então você provavelmente vai querer ler e entender esses documentos antes de ler este aqui.

Por toda essa referência nós iremos usar os *models do exemplo weblog* apresentado em *guia de consulta de banco de dados*.

Quando QuerySets são avaliados

Internamente, um `QuerySet` pode ser construído, filtrado, dividido, e geralmente passado adiante sem, bem na verdade, atingir o banco de dados. Nenhuma atividade de banco de dados realmente ocorre, até que você faça algo que avalie o queryset.

Você pode avaliar um `QuerySet` das seguintes formas:

- **Iteração.** Um `QuerySet` é iterável, e ele executa suas consultas de banco de dados na primeira vez que você itera sobre ele. Por exemplo, isso irá imprimir o título de todas as entradas do banco de dados:

```
for e in Entry.objects.all():
    print e.headline
```

- **Fatiando/Dividindo.** Como explicado em [Limitando QuerySets](#), um `QuerySet` pode ser dividido, usando a sintaxe de fatiamento do Python. Geralmente fatiar um `QuerySet` retorna um outro (não avaliado) `QuerySet`, mas o Django executará a consulta do banco de dados se você usar o parâmetro “step” da sintaxe de fatiamento.
- **Preservamento/Cacheamento.** Veja a seguinte seção para detalhes que é envolvida quando se [preservando QuerySets](#). A coisa importante para proposta desta seção é que os resultados são lidos do banco de dados.
- **repr().** Um `QuerySet` é avaliado quando você chama `repr()` sobre ele. Este é por conveniência um interpretador Python interativo, então você pode imediatamente ver seus resultados quando estiver usando a API interativamente.
- **len().** Um `QuerySet` é avaliado quando você chama `len()` sobre ele. Isso, como você pode esperar, retorna o comprimento da lista de resultados.

Nota: Não use `len()` sobre `QuerySets` se tudo que você quer fazer é determinar o número de dados de um conjunto. É muito mais eficiente manipular um contador a nível de banco de dados, usando um SQL `SELECT COUNT(*)`, e o Django provê um método `count()` justamente por esta razão. Veja `count()` abaixo.

- **list().** Força a avaliação de um `QuerySet` chamando `list()` sobre ele. Por exemplo:

```
entry_list = list(Entry.objects.all())
```

Esteja avisado, no entanto, que isso poderia resultar num grande consumo de memória, porque o Django carregará cada elemento da lista na sua memória. Em contrast, iterando sobre um `QuerySet` você tirará vantagem de seu banco de dados para carregar os dados e instanciar os objetos sobre quando precisar deles.

Preservando QuerySets

Se você [preserva](#) um `QuerySet`, este forçará todos os resultados a serem carregados em memória para conservação. O preservamento é geralmente usado como um precursor para o cache, quando o cache do queryset é recarregado, você terá os resultados já presentes e prontos para o uso (lendo do banco de dados pode tomar algum tempo, anulando a proposta do cache). Isto significa que quando você desconserva um `QuerySet`, ele contém os resultados do momento em que foi preservado, ao invés dos resultados que estão atualmente no banco de dados.

Se você somente quer preservar as informações necessárias para recriar o `QuerySet` do banco de dados desde a última vez, conserve o atributo `query` do `QuerySet`. Você pode então recriar o `QuerySet` original (sem qualquer resultado carregado) usando algum código como este:

```
>>> import pickle
>>> query = pickle.loads(s)          # Assumindo 's' como uma string conservada.
>>> qs = MyModel.objects.all()
>>> qs.query = query                 # Restaura o 'query' original.
```

O atributo `query` é um objeto opaco. Ele representa a construção interna da query e não parte da API pública. Entretanto, ele é seguro (e completamente suportado) para preservar e reconstruir os conteúdos dos atributos como descrito aqui.

API QuerySet

Embora você geralmente não criará uma manualmente – você passará um *Manager* – aqui tem a declaração formal de um `QuerySet`:

```
class QuerySet ([model=None ])
```

Normalmente quando você for interagir com um `QuerySet` você o usará com *filtros encadeados*. Para fazer isto funcionar, a maioria dos métodos do `QuerySet` retorna novos `querysets`.

Métodos de QuerySet que retornam novos QuerySets

O Django fornece uma gama de métodos de refinamento do `QuerySet` que modifica cada tipo de resultados retornados pelo `QuerySet` ou a forma como sua consulta SQL é executada.

`filter(**kwargs)`

Retorna um novo `QuerySet` contendo objetos que combinam com os parâmetros dados.

Os parâmetros (`**kwargs`) devem ser no formato descrito em *Campos de Pesquisa* abaixo. Vários parâmetros são mesclados via AND e regras SQL subjacentes.

`exclude(**kwargs)`

Retorna um novo `QuerySet` contendo objetos que *não* combinam com os parâmetros dados.

Os parâmetros (`**kwargs`) devem ser no formato descrito em *Campos de Pesquisa* abaixo. Vários parâmetros são mesclados via AND e regras SQL subjacentes, e toda coisa é envolvida por um `NOT()`.

Este exemplo exclui todas as entradas cujo `pub_date` é maior que 2005-1-3 e (AND) cujo `headline` é igual “Hello”:

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3), headline='Hello')
```

Em termos SQL, isso seria:

```
SELECT ...
WHERE NOT (pub_date > '2005-1-3' AND headline = 'Hello')
```

Este exemplo exclui entradas cujo `pub_date` é maior que 2005-1-3 ou (OR) cujo `headline` é “Hello”:

```
Entry.objects.exclude(pub_date__gt=datetime.date(2005, 1, 3)).exclude(headline=
↪ 'Hello')
```

Em termos SQL, isso seria:

```
SELECT ...
WHERE NOT pub_date > '2005-1-3'
OR NOT headline = 'Hello'
```

Note que o segundo exemplo é mais restritivo.

`order_by(*fields)`

Por padrão, os resultados retornados por um `QuerySet` são ordenado por uma tupla dada pela opção `ordering` no `Meta` do `model`. Você pode sobrescrever isso para uma `QuerySet` basicamente usando o método `order_by`. Exemplo:

```
Entry.objects.filter(pub_date__year=2005).order_by('-pub_date', 'headline')
```

O resultado acima será ordenado por `pub_date` decrescendo, e então por `headline` ascendendo. O sinal negativo na frente do `-pub_date` indica o ordenamento *descendente*. O ordenamento ascendente é implícito. Para ordenar randômicamente, use `"?"`, desta forma:

```
Entry.objects.order_by('?')
```

Nota: a consulta `order_by('?')` pode ser custosa e lenta, dependendo do backend e banco de dados que estiver usando.

Para ordenar por um campo de um model diferente, use a mesma sintaxe como quando você consulta através de relações de models. Isto é, o nome do campo, seguido por um underscore duplo (`__`), seguido pelo nome do campo no novo model, faça assim para todos os models que você deseja juntar na consulta. Por exemplo:

```
Entry.objects.order_by('blog__name', 'headline')
```

Se você tentar ordenar por um campo que é relacionado a outro model, o Django usará o ordenamento padrão do model relacionado (ou ordenará pela chave primária do model relacionado se ele não tiver um `Meta.ordering` especificado. Por exemplo:

```
Entry.objects.order_by('blog')
```

...é idêntico a:

```
Entry.objects.order_by('blog__id')
```

...desde que o model `Blog` não tenha o ordenamento especificado.

Seja cauteloso quando ordenar por campos de models relacionados se você também estiver usando `distinct()`. Veja a nota na seção [distinct\(\)](#) para uma explicação e como um model relacionado ordenado pode mudar os resultados esperados.

É permissível especificar um campo com valores múltiplos para ordenar os resultados (por exemplo, um campo `ManyToMany`). Normalmente isso não será uma coisa sensível a se fazer e realmente não é uma funcionalidade avançada. Entretanto, se você sabe que a filtragem de seu queryset ou dados absolutos implica que haverá somente um ordenamento de parte dos dados para cada um dos itens principais que você estiver selecionando, o ordenamento pode bem ser exatamente o que você precisa fazer. use o ordenamento em campos com multi-valores com cuidado e assegure-se de que os resultados sejam os que você espera.

Admite-se especificar um campo com multi-valor para ordenar resultados (por exemplo, um campo `ManyToMany`). Normalmente isso não será uma coisa sensível a se fazer e ele realmente é uma funcionalidade avançada de uso. *Please, see the release notes* Se você não quer que qualquer ordenamento seja aplicado a consulta, nem mesmo o ordenamento padrão, chame `order_by()` sem parametros. .. versionadded:: 1.0

A syntax de para ordenamento através de models relacionados mudou. Veja a [documentação do Django 0.96](#) para os comportamentos antigos.

Não há formas de especificar se o ordenamento deve ser sensível a maiúsculas e minúsculas. Com respeito a case-sensitivity, o Django não ordena resultados por mais que seu backend de banco de dados normalmente o faça.

`reverse()`

Please, see the release notes Use o método `reverse()` para reverter a ordem em que os elementos de uma queryset são retornados. Chamando `reverse()` uma segunda vez restaura a ordem de volta a direção normal.

Para receber os “últimos” cinco elementos de um queryset, você pode fazer isso:

```
my_queryset.reverse()[ :5]
```

Perceba que isto não é a mesma coisa que tirar um pedaço ao final de uma sequência no Python. O exemplo acima retornará o último item primeiro, e então o penúltimo item e assim por diante. Se nós temos uma sequência do Python e olhamos em `seq[-5:]`, nós veríamos o quinto-último elemento primeiro. O Django não suporta este modo de acesso (fatiamento ao final), porque ele não é possível de ser feito eficientemente no SQL.

Também, note que o `reverse()` geralmente só deve ser chamado sobre um `QuerySet` que tem um ordenamento definido (e.g. quando a consulta é feita contra um model que tem o ordenamento definido, ou quando usa-se o `order_by()`). Se nenhum ordenamento foi definido para o `QuerySet`, chamar o `reverse()` sobre ele não terá o efeito real (o ordenamento estava indefinido antes da chamada do `reverse()`, e permanecerá indefinido depois).

`distinct()`

Retorna um novo `QuerySet` que usa `SELECT DISTINCT` na sua consulta SQL. Isso elimina duplicatas nos resultados da consulta.

Por padrão, um `QuerySet` não eliminará linhas duplicadas. Na prática, isto raramente é um problema, porque consultas simples como `Blog.objects.all()` não introduzem a possibilidade de duplicatas no resultado. No entanto, se sua consulta atinge várias tabelas, é possível receber resultados duplicados quando um `QuerySet` for avaliado. Nessas horas você deve usar `distinct()`.

Note: Quaisquer campos usados numa chamada `order_by(*fields)` são incluídos nas colunas do SQL `SELECT`. Isso pode conduzir, as vezes, a resultados inesperados quando usa em conjunto com `distinct()`. Se você ordena por campos de um model relacionado, estes campos podem ser adicionados as colunas selecionadas e eles podem fazer, de outra forma, as linhas duplicadas parecerem ser distintas. Desde que as colunas extras não aparecem nos resultados retornados (eles somente estão lá para apoiar o ordenamento), algumas vezes parecem como se resultados não distintos estão sendo devolvidos.

Similarmente, se você usa uma consulta `values()` para as colunas restritas selecionadas, as colunas usadas em qualquer `order_by()` (ou ordenamento padrão do model) ainda estarão envolvidos e pode afetar a unicidade dos resultados.

A moral aqui é que se você estiver usando `distinct()` deve ser cuidadoso com o ordenamento por models relacionados. Identicamente, quando usamos `distinct()` e `values()` juntos, deve-se ter cuidado ao ordenar por campos que não estão na chamada `values()`.

`values(*fields)`

Retorna um `ValuesQuerySet()` – um `QuerySet` que avalia para uma lista de dicionários ao invés de objetos de instância de models.

Cada um destes dicionários representa um objeto, com as chaves correspondendo aos nomes dos atributos dos objetos de model.

Este exemplo compara os dicionários de `values()` com os objetos normais de model:

```
# Esta lista contém um objeto Blog.
>>> Blog.objects.filter(name__startswith='Beatles')
[<Blog: Beatles Blog>]

# Esta lista contém um dicionário.
>>> Blog.objects.filter(name__startswith='Beatles').values()
[{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]
```

`values()` recebe argumentos posicionais opcionais, `*fields`, que especificam nomes de campos aos quais o `SELECT` deve ser limitado. Se você especificar os `fields`, cada dicionário conterá somente o campo chave/valor para os campos que você determinou. Se você não especificar os campos, cada dicionário terá uma chave e um valor para todos os campos da tabela do banco de dados.

Exemplo:

```
>>> Blog.objects.values()
[{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}],
>>> Blog.objects.values('id', 'name')
[{'id': 1, 'name': 'Beatles Blog'}]
```

Algumas sutilezas que valem a pena mencionar:

- O método `values()` não retorna nada para atributos *ManyToManyField* e gerará um erro se você tentar passar este tipo de campo para ele.
- Se você tem um campo chamado `foo` que é um *ForeignKey*, a chamada padrão de `values()` retornará uma chave de dicionário chamada `foo_id`, uma vez que este é o nome do atributo oculto do model que armazena o valor real (o atributo `foo` refere-se ao model relacionado). Quando você estiver chamando `values()` e passando em nomes de campos, você pode passar ambos `foo` ou `foo_id` e você terá de volta a mesma coisa (a chave do dicionário combinará o nome do campo que você passou).

Por Exemplo:

```
>>> Entry.objects.values()
[{'blog_id': 1, 'headline': u'First Entry', ...}, ...]

>>> Entry.objects.values('blog')
[{'blog': 1}, ...]

>>> Entry.objects.values('blog_id')
[{'blog_id': 1}, ...]
```

- Quando estiver usando `values()` junto com `distinct()`, esteja alerta pois o ordenamento pode afetar os resultados. Veja a nota na seção *distinct()*, acima, para detalhes.

Please, see the release notes Anteriormente, não seria possível passar `blog_id` ao `values()` no exemplo acima, somente `blog`.

Um `ValuesQuerySet` é útil quando você sabe que vai precisar de valores de um pequeno número de campos disponíveis e você não precisa de funcionalidade de um objeto de instância de model. É mais eficiente selecionar somente os campos que você precisa usar.

Finalmente, perceba que um `ValuesQuerySet` é uma subclasse de `QuerySet`, então ele terá todos os métodos do `QuerySet`. Você pode chamar o `filter()` sobre ele, ou `order_by()`, ou ou que seja. Sim, isto significa que essas duas chamadas são idênticas:

```
Blog.objects.values().order_by('id')
Blog.objects.order_by('id').values()
```

As pessoas que fizeram o Django preferem colocar todos os métodos que afetam SQL primeiro, seguidos (opcionalmente) por quaisquer métodos que afetam a saída (como o `values()`), mas isso não importa realmente. Esta é a sua chance de ostentar seu individualismo.

`values_list(*fields)`

Please, see the release notes Este é similar ao `values()` exceto que ao invés de retornar uma lista de dicionários, ele retorna uma lista de tuplas. Cada tupla contém o valor do respectivo campo passado dentro da chamada `values_list()` – então o primeiro item é o primeiro campo, etc. Por exemplo:

```
>>> Entry.objects.values_list('id', 'headline')
[(1, u'First entry'), ...]
```

Se você somente passa um campo singular, você também pode passar num parâmetro `flat`. Se `True`, isto significará que os resultados retornados são valores únicos, ao invés de uma tupla. Um exemplo deve fazer a diferença:

```
>>> Entry.objects.values_list('id').order_by('id')
[(1,), (2,), (3,), ...]

>>> Entry.objects.values_list('id', flat=True).order_by('id')
[1, 2, 3, ...]
```

É um erro passar o `flat` quando há mais de um campo.

Se você não passar quaisquer valores ao `values_list()`, ele irá retornar todos os campos no model, na ordem em que foram declarados.

`dates(field, kind, order='ASC')`

Retorna um `DateQuerySet` – um `QuerySet` que avalia uma lista de objetos `datetime.datetime` representando todas as datas disponíveis de um tipo particular dentro dos conteúdos do `QuerySet`.

O `field` deve ser o nome de um `DateField` ou `DateTimeField` de seu model.

O `kind` deve ser "year", "month" ou "day". Cada objeto `datetime.datetime` na lista de resultados é “trucado” para o type dado.

- "year" retorna uma lista de todos os valores de anos distintos para o campo.
- "month" retorna uma lista de todos os valores anos/mês para o campo.
- "day" retorna uma lista de todos os valores ano/mês/dia para o campo.

O `order`, que por padrão é 'ASC', deve ser 'ASC' ou 'DESC'. Isto especifica como ordenar os resultados.

Exemplos:

```
>>> Entry.objects.dates('pub_date', 'year')
[datetime.datetime(2005, 1, 1)]
>>> Entry.objects.dates('pub_date', 'month')
[datetime.datetime(2005, 2, 1), datetime.datetime(2005, 3, 1)]
>>> Entry.objects.dates('pub_date', 'day')
[datetime.datetime(2005, 2, 20), datetime.datetime(2005, 3, 20)]
>>> Entry.objects.dates('pub_date', 'day', order='DESC')
[datetime.datetime(2005, 3, 20), datetime.datetime(2005, 2, 20)]
>>> Entry.objects.filter(headline__contains='Lennon').dates('pub_date', 'day')
[datetime.datetime(2005, 3, 20)]
```

`none()`

Please, see the release notes Retorna um `EmptyQuerySet` – um `QuerySet` que sempre avalia para uma lista vazia. Este pode ser usado nos casos onde você sabe que você deve retornar uma lista de resultados vazia e sua chamada está esperando um objeto `QuerySet` (ao invés de retornar uma lista vazia, por exemplo.)

Exemplos:

```
>>> Entry.objects.none()
[]
```

`all()`

Please, see the release notes Retorna uma “cópia” do `QuerySet` atual (ou estende o `QuerySet` que você passou). Isso pode ser útil em algumas situações onde você pode querer passar em cada model manager ou um `QuerySet` e fazer uma filtragem maior do resultado. Você pode seguramente chamar `all()` em cada objeto e então você definitivamente terá um `QuerySet` para trabalhar com.

`select_related()`

Retorna um `QuerySet` que automaticamente “seguirá” chaves estrangeiras de relacionamentos, selecionando os dados adicionais de objetos relacionados quando executar sua consulta. Isso é um aumentador de performance com resultados em (algumas vezes mais) consultas maiores, mas significa mais uso de relacionamentos de chave estrangeira que não vão exigir consultas de banco de dados.

O seguinte exemplo ilustra a diferença entre pesquisas planas e pesquisas `select_related()`. Aqui a pesquisa padrão:

```
# Acerta o banco de dados.
e = Entry.objects.get(id=5)

# Acerta o banco de dados novamente para pegar o objeto Blog relacionado.
b = e.blog
```

E aqui uma pesquisa `select_related`:

```
# Acerta o banco de dados.
e = Entry.objects.select_related().get(id=5)

# Não acerta o banco de dados, porque e.blog já foi populado numa consulta
# anterior.
b = e.blog
```

O `select_related()` segue chaves estrangeiras tão longe quanto possível. Se você tem os seguintes models:

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

class Person(models.Model):
    # ...
    hometown = models.ForeignKey(City)

class Book(models.Model):
    # ...
    author = models.ForeignKey(Person)
```

...então uma chamada para `Book.objects.select_related().get(id=4)` fará cache do objeto relacionado `Person` e do `City`:

```
b = Book.objects.select_related().get(id=4)
p = b.author          # Não acerta o banco de dados.
c = p.hometown        # Não acerta o banco de dados.

b = Book.objects.get(id=4) # Nenhum select_related() neste exemplo.
p = b.author          # Acerta o banco de dados.
c = p.hometown        # Acerta o banco de dados.
```

Note que, por padrão, `select_related()` não segue chaves estrangeiras que tenham `null=True`.

Normalmente, usar o `select_related()` pode melhorar muito a performance pois sua aplicação pode evitar muitas chamadas de banco de dados. Entretanto, em situações com conjuntos profundamente aninhados de relacionamentos, o `select_related` pode algumas vezes acabar seguindo “relacionamentos de mais”, e pode gerar consultas tão grandes que elas acabam sendo lentas.

Nestas situações, você pode usar o argumento `depth` para controlar quantos “níveis” de relações o `select_related()` realmente irá seguir:

```
b = Book.objects.select_related(depth=1).get(id=4)
p = b.author          # Não acerta o banco de dados.
c = p.hometown        # Requer uma chamada de banco de dados.
```

Algumas vezes você somente quer acessar um model específico que está relacionado ao seu model raiz, não todos os models relacionados. Nestes casos, você pode passar os nomes dos campos ao `select_related()` e ele somente seguirá estas relações. Você pode até fazer isso para os models que estão a mais de uma relação de distância, separando os nomes dos campos com underscores duplos, assim como nos filtros. Por exemplo, se você tem este model:

```
class Room(models.Model):
    # ...
    building = models.ForeignKey(...)

class Group(models.Model):
    # ...
    teacher = models.ForeignKey(...)
    room = models.ForeignKey(Room)
    subject = models.ForeignKey(...)
```

...e você somente precisa trabalhar com os atributos `room` e `subject`, você poderia escrever isso:

```
g = Group.objects.select_related('room', 'subject')
```

Isso também é válido:

```
g = Group.objects.select_related('room__building', 'subject')
```

...e também puxar a relação `building`.

Você somente pode referenciar à relações `ForeignKey` na lista de campos passada ao `select_related`. Você *pode* referenciar chaves estrangeiras que tenham `null=True` (indiferente da chamada padrão do `select_related()`). É um erro usar ambos, uma lista de campos e o parâmetro `depth`, na mesma chamada do `select_related()`, já que eles são opções conflitantes. *Please, see the release notes* Ambos, o argumento `depth` e a habilidade de especificar nomes de campos em chamadas do `select_related()`, são novidades do Django 1.0.

extra(select=None, where=None, params=None, tables=None, order_by=None, select_params=None)

Algumas vezes, a sintaxe de consulta do Django, por si só, não expressa uma cláusula complexa `WHERE`. Para esta orla de casos, o Django fornece o modificador `extra()` do `QuerySet` – um hook para injetar cláusulas específicas dentro do SQL gerado pelo `QuerySet`.

Por definição, estas pesquisas extra podem não ser portáteis para diferentes bancos de dados (pois você está escrevendo código SQL explicitamente) e violando o princípio DRY, então você deve evitá-los se possível.

Especificar um ou mais de `params`, `select`, `where` ou `tables`. Nenhum dos argumentos é obrigatório, mas você deve usar pelo menos um deles.

select O argumento `select` deixa você colocar campos extras na cláusula `SELECT`. Ele deve ser um dicionário que mapeia nomes de atributos para que a cláusula SQL calcule estes atributos.

Exemplo:

```
Entry.objects.extra(select={'is_recent': "pub_date > '2006-01-01'"})
```

Como um resultado, cada objeto `Entry` terá um atributo extra, `is_recent`, um booleano representando se o `pub_date` da entrada é maior que Jan. 1, 2006.

O Django insere o dado pedaço de SQL diretamente dentro da regra `SELECT`, então o SQL resultante do exemplo acima deveria ser:

```
SELECT blog_entry.*, (pub_date > '2006-01-01')
FROM blog_entry;
```

o próximo exemplo é mais avançado; ele faz uma sub-consulta para cada objeto `Blog` resultando num atributo `entry_count`, um contador inteiro de objetos `Entry` associados:

```
Blog.objects.extra(
    select={
        'entry_count': 'SELECT COUNT(*) FROM blog_entry WHERE blog_entry.blog_
        ↳id = blog_blog.id'
    },
)
```

(Neste caso particular, nós estamos explorando o fato de que a consulta já conterá a tabela `blog_blog` na cláusula `FROM`.)

O SQL resultante do exemplo acima deveria ser:

```
SELECT blog_blog.*, (SELECT COUNT(*) FROM blog_entry WHERE blog_entry.blog_id_
↳= blog_blog.id) AS entry_count
FROM blog_blog;
```

Note que os parênteses requeridos pela maioria dos bancos de dados a volta da sub-consulta não são obrigatório na cláusula `select` do Django. Também note que alguns backends de banco de dados, como algumas versões do MySQL, não suportam sub-consultas. *Please, see the release notes* Em alguns casos raros, você pode querer passar parâmetros para os fragmentos SQL no `extra(select=...)`. Para este propósito, use o parâmetro `select_params`. Já que `select_params` é uma sequência de atributos `select` como um dicionário, alguns cuidados são requeridos para que os parâmetros sejam combinados corretamente com as partes do `select extra`. Nesta situação, você deve usar um `django.utils.datastructures.SortedDict` para o valor `select`, não somente um dicionário normal do Python.

Isso funcionará, por exemplo:

```
Blog.objects.extra(
    select=SortedDict([('a', '%s'), ('b', '%s')]),
    select_params=('one', 'two'))
```

A única coisa que se deve ter cuidado quando usar parâmetros de `select` no `extra()` é para evitar usar um substring `"%s"` (que são dois caracteres por cento antes do `s`) nas strings `select`. O monitoramento de parâmetros do Django procura por `%s` e um caractere de escape `%` como este não detectado. Isso conduzirá a resultados incorretos.

where / tables Você pode definir explicitamente uma cláusula SQL `WHERE` – talvez para executar joins não explícitos – usando `where`. Você pode manual adicionar tabelas a cláusula SQL `FROM` usando `tables`.

`where` e `tables` ambos recebem uma lista de strings. Todos os parâmetros de `where` são separados por “AND” para qualquer critério de busca.

Exemplo:

```
Entry.objects.extra(where=['id IN (3, 4, 5, 20)'])
```

...é traduzido (aproximadamente) para o seguinte SQL:

```
SELECT * FROM blog_entry WHERE id IN (3, 4, 5, 20);
```

Seja cuidadoso quando estiver usando o parâmetro `tables` se você estiver especificando tabelas que já são usadas na consulta. Quando você adiciona tabelas extras via parâmetro `tables`, o Django assume que você quer que aquela tabela tenha uma inclusão extra, se ela já estiver incluída. Isto cria um problema, já que o nome da tabela terá de ser um alias. Se a tabela aparece várias vezes no SQL, a segunda e subsequentes ocorrências devem usar pseudônimos, desta forma o banco de dados pode distingui-las. Se você está referindo-se a tabelas extras que você adicionou no parâmetro `where` isto causará erros.

Normalmente você somente adicionará tabelas extra que já não estão aparecendo na consulta. Entretanto, se o caso mostrado acima ocorrer, há poucas soluções. Primeiro, veja se você pode obter resultado sem incluir a tabela extra que já esteja na consulta. Se isso não for possível, coloque sua chamada `extra()` na

frente da construção do queryset de modo que esse será o primeiro uso desta tabela. O alias será o mesmo cada vez que você construir o queryset da mesma forma, então você pode confiar que o nome do alias não mudará.

order_by Se você precisa ordenar o queryset resultante usando algum dos novos campos ou tabelas que você incluiu via `extra()` use o parâmetro `order_by()` e passe uma sequência de strings. Estas strings devem ser os campos dos models (como um método `order_by` normal de queryset), no formato `nome_da_tabela.nome_da_coluna` ou um alias para uma coluna que você especificou no parâmetro `select` do `extra()`.

Por exemplo:

```
q = Entry.objects.extra(select={'is_recent': "pub_date > '2006-01-01'"})
q = q.extra(order_by = ['-is_recent'])
```

Isso sortaria todos os itens para que `is_recent` seja verdadeiro e postos na frente do conjunto de resultados (`True` é ordenado na frente, e `False` no ordenamento descendente).

Isso mostra, a propósito, que você pode fazer várias chamadas para o `extra()` e ele se comportará como você espera (adicionando novos limitadores a cada vez).

params O parâmetro `where` descrito acima pode usar marcadores de string de banco de dados padrão do Python – `'%s'` para indicar parâmetros do banco de dados que devem ser citados automaticamente. O argumento `params` é uma lista de qualquer parâmetro `extra` a ser substituído.

Exemplo:

```
Entry.objects.extra(where=['headline=%s'], params=['Lennon'])
```

Sempre use `params` ao invés de embutir valores diretamente dentro do `where`, porque `params` irá assegurar que os valores são colocados entre aspas corretamente de acordo com o seu backend particular. (Por exemplo, aspas podem ser escapadas corretamente.)

Ruim:

```
Entry.objects.extra(where=["headline='Lennon'"])
```

Bom:

```
Entry.objects.extra(where=['headline=%s'], params=['Lennon'])
```

Métodos do QuerySet que não retornam QuerySets

Os seguintes métodos do `QuerySet` avaliam o `QuerySet` e retornam *algo* diferente de um `QuerySet`.

Estes métodos não usam um cache (veja [Cacheamento e QuerySets](#)). Ao invés, eles consultam o banco de dados toda vez que são chamados.

`get(**kwargs)`

Retorna o objeto combinado com o dado parâmetro, que deve estar no formato descrito em [Campos de pesquisa](#).

O `get()` lança um `MultipleObjectsReturned` se mais de um objeto for encontrado. A exceção `MultipleObjectsReturned` é um atributo da classe `model`.

O `get()` lança uma exceção `DoesNotExist` se um objeto não foi encontrado para os parâmetros dados. Esta exceção também é um atributo da classe `model`. Exemplo:

```
Entry.objects.get(id='foo') # raises Entry.DoesNotExist
```

A exceção `DoesNotExist` herda do `django.core.exceptions.ObjectDoesNotExist`, então você pode atingir várias exceções `DoesNotExist`. Exemplo:

```
from django.core.exceptions import ObjectDoesNotExist
try:
    e = Entry.objects.get(id=3)
    b = Blog.objects.get(id=1)
except ObjectDoesNotExist:
    print "Ou entry ou blog não existe."
```

`create(**kwargs)`

Um método conveniente para criar um objeto e salvá-lo em um passo. Deste modo:

```
p = Person.objects.create(first_name="Bruce", last_name="Springsteen")
```

e:

```
p = Person(first_name="Bruce", last_name="Springsteen")
p.save(force_insert=True)
```

são equivalentes.

O parâmetro *force_insert* é documentado em outro lugar, mas tudo isso significa que um novo objeto sempre será criado. Normalmente você não vai precisar se preocupar com isto. Entretanto, se seu model contém uma chave primária de valor manual, que você seta e se este valor já existe no banco de dados, uma chamada do `create()` falhará com um `IntegrityError` já que a chave primária deve ser única. Então lembre-se de estar preparado para tratar essa exceção se você estiver usando chaves primárias manuais.

`get_or_create(**kwargs)`

Um método conveniente para procurar um objeto com os argumentos, e criá-lo se necessário.

Retorna um tupla de (`object`, `created`), onde `object` é objeto recebido ou criado e `created` é um booleano especificando se um novo objeto foi criado.

Isto age como um atalho de código e é mais útil para scripts de importação de dados. Por exemplo:

```
try:
    obj = Person.objects.get(first_name='John', last_name='Lennon')
except Person.DoesNotExist:
    obj = Person(first_name='John', last_name='Lennon', birthday=date(1940, 10, 9))
    obj.save()
```

Este padrão fica muito pesado quando o número de campos de um model se eleva. O exemplo acima pode ser reescrito usando `get_or_create()` desta forma:

```
obj, created = Person.objects.get_or_create(first_name='John', last_name='Lennon',
                                           defaults={'birthday': date(1940, 10, 9)})
```

Qualquer argumento nomeado passado para o `get_or_create()` – *exceto* um opcional chamado `defaults` – serão usado numa chamada `get()`. Se um objeto é encontrado, `get_or_create()` retorna uma tupla deste objeto e `False`. Se um objeto *não* é encontrado, `get_or_create()` instanciará e salvará um novo objeto, retornando uma tupla do novo objeto e `True`. O novo objeto será criado aproximadamente de acordo com este algoritmo:

```
defaults = kwargs.pop('defaults', {})
params = dict([(k, v) for k, v in kwargs.items() if '__' not in k])
params.update(defaults)
obj = self.model(**params)
obj.save()
```

Em inglês, isso significa começar com um argumento nomeado não-`'defaults'` que não contém um underscore duplo (o que poderia indicar uma pesquisa inexata). Então adicione os conteúdos do `defaults`, sobrecrevendo qualquer chave se necessário, e use o resultado como um argumento nomeado para a classe `model`. Como insinuado acima, esta é uma simplificação do algoritmo que é usado, mas ele contém todos os detalhes pertinentes. A implementação interna tem mais checagens de erro que estas e manipula algumas condições extras afiadas; se você estiver interessado, leia o código.

Se você tem um campo chamado `defaults` e quer usá-lo numa pesquisa exata no `get_or_create()`, é só usar `'defaults__exact'`, desta forma:

```
Foo.objects.get_or_create(defaults__exact='bar', defaults={'defaults': 'baz'})
```

o método `get_or_create()` tem um comportamento de erro semelhante ao `create()` quando você está especificando chaves primárias manualmente. Se um objeto precisa ser criado e a chave já existe no banco de dados, um `IntegrityError` será lançado.

Finalmente, uma palavra em uso `get_or_create()` nos views do Django. Como mencionado antes, `get_or_create()` é útil principalmente em scripts que precisam parsear dados e criar novos registros se não existirem disponíveis. Porém, se você precisa usar `get_or_create()` num view, por favor, esteja certo de usá-lo em requisições POST exceto que você tenha uma boa razão para não fazê-lo. Requisições GET não devem ter dados efetivos; Use POST sempre numa requisição a uma página que tenha efeito unilateral nos seus dados. Para mais, veja [Métodos seguros](#) na especificação do HTTP.

count()

Retorna um inteiro representando o número de objetos no banco de dados combinando com o `QuerySet`. O `count()` nunca lança exceções.

Exemplo:

```
# Retorna o numero total de entradas no banco de dados.
Entry.objects.count()

# Retorna o número de entradas cujo headline contém 'Lennon'
Entry.objects.filter(headline__contains='Lennon').count()
```

O `count()` realiza um `SELECT COUNT(*)` por trás das cenas, então você deve sempre usar `count()` ao invés de carregar todos os objetos do banco e chamar `len()` sobre o resultado.

Dependendo de qual banco de dados você estiver usando (e.g. PostgreSQL vs. MySQL), `count()` pode retornar um inteiro long ao invés de um inteiro normal do Python. Esta é uma implementação evasiva subjacente que não deve causar quaisquer problemas no mundo real.

in_bulk(id_list)

Recebe uma lista de valores de chaves primárias e retorna um dicionário mapeando cada valor de chave primária para uma instância de objeto com o dado ID.

Exemplo:

```
>>> Blog.objects.in_bulk([1])
{1: <Blog: Beatles Blog>}
>>> Blog.objects.in_bulk([1, 2])
{1: <Blog: Beatles Blog>, 2: <Blog: Cheddar Talk>}
>>> Blog.objects.in_bulk([])
{}
```

Se você passar ao `in_bulk()` uma lista vazia, você terá um dicionário vazio.

`iterator()`

Avalia o `QuerySet` (para realizar uma consulta) e retorna um [iterador](#) sobre os resultados. Um `QuerySet` tipicamente lê todos os seus resultados e instancia todos os objetos correspondentes na primeira vez que você acessá-los; o `iterator()`, no entanto, lerá os resultados e instanciará objetos em pedaços discretos, fornecendo-os um por vez. Para um `QuerySet` que retorna um número grande de objetos, este frequentemente resulta em melhor performance e redução significativa no uso de memória.

Note que usando o `iterator()` sobre um `QuerySet` que já foi avaliado, irá forçá-lo a avaliá-lo novamente, repetindo a consulta.

`latest(field_name=None)`

Retorna o último objeto na tabela, por data, usando o `field_name` fornecido como o campo de data.

Este exemplo retorna a última `Entry` na tabela, de acordo com o campo `pub_date`:

```
Entry.objects.latest('pub_date')
```

Se o `Meta` do seu model especifica `get_latest_by`, você pode deixar vazio o argumento `field_name` do `latest()`. O Django usará o campo especificado em `get_latest_by` por padrão.

Assim como `get()`, `latest()` lançam `DoesNotExist` se um objeto não existe com os parâmetros fornecidos.

Note que `latest()` existe puramente por conveniência e legibilidade.

Campos de Pesquisa

Campos de pesquis são como você especifica o cerne de uma cláusula SQL `WHERE`. Eles são especificados como argumentos nomeados dos métodos do `QuerySet` `filter()`, `exclude()` e `get()`.

Para uma introdução, veja [Campos de pesquisa](#).

`exact`

Combinação exata. Se o valor fornecido para comparação for `None`, ele será interpretado como um SQL `NULL` (Veja [isnull](#) para mais detalhes).

Exemplos:

```
Entry.objects.get(id__exact=14)
Entry.objects.get(id__exact=None)
```

SQL equivalente:

```
SELECT ... WHERE id = 14;
SELECT ... WHERE id IS NULL;
```

A semântica de `id__exact=None` mudou no Django 1.0. Anteriormente, ela era (intencionalmente) convertido para `WHERE id = NULL` a nível de SQL, o que nunca combinaria com nada. Agora ela foi mudada para fazer o mesmo utilizando `id__isnull=True`.

MySQL comparisons

No MySQL, uma configuração de “collation” de tabela do banco de dados determina se comparações `exact` são sensíveis a maiúsculas. Essa é uma configuração de banco de dados, *não* uma configuração do Django. É

possível configurar suas tabelas MySQL para usar a comparação sensível a maiúsculas, mas algumas pecinhas estão envolvidas. Para mais informação sobre isto, veja a [seção *collation*](#) na documentação de [banco de dados](#).

exact

Combinação exata sensível a maiúsculas.

Exemplo:

```
Blog.objects.get(name__exact='beatles blog')
```

SQL equivalente:

```
SELECT ... WHERE name ILIKE 'beatles blog';
```

Note que isto combinará 'Beatles Blog', 'beatles blog', 'BeAtLes BLoG', etc.

SQLite users

Quando estiver usando o backend SQLite e strings Unicode (não ASCII), tenha em mente a [anotação de banco de dados](#) sobre comparação de strings. O SQLite não faz checagem não sensível a maiúsculas em strings Unicode.

contains

Teste de contimento sensível a maiúsculas.

Exemplo:

```
Entry.objects.get(headline__contains='Lennon')
```

SQL equivalente:

```
SELECT ... WHERE headline LIKE '%Lennon%';
```

Note que este combinará o headline 'Today Lennon honored' mas não 'today lennon honored'. O SQLite não suporta regras LIKE sensíveis a maiúsculas; `contains` age como `icontains` no SQLite.

icontains

Teste de contimento insensível a maiúsculas.

Exemplo:

```
Entry.objects.get(headline__icontains='Lennon')
```

SQL equivalente:

```
SELECT ... WHERE headline ILIKE '%Lennon%';
```

SQLite users

Quando estiver usando o backend SQLite e strings Unicode (não ASCII), tenha em mente a [anotação de banco de dados](#) sobre comparação de strings.

in

Numa dada lista.

Exemplo:

```
Entry.objects.filter(id__in=[1, 3, 4])
```

SQL equivalente:

```
SELECT ... WHERE id IN (1, 3, 4);
```

Você pode também usar um queryset para avaliar dinamicamente a lista de valores ao invés de fornecer uma lista de valores literais. O queryset deve ser reduzido a uma lista de valores individuais usando o método `values()`, e então convertido dentro de uma consulta usando o atributo `query`:

```
q = Blog.objects.filter(name__contains='Cheddar').values('pk').query
e = Entry.objects.filter(blog__in=q)
```

Warning: Este atributo `query` deve ser considerado um atributo interno opaco. É legal usá-lo como acima, mas sua API pode mudar entre as versões do Django.

Este queryset será avaliado como uma regra de subselect:

```
SELECT ... WHERE blog.id IN (SELECT id FROM ... WHERE NAME LIKE '%Cheddar%')
```

gt

Maior que.

Exemplo:

```
Entry.objects.filter(id__gt=4)
```

SQL equivalente:

```
SELECT ... WHERE id > 4;
```

gte

Maior ou igual a.

lt

Menor que.

lte

Menor ou igual a.

startswith

Começa com sensível a maiúsculas.

Exemplo:

```
Entry.objects.filter(headline__startswith='Will')
```

SQL equivalente:

```
SELECT ... WHERE headline LIKE 'Will%';
```

O SQLite não suporta regras LIKE sensíveis a maiúsculas; o `startswith` age como `istartswith` no SQLite.

istartswith

Começa com insensível a maiúsculas.

Exemplo:

```
Entry.objects.filter(headline__istartswith='will')
```

SQL equivalente:

```
SELECT ... WHERE headline ILIKE 'Will%';
```

SQLite users

Quando estiver usando o backend SQLite e strings Unicode (não ASCII), tenha em mente a *anotação de banco de dados* sobre comparação de strings.

endswith

Termina com sensível a maiúsculas.

Exemplo:

```
Entry.objects.filter(headline__endswith='cats')
```

SQL equivalente:

```
SELECT ... WHERE headline LIKE '%cats';
```

O SQLite não suporta regras LIKE sensíveis a maiúsculas; o `endswith` age como `iendswith` no SQLite.

iendswith

Termina com insensível a maiúsculas.

Exemplo:

```
Entry.objects.filter(headline__iendswith='will')
```

SQL equivalente:

```
SELECT ... WHERE headline ILIKE '%will'
```

SQLite users

Quando estiver usando o backend SQLite e strings Unicode (não ASCII), tenha em mente a *anotação de banco de dados* sobre comparação de strings.

range

Teste de faixa (inclusivo).

Exemplo:

```
start_date = datetime.date(2005, 1, 1)
end_date = datetime.date(2005, 3, 31)
Entry.objects.filter(pub_date__range=(start_date, end_date))
```

SQL equivalente:

```
SELECT ... WHERE pub_date BETWEEN '2005-01-01' and '2005-03-31';
```

Você pode usar `range` onde você puder usar `BETWEEN` no SQL – para datas, números ou mesmo caracteres.

year

Para campos `date/datetime`, combinação exata de ano. Recebe um ano com quatro dígitos.

Exemplo:

```
Entry.objects.filter(pub_date__year=2005)
```

SQL equivalente:

```
SELECT ... WHERE EXTRACT('year' FROM pub_date) = '2005';
```

(A sintaxe exata do SQL varia para cada motor de banco de dados.)

month

Para campos `date/datetime`, combinação exata de mês. Recebe um inteiro 1 (Janeiro) até 12 (Dezembro).

Exemplo:

```
Entry.objects.filter(pub_date__month=12)
```

SQL equivalente:

```
SELECT ... WHERE EXTRACT('month' FROM pub_date) = '12';
```

(A sintaxe exata do SQL varia para cada motor de banco de dados.)

day

Para campos `date/datetime`, combinação exata de dia.

Exemplo:

```
Entry.objects.filter(pub_date__day=3)
```

SQL equivalente:

```
SELECT ... WHERE EXTRACT('day' FROM pub_date) = '3';
```

(A sintaxe exata do SQL varia para cada motor de banco de dados.)

Note que isto combinará com qualquer dado com um `pub_date` no terceiro dia do mês, como Janeiro 3, Julho 3, etc.

isnull

Recebe um `True` ou `False`, que corresponde a consultas SQL `IS NULL` e `IS NOT NULL`, respectivamente.

Exemplo:

```
Entry.objects.filter(pub_date__isnull=True)
```

SQL equivalente:

```
SELECT ... WHERE pub_date IS NULL;
```

search

Um booleano de busca full-text, recebendo vantagem do indexamento full-text. Este é como o `contains` mas é significativamente mais rápido, devido ao indexamento full-text.

Exemplo:

```
Entry.objects.filter(headline__search="+Django -jazz Python")
```

SQL equivalente:

```
SELECT ... WHERE MATCH(tablename, headline) AGAINST (+Django -jazz Python IN_
↪BOOLEAN MODE);
```

Note que isso somente está disponível no MySQL e requer manipulação direta do banco de dados para adicionar o index full-text. Por padrão o Django usa `BOOLEAN MODE` nas buscas em full-text. [Por favor verifique a documentação do MySQL para detalhes adicionais.](#)

regex

Please, see the release notes Combinação por expressão regular sensível a maiúsculas.

A sintaxe de expressão regular é a que o backend do banco de dados usa. No caso do SQLite, que não suporta nativamente, pesquisa com expressões regulares, a sintaxe é a do módulo Python `re`.

Exemplo:

```
Entry.objects.get(title__regex=r'^(An?|The) +')
```

SQL equivalentes:

```
SELECT ... WHERE title REGEXP BINARY '^(An?|The) +'; -- MySQL
SELECT ... WHERE REGEXP_LIKE(title, '^(an?|the) +', 'c'); -- Oracle
SELECT ... WHERE title ~ '^(An?|The) +'; -- PostgreSQL
```

```
SELECT ... WHERE title REGEXP '^(An?|The) +'; -- SQLite
```

Using raw strings (e.g., `r'foo'` instead of `'foo'`) for passing in the regular expression syntax is recommended.

iregex

Please, see the release notes Combinação por expressão regular insensível a maiúsculas.

Exemplo:

```
Entry.objects.get(title__iregex=r'^(an?|the) +')
```

SQL equivalentes:

```
SELECT ... WHERE title REGEXP '^(an?|the) +'; -- MySQL
SELECT ... WHERE REGEXP_LIKE(title, '^(an?|the) +', 'i'); -- Oracle
SELECT ... WHERE title ~* '^(an?|the) +'; -- PostgreSQL
SELECT ... WHERE title REGEXP '^(?i)(an?|the) +'; -- SQLite
```

Objetos de requisição e resposta

Visão Geral

O Django utiliza objetos de requisição e resposta para passar estado através do sistema.

Quando uma página é requisitada, o Django cria um objeto *HttpRequest* que contém metadados sobre a requisição. Então o Django carrega a view apropriada, passando o *HttpRequest* como o primeiro argumento para a função de view. Cada view é responsável por devolver um objeto *HttpResponse*.

Este documento explica as APIs para os objetos *HttpRequest* e *HttpResponse*.

Objetos HttpRequest

class HttpRequest

Atributos

Todos os atributos, exceto *session*, devem ser considerados somente para leitura.

HttpRequest.path

Uma string representando o caminho completo para a página requisitada, não incluindo o domínio.

Exemplo: `"/music/bands/the_beatles/"`

HttpRequest.method

Uma string representando o método HTTP usado na requisição. Este valor está sempre em maiúsculo.

Exemplo:

```
if request.method == 'GET':
    do_something()
elif request.method == 'POST':
    do_something_else()
```

HttpRequest.encoding

Please, see the release notes Uma string representando o valor atual de codificação utilizado para decodificar o envio de dados de formulário (ou *None*, que quer dizer que o parâmetro de configuração `DEFAULT_CHARSET` é utilizado). Você pode alterar este atributo para modificar a codificação usada

quando acessar os dados do formulário. Quaisquer acessos subsequentes a atributos (como ler de GET ou POST) utilizará no novo valor de encodig. Isto é útil se você sabe que os dados do formulário não estão na codificação `DEFAULT_CHARSET`.

`HttpRequest.GET`

Um objeto, tipo dicionário, contendo todos os parâmetros HTTP GET. Veja a documentação do `QueryDict` abaixo.

`HttpRequest.POST`

Um objeto dicionário contendo todos os parametros passados por HTTP POST. Veja a documentação do `QueryDict` abaixo.

It's possible that a request can come in via POST with an empty POST dictionary – if, say, a form is requested via the POST HTTP method but does not include form data. Therefore, you shouldn't use `if request.POST` to check for use of the POST method; instead, use `if request.method == "POST"` (see above). É possível que um requisição POST possa vir com um dicionário POST vazio – se, digo, um formulário é requisitado via metodo HTTP POST mas não inclui dados. Portanto, você não deve usar `if request.POST` para checar o uso do método POST; ao invés, use `if request.method == "POST"` (veja acima).

Perceba: POST *não* inclui informações de upload de arquivos. Veja FILES.

`HttpRequest.REQUEST`

Por conveniência, um objeto dicionário que procura POST primeiro, somente depois GET. Inspirado no `$_REQUEST` do PHP.

Por exemplo, se `GET = {"name": "john"}` and `POST = {"age": '34'}`, `REQUEST["name"]` poderia ser "john", e `REQUEST["age"]` poderia ser "34".

É fortemente sugerido que você use o GET e POST ao invés de REQUEST, porque são mais explicitos.

`HttpRequest.COOKIES`

Um dicionário padrão do Python contendo todos os cookies. Chaves e valores são strings.

`HttpRequest.FILES`

Um objeto dicionário contendo todos os arquivos enviados. Cada chave em FILES é o name do `<input type="file" name="" />`. Cada valor em FILES é um objeto `UploadedFile` contendo os seguintes atributos:

- `read(num_bytes=None)` – Lê um número de bytes do arquivo.
- `name` – O nome do arquivo enviado.
- `size` – O tamanho, em bytes, do arquivo enviado.
- `chunks(chunk_size=None)` – Um gerado que fornece pedaços sequenciais de dados.

Veja *Gerenciando arquivos* para mais informações.

Note que FILES conterá somente dados se o método de requisição for POST e o `<form>` que postou a requisição tenha `enctype="multipart/form-data"`. De outra forma, FILES será um dicionário em branco. *Please, see the release notes* Nas versões anteriores do Django, `request.FILES` contendo um simples dict representando os arquivos enviados. Isto já não é verdade – os arquivos são representados por objetos `UploadedFile` como decrito abaixo.

Estes objetos `UploadedFile` emulam a interface do velho dict, mas que é depreciada e será removida no próximo lançamento do Django.

`HttpRequest.META`

Um dicionário padrão do Python contendo todos cabeçalhos disponíveis do HTTP. Os cabeçalhos disponíveis dependem do cliente e do servidor, mas aqui há alguns exemplos:

- `CONTENT_LENGTH`
- `CONTENT_TYPE`
- `HTTP_ACCEPT_ENCODING`
- `HTTP_ACCEPT_LANGUAGE`

- `HTTP_HOST` – O cabeçalho HTTP Host enviado pelo cliente.
- `HTTP_REFERER` – A página remetente, se houver uma.
- `HTTP_USER_AGENT` – A string do user-agent do cliente.
- `QUERY_STRING` – A query string, como uma string única (não parseada).
- `REMOTE_ADDR` – O endereço IP do cliente.
- `REMOTE_HOST` – O hostname do cliente.
- `REQUEST_METHOD` – Uma string como "GET" ou "POST".
- `SERVER_NAME` – O hostname do servidor.
- `SERVER_PORT` – A porta do servidor.

Com a exceção do `CONTENT_LENGTH` e `CONTENT_TYPE`, mostrados acima, qualquer cabeçalho HTTP na requisição é convertido para a chave `META`, tendo todos os seus caracteres passados para maiúsculo, substituindo os hífens por underscores e adicionando um `HTTP_` como prefixo do nome. Então, por exemplo, um cabeçalho chamado X-Bender seria mapeado para a chave `META` como `HTTP_X_BENDER`.

`HttpRequest.user`

A `django.contrib.auth.models.User` object representing the currently logged-in user. If the user isn't currently logged in, user will be set to an instance of `django.contrib.auth.models.AnonymousUser`. You can tell them apart with `is_authenticated()`, like so:: Um objeto `django.contrib.auth.models.User` representando o usuário atual logado. Se o usuário não estiver logado, o user conterá uma instância do `django.contrib.auth.models.AnonymousUser`. Você pode usar o método `is_authenticated()`, tipo:

```
if request.user.is_authenticated():
    # Faça algo para usuários logados.
else:
    # Faça algo para usuários anônimos.
```

O user é somente disponível se sua instalação do Django tem o middleware `AuthenticationMiddleware` ativado. Para mais, veja [Autenticação de Usuário no Django](#).

`HttpRequest.session`

Um objeto dicionário, onde é possível ler e escrever dados, que representa a sessão corrente. Ele somente é disponível se sua instalação do Django tiver o suporte a sessão ativado. Veja a [documentação do session](#) para detalhes completos.

`HttpRequest.raw_post_data`

Os dados do HTTP POST puros. Este é usual somente para processamentos avançados. Use POST no lugar dele.

`HttpRequest.urlconf`

Não definido pelo Django em si, mas será lido se outro código (e.g., uma classe middleware) setá-lo. Quando presente, será usado como o `URLconf` raiz para a requisição corrent, sobrescrever a configuração `ROOT_URLCONF`. Veja [Como o Django processa uma requisição](#) para detalhes.

Métodos

`HttpRequest.get_host()`

Please, see the release notes Retorna o servidor originador da requisição usando informações dos cabeçalhos `HTTP_X_FORWARDED_HOST` e `HTTP_HOST` (nesta ordem). Se eles não fornecem um valor, o método usa uma combinação de `SERVER_NAME` e `SERVER_PORT` como detalhado em [PEP 333](#).

Exemplo: "127.0.0.1:8000"

`HttpRequest.get_full_path()`

Retorna o path, mais uma query string anexa, se aplicável.

Exemplo: "/music/bands/the_beatles/?print=true"

`HttpRequest.build_absolute_uri(location)`

Please, see the release notes Retorna a URI absoluta de `location`. Se nenhum `location` é fornecido, o `location` será setado para `request.get_full_path()`.

Se o `location` já é uma URI absoluta, ele não será alterado. De outra forma a URI absoluta é construída usando as variáveis de servidor disponíveis nesta requisição.

Exemplo: `"http://example.com/music/bands/the_beatles/?print=true"`

`HttpRequest.is_secure()`

Retorna `True` se a requisição é segura; isto é, se ela foi feita com HTTPS.

`HttpRequest.is_ajax()`

Please, see the release notes Retorna `True` se a requisição foi feita via um `XMLHttpRequest`, checando o cabeçalho `HTTP_X_REQUESTED_WITH` por um string `'XMLHttpRequest'`. As seguintes bibliotecas JavaScript enviam seus cabeçalhos:

- jQuery
- Dojo
- MochiKit
- MooTools
- Prototype
- YUI

Se você escrever sua própria chamada `XMLHttpRequest` (no lado do navegador), terá de setar este cabeçalho manualmente se quiser que o `is_ajax()`.

Objetos QueryDict

class QueryDict

Num objeto `HttpRequest`, os atributos `GET` e `POST` são instâncias do `django.http.QueryDict`. O `QueryDict` é uma classe tipo dicionário customizada para lidar com múltiplos valores para a mesma chave. Isto é necessário porque alguns elementos de formulário HTML, notavelmente, `<select multiple="multiple">`, passam múltiplos valores para a mesma chave.

`QueryDict` instances are immutable, unless you create a `copy()` of them. That means you can't change attributes of `request.POST` and `request.GET` directly. A instância `QueryDict` é imutável, a menos que você crie uma `copy()` deles. O que significa que você não poderá mudar atributos do `request.POST` e `request.GET` diretamente.

Métodos

O `QueryDict` implementa todo os métodos padrão de dicionários, porque ele é uma subclasse de dicionário. Exceções são esboçadas aqui:

`QueryDict.__getitem__(key)`

Retorna o valor para a chave dada. Se a chave tem mais de um valor, `__getitem__()` retorna o último valor. Ele lança `django.utils.datastructure.MultiValueDictKeyError` se a chave não existe. (Esta é uma subclasse da classe `KeyError` padrão do Python, então você pode cutucá-la para pegar o `KeyError`.)

`QueryDict.__setitem__(key, value)`

Seta a chave data para `[value]` (uma lista do Python cujo o único elemento é `value`). Note que esta, como outras funções de dicionário que têm efeitos secundários, ela somente pode ser chamada num `QueryDict` mutável (um que foi criado via `copy()`).

`QueryDict.__contains__(key)`

Retorna `True` se a chave dada estiver setada. Este te permite fazer, e.g., `if "foo" in request.GET`.

`QueryDict.get(key, default)`

Use a mesma lógica em `__getitem__()` acima, com um hook para retornar um valor padrão se a chave não existe.

`QueryDict.setdefault(key, default)`

É como um método `setdefault()` padrão de dicionário, exceto por ele usar `__setitem__` internamente.

`QueryDict.update(other_dict)`

Recebe ambos `QueryDict` ou um dicionário padrão. É como o método `update()` padrão de dicionário, exceto por *atachar* os itens no dicionário atual ao invés de substituí-los. Por exemplo:

```
>>> q = QueryDict('a=1')
>>> q = q.copy() # para torná-lo mutável
>>> q.update({'a': '2'})
>>> q.getlist('a')
['1', '2']
>>> q['a'] # retorna o último
['2']
```

`QueryDict.items()`

É como o método `items()` padrão de dicionários, exceto por usar o mesmo último valor lógico como `__getitem__()`. Por exemplo:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.items()
[('a', '3')]
```

`QueryDict.iteritems()`

É como o método `iteritems()` padrão de dicionários. Como o `QueryDict.items()` este usa o mesmo último valor lógico como o `QueryDict.__getitem__()`.

`QueryDict.iterlists()`

Como `QueryDict.iteritems()` exceto por incluir todos os valores, como uma lista, para cada membro do dicionário.

`QueryDict.values()`

Como o método `values()` padrão de dicionários, exceto por usar o mesmo último valor lógico como `__getitem__()`. Por exemplo:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.values()
['3']
```

`QueryDict.itervalues()`

Como `QueryDict.values()`, exceto por ser um iterador.

Além disso, o `QueryDict` tem os seguintes métodos:

`QueryDict.copy()`

Retorna uma cópia do objeto, usando `copy.deepcopy()` da biblioteca padrão do Python. A cópia será mutável – isto é, você poderá mudar seus valores.

`QueryDict.getlist(key)`

Retorna os dados da chave requisitada, como uma lista Python. Retorna uma lista vazia se a chave não existe. É garantido o retorno de uma lista de algum tipo.

`QueryDict.setlist(key, list_)`

Seta a chave dada com `list_` (diferentemente de `__setitem__()`).

`QueryDict.appendlist(key, item)`

Appends an item to the internal list associated with key. Atache um item para a lista interna associada com a chave.

`QueryDict.setlistdefault(key, default_list)`

Como `setdefault`, exceto por receber uma lista de valores ao invés de um valor único.

`QueryDict.lists()`

Como `items()`, exceto por incluir todos os valores, como uma lista, para cada membro do dicionário. Por exemplo:

```
>>> q = QueryDict('a=1&a=2&a=3')
>>> q.lists()
[('a', ['1', '2', '3'])]
```

`QueryDict.urlencode()`

Retorna uma string de dados no formato query string. Exemplo: "a=2&b=3&b=5".

Objetos `HttpResponse`

`class HttpResponse`

Em contraste com objetos `HttpRequest`, que são criados automaticamente pelo Django, os objetos `HttpResponse` são sua responsabilidade. Cada view que você escrever é responsável por instanciar, popular e retornar um `HttpResponse`.

The `HttpResponse` class lives in the `django.http` module. A classe `HttpResponse` reside o módulo `django.http`.

Uso

Passando strings

Um uso típico é passar os conteúdos de uma página, como uma string, para o construtor `HttpResponse`:

```
>>> response = HttpResponse("Aqui tem um texto para a página Web.")
>>> response = HttpResponse("Somente texto, por favor.", mimetype="text/plain")
```

Mas se você quiser adicionar conteúdo incrementalmente, você pode usar o `response` como um objeto:

```
>>> response = HttpResponse()
>>> response.write("<p>Aqui tem um texto para a página Web.</p>")
>>> response.write("<p>Aqui tem outro parágrafo.</p>")
```

Você pode adicionar e deletar cabeçalhos usando sintaxe de dicionário:

```
>>> response = HttpResponse()
>>> response['X-DJANGO'] = "Ele é o melhor."
>>> del response['X-PHP']
>>> response['X-DJANGO']
"Ele é o melhor."
```

Note que `del` não lança um `KeyError` se o cabeçalho não existe.

Passando iteradores

Finalmente, você pode passar ao `HttpResponse` um iterador ao invés de passar strings na mão. Se você usar esta técnica, siga estas dicas:

- O iterador deve retornar strings.
- Se um `HttpResponse` foi inicializado com um iterador como seu conteúdo, você não pode usar a instância `HttpResponse` como um objeto. Fazendo-o então lançar uma `Exception`.

Configurando cabeçalhos

Para setar um cabeçalho em sua resposta, é só tratá-la como um dicionário:

```
>>> response = HttpResponse()
>>> response['Pragma'] = 'no-cache'
```

Dizendo ao navegador para tratar a resposta como um arquivo anexado

Para dizer ao navegador para tratar a resposta como um arquivo anexado, use o argumento `mimetype` e set o cabeçalho `Content-Disposition`. Por exemplo, esta é a forma como você pode retornar uma planilha do Microsoft Excel:

```
>>> response = HttpResponse(my_data, mimetype='application/vnd.ms-excel')
>>> response['Content-Disposition'] = 'attachment; filename=foo.xls'
```

Não há nada específico sobre o cabeçalho `Content-Disposition` do Django, mas é fácil esquecer a sintaxe, então nós incluímos ela aqui.

Atributos

`HttpResponse.content`

Uma string normal do Python representando o conteúdo, codificado como um objeto Unicode se necessário.

Métodos

`HttpResponse.__init__(content='', mimetype=None, status=200, content_type=DEFAULT_CONTENT_TYPE)`

Instancia um objeto `HttpResponse` com o conteúdo da página fornecida (uma string) e o “MIME type”. O `DEFAULT_CONTENT_TYPE` é `'text/html'`.

O `content` pode ser um iterador ou string. Se for um iterador, ele deve retornar strings, e suas strings serão unidas para formar o conteúdo da resposta.

O `status` é o [Status code HTTP](#) para a resposta. *Please, see the release notes* O `content_type` é um alias para `mimetype`. Historicamente, este parâmetro foi chamado somente `mimetype`, mas como na verdade ele é o valor incluso no cabeçalho HTTP `Content-Type`, também pode incluir o conjunto de caracteres de codificação.

`HttpResponse.__setitem__(header, value)`

Seta o nome do cabeçalho dado com o valor fornecido. Ambos `header` e `value` devem ser strings.

`HttpResponse.__delitem__(header)`

Deleta o cabeçalho com o nome dado. Falha silenciosamente se o cabeçalho não existe. É case-sensitive.

`HttpResponse.__getitem__(header)`

Retorna o valor para o nome do cabeçalho dado. É case-sensitive.

`HttpResponse.has_header(header)`

Retorna `True` ou `False` baseado em uma checagem case-insensitive para o cabeçalho com o nome fornecido.

`HttpResponse.set_cookie(key, value='', max_age=None, expires=None, path='/', domain=None, secure=None)`

Seta um cookie. Os parametros são os mesmo do objeto [cookie Morsel](#) da biblioteca padrão do Python.

- O `max_age` deve ser um número de segundo, ou `None` (padrão) se o cookie deve durar somente enquanto a sessão do browser do cliente estiver aberta.
- `expires` deve ser uma string no formato `"Wdy, DD-Mon-YY HH:MM:SS GMT"`.

- Usa o `domain` se você quiser setar um cross-domain cookie. Por exemplo, `domain=".lawrence.com"` setará um cookie que é legível pelo domínio `www.lawrence.com`, `blogs.lawrence.com` e `calendars.lawrence.com`. Caso contrário, um cookie somente será acessível pelo domínio que o setar.

`HttpResponse.delete_cookie(key, path='/', domain=None)`

Deleta o cookie com o a chave fornecida. Falha silenciosamente se a chave não existe.

Devido a forma como os cookies funcionam, `path` e `domain` devem ser o mesmo valor usado em `set_cookie()` – caso contrário o cookie pode não ser deletado.

`HttpResponse.write(content)`

Este método monta uma instância de um objeto `HttpResponse` em um arquivo.

`HttpResponse.flush()`

Este método monta uma instância do objeto `HttpResponse` em um arquivo.

`HttpResponse.tell()`

Este método monta uma instância de objeto `HttpResponse` em um arquivo.

Subclasses do `HttpResponse`

O Django inclui um número de subclasses `HttpResponse` que manipulam direntes tipos de respostas HTTP. Como `HttpResponse`, estas subclasses residem o módulo `django.http`.

class `HttpResponseRedirect`

O construtor recebe um único argumento – o caminho para onde deve ser redirecionado. Este pode ser uma URL completa (e.g. `'http://www.yahoo.com/search/'`) ou uma URL absoluta sem domínio (e.g. `'/search'`). Note que ele retorna o status code 302.

class `HttpResponsePermanentRedirect`

Como um `HttpResponseRedirect`, mas ele retorna um redirecionamento permanente (HTTP status code 300) ao invés de um redirecionamento “found” (status code 302).

class `HttpResponseNotModified`

O construtor não recebe qualquer argumento. Use isto para designar que uma página não foi modificada desde a última requisição do usuário (status code 304).

class `HttpResponseBadRequest`

Please, see the release notes Age como um `HttpResponse` mas usa um “status code” 400.

class `HttpResponseNotFound`

Age como um `HttpResponse` mas usa um “status code” 404.

class `HttpResponseForbidden`

Age como um `HttpResponse` mas usa o “status code” 403.

class `HttpResponseNotAllowed`

Como o `HttpResponse`, mas usa um “status code” 405. Recebe um argumento único, obrigatório: uma lista de métodos permitidos (e.g. `['GET', 'POST']`).

class `HttpResponseGone`

Age como um `HttpResponse` mas usa um “status code” 410.

class `HttpResponseServerError`

Age como um `HttpResponse` mas usa um “status code” 500.

Configurações disponíveis

Aqui temos uma lista completa de todas as configurações disponíveis, em ordem alfabética, e seus valores padrões.

ABSOLUTE_URL_OVERRIDES

Padrão: `{}` (Dicionário vazio)

Um mapeamento de strings num dicionário `"app_label.model_name"` para funções que recebem um objeto `model` e retornam sua URL. Esta é a forma de sobrescrever métodos `get_absolute_url()` numa instalação básica. Exemplo:

```
ABSOLUTE_URL_OVERRIDES = {
    'blogs.weblog': lambda o: "/blogs/%s/" % o.slug,
    'news.story': lambda o: "/stories/%s/%s/" % (o.pub_year, o.slug),
}
```

Perceba que o nome do `model` usado nesta configuração deve ser todo em minúsculo, indiferente da configuração do nome da classe atual do `model`.

ADMIN_FOR

Padrão: `()` (Tupla vazia)

Usado para configurar módulos do site admin, ele deve ser uma tupla de módulos de configuração (no formato `'foo.bar.baz'`) para que este site esteja no admin.

O site admin usa isto na introspecção automática de documentação dos `models`, `views` e `tags` de templates.

ADMIN_MEDIA_PREFIX

Padrão: `'/media/'`

Prefixo de URL para as mídias do admin – CSS, Javascript e imagens usadas pela interface administrativa do Django. Assegure-se de usar um barra no final do caminho, e de ele ser diferente da configuração `MEDIA_URL` (uma vez que uma mesma URL não pode ser mapeada para dois conjuntos diferentes de arquivos).

ADMINS

Padrão: `()` (Tupla vazia)

Uma tupla que lista pessoas que receberam notificações de erro. Quando o `DEBUG=False` e uma view lança uma exceção, o Django enviará um e-mail para estas pessoas com a informação completa da exceção. Cada item da tupla deve ser uma tupla com (Nome completo, endereço de e-mail). Exemplo:

```
(( 'John', 'john@example.com'), ('Mary', 'mary@example.com'))
```

Perceba que o Django enviará um e-mail para *todos* que estiverem na lista, toda vez que um erro ocorrer. Veja [Reporte de erros via e-mail](#) para mais informações.

ALLOWED_INCLUDE_ROOTS

Padrão: `()` (Tupla vazia)

Uma tupla de strings representando prefixos permitidos para a template tag `{% ssi %}`. Esta é uma medida de segurança, de modo que os autores de templates não possam acessar arquivos que não deveriam acessar.

Por exemplo, se `ALLOWED_INCLUDE_ROOTS` é `('/home/html', '/var/www')`, então o `{% ssi /home/html/foo.txt %}` funcionaria, mas `{% ssi /etc/passwd %}` não.

APPEND_SLASH

Padrão: `True`

Para atachar uma barra ao final das URLs. Isto é utilizado somente se o `CommonMiddleware` estiver instalado (veja [Middleware](#)). veja também `PREPEND_WWW`.

AUTHENTICATION_BACKENDS

Padrão: `('django.contrib.auth.backends.ModelBackend',)`

Uma tupla de classes de backend de autenticação (como strings) para serem usadas nas tentativas de autenticação de usuários. Veja a [documentação de backends de autenticação](#) para mais detalhes.

AUTH_PROFILE_MODULE

Padrão: `Not defined`

O model de “profile” de usuário específico para um site, e usado para este site. Veja [Armazenando informações adicionais sobre usuários](#).

CACHE_BACKEND

Padrão: `'locmem://'`

O backend de cache a ser utilizado. Veja *O framework de cache do Django*.

CACHE_MIDDLEWARE_KEY_PREFIX

Padrão: `''` (Empty string)

A chave de prefixo do cache, que o middleware de cache deverá usar. Veja *O framework de cache do Django*.

CACHE_MIDDLEWARE_SECONDS

Padrão: `600`

O número padrão de segundos para fazer cache e uma página quando o middleware de cache ou o decorador `cache_page()` for usado.

DATABASE_ENGINE

Padrão: `''` (Empty string)

O backend de banco de dados a ser utilizado. Os backends de banco de dados embutidos no Django são `'postgresql_psycopg2'`, `'postgresql'`, `'mysql'`, `'sqlite3'`, e `'oracle'`.

Você pode usar um backend de banco de dados que não é entregue com o Django setando o `DATABASE_ENGINE` com um caminho completo (i.e. `mypackage.backends.whatever`). Escrever todo um novo backend de banco de dados do zero é deixado como um exercício para o leitor; estude os outros backends como exemplos. Suporte a backends de bancos de dados externos é novidade da versão 1.0.

DATABASE_HOST

Padrão: `''` (Empty string)

Qual host será usado quando conectar-se a um banco de dados. Uma string vazia significa localhost. Não é utilizado com SQLite.

Se este valor começa com uma barra (`'/'`) e você estiver usando MySQL, o MySQL se conectará a um socket Unix no socket especificado. Por exemplo:

```
DATABASE_HOST = '/var/run/mysql'
```

Se você estiver usando o MySQL e este valor *não* começar com uma barra, então este valor é assumido como um host.

Se você estiver usando o PostgreSQL, uma string vazia significa usar um socket de domínio Unix para a conexão, ao invés de uma conexão de rede para localhost. Se você precisa ser explícito em usar uma conexão TCP/IP na máquina local com PostgreSQL, especifique `localhost` aqui.

DATABASE_NAME

Padrão: '' (Empty string)

O nome do banco de dados que será usado. Para o SQLite, este valor é o caminho completo do arquivo de banco de dados. Quando especificar o caminho, sempre use barras ('/'), mesmo no ambiente Windows (e.g. C:/homes/user/mysite/sqlite3.db).

DATABASE_OPTIONS

Padrão: {} (Empty dictionary)

Parâmetros extra para usar quando conectar-se ao banco de dados. Consulte a documentação dos módulos back-ends para chaves disponíveis.

DATABASE_PASSWORD

Padrão: '' (Empty string)

A senha a ser usada quando conectar com o banco de dados. Não é usado no SQLite.

DATABASE_PORT

Padrão: '' (Empty string)

A porta a ser usada quando conectar ao banco de dados. Uma string vazia significa a porta padrão. Não é usado com SQLite.

DATABASE_USER

Padrão: '' (Empty string)

O nome de usuário a ser usado quando conectar com o banco de dados. Não é usado com SQLite.

DATE_FORMAT

Padrão: 'N j, Y' (e.g. Feb. 4, 2003)

O formato padrão a ser utilizado para os campos de data no admin do Django, nas páginas de listagem – e, possivelmente, por outras partes do sistema. Veja *formatos de string permitidos para datas*.

Veja também `DATETIME_FORMAT`, `TIME_FORMAT`, `YEAR_MONTH_FORMAT` e `MONTH_DAY_FORMAT`.

DATETIME_FORMAT

Padrão: 'N j, Y, P' (e.g. Feb. 4, 2003, 4 p.m.)

O formato padrão usado nos campos de data no admin do Django, nas páginas de listagem – e, possivelmente, por outras partes do sistema. Veja *formatos de string permitidos para datas*.

Veja também `DATE_FORMAT`, `DATETIME_FORMAT`, `TIME_FORMAT`, `YEAR_MONTH_FORMAT` e `MONTH_DAY_FORMAT`.

DEBUG

Padrão: `False`

Um booleano que liga ou desliga o modo debug.

Se você define uma configuração personalizada, `django/views/debug.py` tem uma expressão regular `HIDDEN_SETTINGS` será escondida da visão de `DEBUG` tudo que contém `'SECRET'`, `'PASSWORD'`, ou `'PROFANITIES'`. Isso permite que usuários, não confiáveis, recebam backtraces sem ver configurações sensíveis do sistema.

Ainda, não que há seções de sua saída de depuração que são inapropriadas para o consumo do público. Caminhos de arquivos, opções de configuração, é como dar informações extra sobre o seu servidor aos atacantes.

É importante lembrar também, que quando estiver rodando com o `DEBUG` ligado, o Django evoca todas as consultas SQL executadas. Isto é usual quando você está debugando, mas no servidor de produção, irá consumir rapidamente muita memória.

Nunca coloque um site em produção com o `DEBUG` ligado.

DEBUG_PROPAGATE_EXCEPTIONS

Please, see the release notes Padrão: `False`

Se for `True`, a manipulador de exceção normal do Django das funções view será suprimido, e as exceções serão propagadas para os níveis superiores. Isto pode ser útil numa configuração de testes, e nunca deve ser usado num site em produção.

DEFAULT_CHARSET

Padrão: `'utf-8'`

O charset a ser utilizado para todas os objetos `HttpResponse`, se um tipo MIME não for manualmente especificado. Usado com `DEFAULT_CONTENT_TYPE` para construir o cabeçalho `Content-Type`.

DEFAULT_CONTENT_TYPE

Padrão: `'text/html'`

Tipo de conteúdo padrão a ser utilizado por todos os objetos `HttpResponse`, se um tipo MIME não for manualmente especificado. Usado como `DEFAULT_CHARSET` para construir o cabeçalho `Content-Type`.

DEFAULT_FILE_STORAGE

Padrão: `'django.core.files.storage.FileSystemStorage'`

Arquivo padrão de classe storage a ser usado por quaisquer operações relacionadas a arquivos que não especificam um sistema de armazenamento particular. Veja *Gerenciando arquivos*.

DEFAULT_FROM_EMAIL

Padrão: `'webmaster@localhost'`

Endereço de email padrão para ser usado por várias correspondências automáticas para o(s) gerente(s) do site.

DEFAULT_TABLESPACE

Please, see the release notes Padrão: '' (Empty string)

Tablespace padrão a ser usado pelos models que não especificarem um, caso o backend suporte.

DEFAULT_INDEX_TABLESPACE

Please, see the release notes Padrão: '' (Empty string)

Tablespace padrão a ser usado para indexes dos campos que não especificar um, caso o backend suporte.

DISALLOWED_USER_AGENTS

Padrão: () (Empty tuple)

Lista de objetos de expressões regulares compiladas representando strings de User-Agent que não são permitidos para visitar qualquer página, ao longo do sistema. Use isto contra robots/crawlers. Isso somente será usado se o `CommonMiddleware` estiver instalado (veja [Middleware](#)).

EMAIL_HOST

Padrão: 'localhost'

O servidor usado para enviar e-mail.

Veja também `EMAIL_PORT`.

EMAIL_HOST_PASSWORD

Padrão: '' (Empty string)

Password do servidor SMTP definido no `EMAIL_HOST`. Esta configuração é usada na conjunção com o `EMAIL_HOST_USER` durante a autenticação do servidor SMTP. Se ambas configurações forem vazias, o Django não tenta fazer a autenticação.

Veja também `EMAIL_HOST_USER`.

EMAIL_HOST_USER

Padrão: '' (Empty string)

Username para o servidor SMTP definido em `EMAIL_HOST`. Se vazio, o Django não tentará a autenticação.

Veja também `EMAIL_HOST_PASSWORD`.

EMAIL_PORT

Padrão: 25

Porta do servidor SMTP definida no `EMAIL_HOST`.

EMAIL_SUBJECT_PREFIX

Padrão: `'[Django] '`

Prefixo de assunto para as mensagens de e-mail enviadas com `django.core.mail.mail_admins` ou `django.core.mail.mail_managers`. Você provavelmente precisará incluir o espaço no final da linha.

EMAIL_USE_TLS

Please, see the release notes Padrão: `False`

Se for usar conexão TLS (segura) quando falar com o servidor SMTP.

FILE_CHARSET

Please, see the release notes Padrão: `'utf-8'`

A codificação de caracteres usado para decodificar qualquer arquivo lido do disco. Este inclui os arquivos de template e arquivos de dados SQL iniciais.

FILE_UPLOAD_HANDLERS

Please, see the release notes Padrão:

```
("django.core.files.uploadhandler.MemoryFileUploadHandler",  
 "django.core.files.uploadhandler.TemporaryFileUploadHandler",)
```

Uma tupla de manipuladores de upload. Veja *Gerenciando arquivos* para detalhes.

FILE_UPLOAD_MAX_MEMORY_SIZE

Please, see the release notes Padrão: `2621440` (i.e. 2.5 MB).

O tamanho máximo (em bytes) de um upload terá, antes de ser redirecionado para o sistema de arquivo. Veja *Gerenciando arquivos* para detalhes.

FILE_UPLOAD_TEMP_DIR

Please, see the release notes Padrão: `None`

O diretório para armazenar os arquivos durante o processo de upload. Se `None`, o Django usará o diretório temporário padrão do sistema operacional. Por exemplo, ele será `'/tmp'` em sistemas operacionais `*nix`.

Veja *Gerenciando arquivos* para detalhes.

FILE_UPLOAD_PERMISSIONS

Padrão: `None`

O modo numérico (i.e. `0644`) para setar o novo arquivo enviado. Para mais informações sobre o que significa esses modos, veja a [documentação do `os.chmod`](#)

Se este não for dado ou for `None`, você terá um comportamento dependendo do sistema operacional. Na maioria das plataformas, arquivos temporários possuem um modo `0600`, e arquivos salvos a partir da memória serão salvos usando o umask padrão do sistema.

Warning: Sempre prefixe o modo com um `0`.

Se você não está familiarizado com os modos de arquivos, por favor perceba que o zero, `0`, que vai na frente é muito importante: ele indica um número octal, pois é como os modos são especificados. Se você tentar usar `644`, você terá um comportamento totalmente errado.

FIXTURE_DIRS

Padrão: `()` (Empty tuple)

Lista de localizações de arquivos de dados para fixtures, na ordem de busca. Note que estes caminhos devem usar o estilo Unix, com barras `'/'`, mesmo no Windows. Veja [Testando aplicações Django](#).

FORCE_SCRIPT_NAME

Padrão: `None`

Se não for `None`, ele será usado como o valor de ambiente `SCRIPT_NAME` em qualquer requisição HTTP. Esta configuração pode ser usada para sobrescrever o valor fornecido pelo servidor `SCRIPT_NAME`, que pode ser uma versão reescrita do valor preferido ou não fornecido.

IGNORABLE_404_ENDS

Padrão: `('mail.pl', 'mailform.pl', 'mail.cgi', 'mailform.cgi', 'favicon.ico', '.php')`

Veja também `IGNORABLE_404_STARTS` e [Error reporting via e-mail](#).

IGNORABLE_404_STARTS

Padrão: `('/cgi-bin/', '/_vti_bin', '/_vti_inf')`

Uma tupla de strings que especificam o início de URLs que devem ser ignoradas pelo e-mailer 404. Veja `SEND_BROKEN_LINK_EMAILS`, `IGNORABLE_404_ENDS` e o [Reporte de erros via e-mail](#).

INSTALLED_APPS

Padrão: `()` (Empty tuple)

Uma tupla de strings designando todas as aplicações que estão habilitadas nesta instalação do Django. Cada string deve ser um caminho do Python completo para o pacote que contém uma aplicação Django, como criada pelo `django-admin.py startapp`.

INTERNAL_IPS

Padrão: `()` (Empty tuple)

Uma tupla de endereços de IP, como strings, que:

- Veja os comentários do debug, quando `DEBUG` for `True`
- Recebe um cabeçalho `X` se o `XViewMiddleware` estiver instalado (veja [Middleware](#))

JING_PATH

Padrão: `'/usr/bin/jing'`

Caminho para o executável “Jing”. Jing é um validador RELAX NG, e o Django usa ele para validar cada `XMLField` nos seus models. Veja <http://www.thaiopensource.com/relaxng/jing.html>.

LANGUAGE_CODE

Padrão: `'en-us'`

Uma string representando o código do idioma para esta instalação. Este deve ser no formato de linguagens padrão. Por exemplo, para U.S. English é `"en-us"`. Para [Internacionalização](#).

LANGUAGE_COOKIE_NAME

Please, see the release notes Padrão: `'django_language'`

O nome que será usado para o cookie de linguagem. Este pode ser qualquer nome que quiser (mas deve ser diferente do `SESSION_COOKIE_NAME`). Veja [Internacionalização](#).

LANGUAGES

Padrão: A tuple of all available languages. This list is continually growing and including a copy here would inevitably become rapidly out of date. You can see the current list of translated languages by looking in `django/conf/global_settings.py` (or view the [online source](#)).

A lista é uma tupla de tuplas duplas no formato (código do idioma, nome do idioma) – por exemplo, `('ja', 'Japanese')`. Isto especifica quais idiomas estão disponíveis para seleção. Veja [Internacionalização](#).

Geralmente, o valor padrão deveria bastar. Somente se esta configuração se você quer restringir a seleção de idiomas para um subconjunto de línguas fornecidas pelo Django.

Se você define uma configuração de `LANGUAGES` personalizada, ela está OK para marcar as línguas como strings de tradução (como no valor padrão mostrado acima) – mas use uma função `gettext()` “dummy”, não a que está em `django.utils.translation`. Você *nunca* deve importar o `django.utils.translation` dentro do seu arquivo de configuração, porquê este módulo em si depende do settings, e que poderia causar uma importação circular.

A solução é usar uma função `gettext()` “dummy”. Aqui temos um exemplo:

```
gettext = lambda s: s

LANGUAGES = (
    ('de', gettext('German')),
```



```
( 'en', gettext('English')),  
)
```

Com este arranjo, o `django-admin.py makemessages` ainda encontrará e marcará estas strings para tradução, mas a tradução não acontecerá em tempo de execução – então você terá de lembrar de envolver os idiomas no `gettext()` *real* em qualquer código que usar o `LANGUAGES` em tempo de execução.

LOCALE_PATHS

Padrão: `()` (Empty tuple)

Uma tupla de diretórios onde o Django procura por arquivos de tradução. Veja *Usando traduções em seus próprios projetos*.

LOGIN_REDIRECT_URL

Please, see the release notes Padrão: `'/accounts/profile/'`

A URL para onde a requisição é redirecionada depois do login, quando o view `contrib.auth.login` não recebe um parametro `next`.

Este é usado pelo decorador `login_required()`, por exemplo.

LOGIN_URL

Please, see the release notes Padrão: `'/accounts/login/'`

A URL onde as requisições são redirecionadas para o login, especialmente quando se usa o decorador `login_required()`.

LOGOUT_URL

Please, see the release notes Padrão: `'/accounts/logout/'`

O contrário do `LOGIN_URL`.

MANAGERS

Padrão: `()` (Empty tuple)

Uma tupla no mesmo formato do `ADMINS` que especifica quem deve receber notificações de links quebrados quando `SEND_BROKEN_LINK_EMAILS=True`.

MEDIA_ROOT

Padrão: `''` (Empty string)

Caminho absoluto para o diretório que mantem as mídias desta instalação. Exemplo: `"/home/media/media.lawrence.com/"` Veja também `MEDIA_URL`.

MEDIA_URL

Padrão: '' (Empty string)

A URL que manipula a mídia servida por MEDIA_ROOT. Exemplo: "http://media.lawrence.com"

Note que este deve ter uma barra no final, se ele tem um componente de caminho.

Bom: "http://www.example.com/static/" Ruim: "http://www.example.com/static"

MIDDLEWARE_CLASSES

Padrão:

```
("django.contrib.sessions.middleware.SessionMiddleware",
 "django.contrib.auth.middleware.AuthenticationMiddleware",
 "django.middleware.common.CommonMiddleware",
 "django.middleware.doc.XViewMiddleware")
```

Uma tupla de classes middleware para usar. Veja *Middleware*.

MONTH_DAY_FORMAT

Padrão: 'F j'

O formato padrão a ser usado para campos de data nas páginas de listagens do Django admin – e, possivelmente, por outras partes do sistema – em casos onde somente o mês e o dia são mostrados.

Por exemplo, quando uma página de listagem do Django admin está sendo filtrada por data, o cabeçalho para um dado dia mostra o dia e o mês. Diferentes localidades possuem diferentes formatos. Por exemplo, nos E.U.A costumam dizer “January 1,” em áreas espanholas pode ser dito “1 Enero.”

Veja *formatos de strings de data permitidos*. Veja também DATE_FORMAT, DATETIME_FORMAT, TIME_FORMAT and YEAR_MONTH_FORMAT.

PREPEND_WWW

Padrão: False

Para prefixar o subdomínio “www.” nas URLs que não o possuem. Este é somente usado se CommonMiddleware estiver instalado (veja *Middleware*). Veja também APPEND_SLASH.

PROFANITIES_LIST

Uma tupla de profanities, como strings, que engatilharão um erro de validação quando o validador hasNoProfanities for chamado.

Nós não listamos os valores padrão aqui, porque eles podem ser profanos. Para ver os valores padrões, veja o arquivo `django/conf/global_settings.py`.

ROOT_URLCONF

Padrão: Not defined

Uma string representando o caminho de import completo do Python para o seu URLconf. Por exemplo: `"mydjangoapps.urls"`. Pode ser sobrescrito caso necessário, definindo o atributo `urlconf` no objeto `HttpRequest` que chega. Veja [Como o Django processa uma requisição](#) para detalhes.

SECRET_KEY

Padrão: '' (Empty string)

Uma chave secreta para esta instalação do Django em particular. Usada para prover uma chave secreta para o algoritmo de hash. Defina como uma string aleatória – quanto maior, melhor. O `django-admin.py startproject` cria uma automaticamente.

SEND_BROKEN_LINK_EMAILS

Padrão: False

Para enviar um e-mail para os `MANAGERS` toda vez que alguém visita uma página feita com o Django que retornou um erro 404 com um referer não vazio (i.e. um link quebrado). Isso só é usado se o `CommonMiddleware` estiver instalado (veja [Middleware](#)). Veja também `IGNORABLE_404_STARTS`, `IGNORABLE_404_ENDS` e [Reporte de erros via e-mail](#).

SERIALIZATION_MODULES

Padrão: Not defined.

A dictionary of modules containing serializer definitions (provided as strings), keyed by a string identifier for that serialization type. For example, to define a YAML serializer, use:

```
SERIALIZATION_MODULES = { 'yaml' : 'path.to.yaml_serializer' }
```

SERVER_EMAIL

Padrão: 'root@localhost'

O endereço de e-mail de onde vem as mensagens de erro, como um dos que envia para os `ADMINS` e `MANAGERS`.

SESSION_ENGINE

Please, see the release notes Padrão: `django.contrib.sessions.backends.db`

Controla onde o Django armazena os dados da sessão. Valores válidos são:

- `'django.contrib.sessions.backends.db'`
- `'django.contrib.sessions.backends.file'`
- `'django.contrib.sessions.backends.cache'`

Veja [Como utilizar sessões](#).

SESSION_COOKIE_AGE

Padrão: 1209600 (2 weeks, in seconds)

A longevidade dos cookies de sessão, em segundos. Veja *Como utilizar sessões*.

SESSION_COOKIE_DOMAIN

Padrão: None

O domínio dos cookies de sessão. Defina isso como uma string, por exemplo ".lawrence.com", para cookies cross-domain, ou use None para um nome de domínio padrão. Veja o *Como utilizar sessões*.

SESSION_COOKIE_NAME

Padrão: 'sessionid'

O nome do cookie das sessões. Este pode ser qualquer um que você queira (mas deve ser diferente do LANGUAGE_COOKIE_NAME). Veja *Como utilizar sessões*.

SESSION_COOKIE_PATH

Please, see the release notes Padrão: '/'

O caminho setado no cookie da sessão. Isso deve corresponder ao caminho ou a URL de sua instalação do Django ou ser antecessor deste caminho.

Isso é útil se você tem várias instâncias do Django rodando sob o mesmo hostname. Eles podem usar diferentes caminhos de cookie, e cada instância somente verá seu próprio cookie de sessão.

SESSION_COOKIE_SECURE

Padrão: False

Para usar um cookie seguro como cookie de sessão. Se este é setado como True, o cookie será marcado como "secure", o que significa que os navegadores podem assegurar que o cookie é somente enviado sob uma conexão HTTPS. Veja o *Como utilizar sessões*.

SESSION_EXPIRE_AT_BROWSER_CLOSE

Padrão: False

Para expirar a sessão quando o usuário fecha seu navegador. Veja o *Como utilizar sessões*.

SESSION_FILE_PATH

Please, see the release notes Padrão: None

Se você estiver usando armazenamento de sessão baseado em arquivos, este seta o diretório em que o Django armazenará os dados de sessão. Veja *Como utilizar sessões*. Quando o valor padrão None é usado, o Django usará o diretório padrão do sistema.

SESSION_SAVE_EVERY_REQUEST

Padrão: `False`

Para salvar os dados da sessão em toda requisição. Veja *Como utilizar sessões*.

SITE_ID

Padrão: `Not defined`

O `site_id`, como um inteiro, do site atual na tabela do banco de dados `django_site`. Isso é usado de modo que os dados da aplicação possam ser presos a um ou mais sites específicos. Um mesmo banco de dados pode gerenciar o conteúdo de múltiplos sites.

Veja *The “sites” framework*.

TEMPLATE_CONTEXT_PROCESSORS

Padrão:

```
("django.core.context_processors.auth",
"django.core.context_processors.debug",
"django.core.context_processors.i18n",
"django.core.context_processors.media")
```

Uma tupla com funções chamáveis que são usadas para popular o contexto no `RequestContext`. Estas funções recebem um objeto `request` como seu argumento e retornam um dicionário de itens a serem mesclados ao contexto.

TEMPLATE_DEBUG

Padrão: `False`

Um booleano que liga/desliga o modo debug do template. Se for `True`, a página de erro mostrará um relatório detalhado para qualquer `TemplateSyntaxError`. Este relatório contém o fragmento relevante do template, com a linha apropriada em destaque.

Note que o Django somente mostra páginas de erros se o `DEBUG` for `True`, de modo que você precisará setar isso para obter vantagem desta configuração.

Veja também `DEBUG`.

TEMPLATE_DIRS

Padrão: `()` (Empty tuple)

Lista de localizações de arquivos fontes de templates, na ordem de busca. Note que estes caminhos devem usar o estilo Unix, com uma barra na frente, mesmo no Windows.

Veja *The Django template language*.

TEMPLATE_LOADERS

Padrão:

```
('django.template.loaders.filesystem.load_template_source',  
 'django.template.loaders.app_directories.load_template_source')
```

Uma tupla de funções (como strings) que sabem como importar templates de várias fontes. Veja *The Django template language: For Python programmers*.

TEMPLATE_STRING_IF_INVALID

Padrão: '' (Empty string)

Saída, como uma string, que o sistema de template deve usar para variáveis inválidas (e.g. com erro de escrita). Veja *How invalid variables are handled*.

TEST_DATABASE_CHARSET

Please, see the release notes Padrão: None

A codificação de caracteres usado para criar as tabelas de testes. O valor desta string é passado diretamente para banco de dados, então seu formato é específico para os backends.

Suportado pelos backends PostgreSQL (postgresql, postgresql_psycopg2) e MySQL (mysql).

TEST_DATABASE_COLLATION

Please, see the release notes Padrão: None

A forma de agrupamento a ser usada quando criar as tabelas no banco de dados de testes. Este valor é passado diretamente para o backend, de modo que seu formato é específico para os backends.

Somente suportado pelo backend do mysql (veja [seção 10.3.2](#) do manual do MySQL para detalhes).

TEST_DATABASE_NAME

Padrão: None

O nome do banco de dados a ser usado quando se roda os testes.

Se o valor padrão (None) for usado como o motor de banco de dados SQLite, os testes usarão um banco de dados em memória. Para todos os outros bancos de dados o banco de teste usará o nome 'test_' + settings.DATABASE_NAME.

Veja *Testando aplicações Django*.

TEST_RUNNER

Padrão: 'django.test.simple.run_tests'

O nome do método a ser usado para começar a suite de testes. Veja *Testando aplicações Django*.

TIME_FORMAT

Padrão: 'P' (e.g. 4 p.m.)

O formato padrão para usar em campos de tempo nas páginas de listagem do Django admin – e, possivelmente, por outras partes do sistema. Veja *formatos de string de data permitidos*.

Veja também `DATE_FORMAT`, `DATETIME_FORMAT`, `TIME_FORMAT`, `YEAR_MONTH_FORMAT` and `MONTH_DAY_FORMAT`.

TIME_ZONE

Padrão: 'America/Chicago'

Uma string representando o fuso horário para esta instalação. Veja *opções disponíveis*. (Note que a lista de escolhas disponíveis lista mais de uma na mesma linha; você vai querer usar só uma das escolhas para um dado fuso horário. Por instância, uma linha diz 'Europe/London GB GB-Eire', mas você deve usar o primeiro pedaço dela – 'Europe/London' – como sua configuração de `TIME_ZONE`.)

Note que este é o mesmo fuso horário para o qual o Django converterá todas as datas/horas – não necessariamente o fuso horário do servidor. Pore exemplo, um servidor pode servir vários sites feitos com o Django, cada um com uma configuração de fuso diferentes.

Normalmente, o Django seta a variável `os.environ['TZ']` para o fuso que você especificar na configuração `TIME_ZONE`. Deste modo, todos os seus views e models automaticamente trabalharão no fuso horário correto. Entretanto, se você está manualmente, *configurando o settings*, o Django *não* tocará na variável de ambiente `TZ`, e ele deixará para você assegurar que seu processo está rodando no ambiente correto.

Note: O Django não pode usar com segurança um fuso alternativo no ambiente Windows. Se você estiver rodando o Django no Windows, esta variável deve ser setada pra combinar com o fuso horário do sistema.

URL_VALIDATOR_USER_AGENT

Padrão: Django/<version> (<http://www.djangoproject.com/>)

A string a ser usada como cabeçalho User-Agent quando checando a existência de URLs (veja a opção `verify_exists` em *URLField*).

USE_ETAGS

Padrão: False

Um booleano que especifica a saída do cabeçalho “Etag”. Este poupa largura de banda mas diminui a performance. Isso somente é usado se `CommonMiddleware` estiver instalado (veja *Middleware*).

USE_I18N

Padrão: True

Um booleano que especifica se o sistema de internacionalização do Django deve ser habilitado. Este fornece uma forma fácil de desligá-lo, por questões de performance. Se estiver setado como `False`, o Django fará algumas otimizações para não carregar os mecanismos de internacionalização.

YEAR_MONTH_FORMAT

Padrão: 'F Y'

O formato padrão a ser usado por campos de datas nas páginas de listagens do Django admin – e, possivelmente, por outras partes do sistema – em casos onde somente o ano e mês são exibidos.

Por exemplo, quando numa página de listagem do Django admin é feita uma filtragem por detalhamento de data, o cabeçalho para um dado mês mostra o mês e o ano. Diferentes locais têm diferentes formatos. Por exemplo, Nos E.U.A costuma-se dizer “Janeiro 2006”, e noutros lugares podem dizer “2006/Janeiro.”

Veja *formato de strings de datas permitidos*. Veja também `DATE_FORMAT`, `DATETIME_FORMAT`, `TIME_FORMAT` and `MONTH_DAY_FORMAT`.

Referência de sinais embutidos

Uma lista de todos os sinais que o Django envia.

See also:

O *framework de comentários* envia um *conjuntos de sinais relacionados a ele*.

Sinais de model

O módulo `django.db.models.signals` define um conjunto de sinais enviados pelo sistema de modelos.

Warning: Muitos destes sinais são enviado por vários métodos de model como `__init__()` ou `save()` que você pode sobrescrever no seu próprio código.

Se você sobrescreve estes métodos no seu próprio model, você deve chamar os métodos da classe pai para que este sinal seja enviado.

Note também que o Django armazena manipuladores de sinais como referências fracas por padrão, então se seu manipulador for uma função local, ela poderia ser “garbage collected”. Para prevenir isso, passe `weak=False` quando você chamar o método `connect()`.

pre_init

`django.db.models.signals.pre_init`

Sempre que você instanciar um model do Django, este sinal é enviado no início do método `__init__()` do model.

Argumentos enviados com este sinal:

sender A classe de model que acabou de ter uma instância criada.

args Uma lista de argumentos posicionais passados para `__init__()`:

kwargs Um dicionário de argumentos nomeados passados para `__init__()`.

Por exemplo, o *tutorial* tem esta linha:

```
p = Poll(question="What's up?", pub_date=datetime.now())
```

Os argumentos enviados para um manipulador `pre_init` seriam:

Argumento	Valor
<code>sender</code>	<code>Poll</code> (a classe em si)
<code>args</code>	<code>[]</code> (uma lista vazia, porque não houve argumentos posicionais passados para <code>__init__</code> .)
<code>kwargs</code>	<code>{'question': 'What's up?', 'pub_date': datetime.now() }</code>

post_init

`django.db.models.signals.post_init`

Como o `pre_init`, mas este é enviado quando o método `__init__()`: finaliza.

Argumentos enviados com este sinal:

sender Como acima: a classe que acabou de ter uma instância criada.

instance A instância atual do model que acabou de ser criado.

pre_save

`django.db.models.signals.pre_save`

Este é enviado no início do método `save()`.

Argumentos enviados com este sinal:

sender A classe model.

instance A instância atual sendo salva.

post_save

`django.db.models.signals.post_save`

Como `pre_save`, mas enviado ao final do método `save()`.

Argumentos enviados com este sinal:

sender A classe model.

instance A instância atual sendo salva.

created Um booleano; `True` se o novo dado foi criado.

pre_delete

`django.db.models.signals.pre_delete`

Enviado no início do método de model `delete()`. Argumentos enviados com este sinal:

sender A classe model.

instance A instância atual sendo criada.

post_delete

`django.db.models.signals.post_delete`

Como `pre_delete`, mas enviado ao final do método `delete()`.

Argumentos enviados com este sinal:

sender A classe model.

instance A instância atual sendo criada.

Note que o objeto não estará mais no banco de dados, portanto seja muito cuidadoso com o que faz com esta instância.

class_prepared

`django.db.models.signals.class_prepared`

Enviado sempre que uma classe de model estiver sendo “preparada” – isto é, uma vez que o model tenha sido definido e registrado no sistema de model do Django. O Django usa este sinal internamente; ele geralmente não é usado em aplicações de terceiros.

Argumentos que são enviados com este sinal:

sender A classe model que acabou de ser preparada.

Gerenciamento de sinais

Sinais enviados pelo *django-admin*.

post_syncdb

`django.db.models.signals.post_syncdb`

Enviado pelo `syncdb` depois de instalar uma aplicação.

Qualquer manipulador que ouve este sinal, precisa ser escrito num lugar específico: num módulo `management` numa de suas `INSTALLED_APPS`. Se o manipulador estiver registrado em algum outro lugar eles podem não ser carregados pelo `syncdb`.

Argumentos enviados com este sinal:

sender O módulo `models` que acabou de ser instalado. Isto é, se o `syncdb` acabou de instalar uma aplicação chamada `"foo.bar.myapp"`, o `sender` será o módulo `foo.bar.myapp.models`.

app O mesmo que o `sender`.

created_models Uma lista de classes model de qualquer aplicação que o `syncdb` tenha criado até agora.

verbosity Indica a quantidade de informações que o `manage.py` imprime na tela. Veja o flag `--verbosity`` para detalhes.

Funções que ouvem o `post_syncdb` deve ser ajustar o que vão mostrar na tela baseado no valor deste argumento.

interactive Se `interactive` for `True`, ele assegura-se de abrir um prompt para usuário digitar coisas na linha de comando. Se o `interactive` for `False`, as funções que ouvem este sinal não devem tentar abrir um prompt para nada.

Por exemplo, a aplicação `django.contrib.auth` somente abre um prompt para criar um superusuário quando o `interactive` é `True`.

Sinais Request/response

Sinais enviados pelo core do framework quando processa uma requisição.

request_started

`django.core.signals.request_started`

Enviado quando o Django começar a processar uma requisição HTTP.

Argumentos enviados com este sinal:

sender A classe manipuladora – i.e. `django.core.handlers.modpython.ModPythonHandler` ou `django.core.handlers.wsgi.WsgiHandler` – que manipula a requisição.

request_finished

`django.core.signals.request_finished`

Enviado quando o Django finaliza o processamento da requisição HTTP.

Argumentos enviados com este sinal:

sender A classe manipuladora, como acima.

got_request_exception

`django.core.signals.got_request_exception`

Este sinal é enviado sempre que o Django encontra uma exceção enquanto processa uma chegada de requisição HTTP.

Argumentos enviados com este sinal:

sender A classe manipuladora, como acima.

request O objeto *HttpRequest*.

Sinais de Test

Sinais que somente são enviados quando estão *rodando testes*.

template_rendered

`django.test.signals.template_rendered`

Enviados quando o sistema de teste renderiza um template. Este sinal não é emitido durante uma operação normal do servidor do Django – ele somente está disponível durante o teste.

Argumentos enviados com este sinal:

sender O objeto *Template* que foi renderizado.

template O mesmo que o sender

context O *Context* com que o template foi renderizado.

Referência do Template

O motor de template do Django provê uma poderosa mini-linguagem para definir as camadas de sua aplicação, encorajando uma separação limpa entre lógica e apresentação. Os templates podem ser mantidos por qualquer um que tenha conhecimento de HTML; não é necessário saber Python.

Tags e filtros de template embutidos (Built-in)

Este documento descreve as tags e filtros de templates embutidos do Django. É recomendado o uso da *documentação automática*, se disponível, que irá incluir a documentação de qualquer tag ou filtro personalizado instalado.

Referência de tags nativas

autoescape

Please, see the release notes Controla o comportamento de auto-scaping atual. Esta tag tem que receber `on` ou `off` como um argumento que determina se o auto-scaping surtirá efeito dentro do bloco.

Quando o auto-scaping surte efeito, todo conteúdo de variável tem seu HTML escapado, antes do resultado ser jogado para saída (mas depois dos filtros serem aplicados). Isto equivale a você aplicar manualmente o filtro `escape` em cada variável.

A única exceção são variáveis que já estão marcadas como “safe”, que através de código foram povoadas, ou porque tem os filtros `safe` ou `escape` aplicados.

block

Define um bloco que pode ser sobrescrito por um template filho. Veja *Herança de template* para mais informações.

comment

Ignora tudo entre dois `{% comment %}` e `{% endcomment %}`

cycle

Ciclo entre dadas strings ou variáveis a cada vez que esta tag é encontrada. Dentro de um loop, ciclos entre dadas strings a cada volta dentro do loop:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' 'row2' %}">
        ...
    </tr>
{% endfor %}
```

Você pode usar variáveis, também. Por exemplo, se você tem duas variáveis de template, `rowvalue1` e `rowvalue2`, você pode montar ciclos entre seus valores desta forma:

```
{% for o in some_list %}
    <tr class="{% cycle rowvalue1 rowvalue2 %}">
        ...
    </tr>
{% endfor %}
```

Sim, você pode mesclar variáveis e strings:

```
{% for o in some_list %}
    <tr class="{% cycle 'row1' rowvalue2 'row3' %}">
        ...
    </tr>
{% endfor %}
```

Em alguns casos você pode querer referenciar o próximo valor de um ciclo de fora de um loop. Para fazer isto, é só ter uma tag `{% cycle %}`, usando “as”, desta forma:

```
{% cycle 'row1' 'row2' as rowcolors %}
```

A partir daqui, você pode inserir o valor atual de um ciclo onde você quiser dentro do seu template:

```
<tr class="{% cycle rowcolors %}">...</tr>
<tr class="{% cycle rowcolors %}">...</tr>
```

Você pode usar qualquer número de valores em uma tag `{% cycle %}`, separados por espaços. Os valores colocados dentro de aspas simples (') ou duplas (") são tratados como strings, enquanto que valores sem aspas serão tratados como variáveis de template.

Perceba que as variáveis incluídas no `cycle` não serão escapadas. Isto porque a tag de template não escapa seus conteúdos. Se você deseja escapar as variáveis no `cycle`, você deve fazê-lo explicitamente:

```
{% filter force_escape %}
    {% cycle var1 var2 var3 %}
{% endfilter %}
```

Para retro-compatibilidade, a tag `{% cycle %}` suporta a mais antiga sintaxe vinda de versões antigas do Django. Você não deve usar isto em qualquer novo projeto, porém, para saúde das pessoas que ainda estão usando-o, aqui está a forma antiga de se usar:

```
{% cycle row1,row2,row3 %}
```

Nesta sintaxe, cada valor é interpretado como uma string, e não há meios de especificar valores de variáveis. Ou vírgulas de fato. Ou espaços. Nós mencionamos que você não deve usar esta sintaxe em nenhum novo projeto?

debug

Mostra todas as informações carregadas de debug, incluindo o contexto atual e os módulos importados.

extends

Sinal que este template estende um template pai.

Esta tag pode ser usada de duas formas:

- `{% extends "base.html" %}` (sem aspas) usa o valor literal `"base.html"` como o nome do template pai a ser estendido.
- `{% extends variable %}` usa o valor da `variable`. Se a variável é uma string, o Django irá usá-la como o nome do template pai. Se a variável for um objeto `Template`, o Django irá usar o objeto como o template pai.

Veja *Template inheritance* para mais informações.

filter

Filtra os conteúdos da variável através da variável filtros.

Filtros podem também ser combinados entre eles, e eles podem ter argumentos – assim como na sintaxe de variáveis.

Exemplo de uso:

```
{% filter force_escape|lower %}
    Este texto terá o HTML escapado, e irá aparecer todo em minúsculo.
{% endfilter %}
```

firstof

Mostra a primeira variável passada que não for False, sem escape.

Não mostra nada se todas as variáveis recebidas forem False.

Exemplo de uso:

```
{% firstof var1 var2 var3 %}
```

Isto é equivalente a:

```
{% if var1 %}
    {{ var1|safe }}
{% else %}{% if var2 %}
    {{ var2|safe }}
{% else %}{% if var3 %}
    {{ var3|safe }}
{% endif %}{% endif %}{% endif %}
```

Você pode também usar uma string como um valor de segurança no caso de todas as variáveis passadas serem False:

```
{% firstof var1 var2 var3 "valor de segurança" %}
```

Note que as variáveis incluídas na tag `firstof` não serão escapadas. Isto porque as tags de template não são escapam seus conteúdos. Se você deseja escapar o conteúdo das variáveis da tag `firstof`, você deve fazê-lo explicitamente:

```
{% filter force_escape %}
    {% firstof var1 var2 var3 "fallback value" %}
{% endfilter %}
```


for

Faz um loop sobre cada item de um array. Por exemplo, para mostrar uma lista de atletas vindos de `athlete_list`:

```
<ul>
{% for athlete in athlete_list %}
    <li>{{ athlete.name }}</li>
{% endfor %}
</ul>
```

Você pode iterar a lista ao contrário usando `{% for obj in list reversed %}`. *Please, see the release notes* Se você precisa iterar uma lista de listas, você pode descompactar os valores de cada sub-lista em variáveis individuais. Por exemplo, se seu contexto contém uma lista de coordenadas (x,y) chamada `points`, você poderia usar a seguinte saída de lista de pontos:

```
{% for x, y in points %}
    Este é um ponto em {{ x }}, {{ y }}
{% endfor %}
```

Isso também pode ser útil se você precisa acessar os itens de um dicionário. Por exemplo, se seu contexto contém um dicionário `data`, o seguinte código irá mostrar as chaves e valores do dicionário:

```
{% for key, value in data.items %}
    {{ key }}: {{ value }}
{% endfor %}
```

O loop `for` cria algumas variáveis dentro do loop:

Variável	Descrição
<code>forloop.counter</code>	A iteração atual do loop (começando de 1)
<code>forloop.counter0</code>	A iteração atual do loop (começando de 0)
<code>forloop.revcounter</code>	O número de iterações começando do final do loop (começando do 1)
<code>forloop.revcounter0</code>	O número de iterações começando do final do loop (começando do 0)
<code>forloop.first</code>	True se esta é a primeira volta do loop
<code>forloop.last</code>	True se esta é a última volta do loop
<code>forloop.parentloop</code>	Para loops aninhados, este é o loop “acima” do loop atual

if

A tag `{% if %}` avalia uma variável, e se a variável for “verdadeira” (i.e. existe, não está vazia, e não tem um valor booleano falso) o conteúdo do bloco é mostrado:

```
{% if athlete_list %}
    Número de atletas: {{ athlete_list|length }}
{% else %}
    Não há atletas.
{% endif %}
```

Acima, se `athlete_list` não for vazio, o numero de atletas será mostrado pela variável `{{ athlete_list|length }}`.

Como você pode ver, a tag `if` pode ter uma clausula opcional `{% else %}` que será mostrado se o teste falhar.

As tags `if` podem usar `and`, `or` ou `not` para testar um várias variáveis ou para negá-las:

```
{% if athlete_list and coach_list %}
    Ambos, atletas e treinadores estão disponíveis.
{% endif %}

{% if not athlete_list %}
```

```

    Não há atletas.
{% endif %}

{% if athlete_list or coach_list %}
    Há alguns atletas ou alguns treinadores.
{% endif %}

{% if not athlete_list or coach_list %}
    Não á atletas ou há alguns treinadores (OK, escrever lógica
    booleana em português parece idiota. Mas não é falha nossa.
    Escreveram isso em inglês antes, hehehe).
{% endif %}

{% if athlete_list and not coach_list %}
    There are some athletes and absolutely no coaches.
    Não há qualquer atleta e absolutamente nenhum treinador.
{% endif %}

```

As tags `if` não permitem cláusulas `and` e `or` dentro da mesma tag, pois a ordem da lógica pode ser ambígua. Por exemplo, isto é inválido:

```
{% if athlete_list and coach_list or cheerleader_list %}
```

Se você precisa combinar `and` e `or` para lógica avançada, use as tags `if` aninhadas. Por exemplo:

```

{% if athlete_list %}
    {% if coach_list or cheerleader_list %}
        Nós temos atletas, e também treinadores ou líderes de torcida!
    {% endif %}
{% endif %}

```

Múltiplos usos do mesmo operador lógico são bons, desde que você use o mesmo operador. Por exemplo, isto é válido:

```
{% if athlete_list or coach_list or parent_list or teacher_list %}
```

ifchanged

Checa se o valor mudou desde a última iteração de um loop.

O tag de bloco ‘`ifchanged`’ é usada dentro de um loop. Existem duas possibilidades de uso.

1. Checa seu próprio conteúdo renderizado contra seu estado anterior e somente mostra o conteúdo se ele mudou. Por exemplo, isto mostra uma lista de dias, somente mostrando o mês se ele muda:

```

<h1>Archive for {{ year }}</h1>

{% for date in days %}
    {% ifchanged %}<h3>{{ date|date:"F" }}</h3>{% endifchanged %}
    <a href="{{ date|date:"M/d"|lower }}">{{ date|date:"j" }}</a>
{% endfor %}

```

2. Para uma determinada variável, checka e essa variável mudou. Por exemplo, a seguinte código mostra a data toda vez que ela muda, mas somente mostra a hora se ambos, hora e data, tiverem mudado:

```

{% for date in days %}
    {% ifchanged date.date %} {{ date.date }} {% endifchanged %}
    {% ifchanged date.hour date.date %}
        {{ date.hour }}
    {% endifchanged %}
{% endfor %}

```

A tag `ifchanged` pode também ter uma cláusula opcional `{% else %}` que será mostrada se o valor não mudou:

```
{% for match in matches %}
    <div style="background-color:
        {% ifchanged match.ballot_id %}
            {% cycle "red" "blue" %}
        {% else %}
            grey
        {% endifchanged %}
    ">{{ match }}</div>
```

ifequal

Mostra os conteúdos do bloco se os dois argumentos forem iguais entre si.

Exemplo:

```
{% ifequal user.id comment.user_id %}
    ...
{% endifequal %}
```

Assim como na tag `{% if %}`, uma cláusula `{% else %}` é opcional.

Os argumentos podem ser strings hard-coded, então o seguinte é válido:

```
{% ifequal user.username "adrian" %}
    ...
{% endifequal %}
```

Somente é possível comparar um argumentos que sejam variáveis de template ou strings. Você não pode checar a igualdade entre objetos Python com `True` ou `False`. Se você precisa testar se algo é verdadeiro ou falso, use a tag `if`.

ifnotequal

Exatamente como `ifequal`, exceto que testa se os dois argumentos não são iguais.

include

Carrega um template e o renderiza com o contexto atual. Este é o meio que se tem para “incluir” um template dentro do outro.

O nome do template pode ser tanto uma variável quanto uma string entre aspas, simples ou duplas.

Este exemplo inclui o conteúdo do template `"foo/bar.html"`:

```
{% include "foo/bar.html" %}
```

Este exemplo inclui o conteúdo do template cujo o nome está contido na variável `template_name`:

```
{% include template_name %}
```

Um template incluído é renderizado com o contexto do template que o incluí. Este exemplo produz uma saída `"Hello, John"`:

- Context: variável `person` é setada para `"john"`.
- Template:

```
{% include "name_snippet.html" %}
```

- O template `name_snippet.html`:

```
Hello, {{ person }}
```

Veja também: `{% ssi %}`.

load

Carrega um conjunto de tags de template customizadas.

Veja *Bibliotecas de tags e filtros customizados* para mais informações.

now

Mostra a data, formatada de acordo com a string dada.

Usa o mesmo formato da função `date()` do PHP (<http://php.net/date>) com algumas extensões customizadas.

Formatos de string disponíveis:

Caractere de formatação	Descrição
a	'a.m.' ou 'p.m.' (Note que isso é um pouco diferente de uma saída do PHP, pois ela inclui per
A	'AM' ou 'PM'.
b	Mês, textual, 3 letras, em minúsculo.
B	Não implementado.
d	Dia do mês, 2 dígitos com zeros na frente
D	Dia da semana, textual 3 letras.
f	Tempo, em 12-horas e minutos, os minutos não aparecem se forem zero. Extensão proprietária.
F	Ms, textual, longo.
g	Hora, formato 12-horas sem zero na frente zeros.
G	Hora, formato 24-horas sem zero na frente
h	Hora, formato 12-horas.
H	Hora, formato 24-horas.
i	Minutos.
I	Não implementado.
j	Dia do mês sem zero na frente.
l	Dia da semana, textual, longo.
L	Boolean para saber se é um ano bissexto.
m	Mês, 2 dígitos com zero na frente.
M	Mês, textual, 3 letras.
n	Mês sem zeros na frente.
N	Mês abreviação no estilo Associated Press Extensão proprietária.
O	Diferença para Greenwich em horas.
P	Tempo, no formato 12-horas, minutos e 'a.m.'/'p.m.', sem os minutos se forem zero e em caso espe
r	Data formatada seguindo RFC 2822.
s	Segundos, 2 dígitos com zero na frente.
S	Sufixo ordinal inglês para dia do mês, 2 caracteres.
t	Numero de dias em um dado mês.
T	Fuso-horário desta máquina.
U	Não implementado.
w	Dia da semana, dígitos sem zeros na frente.
W	Número da semana do ano segundo o padrão ISO-8601 com semanas começando na Segunda-feira
y	Ano, 2 dígitos.
Y	Ano, 4 dígitos.

Caractere de formatação	Descrição
z	Dia do ano.
Z	Compensação de fuso-horários em segundos. A compensação de fuso-horários a oeste de UTC é se

exemplo:

```
It is {% now "jS F Y H:i" %}
```

Note que você pode escapar uma string de formatação utilizando contra-barra (\), isso se você quiser usar um valor “cru”. Neste exemplo, “f” é escapado com contra-barra, pois “f” é um formato que mostra o tempo.

It is the {% now “jS of F” %}

Isto geraria uma saída “It is the 4th of September”.

regroup

Agrupar uma lista de objetos similares por um atributo comum.

Esta tag complexa é melhor ilustrada em uso num exemplo: digamos que `people` é uma lista de pessoas representadas em um dicionário com as chaves `first_name`, `last_name`, e `gender`:

```
people = [
    {'first_name': 'George', 'last_name': 'Bush', 'gender': 'Male'},
    {'first_name': 'Bill', 'last_name': 'Clinton', 'gender': 'Male'},
    {'first_name': 'Margaret', 'last_name': 'Thatcher', 'gender': 'Female'},
    {'first_name': 'Condoleezza', 'last_name': 'Rice', 'gender': 'Female'},
    {'first_name': 'Pat', 'last_name': 'Smith', 'gender': 'Unknown'},
]
```

...e você gostaria de mostrar uma lista hierarquica ordenada por gender, como esta:

- **Male:**
 - George Bush
 - Bill Clinton
- **Female:**
 - Margaret Thatcher
 - Condoleezza Rice
- **Unknown:**
 - Pat Smith

Você pode usar a tag `{% regroup %}` para agrupar a lista de pessoas por gender. O fragmento de template poderia efetuar isto:

```
{% regroup people by gender as gender_list %}

<ul>
{% for gender in gender_list %}
  <li>{{ gender.grouper }}
  <ul>
    {% for item in gender.list %}
      <li>{{ item.first_name }} {{ item.last_name }}</li>
    {% endfor %}
  </ul>
</li>
{% endfor %}
</ul>
```

Vamos examinar este exemplo. `{% regroup %}` recebe três argumentos: a lista que você quer agrupar, o atributo ao qual será agrupado, e o nome da lista resultante. Aqui, nós estamos agrupando a lista de `people` pelo atributo `gender` e chamando o resultado de `gender_list`.

`{% regroup %}` produz uma lista (neste caso, `gender_list`) de **objetos grupo**. Cada objeto grupo possui dois atributos:

- `grouper` – o item que foi agrupado por (e.g., a string “Male” ou “Female”).
- `list` – uma lista de todos os itens deste grupo (e.g., uma lista de todas as pessoas com `gender='Male'`).

Note que `{% regroup %}` não ordena sua entrada! Nosso exemplo se baseia no fato de que a lista `people` foi ordenada por `gender`, em primeiro lugar. Se a lista `people` *não* foi ordenada por `gender`, o agrupamento poderia ingenuamente gerar uma lista com mais de um grupo com um único `gender`. Por exemplo, digamos que a lista `people` foi setada desta forma (note que os *males* não foram agrupados):

```
people = [
    {'first_name': 'Bill', 'last_name': 'Clinton', 'gender': 'Male'},
    {'first_name': 'Pat', 'last_name': 'Smith', 'gender': 'Unknown'},
    {'first_name': 'Margaret', 'last_name': 'Thatcher', 'gender': 'Female'},
    {'first_name': 'George', 'last_name': 'Bush', 'gender': 'Male'},
    {'first_name': 'Condoleezza', 'last_name': 'Rice', 'gender': 'Female'},
]
```

Com esta entrada para `people`, o exemplo de código de template `{% regroup %}` acima, poderia resultar na seguinte saída:

- **Male:**
 - Bill Clinton
- **Unknown:**
 - Pat Smith
- **Female:**
 - Margaret Thatcher
- **Male:**
 - George Bush
- **Female:**
 - Condoleezza Rice

A solução mais fácil para este problema é ter certeza de que seu código no view ordenou a data de acordo com o que você deseja exibir.

Outra solução é classificar os dados no template usando o filtro `dictsort`, se seus dados estão em um dicionário:

```
{% regroup people|dictsort:"gender" by gender as gender_list %}
```

spaceless

Remove espaços em branco entre tags HTML. Isto inclui tabs e novas linhas.

exemplo de uso:

```
{% spaceless %}
<p>
  <a href="foo/">Foo</a>
</p>
{% endspaceless %}
```

Este exemplo poderia retornar este HTML:

```
<p><a href="foo/">Foo</a></p>
```

Somente espaços entre *tags* são removidos – não espaços entre tags e textos. Neste exemplo, o espaço ao redor de `Hello` não foi removido:

```
{% spaceless %}
  <strong>
    Hello
  </strong>
{% endspaceless %}
```

ssi

Mostra os conteúdos de um dado arquivo dentro da página.

Como uma simples tag “include”, `{% ssi %}` inclui o conteúdo de outro arquivo – que devem ser especificados usando um caminho absoluto – na página atual:

```
{% ssi /home/html/ljworld.com/includes/right_generic.html %}
```

Se o parâmetro opcional “parsed” for informado, o conteúdo do arquivo incluído será validado como código de template, dentro do contexto atual:

```
{% ssi /home/html/ljworld.com/includes/right_generic.html parsed %}
```

Note que se você usar `{% ssi %}`, você precisará definir `ALLOWED_INCLUDE_ROOTS` nas suas configurações do Django, como uma medida de segurança.

Veja também: `{% include %}`.

templatetag

Mostra um dos caracteres de sintaxe usados para compor tags de template.

Uma vez que o sistema de template não tem nenhum conceito de “escape”, para mostrar um pouco do que está em uso nas tags de template, você deve usar a tag `{% templatetag %}`.

O argumento diz qual parte do template deve ser mostrado:

Argumento	Saídas
<code>openblock</code>	<code>{%</code>
<code>closeblock</code>	<code>%}</code>
<code>openvariable</code>	<code>{{</code>
<code>closevariable</code>	<code>}}</code>
<code>openbrace</code>	<code>{</code>
<code>closebrace</code>	<code>}</code>
<code>opencomment</code>	<code>{#</code>
<code>closecomment</code>	<code>#}</code>

url

Retorna uma URL absoluta (i.e, uma URL sem o nome do domínio) combinando uma certa função view e parâmetros opcionais. Esta é uma forma de gerar links sem violar os princípios do DRY, ao escrever na mão as URLs nos seus templates:

```
{% url path.to.some_view arg1,arg2,name1=value1 %}
```

O primeiro argumento é um caminho para uma função view no formato `package.package.module.function`. Argumentos adicionais são opcionais e devem ser separados por vírgula, pois serão usados como argumentos posicionais e nomeados na URL. Todos os argumentos obrigatórios pelo URLConf devem estar presentes.

Por exemplo, suponha que você tem um view, `app_views.client`, cujo URLConf recebe um `client ID` (aqui, `client()` é um método dentro do arquivo view `app_views.py`). A linha do URLConf pode parecer com isso:

```
('^client/(\d+)/$', 'app_views.client')
```

Se o URLConf desta aplicação está incluído no URLConf do projeto com um caminho como este:

```
('^clients/', include('project_name.app_name.urls'))
```

...então, em um template, você pode criar um link para este view desta forma:

```
{% url app_views.client client.id %}
```

A tag do template gerará uma string `/clients/client/123/`. *Please, see the release notes* Se você está usando *padrões de URL nomeados*, você pode referenciar o nome do padrão na tag `url` ao invés de usar o caminho do view.

Note que se a URL que você está revertendo não existe, você irá receber uma exceção `NoReverseMatch`, que fará seu site mostrar uma página de erro. *Please, see the release notes* Se você gostaria de receber uma URL sem mostrá-la, você pode usar uma chamada ligeiramente diferente:

```
{% url path.to.view arg, arg2 as the_url %}

<a href="{{ the_url }}">I'm linking to {{ the_url }}</a>
```

Esta sintaxe `{% url ... as var %}` não causará um erro se o view não existir. Na prática você usará isto para linkar views que são opcionais:

```
{% url path.to.view as the_url %}
{% if the_url %}
  <a href="{{ the_url }}">Link to optional stuff</a>
{% endif %}
```

widthratio

Para criar barras de gráficos e tal, esta tag calcula a razão entre um dado valor e o máximo valor, e então aplica esta proporção a uma constante.

Por exemplo:

```

```

Acima, se `this_value` é 175 e `max_value` é 200, a imagem do exemplo acima terá 88 pixels de largura (porque $175/200 = .875$; $.875 * 100 = 87.5$ que é arredondado para cima, 88).

with

Please, see the release notes Cacheia uma variável complexa sob um simples nome. Isto é usual quando se acessa um método muito “custoso” (e.g, um que consulta o banco de dados) múltiplas vezes.

Por exemplo:

```
{% with business.employees.count as total %}
  {{ total }} employee{{ total|pluralize }}
{% endwith %}
```


A variável populada (do exemplo acima, `total`) somente está disponível entre as tags `{% with %}` e `{% endwith %}`.

Referência de filtros embutidos

add

Adiciona o argumento ao valor.

Por exemplo:

```
{{ value|add:"2" }}
```

Se `value` é 4, então a saída será 6.

addslashes

Adiciona barras antes das aspas. Usual para escapar strings em CSV, por exemplo.

capfirst

Torna a primeira letra do valor maiúscula.

center

Centraliza o valor no campo conforme uma dada largura.

cut

Remove todos os valores do argumento de uma dada string.

Por exemplo:

```
{{ value|cut:" " }}
```

Se `value` é "String with spaces", a saída será "Stringwithspaces".

date

Formata a data de acordo com o formato dado (o mesmo que a tag *now*).

Por exemplo:

```
{{ value|date:"D d M Y" }}
```

Se `value` é um objeto `datetime` (e.g., o resultado de `datetime.datetime.now()`), a saída será a string 'Wed 09 Jan 2008'.

Quando usado sem uma string de formatação:

```
{{ value|date }}
```

...a string de formatação definida na configuração `DATE_FORMAT` será utilizada.

default

Se o valor é `False`, usa o dado padrão. Do contrário, usa o valor.

Por exemplo:

```
{{ value|default:"nothing" }}
```

Se `value` é `" "` (uma string vazia), a saída será `nothing`.

default_if_none

Se (e somente se) o valor for `None`, use o dado padrão. Do contrário, use o valor.

Note que se uma string vazia for passada, o valor padrão *não* será usado. Use o filtro `default` se você quiser suportar strings vazias.

Por exemplo:

```
{{ value|default_if_none:"nothing" }}
```

Se `value` é `None`, a saída será a string `"nothing"`.

dictsort

Recebe uma lista de dicionários e retorna a lista ordenada por uma chave fornecida no argumento.

Por exemplo:

```
{{ value|dictsort:"name" }}
```

Se `value` é:

```
[
  { 'name': 'zed', 'age': 19 },
  { 'name': 'amy', 'age': 22 },
  { 'name': 'joe', 'age': 31 },
]
```

então a saída poderia ser:

```
[
  { 'name': 'amy', 'age': 22 },
  { 'name': 'joe', 'age': 31 },
  { 'name': 'zed', 'age': 19 },
]
```

dictsortreversed

Recebe uma lista de dicionários e retorna uma lista ordenada reversamente, por uma chave fornecida em um argumento. Este funciona exatamente igual ao filtro acima, mas o valor retornado será na ordem contrária.

divisibleby

Retorna `True` se o valor é divisível por um argumento.

Por exemplo:

```
{{ value|divisibleby:"3" }}
```

Se `value` é 21, a saída poderia ser `True`.

escape

Escapa strings HTML. Especificamente, fazendo substituições:

- `<` é convertido para `<`;
- `>` é convertido para `>`;
- `'` (aspas simples) é convertido para `'`;
- `"` (aspas duplas) é convertido para `"`;
- `&` é convertido para `&`;

O `escape` é somente aplicado quando a string é exibida, então não importa onde, dentro de uma sequência encadeada de filtros, você coloca `escape`: ele sempre será aplicado como se fosse o último filtro. Se você deseja que o `escape` seja aplicado imediatamente, use o filtro `force_escape`.

Aplicando `escape` para uma variável que normalmente teria aplicado um auto-escaping no seu conteúdo, somente resultará em uma rodada de `escape` a ser feita. Então é seguro usar esta função mesmo em ambientes que possuem `escape` automático. Se você deseja multiplicar os passes do `escape` a serem aplicados, use o filtro `force_escape`. Devido ao auto-escape, o comportamento deste filtro foi ligeiramente mudado. As substituições são feitas somente uma vez, depois de todos os outros filtros – incluindo os filtros antes e depois dele.

escapejs

Please, see the release notes Escapa caracteres para uso em strings de JavaScript. Ele *não* cria a string pronta para usar em HTML, mas protege você de erros de sintaxe quando são usados templates para gerar JavaScript/JSON.

filesizeformat

Formata o valor como um tamanho de arquivo ‘legível por humanos’ (i.e. `'13 KB'`, `'4.1 MB'`, `'102 bytes'`, etc).

Por exemplo:

```
{{ value|filesizeformat }}
```

Se `value` é 123456789, a saída poderia ser `117.7 MB`.

first

Retorna o primeiro item de uma lista.

Por exemplo:

```
{{ value|first }}
```

Se `value` é a lista `['a', 'b', 'c']`, a saída será `'a'`.

fix_ampersands

Este raramente é útil pois comerciais (`&`) são automaticamente escapados agora. Veja *escape* para mais informações. Substitui comercial (`&`) pela entidade `&`.

Por exemplo:

```
{{ value|fix_undersands }}
```

Se `value` é `Tom & Jerry`, a saída será `Tom & Jerry`.

floatformat

Quando usado sem um argumento, arredonda um número em ponto flutuante para uma casa decimal – mas somente se houver uma parte decimal para ser mostrada. Por exemplo:

value	Template	Saída
34.23234	{{ value floatformat }}	34.2
34.00000	{{ value floatformat }}	34
34.26000	{{ value floatformat }}	34.3

Se usado com um argumento numérico inteiro, `floatformat` arredonda um número para várias casas decimais. Por exemplo:

value	Template	Saída
34.23234	{{ value floatformat:3 }}	34.232
34.00000	{{ value floatformat:3 }}	34.000
34.26000	{{ value floatformat:3 }}	34.260

Se o argumento passado para `floatformat` é negativo, ele irá arredondar o número para muitas casas decimais – mas comente se houver uma parte decimal a ser mostrada. Por exemplo:

value	Template	Saída
34.23234	{{ value floatformat:"-3" }}	34.232
34.00000	{{ value floatformat:"-3" }}	34
34.26000	{{ value floatformat:"-3" }}	34.260

Usando `floatformat` sem argumento é equivalente a usar `floatformat` com um argumento `-1`.

force_escape

Please, see the release notes Escapa o HTML para uma string (veja o filtro `escape` para detalhes). Este filtro é aplicado *imediatamente* e retorna uma nova, string escapada. Ele é usual em raros casos onde você precisa dar múltiplos escapes ou quer aplicar outros filtros sobre resultados escapados. Normalmente, você quer usar o filtro `escape`.

get_digit

Dado um número inteiro, retorna o dígito requisitado, onde 1 é o dígito mais a direita, 2 é o segundo mais a direita, etc. Retorna o valor original no caso de entradas inválidas (se a entrada ou argumento não for um inteiro, ou se o argumento é menor que 1). Todo caso, a saída será sempre um inteiro.

Por exemplo:

```
{{ value|get_digit:"2" }}
```

Se `value` é `123456789`, a saída será `8`.

iriencode

Converte uma IRI (Internationalized Resource Identifier) para uma string que é adequada para uma URL. Isto é necessário se você está tentando usar strings que possuem caracteres não ASCII na URL.

É seguro usar este filtro em uma string que também passou pelo filtro `urlencode`.

join

Junta uma lista em uma string, como o `str.join(list)` do Python.

Por exemplo:

```
{{ value|join:" // " }}
```

Se `value` é a lista `['a', 'b', 'c']`, a saída será a string `"a // b // c"`.

last

Please, see the release notes Retorna o último item de uma lista.

Por exemplo:

```
{{ value|last }}
```

Se `value` é a lista `['a', 'b', 'c', 'd']`, a saída será a string `"d"`.

length

Retorna o comprimento de um valor. Ela funciona para ambos strings e listas.

Por exemplo:

```
{{ value|length }}
```

Se `value` é `['a', 'b', 'c', 'd']`, a saída será `4`.

length_is

Retorna `True` se o valor do comprimento é o argumento, ou `False` caso contrário.

Por exemplo:

```
{{ value|length_is:"4" }}
```

Se `value` é `['a', 'b', 'c', 'd']`, a saída será `True`.

linebreaks

Substitui quebras de linha em textos planos pelo HTML apropriado; uma simples quebra de linha se torna uma quebra de linha do HTML (`
`) e uma nova linha seguida por uma linha em branco se tornará um parágrafo (`<p>`).

Por exemplo:

```
{{ value|linebreaks }}
```

Se `value` é `Joel\nis a slug`, a saída será `<p>Joel
is a slug</p>`.

linebreaksbr

Converte todas as novas linhas em uma amostra de texto plano para quebras de linhas HTML (`
`).

linenumbers

Mostra o texto com número de linhas.

ljust

Alinha a esquerda o valor no campo de uma dada largura.

Argumento: tamanho do campo

lower

Converte uma string em minúsculo.

Por exemplo:

```
{{ value|lower }}
```

Se `value` é `Still MAD At Yoko`, a saída será `still mad at yoko`.

make_list

Retorna o valor em uma lista. Para um inteiro, será uma lista de dígitos. Para uma string, será uma lista de caracteres.

Por exemplo:

```
{{ value|make_list }}
```

Se `value` é a string `"Joel"`, a saída poderá ser a lista `[u'J', u'o', u'e', u'l']`. Se `value` é `123`, a saída será a lista `[1, 2, 3]`.

phone2numeric

Converte um número telefonico (possivelmente contendo letras) para seu numeral equivalente. Por exemplo, `'800-COLLECT'` será convertido para `'800-2655328'`.

A entrada não tem de ser um número de telefone válido. Isto irá converter alegremente qualquer sequência.

pluralize

Retorna o sufixo plural se o valor não é 1. Por padrão, este sufixo é `'s'`.

Exemplo:

```
You have {{ num_messages }} message{{ num_messages|pluralize }}.
```

Para palavras que requerem um outro sufixo que não `'s'`, você pode prover um sufixo alternativo como um parâmetro para o filtro.

Exemplo:

```
You have {{ num_walruses }} walrus{{ num_walruses|pluralize:"es" }}.
```

Para palavras que não são pluralizadas por sufixos simples, você pode especificar ambos sufixos, singular e plural, separados por vírgula.

Exemplo:

```
You have {{ num_cherries }} cherr{{ num_cherries|pluralize:"y,ies" }}.
```

pprint

Um contorno para `pprint.pprint` – para debugging, realmente.

random

Retorna um item randômico de uma lista.

Por exemplo:

```
{{ value|random }}
```

Se `value` é a lista `['a', 'b', 'c', 'd']`, a saída poderá ser `"b"`.

removetags

Remove uma lista de tags [X]HTML separadas por espaços da saída.

Por exemplo:

```
{{ value|removetags:"b span"|safe }}
```

Se `value` é `"Joel <button>is</button> a slug"` a saída será `"Joel <button>is</button> a slug"`.

rjust

Alinha a direita o valor no campo de uma dada largura.

Argumento: tamanho do campo

safe

Marca uma string que não exige escape de HTML antes da saída. Quando autoscaping está desligado, este filtro não surte efeito.

slice

Retorna um pedaço da lista.

Usa a mesma sintaxe do fatiamento de listas do Python. Veja http://diveintopython.org/native_data_types/lists.html#odbchelper.list.slice para uma introdução

Exemplo:

```
{{ some_list|slice:"2" }}
```

slugify

Converts to lowercase, removes non-word characters (alphanumerics and underscores) and converts spaces to hyphens. Also strips leading and trailing whitespace.

Por exemplo:

```
{{ value|slugify }}
```

Se `value` é "Joel is a slug", a saída será "joel-is-a-slug".

stringformat

Formata a variável de acordo com o argumento, um especificador de formatação de strings. Este especificador usa a sintaxe de formatação de strings do Python, com a exceção de não usar o guia "%".

Veja See <http://docs.python.org/library/stdtypes.html#string-formatting-operations> para documentação de formatação de strings do Python.

Por exemplo:

```
{{ value|stringformat:"s" }}
```

Se `value` é "Joel is a slug", a saída será "Joel is a slug".

striptags

Mostra todas as tags [X]HTML.

Por exemplo:

```
{{ value|striptags }}
```

Se `value` é "Joel <button>is</button> a slug", a saída será "Joel is a slug".

time

Formata um tempo de acordo com o formato dado (o mesmo que a tag *now*). O filtro `time` somente aceitará parâmetros no formato de string relacionados a hora do dia, não a data (por razões óbvias). Se você precisa formatar uma data, use o filtro *date*.

Por exemplo:

```
{{ value|time:"H:i" }}
```

Se `value` é equivalente a `datetime.datetime.now()`, a saída será a string "01:23".

When used without a format string:

```
{{ value|time }}
```

...the formatting string defined in the `TIME_FORMAT` setting will be used.

timesince

Formata a data como o tempo desde essa data (e.g., "4 days, 6 hours").

Recebe um argumento opcional que é uma variável contendo a data para ser usada como ponto de comparação (sem o argumento, o ponto de comparação é *now*). Por exemplo, se `blog_date` é uma instancia de `date` representando meia-noite em 1 de Junho de 2006, e `comment_date` é uma instancia de `date` para 08:00 em 1 Junho de 2006, então `{{ blog_date|timesince:comment_date }}` retornaria “8 hours”.

Comparando datetimes offset-naive e offset-aware irão retornar uma string vazia.

Minutos é a menor unidade usada, e “0 minutes” será retornado por qualquer `date` que está num futuro relativo ao ponto de comparação.

timeuntil

Similar ao `timesince`, exceto que ele mede o tempo de agora até a data ou `datetime` dada. Por exemplo, se hoje é 1 June 2006 e `conference_date` é uma instância de `date` marcando 29 June 2006, então `{{ conference_date|timeuntil }}` retornará “4 semanas”.

Recebe um argumento opcional que é uma variável contendo a data a ser usada como o ponto de comparação (ao invés de *now*). Se `from_date` contém 22 June 2006, então `{{ conference_date|timeuntil:from_date }}` retornará “1 week”.

Comparing offset-naive and offset-aware datetimes will return an empty string.

Minutos são a menor unidade usada, e “0 minutos” será retornado por qualquer data que estiver num passado relativo ao ponto de comparação.

title

Converte uma string em titlecase.

truncatewords

Trunca uma string depois de um certo número de palavras.

Argumento: Número de palavras para trancar depois.

Por exemplo:

```
{{ value|truncatewords:2 }}
```

Se `value` é “Joel is a slug”, a saída será “Joel is ...”.

truncatewords_html

Semelhante ao `truncatewords`, exceto que ele se preocupa com as tags HTML. Qualquer tag que estiver aberta na string, e não foi fechada antes do ponto de corte, serão fechadas imediatamente após o truncament.

Isto é menos eficiente que `truncatewords`, então deve ser usado somente quando estiver sendo passado textos HTML.

unordered_list

Recursivamente recebe uma lista auto-aninhada e retorna uma lista HTML não-ordenada – SEM abrir e fechar tags ``. O formato aceito por `unordered_list` mudou para facilitar o entendimento. A lista é para ser assumida no formato correto. Por exemplo, se `var` contém `['States', ['Kansas', ['Lawrence', 'Topeka']], 'Illinois']`, então `{{ var|unordered_list }}` retornaria:

```
<li>States
<ul>
    <li>Kansas
    <ul>
        <li>Lawrence</li>
        <li>Topeka</li>
    </ul>
    </li>
    <li>Illinois</li>
</ul>
</li>
```

Nota: o formato anterior mais restritivo e verboso ainda é suportado: `['States', [['Kansas', [['Lawrence', []], ['Topeka', []]]], ['Illinois', []]]],`

upper

Converte uma string em maiúsculo.

Por exemplo:

```
{{ value|upper }}
```

Se `value` é `"Joel is a slug"`, a saída será `"JOEL IS A SLUG"`.

urlencode

Escapa um valor para ser usado em uma URL.

urlize

Converte URLs em texto plano dentro de links clicáveis.

Note que se `urlize` é aplicado em um texto que já contém marcação HTML, as coisas não irão funcionar como esperado. Aplique este filtro somente no texto *plano*.

Por exemplo:

```
{{ value|urlize }}
```

Se `value` é `"Check out www.djangoproject.com"`, a saída será `"Check out www.djangoproject.com"`.

urlizetrunc

Converte URLs dentro de links clicáveis, truncando URLs mais longas que um determinado limite de caracteres.

Como em `urlize`, este filtro deve ser somente aplicado em texto *plano*.

Argumento: Comprimento que as URLs devem ter

Por exemplo:

```
{{ value|urlizetrunc:15 }}
```

Se `value` é `"Check out www.djangoproject.com"`, a saída seria `"Check out www.djangopr..."`.

wordcount

Retorna o número de palavras.

wordwrap

Quebra palavras em um comprimento de linha específico.

Argumento: número de caracteres em que ocorrerá a quebra do texto

Por exemplo:

```
{{ value|wordwrap:5 }}
```

Se value é Joel is a slug, a saída seria:

```
Joel
is a
slug
```

yesno

Dada uma string de mapeamento de valores true, false e (opcionalmente) None, retorna uma destas strings, de acordo com o valor:

Valor	Argumento	Saída
True	"yeah, no, maybe"	yeah
False	"yeah, no, maybe"	no
None	"yeah, no, maybe"	maybe
None	"yeah, no"	"no" (converte None para False se nenhum mapeamento para None for dado)

Outras bibliotecas de tags e filtros

O Django vem com algumas outras bibliotecas de tags de templates, que você tem de habilitar explicitamente em seu `INSTALLED_APPS` e habilitar em seu template com a tag `{% load %}`.

django.contrib.humanize

Um conjunto de filtros de templates Django usuais para adicionar um “toque humano” aos dados. Veja [*django.contrib.humanize*](#).

django.contrib.markup

Uma coleção de filtros de templates que implementam estas linguagens de marcação comuns:

- Textile
- Markdown
- ReST (ReStructured Text)

Veja [*markup*](#).

django.contrib.webdesign

Uma coleção de tags de template que podem ser úteis quando se faz o design de um website, como um gerador de texto Lorem Ipsum. Veja [django.contrib.webdesign](#).

The Django template language: For Python programmers

This document explains the Django template system from a technical perspective – how it works and how to extend it. If you’re just looking for reference on the language syntax, see [The Django template language](#).

If you’re looking to use the Django template system as part of another application – i.e., without the rest of the framework – make sure to read the [configuration](#) section later in this document.

Basics

A **template** is a text document, or a normal Python string, that is marked-up using the Django template language. A template can contain **block tags** or **variables**.

A **block tag** is a symbol within a template that does something.

This definition is deliberately vague. For example, a block tag can output content, serve as a control structure (an “if” statement or “for” loop), grab content from a database or enable access to other template tags.

Block tags are surrounded by "{%" and "%}".

Example template with block tags:

```
{% if is_logged_in %}Thanks for logging in!{% else %}Please log in.{% endif %}
```

A **variable** is a symbol within a template that outputs a value.

Variable tags are surrounded by "{{" and "}}".

Example template with variables:

```
My first name is {{ first_name }}. My last name is {{ last_name }}.
```

A **context** is a “variable name” -> “variable value” mapping that is passed to a template.

A template **renders** a context by replacing the variable “holes” with values from the context and executing all block tags.

Using the template system

Using the template system in Python is a two-step process:

- First, you compile the raw template code into a `Template` object.
- Then, you call the `render()` method of the `Template` object with a given context.

Compiling a string

The easiest way to create a `Template` object is by instantiating it directly. The class lives at `django.template.Template`. The constructor takes one argument – the raw template code:

```
>>> from django.template import Template
>>> t = Template("My name is {{ my_name }}.")
>>> print t
<django.template.Template instance>
```

Behind the scenes

The system only parses your raw template code once – when you create the `Template` object. From then on, it's stored internally as a “node” structure for performance.

Even the parsing itself is quite fast. Most of the parsing happens via a single call to a single, short, regular expression.

Rendering a context

Once you have a compiled `Template` object, you can render a context – or multiple contexts – with it. The `Context` class lives at `django.template.Context`, and the constructor takes one (optional) argument: a dictionary mapping variable names to variable values. Call the `Template` object's `render()` method with the context to “fill” the template:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ my_name }}.")

>>> c = Context({"my_name": "Adrian"})
>>> t.render(c)
"My name is Adrian."

>>> c = Context({"my_name": "Dolores"})
>>> t.render(c)
"My name is Dolores."
```

Variable names must consist of any letter (A-Z), any digit (0-9), an underscore or a dot.

Dots have a special meaning in template rendering. A dot in a variable name signifies **lookup**. Specifically, when the template system encounters a dot in a variable name, it tries the following lookups, in this order:

- Dictionary lookup. Example: `foo["bar"]`
- Attribute lookup. Example: `foo.bar`
- Method call. Example: `foo.bar()`
- List-index lookup. Example: `foo[bar]`

The template system uses the first lookup type that works. It's short-circuit logic.

Here are a few examples:

```
>>> from django.template import Context, Template
>>> t = Template("My name is {{ person.first_name }}.")
>>> d = {"person": {"first_name": "Joe", "last_name": "Johnson"}}
>>> t.render(Context(d))
"My name is Joe."

>>> class PersonClass: pass
>>> p = PersonClass()
>>> p.first_name = "Ron"
>>> p.last_name = "Nasty"
>>> t.render(Context({"person": p}))
"My name is Ron."

>>> class PersonClass2:
...     def first_name(self):
...         return "Samantha"
>>> p = PersonClass2()
>>> t.render(Context({"person": p}))
"My name is Samantha."
```

```
>>> t = Template("The first stooge in the list is {{ stooges.0 }}.")
>>> c = Context({"stooges": ["Larry", "Curly", "Moe"]})
>>> t.render(c)
"The first stooge in the list is Larry."
```

Method lookups are slightly more complex than the other lookup types. Here are some things to keep in mind:

- If, during the method lookup, a method raises an exception, the exception will be propagated, unless the exception has an attribute `silent_variable_failure` whose value is `True`. If the exception *does* have a `silent_variable_failure` attribute, the variable will render as an empty string. Example:

```
>>> t = Template("My name is {{ person.first_name }}.")
>>> class PersonClass3:
...     def first_name(self):
...         raise AssertionError, "foo"
>>> p = PersonClass3()
>>> t.render(Context({"person": p}))
Traceback (most recent call last):
...
AssertionError: foo

>>> class SilentAssertionError(Exception):
...     silent_variable_failure = True
>>> class PersonClass4:
...     def first_name(self):
...         raise SilentAssertionError
>>> p = PersonClass4()
>>> t.render(Context({"person": p}))
"My name is ."
```

Note that `django.core.exceptions.ObjectDoesNotExist`, which is the base class for all Django database API `DoesNotExist` exceptions, has `silent_variable_failure = True`. So if you're using Django templates with Django model objects, any `DoesNotExist` exception will fail silently.

- A method call will only work if the method has no required arguments. Otherwise, the system will move to the next lookup type (list-index lookup).
- Obviously, some methods have side effects, and it'd be either foolish or a security hole to allow the template system to access them.

A good example is the `delete()` method on each Django model object. The template system shouldn't be allowed to do something like this:

```
I will now delete this valuable data. {{ data.delete }}
```

To prevent this, set a function attribute `alters_data` on the method. The template system won't execute a method if the method has `alters_data=True` set. The dynamically-generated `delete()` and `save()` methods on Django model objects get `alters_data=True` automatically. Example:

```
def sensitive_function(self):
    self.database_record.delete()
sensitive_function.alters_data = True
```

How invalid variables are handled

Generally, if a variable doesn't exist, the template system inserts the value of the `TEMPLATE_STRING_IF_INVALID` setting, which is set to `' '` (the empty string) by default.

Filters that are applied to an invalid variable will only be applied if `TEMPLATE_STRING_IF_INVALID` is set to `' '` (the empty string). If `TEMPLATE_STRING_IF_INVALID` is set to any other value, variable filters will be ignored.

This behavior is slightly different for the `if`, `for` and `regroup` template tags. If an invalid variable is provided to one of these template tags, the variable will be interpreted as `None`. Filters are always applied to invalid variables within these template tags.

If `TEMPLATE_STRING_IF_INVALID` contains a `'%s'`, the format marker will be replaced with the name of the invalid variable.

For debug purposes only!

While `TEMPLATE_STRING_IF_INVALID` can be a useful debugging tool, it is a bad idea to turn it on as a ‘development default’.

Many templates, including those in the Admin site, rely upon the silence of the template system when a non-existent variable is encountered. If you assign a value other than `' '` to `TEMPLATE_STRING_IF_INVALID`, you will experience rendering problems with these templates and sites.

Generally, `TEMPLATE_STRING_IF_INVALID` should only be enabled in order to debug a specific template problem, then cleared once debugging is complete.

Playing with Context objects

Most of the time, you’ll instantiate `Context` objects by passing in a fully-populated dictionary to `Context()`. But you can add and delete items from a `Context` object once it’s been instantiated, too, using standard dictionary syntax:

```
>>> c = Context({"foo": "bar"})
>>> c['foo']
'bar'
>>> del c['foo']
>>> c['foo']
''
>>> c['newvariable'] = 'hello'
>>> c['newvariable']
'hello'
```

A `Context` object is a stack. That is, you can `push()` and `pop()` it. If you `pop()` too much, it’ll raise `django.template.ContextPopException`:

```
>>> c = Context()
>>> c['foo'] = 'first level'
>>> c.push()
>>> c['foo'] = 'second level'
>>> c['foo']
'second level'
>>> c.pop()
>>> c['foo']
'first level'
>>> c['foo'] = 'overwritten'
>>> c['foo']
'overwritten'
>>> c.pop()
Traceback (most recent call last):
...
django.template.ContextPopException
```

Using a `Context` as a stack comes in handy in some custom template tags, as you’ll see below.

Subclassing Context: RequestContext

Django comes with a special Context class, `django.template.RequestContext`, that acts slightly differently than the normal `django.template.Context`. The first difference is that it takes an *HttpRequest* as its first argument. For example:

```
c = RequestContext(request, {
    'foo': 'bar',
})
```

The second difference is that it automatically populates the context with a few variables, according to your `TEMPLATE_CONTEXT_PROCESSORS` setting.

The `TEMPLATE_CONTEXT_PROCESSORS` setting is a tuple of callables – called **context processors** – that take a request object as their argument and return a dictionary of items to be merged into the context. By default, `TEMPLATE_CONTEXT_PROCESSORS` is set to:

```
("django.core.context_processors.auth",
"django.core.context_processors.debug",
"django.core.context_processors.i18n",
"django.core.context_processors.media")
```

Each processor is applied in order. That means, if one processor adds a variable to the context and a second processor adds a variable with the same name, the second will override the first. The default processors are explained below.

Also, you can give `RequestContext` a list of additional processors, using the optional, third positional argument, `processors`. In this example, the `RequestContext` instance gets a `ip_address` variable:

```
def ip_address_processor(request):
    return {'ip_address': request.META['REMOTE_ADDR']}

def some_view(request):
    # ...
    c = RequestContext(request, {
        'foo': 'bar',
    }, [ip_address_processor])
    return HttpResponse(t.render(c))
```

Note: If you're using Django's `render_to_response()` shortcut to populate a template with the contents of a dictionary, your template will be passed a `Context` instance by default (not a `RequestContext`). To use a `RequestContext` in your template rendering, pass an optional third argument to `render_to_response()`: a `RequestContext` instance. Your code might look like this:

```
def some_view(request):
    # ...
    return render_to_response('my_template.html',
                              my_data_dictionary,
                              context_instance=RequestContext(request))
```

Here's what each of the default processors does:

django.core.context_processors.auth

If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these three variables:

- `user` – An `auth.User` instance representing the currently logged-in user (or an `AnonymousUser` instance, if the client isn't logged in).

- `messages` – A list of messages (as strings) for the currently logged-in user. Behind the scenes, this calls `request.user.get_and_delete_messages()` for every request. That method collects the user's messages and deletes them from the database.

Note that messages are set with `user.message_set.create`.

- `perms` – An instance of `django.core.context_processors.PermWrapper`, representing the permissions that the currently logged-in user has.

`django.core.context_processors.debug`

If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these two variables – but only if your `DEBUG` setting is set to `True` and the request's IP address (`request.META['REMOTE_ADDR']`) is in the `INTERNAL_IPS` setting:

- `debug` – `True`. You can use this in templates to test whether you're in `DEBUG` mode.
- `sql_queries` – A list of `{ 'sql': ..., 'time': ... }` dictionaries, representing every SQL query that has happened so far during the request and how long it took. The list is in order by query.

`django.core.context_processors.i18n`

If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain these two variables:

- `LANGUAGES` – The value of the `LANGUAGES` setting.
- `LANGUAGE_CODE` – `request.LANGUAGE_CODE`, if it exists. Otherwise, the value of the `LANGUAGE_CODE` setting.

See *Internacionalização* for more.

`django.core.context_processors.media`

Please, see the release notes If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain a variable `MEDIA_URL`, providing the value of the `MEDIA_URL` setting.

`django.core.context_processors.request`

If `TEMPLATE_CONTEXT_PROCESSORS` contains this processor, every `RequestContext` will contain a variable `request`, which is the current `HttpRequest`. Note that this processor is not enabled by default; you'll have to activate it.

Writing your own context processors

A context processor has a very simple interface: It's just a Python function that takes one argument, an `HttpRequest` object, and returns a dictionary that gets added to the template context. Each context processor *must* return a dictionary.

Custom context processors can live anywhere in your code base. All Django cares about is that your custom context processors are pointed-to by your `TEMPLATE_CONTEXT_PROCESSORS` setting.

Loading templates

Generally, you'll store templates in files on your filesystem rather than using the low-level `Template` API yourself. Save templates in a directory specified as a **template directory**.

Django searches for template directories in a number of places, depending on your template-loader settings (see “Loader types” below), but the most basic way of specifying template directories is by using the `TEMPLATE_DIRS` setting.

The `TEMPLATE_DIRS` setting

Tell Django what your template directories are by using the `TEMPLATE_DIRS` setting in your settings file. This should be set to a list or tuple of strings that contain full paths to your template directory(ies). Example:

```
TEMPLATE_DIRS = (
    "/home/html/templates/lawrence.com",
    "/home/html/templates/default",
)
```

Your templates can go anywhere you want, as long as the directories and templates are readable by the Web server. They can have any extension you want, such as `.html` or `.txt`, or they can have no extension at all.

Note that these paths should use Unix-style forward slashes, even on Windows.

The Python API

Django has two ways to load templates from files:

`django.template.loader.get_template(template_name)` `get_template` returns the compiled template (a `Template` object) for the template with the given name. If the template doesn't exist, it raises `django.template.TemplateDoesNotExist`.

`django.template.loader.select_template(template_name_list)` `select_template` is just like `get_template`, except it takes a list of template names. Of the list, it returns the first template that exists.

For example, if you call `get_template('story_detail.html')` and have the above `TEMPLATE_DIRS` setting, here are the files Django will look for, in order:

- `/home/html/templates/lawrence.com/story_detail.html`
- `/home/html/templates/default/story_detail.html`

If you call `select_template(['story_253_detail.html', 'story_detail.html'])`, here's what Django will look for:

- `/home/html/templates/lawrence.com/story_253_detail.html`
- `/home/html/templates/default/story_253_detail.html`
- `/home/html/templates/lawrence.com/story_detail.html`
- `/home/html/templates/default/story_detail.html`

When Django finds a template that exists, it stops looking.

Tip

You can use `select_template()` for super-flexible “templatability.” For example, if you've written a news story and want some stories to have custom templates, use something like `select_template(['story_%s_detail.html' % story.id, 'story_detail.html'])`.

That'll allow you to use a custom template for an individual story, with a fallback template for stories that don't have custom templates.

Using subdirectories

It's possible – and preferable – to organize templates in subdirectories of the template directory. The convention is to make a subdirectory for each Django app, with subdirectories within those subdirectories as needed.

Do this for your own sanity. Storing all templates in the root level of a single directory gets messy.

To load a template that's within a subdirectory, just use a slash, like so:

```
get_template('news/story_detail.html')
```

Using the same `TEMPLATE_DIRS` setting from above, this example `get_template()` call will attempt to load the following templates:

- `/home/html/templates/lawrence.com/news/story_detail.html`
- `/home/html/templates/default/news/story_detail.html`

Loader types

By default, Django uses a filesystem-based template loader, but Django comes with a few other template loaders, which know how to load templates from other sources.

Some of these other loaders are disabled by default, but you can activate them by editing your `TEMPLATE_LOADERS` setting. `TEMPLATE_LOADERS` should be a tuple of strings, where each string represents a template loader. Here are the template loaders that come with Django:

`django.template.loaders.filesystem.load_template_source` Loads templates from the filesystem, according to `TEMPLATE_DIRS`. This loader is enabled by default.

`django.template.loaders.app_directories.load_template_source` Loads templates from Django apps on the filesystem. For each app in `INSTALLED_APPS`, the loader looks for a `templates` subdirectory. If the directory exists, Django looks for templates in there.

This means you can store templates with your individual apps. This also makes it easy to distribute Django apps with default templates.

For example, for this setting:

```
INSTALLED_APPS = ('myproject.polls', 'myproject.music')
```

...then `get_template('foo.html')` will look for templates in these directories, in this order:

- `/path/to/myproject/polls/templates/foo.html`
- `/path/to/myproject/music/templates/foo.html`

Note that the loader performs an optimization when it is first imported: It caches a list of which `INSTALLED_APPS` packages have a `templates` subdirectory.

This loader is enabled by default.

`django.template.loaders.eggs.load_template_source` Just like `app_directories` above, but it loads templates from Python eggs rather than from the filesystem.

This loader is disabled by default.

Django uses the template loaders in order according to the `TEMPLATE_LOADERS` setting. It uses each loader until a loader finds a match.

The `render_to_string()` shortcut

To cut down on the repetitive nature of loading and rendering templates, Django provides a shortcut function which largely automates the process: `render_to_string()` in `django.template.loader`, which loads a template, renders it and returns the resulting string:

```
from django.template.loader import render_to_string
rendered = render_to_string('my_template.html', { 'foo': 'bar' })
```

The `render_to_string` shortcut takes one required argument – `template_name`, which should be the name of the template to load and render – and two optional arguments:

dictionary A dictionary to be used as variables and values for the template’s context. This can also be passed as the second positional argument.

context_instance An instance of `Context` or a subclass (e.g., an instance of `RequestContext`) to use as the template’s context. This can also be passed as the third positional argument.

See also the `render_to_response()` shortcut, which calls `render_to_string` and feeds the result into an `HttpResponse` suitable for returning directly from a view.

Configuring the template system in standalone mode

Note: This section is only of interest to people trying to use the template system as an output component in another application. If you’re using the template system as part of a Django application, nothing here applies to you.

Normally, Django will load all the configuration information it needs from its own default configuration file, combined with the settings in the module given in the `DJANGO_SETTINGS_MODULE` environment variable. But if you’re using the template system independently of the rest of Django, the environment variable approach isn’t very convenient, because you probably want to configure the template system in line with the rest of your application rather than dealing with settings files and pointing to them via environment variables.

To solve this problem, you need to use the manual configuration option described in *Usando o settings sem a configuração `DJANGO_SETTINGS_MODULE`*. Simply import the appropriate pieces of the templating system and then, *before* you call any of the templating functions, call `django.conf.settings.configure()` with any settings you wish to specify. You might want to consider setting at least `TEMPLATE_DIRS` (if you’re going to use template loaders), `DEFAULT_CHARSET` (although the default of `utf-8` is probably fine) and `TEMPLATE_DEBUG`. All available settings are described in the *settings documentation*, and any setting starting with `TEMPLATE_` is of obvious interest.

See also:

Para informações de como criar suas próprias tags e filters, veja *Tags e filtros de template personalizados*.

Dados Unicode no Django

Please, see the release notes O Django nativamente suporta dados Unicode por toda parte. Proporcionando ao seu banco de dados uma forma de armazenar os dados, e dando-lhe a segurança de passar strings Unicode aos templates, models e pro banco de dados.

Este documento lhe conta o que você precisa saber se você estiver escrevendo aplicações que usam dados ou templates que estão codificados em algum outro formato ASCII.

Criando o banco de dados

Esteja certo que seu banco de dados está configurado para armazenar strings arbitrárias. Normalmente, isto significa tê-las numa codificação UTF-8 ou UTF-16. Se você usa mais de uma codificação restritiva – por exemplo, latin1 (iso8859-1) – você não poderá armazenar certos caracteres no banco de dados, e informações serão perdidas.

- Usuários do MySQL, consultem o [manual do MySQL](#) (seção 10.3.2 para MySQL 5.1) para detalhes sobre como setar ou alterar a codificação do banco de dados.
- Usuários de PostgreSQL, consultem o [manual do PostgreSQL](#) (seção 21.2.2 no PostgreSQL 8) para detalhes sobre criar bancos de dados com a codificação correta.
- Usuários de SQLite, não há nada que você precise fazer. O SQLite sempre usa o UTF-8 para codificação interna.

Todos os backends de banco de dados do Django automaticamente convertem strings Unicode para o formato apropriado do banco de dados. Eles também convertem automaticamente strings recebidas de bancos de dados em strings Unicode do Python. Você não precisa dizer ao Django qual codificação seu banco de dados usa: isso é manipulado transparentemente.

Par mais, veja a seção “API de banco de dados” abaixo.

Manipulação genérica de string

Sempre que você usa strings com o Django – e.g., no banco de dados, renderização de templates ou algo mais – você tem duas escolhas de codificação de strings. Você pode usar strings Unicode, ou você pode usar strings normais (algumas chamadas “bytestrings”) que são codificadas usando UTF-8.

Warning

Uma `bytestring` não carrega qualquer informação com ela sobre sua codificação. Por esta razão, nós temos que fazer uma suposição, e o Django assume que toda `bytestring` é UTF-8.

Se você passar uma `string` para o Django que já foi codificada em algum outro formato, as coisas podem ir por um caminho errado e de formas interessantes. Normalmente, o Django lançará um erro `UnicodeDecodeError` neste ponto.

Se seu código somente usa dados ASCII, ele é seguro para usar `strings` normais, passando-as a vontade, porque o ASCII é sub-conjunto do UTF-8.

Não se engane em pensar que se sua configuração `DEFAULT_CHARSET` é qualquer coisa diferente de `'utf-8'` que você pode usar nas suas `bytestrings`! `DEFAULT_CHARSET` somente se aplica a `strings` geradas como resultado de renderização de templates (e e-mail). O Django sempre assumirá a codificação UTF-8 para `bytestrings` internas. A razão disso é que na verdade a configuração `DEFAULT_CHARSET` não está sob seu controle (se você é um desenvolvedor de aplicações). Está sob controle da pessoa que instala e usa sua aplicação – e se essa pessoa escolher uma configuração diferente, seu código pode ainda continuar funcionando. Portanto, ela não pode contar com essa configuração.

Na maioria dos casos quando o Django está lidando com `strings`, ele as converterá para Unicode antes de fazer algo mais. Então, como uma regra geral, se você passa uma `bytestring`, esteja preparado para receber uma `string` Unicode de volta no resultado.

Strings traduzidas

Além das `string` Unicode e `bytestrings`, há um terceiro tipo de objeto do tipo `string` que você pode encontrar enquanto usa o Django. As funcionalidades do framework de internacionalização introduzem o conceito de uma “tradução lazy” – uma `string` que foi marcada como traduzida mas cujo resultado atual da tradução não é determinado até que o objeto seja utilizado numa `string`. Esta funcionalidade é útil em casos onde a localização da tradução é desconhecida até que a `string` seja usada, mesmo pensando que a `string` possa ter sido originalmente criada quando o código foi importado pela primeira vez.

Normalmente, você não terá de se preocupar com traduções lazy. Somente esteja alerta que se você examinar um objeto e ele afirma ser um objeto `django.utils.functional.__proxy__`, ele é uma tradução lazy. Chamando `unicode()` com a tradução lazy como argumento gerará uma `string` Unicode na localização atual.

Para mais detalhes sobre traduções tardias de objetos, leia a documentação de [internacionalização](#).

Utilitário de funções úteis

Por algumas operações de `string` serem muito repetitivas, o Django vem acompanhado de funções úteis que devem fazer o trabalho com objetos Unicode e `bytestring` ficar um pouco mais fácil.

Fuções de conversão

O módulo `django.utils.encoding` contém algumas funções que são capazes de realizar conversões entre `strings` Unicode e `bytestrings`.

- `smart_unicode(s, encoding='utf-8', strings_only=False, errors='strict')` converte sua entrada para uma `string` Unicode. O parametro `encoding` especifica a codificação da entrada. (Por exemplo, o Django usa isso internamente quando processa dados de formulários, que podem não ter codificação UTF-8.) O parametro `strings_only` `` , se setado como `True`, resultará em números Python, booleanos e ``None sem conversão para uma `string` (eles mantêm seus tipos originais). O parametro `errors` recebe qualquer um dos valores que são aceitos pela função `unicode()` do Python para sua manipulação de erros.

Se você passar ao `smart_unicode()` um objeto que tem um método `__unicode__`, ele usará este método para fazer a conversão.

- `force_unicode(s, encoding='utf-8', strings_only=False, errors='strict')` é idêntico ao `smart_unicode()` em quase todos os casos. A diferença é quando o primeiro argumento é uma instância de *tradução lazy*. Enquanto `smart_unicode()` preserva a tradução tardia, `force_unicode()` força estes objetos a serem strings Unicode (causando a ocorrência da tradução). Normalmente, você desejará usar `smart_unicode()`. Entretanto, `force_unicode()` é útil em template tags e filtros que *devem* ter absolutamente uma string para trabalhar com, não algo que pode ser convertido numa string.
- `smart_str(s, encoding='utf-8', strings_only=False, errors='strict')` é essencialmente o oposto de `smart_unicode()`. Ele força o primeiro argumento a ser uma bytestring. O parametro `strings_only` tem o mesmo comportamento que do `smart_unicode()` e `force_unicode()`. Essa é uma semântica ligeiramente diferente da função nativa do Python `str()`, mas a diferença é necessária em alguns poucos lugares internos do Django.

Normalmente, você somente precisará usar `smart_unicode()`. Chamando-o tão cedo quanto possível sobre qualquer entrada de dados que podem ser Unicode ou bytestring, e a partir de então, você pode tratar os resultados como sempre faz com Unicode.

Manipulação de URI e IRI

Os frameworks Web tem de lidar com URLs (que são um tipo de [IRI](#)). Um dos requerimentos das URLs é que elas são codificadas usando somente caracteres ASCII. Entretanto, num ambiente internacional, você pode precisar construir uma URL a partir de uma [IRI](#) – falando muito vagamente, um URI que pode conter caracteres Unicode. Colocando entre aspas e convertendo uma IRI para URI pode ser um pouco complicado, então o Django fornece alguma assistência.

- A função `django.utils.encoding.iri_to_uri()` implementa a conversão de uma IRI para URI como é pedido na especificação ([RFC 3987](#)).
- As funções `django.utils.http.urlquote()` e `django.utils.http.urlquote_plus()` são versões ao padrão Python `urllib.quote()` e `urllib.quote_plus()` que trabalham com caracteres não-ASCII. (O dado é convertido para UTF-8 antes da codificação.)

Estes dois grupos de funções têm efeitos ligeiramente diferentes, e é importante mante-los distintos. Normalmente, você usaria `urlquote()` nas porções individuais de caminhos IRI ou URI de modo que quaisquer caracteres reservados como ‘&’ ou ‘%’ sejam corretamente codificados. Depois, você aplica `iri_to_uri()` para o IRI completo e ele converte quaisquer caracteres não-ASCII para os valores codificados corretos.

Note: Tecnicamente, não é correto dizer que `iri_to_uri()` implementa o algoritmo completo da especificação IRI. Ele (ainda) não faz a codificação de nomes de domínios internacionais, que é uma porção do algoritmo.

A função `iri_to_uri()` não mudará caracteres ASCII que são de outra forma permitidos em uma URL. Então, por exemplo, o caractere ‘%’ não é mais codificado quando passado para `iri_to_uri()`. Isso significa que você pode passar uma URL completa para essa função e ela não bagunçará a query string ou qualquer coisa do tipo.

Um exemplo pode clarear as coisas aqui:

```
>>> urlquote(u'Paris & Orléans')
u'Paris%20%26%20Orl%C3%A9ans'
>>> iri_to_uri(u'/favorites/François/%s' % urlquote(u'Paris & Orléans'))
'/favorites/Fran%C3%A7ois/Paris%20%26%20Orl%C3%A9ans'
```

Se você olhar cuidadosamente, você poderá ver que a porção que foi gerada pelo `urlquote()` no segundo exemplo, não foi duplamente cotada quando passou pelo `iri_to_uri()`. Essa é uma funcionalidade muito importante e útil. Ela significa que você pode construir suas IRI sem se preocupar se ela contém caracteres não-ASCII e então, bem no final, chame `iri_to_uri()` sobre o resultado.

A função `iri_to_uri()` é também imutável, o que significa que o seguinte é sempre verdadeiro:


```
iri_to_uri(iri_to_uri(some_string)) = iri_to_uri(some_string)
```

Então você pode seguramente chamá-lo várias vezes sobre a mesma IRI sem o risco de problemas com duplicação.

Models

Como todas as strings são retornadas de um banco de dados como strings Unicode, os campos de model que são baseados em caracteres (CharField, TextField, URLField, etc) conterão valores Unicode quando o Django recebe dados de um banco de dados. O caso é *sempre* esse, mesmo que os dados pudessem caber numa bytestring ASCII.

Você pode passar dados em bytestrings quando estiver criando um model ou populando um campo, mas o Django o converterá para Unicode quando ele precisar.

Escolhendo entre `__str__()` e `__unicode__()`

Uma das consequências de se usar o Unicode por padrão é que você tem de tomar cuidado quando for imprimir dados de um model.

Em particular, ao invés de dar ao seu model um método `__str__()`, nós recomendamos você implementar um método `__unicode__()`. No método `__unicode__()`, você pode seguramente retornar valores de todos os campos sem ter de se preocupar se eles irão caber dentro de uma bytestring ou não. (A forma como o Python trabalha, o resultado de `__str__()` é *sempre* uma bytestring, mesmo se você acidentalmente tentar retornar um objeto Unicode).

Você pode ainda criar um método `__str__()` nos seus models se você quiser, é claro, mas você não precisa fazer isso a menos que tenha uma boa razão. O classe base `Model` do Django fornece automaticamente uma implementação do método `__str__()` que chama `__unicode__()` e codifica o resultado para UTF-8. Isso significa que você normalmente precisará somente implantar um método `__unicode__()` e deixar o Django manipular a coerção para uma bytestring quando for requerido.

Tome cuidado no `get_absolute_url()`

URLs podem somente conter caracteres ASCII. Se você estiver construindo uma URL a partir de dados que podem ser não-ASCII, tenha cuidado para codificar os resultados de uma forma que seja adequada para uma URL. O decorador `django.db.models permalink()` automaticamente manipula isso automaticamente por você.

Se você estiver construindo uma URL manualmente (i.e., *sem* usar o decorador `permalink()`), você precisará se preocupar em codificar você mesmo. Neste caso use, use as funções `iri_to_uri()` e `urlquote()` que foram documentadas [acima](#). Por exemplo:

```
from django.utils.encoding import iri_to_uri
from django.utils.http import urlquote

def get_absolute_url(self):
    url = u'/person/%s/?x=0&y=0' % urlquote(self.location)
    return iri_to_uri(url)
```

Essa função retorna um URL corretamente codificada mesmo se `self.location` seja algo tipo “Jack visited Paris & Orléans”. (De fato, a chamada `iri_to_uri()` não é estritamente necessária no exemplo acima, pois todos os caracteres não-ASCII já foram removidos durante o `urlquote()` na primeira linha.)

A API de banco de dados

Você pode passar tanto strings Unicode ou bytestrings UTF-8 como argumentos para os métodos `filter()` e seus semelhantes da API de banco de dados. Os dois queriesets a seguir são idênticos:

```
qs = People.objects.filter(name__contains=u'Å')
qs = People.objects.filter(name__contains='\xc3\x85') # Codificação UTF-8 de Å
```

Templates

Você pode usar tanto Unicode quanto bytestrings quando criar templates manualmente:

```
from django.template import Template
t1 = Template('This is a bytestring template.')
t2 = Template(u'This is a Unicode template.')
```

Porém o caso mais comum é ler os templates do sistema de arquivos, e isso gera uma ligeira complicação: nem todos os sistemas de arquivos armazenam seus dados codificados em UTF-8. Se seus arquivos de template não são armazenados com codificação UTF-8, configure o `FILE_CHARSET` para codificar os arquivos no disco. Quando o Django lê um arquivo de template, ele converterá os dados dele para Unicode. (`FILE_CHARSET` é configurado como `'utf-8'` por padrão.)

A configuração `DEFAULT_CHARSET` controla a codificação de templates renderizados. Isto é configurado como UTF-8 por padrão.

Template tags e filtros

Algumas dicas para se lembrar quando escrever suas próprias template tags e filtros:

- Sempre retorne strings Unicode de um método render de uma template tag e de um filtro.
- Use `force_unicode()` de preferência, ao invés de `smart_unicode()` nestes lugares. As chamadas de renderização de tags e filtros ocorrem quando o template estiver em renderização, assim não há vantagem em adiar a conversão de objetos de traduções lazy. É mais fácil trabalhar unicamente com strings Unicode neste ponto.

E-mail

O framework de email do Django (em `django.core.mail`) suporta Unicode transparentemente. Você pode usar dados Unicode no corpo das mensagens e qualquer cabeçalho. Entretanto, você ainda será obrigado a respeitar os requerimentos das especificações do e-mail, desta forma, por exemplo, endereços de e-mail devem usar somente caracteres ASCII.

O seguinte código de exemplo demonstra que tudo, exceto endereços de e-mail, podem ser não ASCII:

```
from django.core.mail import EmailMessage

subject = u'My visit to Sør-Trøndelag'
sender = u'Arnbjörg Ráðormsdóttir <arnbjorg@example.com>'
recipients = ['Fred <fred@example.com>']
body = u'...'
EmailMessage(subject, body, sender, recipients).send()
```

Submissão de formulários

Submissão de formulários HTML é uma área complicada. Não há garantias de que a submissão incluirá informações de codificação, o que significa que o framework pode ter de adivinhar a codificação dos dados submetidos.

O Django adota uma abordagem “lazy” para decodificar dados de formulário. Os dados num objeto `HttpRequest` é somente decodificado quando você o acessa. De fato, a maior parte dos dados não é decodificada. Somente as estruturas de dados `HttpRequest.GET` e `HttpRequest.POST` tem qualquer decodificação aplicada nelas. Estes dois campos retornam seus membros como dados Unicode. Todos os outros atributos e método do `HttpRequest` retornam exatamente o dado como foi submetido pelo cliente.

Por padrão, a configuração `DEFAULT_CHARSET` é usada como a codificação assumida pelos dados do formulário. Se você precisa mudar isso para um formulário particular, você pode configurar o atributo `encoding` de uma instância `HttpRequest`. Por exemplo:

```
def some_view(request):
    # Nós sabemos que os dados devem ser codificados como KOI8-R
    # (por alguma razão).
    request.encoding = 'koi8-r'
    ...
```

Você ainda pode mudar a codificação depois de acessado o `request.GET` ou `request.POST`, e todos os acessos subsequentes usarão a nova codificação.

A maioria dos desenvolvedores não precisam se preocupar em mudar a codificação de um formulário, mas essa é uma funcionalidade útil quando falamos de sistemas legadas cuja codificação você não pode controlar.

O Django não codifica dados de arquivos de upload, pois esses dados são normalmente tratados como coleções de bytes, ao invés de strings. Qualquer decodificação automática não alteraria o sentido do fluxo de bytes.

Part VI

Meta-documentação e miscelânea

Documentação que não podemos encontrar um lugar mais organizado para colocá-las. Como aquela gaveta na sua cozinha com tesoura, baterias, fita adesiva, e outras tralhas.

Filosofia de design

Este documento explica algumas filosofias fundamentais dos desenvolvedores do Django, que as tem usado na criação do framework. O objetivo é explicar o passado e guiar o futuro.

Em toda parte

Baixo acoplamento

Um objetivo fundamental da pilha do Django é o **baixo acoplamento e alta coesão**. As várias camadas do framework não devem se “conhecer” a menos que seja absolutamente necessário.

Por exemplo, o sistema de template não sabe nada sobre requisições Web, a camada de banco de dados não sabe nada sobre a exibição dos dados e o sistema de view não se preocupa com que sistema de template o programador está utilizando.

Apesar do Django vir com uma pilha completa por conveniência, as partes da pilha são independentes de quaisquer outras sempre que possível.

Menos código

Aplicações Django devem utilizar o menor código possível; elas devem possuir pouco código “boilerplate” (códigos repetitivos). O Django deve tirar pleno proveito das capacidades dinâmicas do Python, tais como a introspecção.

Desenvolvimento ágil

O ponto forte de um framework Web no século 21 é tornar rápido os aspectos tediosos do desenvolvimento Web. O Django deve permitir um desenvolvimento Web incrivelmente rápido.

Não se repita (*Don't repeat yourself, DRY*)

Cada conceito distinto e/ou porção de dado deve existir em um, e somente em um, lugar. Redundância é ruim. Normalização é bom.

O framework, dentro do possível, deve deduzir o máximo do menor número de informação possível.

See also:

A [discussão sobre DRY no Portland Pattern Repository](#)

Explícito é melhor que implícito

Isto é um [princípio do núcleo do Python](#), que significa que o Django não deve fazer muita “mágica”. Mágica não deveria acontecer a menos que haja realmente uma boa razão para ela. Vale a pena usar mágica apenas quando ela cria uma enorme conveniência inatingível de outras formas, e não é aplicada de uma forma que confunde desenvolvedores que estão tentando aprender a usar o recurso.

Consistência

O framework deve ser consistente em todos os níveis. Consistência aplica-se em tudo, do baixo-nível (o estilo de código Python utilizado) ao alto-nível (a “experiência” de uso do Django).

Models

Explícito é melhor que implícito

Os campos não devem assumir certos comportamentos baseado unicamente no nome do campo. Isto exige muito conhecimento do sistema e é propenso a erros. Ao invés disso, os comportamentos devem ser baseados em argumentos com palavras-chave (keyword arguments) e, em alguns casos, sobre o tipo do campo.

Incluir toda lógica de domínio relevante

Modelos devem encapsular todo aspecto de um “objeto”, seguindo o padrão de projeto [Active Record](#) do Martin Fowler.

É por este motivo que tanto os dados representados por um modelo e a informação sobre o mesmo (seu nome legível por humanos, opções como ordenação padrão, etc.), são definidos em uma classe de modelo; todas as informações necessárias para compreender um determinado modelo deve ser armazenado *no* modelo.

API do banco de dados

Os objetivos do núcleo da API do banco de dados são:

Eficiência do SQL

As declarações SQL devem ser executadas o mais rápido possível, e as consultas devem ser otimizadas internamente.

Esta é a razão pela qual os desenvolvedores precisam executar `save()` explicitamente, ao invés do framework salvar as coisas silenciosamente de trás das cortinas.

Esta também é a razão dos métodos `select_related()` e `QuerySet` existirem. É como um reforço opcional de performance para o caso comum de selecionar “tudo que está relacionado a um objeto”.

Sintaxe poderosa e concisa

A API do banco de dados deve permitir ricas e expressivas consultas em uma sintaxe mais simples possível. Ela não deve depender de outros módulos ou da importação de objetos auxiliares.

“Joins” devem ser executados automaticamente quando necessário, por trás das cortinas.

Todo objeto deve ser capaz de acessar todo conteúdo relacionado a ela. Este acesso deve funcionar nos dois sentidos.

Opção de usar SQL puro facilmente, quando necessário

A API do banco de dados deve realizar isto através de um atalho, mas não necessariamente como um fim para tudo. O framework deve tornar fácil a escrita de SQL personalizado – consultas inteiras, ou simplesmente cláusulas `WHERE` com parâmetros personalizados para chamadas da API.

Design de URL

Baixo acoplamento

URLs nas aplicações Django não devem ser acopladas a códigos Python subjacentes. Subordinando URLs a nomes de funções do Python é uma coisa ruim e feia de se fazer.

Sendo assim, o sistema de URL do Django deve permitir que URLs para uma mesma aplicação sejam diferentes em diferentes contextos. Por exemplo, um site pode ter notícias em `/stories/`, enquanto em outros o caminho utilizado é `/news/`.

Flexibilidade infinita

URLs devem ser tão flexíveis quanto possível. Qualquer URL concebível deve ser permitida.

Encorajar as melhores práticas

O framework deve tornar fácil (ou mesmo mais fácil) o design de URLs bonitas do que feias.

Extensões de arquivos nas URLs das páginas devem ser evitadas.

O estilo Vignette de vírgulas em URLs merecem um castigo severo.

URLs definitivas

Técnicamente, `foo.com/bar` e `foo.com/bar/` são URLs diferentes, e robôs de motores de busca (e alguns analisadores de tráfego na Web) irão tratá-las como páginas distintas. O Django deve fazer um esforço para “normalizar” as URLs, a fim de que os robôs de motores de busca não se confundam.

Este é o raciocínio por trás da configuração `APPEND_SLASH`.

Sistema de template

Separar a lógica da apresentação

Nós vemos um sistema de template como uma ferramenta que controla a apresentação e a lógica relacionada à apresentação – e só. O sistema de template não deve suportar funcionalidades que vão além de seu objetivo básico.

Se quiséssemos colocar tudo em templates, era para estarmos usando PHP. Já passamos por isso.

Desencorajar redundâncias

A maioria dos Web sites dinâmicos utilizam algum tipo de design comum – um cabeçalho comum, rodapé, barra de navegação, etc. O sistema de template do Django deve ser fácil para armazenar estes elementos em um único lugar, eliminando código duplicado.

Esta é a filosofia por trás da *herança de templates*.

Ser dissociado do HTML

O sistema de template não deve ser projetado para gerar somente saída HTML. Ele deve ser igualmente bom para geração de outros formatos baseados em texto, ou só texto plano.

XML não deve ser utilizada como linguagem de template

Usando um motor de XML para fazer análise dos templates, introduz todo um novo mundo de erro humano na edição de templates – e implica um nível inaceitável de overhead no processamento de template.

Presume competência do designer

O sistema de template não deve ser projetado de modo que templates necessariamente sejam exibidos amigavelmente nos editores WYSIWYG, como o Dreamweaver. Isso é tão grave, que passa a ser uma limitação e não permitiria a sintaxe ser tão legal como ela é. O Django espera que os autores de templates estejam confortáveis editando HTML diretamente.

Trate espaços em branco de maneira óbvia

O sistema de template não deve fazer coisas mágicas com espaços em branco. Se um template inclui espaços em branco, o sistema deve tratá-los como se trata um texto – simplesmente exibindo-o. Qualquer espaço em branco que não está em uma tag de template deverá ser exibido.

Não invente uma linguagem de programação

O sistema de template intencionalmente não permite o seguinte:

- Atribuição de variáveis
- Lógica avançada

O objetivo não é inventar uma linguagem de programação. O objetivo é oferecer somente funcionalidades suficientes, como lógica condicional e laços (looping), que são essenciais para contruir representações relacionadas com decisões.

O sistema de template do Django reconhece que templates são mais frequentemente escritos por *designers*, não por *programadores*, e por isso não deve presumir conhecimento em Python.

Proteção e segurança

O sistema de template, *out of the box*, deve proibir a inclusão de códigos maliciosos – como comandos que apagam registros no banco de dados.

Esta é outra razão para que o sistema de template não permita código Python arbitrário.

Extensibilidade

O sistema de template deve reconhecer que autores avançados de templates podem querer estender sua tecnologia. Está é a filosofia por trás das tags e filtros de template personalizados.

Views

Simplicidade

Escrever uma view deve ser tão simples quanto escrever uma função Python. Os desenvolvedores não devem ter que instanciar uma classe quando uma função for suficiente.

Use objetos request

Views devem ter acesso ao objeto *request* – um objeto que armazena metadados sobre a requisição corrente. O objeto deve ser passado diretamente para a função view, ao invés da função view ter que acessar o request de uma variável global. Isto se faz leve, limpo e fácil de testar views passando-a objetos de requisição “falsos”.

Baixo acoplamento

Uma view não deve se preocupar sobre qual sistema de template o desenvolvedor utiliza – ou mesmo se ele utiliza algum.

Diferenciamento entre GET e POST

GET é POST são distintos; os desenvolvedores devem explicitamente usar um ou outro. O framework deve tornar fácil a distinção entre os dados GET e POST.

Estabilidade de API

O release do Django 1.0 vem com a promessa de estabilidade de API e compatibilidade futura. Em resumo, isto significa que o código que você construir utilizando o Django 1.0 irá funcionar com 1.1 sem mudanças, e você precisará fazer somente pequenas mudanças para qualquer release 1.x.

O que “stable” significa

Neste contexto, “stable” significa:

- Todas as APIs públicas – tudo que está documentado nos links abaixo, e todos os métodos que não comecem com um underscore – não serão movidas nem renomeadas sem fornecer aliases de compatibilidade anterior.
- Se novas funcionalidades são adicionadas a essas APIs – o que é bem possível – elas não irão quebrar ou mudar o significado dos métodos existentes. Em outras palavras, “stable” não significa, necessariamente, “completo”.
- Se, por alguma razão, uma API declarada “stable” tiver que ser removida ou substituída, ela será declarada obsoleta, porém irá permanecer até, pelo menos, duas novas versões secundárias. Avisos serão feitos quando um método obsoleto for chamado.

Veja *Official releases* para mais detalhes de como a numeração das versões do Django funciona, e como as funcionalidades entrarão em desuso.

- Nós somente iremos quebrar a compatibilidade anterior dessas APIs se um bug ou uma falha de segurança fizer com que seja completamente inevitável.

APIs “stable”

Geralmente, tudo que foi documentado – com exceção de qualquer item da *area interna* é considerado “stable” na versão 1.0. Isto inclui estas APIs:

- *Autorização*
- *Caching*.
- *Definição de Model, gerenciadores, consultas e transações*
- *Envio de e-mail*.

- *Manipulação de arquivos e armazenamento*
- *Formulários*
- *Manipulação de requisições/respostas HTTP*, incluindo envio de arquivos, middleware, sessões, resolução de URL, view, e “shortcut” APIs.
- *Generic views*.
- *Internacionalização*.
- *Paginação*
- *Serialização*
- *Signals*
- *Templates*, incluindo a linguagem, *APIs de template* a nível do Python, e *bibliotecas e tags customizadas de template*. Nós podemos adicionar novas tags de template no futuro, e os nomes podem inadvertidamente chocar-se com tags de template externas. Antes de adicionar algo como tags, esteja seguro de que o Django mostrará um erro caso tente carregar tags com nomes duplicados.
- *Testes*
- *Utilitário django-admin*.
- *Middleware embutido*
- *Objetos request/response*.
- *Settings*. Note, embora a *lista de configurações* possa ser considerada completa, nós podemos – e provavelmente iremos – adicionar novas configurações nas versões futuras. Este é um dos muitos lugares onde “‘stable’ não significa ‘completo’”.
- *Signals embutidos*. Como nas settings, nós iremos provavelmente adicionar novos sinais no futuro, mas os existentes não devem quebrar.
- *Manipulação de unicode*.
- Tudo coberto pelos *guias HOWTO*.

django.utils

A maioria dos módulos em `django.utils` são projetados para uso interno. Somente as seguintes partes do `django.utils` podem ser consideradas estáveis:

- `django.utils.cache`
- `django.utils.datastructures.SortedDict` – somente esta classe em particular; o resto do módulo é para uso interno.
- `django.utils.encoding`
- `django.utils.feedgenerator`
- `django.utils.http`
- `django.utils.safestring`
- `django.utils.translation`
- `django.utils.tzinfo`

Exceções

Existem algumas poucas exceções para esta promessa de estabilidade e compatibilidade reversa.

Correções de segurança

Se nós tomarmos consciência de um problema de segurança – esperamos sigam a nossa *política de relato de segurança* – nós iremos fazer tudo o que for necessário para corrigí-lo. Isto pode significar quebra de compatibilidade; A segurança triunfa sobre a garantia de compatibilidade.

Aplicações contribuídas (`django.contrib`)

Nós faremos tudo para manter essas APIs estáveis – e não temos planos de quebrar qualquer aplicação do contrib – esta é uma área que terá mais fluxo entre os releases. Como a web evolui, o Django também deve evoluir com ela.

No entanto, eventuais alterações nas aplicações do contrib virão com uma importante garantia: faremos o necessário para que seja possível utilizar uma versão mais antiga de uma aplicação do contrib se precisarmos fazer alterações. Desta forma, se o Django 1.5 vem com uma incompatibilidade no `django.contrib.flatpages`, nós faremos de forma que você continue utilizando o Django 1.4 em paralelo com o Django 1.5. Isto continuará permitindo atualizações fáceis.

Historicamente, aplicações em `django.contrib` têm sido mais estáveis que o núcleo, então, na prática, nós provavelmente nunca teremos que abrir essa exceção. Entretanto, é importante observar se você está construindo aplicações que dependam do `django.contrib`.

APIs marcadas como internal

Algumas APIs são explicitamente marcadas como “internal” (interna, em inglês) de algumas maneiras:

- Algumas documentações referem e mencionam as partes internas como tal. Se a documentação diz que algo é interno, nós reservamos o direito de mudá-lo.
- Funções, métodos, e outros objetos pré-fixados por um sublinhado (`_`). Este é um padrão do Python para indicar que algo é privado; se um método começa com um `_`, ele pertence a API interna.

Distribuições de terceiros do Django

Muitas distribuições de terceiros estão agora disponibilizando versões do Django integradas com seus sistemas de gerenciamento de pacotes. Isso pode tornar a instalação e atualização muito mais fácil para usuários do Django uma vez que a integração inclui a possibilidade de instalação automática de dependências (como drivers de bancos de dados) que o Django requer.

Tipicamente estes pacotes são baseados na última versão estável do Django, então se você deseja usar a versão em desenvolvimento do Django você necessitará seguir as seguintes instruções para *instalar a versão em desenvolvimento* do nosso repositório Subversion.

Se você está usando o GNU/Linux ou uma instalação do Unix, como o OpenSolaris, verifique com o seu fornecedor se ele já tem o Django empacotado. Se você está usando uma distribuição do GNU/Linux e não sabe como descobrir se um pacote está disponível, é uma boa hora pra aprender isso. O Wiki do Django contém uma lista de [Distribuições de terceiros](#) para ajudá-lo.

Para distribuidores

Se você gostaria de empacotar o Django para distribuição, nós ficaremos felizes em ajudar. Por favor, participe da [lista de discussão django-developers](#) e se apresente.

Nós também incentivamos todos os distribuidores a participar da [lista de discussão django-announce](#), uma lista com baixíssima atividade para o anúncio de novos lançamentos do Django e importantes correções de bugs.

Part VII

Glossário

field Um atributo em um *model*; um determinado campo que usualmente é mapeado diretamente para uma única coluna em uma base dados.

Veja *Models*.

generic view Uma função *view* de ordem superior que fornece uma implementação abstrata/genérica de um padrão comum encontrado no desenvolvimento de uma view.

Veja *Generic views (Visões genéricas)*.

model Um modelo armazena os dados de sua aplicação.

Veja *Models*.

MTV Veja *O Django parece ser um framework MVC, mas vocês chama o “controller” de “view” e a View de “template”. Por que vocês não utilizam a nomenclatura padrão?*.

MVC *Model-view-controller*; é um padrão de software. O Django *segue o padrão MVC até certo ponto*.

projeto Um pacote Python – ou seja, um diretório de código – que contém todas as configurações para uma instância do Django. Isto inclui a configuração do banco de dados, opções específicas do Django e configurações específicas das aplicações.

property Também conhecida como “atributos gerenciados”, e também uma característica comum do Python desde a versão 2.2. Retirado da *documentação da property*:

Properties (propriedades) são o modo mais decente de implementar atributos dos quais seu uso assemelha-se com o acesso a atributos, mas cujo a implementação usa chamadas a métodos. [...] Você pode fazer isto apenas sobrescrevendo `__getattr__` e `__setattr__`; mas sobrescrevendo `__setattr__` deixará todas as atribuições de atributos consideravelmente mais lentas, e sobrescrever `__getattr__` é sempre um pouco complicado de fazer corretamente. Properties deixam você fazer isto sem complicações, sem ter que sobrescrever `__getattr__` ou `__setattr__`.

queryset Um objeto representando algum conjunto de linhas a serem obtidas do banco de dados.

Veja *Fazendo consultas*.

slug Uma pequena etiqueta para alguma coisa, contendo apenas letras, números, underscores (sublinhados) ou hifens. São usados geralmente em URLs. Por exemplo, em uma típica URL de uma página de blog:

`http://www.djangoproject.com/weblog/2008/apr/12/spring/`

a última parte (`spring`) é um slug.

template Um fragmento de texto que se comporta como formatador para representação de dados. Um template ajuda a abstrair a apresentação dos dados a partir dos mesmos.

Veja *The Django template language*.

view Uma função responsável por renderizar uma página.

Part VIII

Notas de Lançamento

Notas de lançamento para os releases oficiais do Django. Cada nota irá falar sobre que há de novo em cada versão, e também descreverá qualquer incompatibilidade com versões anteriores por conta das mudanças que ocorrem.

Django version 0.95 release notes

Welcome to the Django 0.95 release.

This represents a significant advance in Django development since the 0.91 release in January 2006. The details of every change in this release would be too extensive to list in full, but a summary is presented below.

Suitability and API stability

This release is intended to provide a stable reference point for developers wanting to work on production-level applications that use Django.

However, it's not the 1.0 release, and we'll be introducing further changes before 1.0. For a clear look at which areas of the framework will change (and which ones will *not* change) before 1.0, see the `api-stability.txt` file, which lives in the `docs/` directory of the distribution.

You may have a need to use some of the features that are marked as “subject to API change” in that document, but that's OK with us as long as it's OK with you, and as long as you understand APIs may change in the future.

Fortunately, most of Django's core APIs won't be changing before version 1.0. There likely won't be as big of a change between 0.95 and 1.0 versions as there was between 0.91 and 0.95.

Changes and new features

The major changes in this release (for developers currently using the 0.91 release) are a result of merging the ‘magic-removal’ branch of development. This branch removed a number of constraints in the way Django code had to be written that were a consequence of decisions made in the early days of Django, prior to its open-source release. It's now possible to write more natural, Pythonic code that works as expected, and there's less “black magic” happening behind the scenes.

Aside from that, another main theme of this release is a dramatic increase in usability. We've made countless improvements in error messages, documentation, etc., to improve developers' quality of life.

The new features and changes introduced in 0.95 include:

- Django now uses a more consistent and natural filtering interface for retrieving objects from the database.
- User-defined models, functions and constants now appear in the module namespace they were defined in. (Previously everything was magically transferred to the `django.models.*` namespace.)

- Some optional applications, such as the FlatPage, Sites and Redirects apps, have been decoupled and moved into `django.contrib`. If you don't want to use these applications, you no longer have to install their database tables.
- Django now has support for managing database transactions.
- We've added the ability to write custom authentication and authorization backends for authenticating users against alternate systems, such as LDAP.
- We've made it easier to add custom table-level functions to models, through a new "Manager" API.
- It's now possible to use Django without a database. This simply means that the framework no longer requires you to have a working database set up just to serve dynamic pages. In other words, you can just use `URLconfs/views` on their own. Previously, the framework required that a database be configured, regardless of whether you actually used it.
- It's now more explicit and natural to override `save()` and `delete()` methods on models, rather than needing to hook into the `pre_save()` and `post_save()` method hooks.
- Individual pieces of the framework now can be configured without requiring the setting of an environment variable. This permits use of, for example, the Django templating system inside other applications.
- More and more parts of the framework have been internationalized, as we've expanded internationalization (i18n) support. The Django codebase, including code and templates, has now been translated, at least in part, into 31 languages. From Arabic to Chinese to Hungarian to Welsh, it is now possible to use Django's admin site in your native language.

The number of changes required to port from 0.91-compatible code to the 0.95 code base are significant in some cases. However, they are, for the most part, reasonably routine and only need to be done once. A list of the necessary changes is described in the [Removing The Magic](#) wiki page. There is also an easy [checklist](#) for reference when undertaking the porting operation.

Problem reports and getting help

Need help resolving a problem with Django? The documentation in the distribution is also available [online](#) at the [Django Web site](#). The [FAQ](#) document is especially recommended, as it contains a number of issues that come up time and again.

For more personalized help, the [django-users](#) mailing list is a very active list, with more than 2,000 subscribers who can help you solve any sort of Django problem. We recommend you search the archives first, though, because many common questions appear with some regularity, and any particular problem may already have been answered.

Finally, for those who prefer the more immediate feedback offered by IRC, there's a `#django` channel on `irc.freenode.net` that is regularly populated by Django users and developers from around the world. Friendly people are usually available at any hour of the day – to help, or just to chat.

Thanks for using Django!

The Django Team July 2006

Django version 0.96 release notes

Welcome to Django 0.96!

The primary goal for 0.96 is a cleanup and stabilization of the features introduced in 0.95. There have been a few small *backwards-incompatible changes* since 0.95, but the upgrade process should be fairly simple and should not require major changes to existing applications.

However, we're also releasing 0.96 now because we have a set of backwards-incompatible changes scheduled for the near future. Once completed, they will involve some code changes for application developers, so we recommend that you stick with Django 0.96 until the next official release; then you'll be able to upgrade in one step instead of needing to make incremental changes to keep up with the development version of Django.

Backwards-incompatible changes

The following changes may require you to update your code when you switch from 0.95 to 0.96:

MySQLdb version requirement

Due to a bug in older versions of the MySQLdb Python module (which Django uses to connect to MySQL databases), Django's MySQL backend now requires version 1.2.1p2 or higher of MySQLdb, and will raise exceptions if you attempt to use an older version.

If you're currently unable to upgrade your copy of MySQLdb to meet this requirement, a separate, backwards-compatible backend, called "mysql_old", has been added to Django. To use this backend, change the `DATABASE_ENGINE` setting in your Django settings file from this:

```
DATABASE_ENGINE = "mysql"
```

to this:

```
DATABASE_ENGINE = "mysql_old"
```

However, we strongly encourage MySQL users to upgrade to a more recent version of MySQLdb as soon as possible. The "mysql_old" backend is provided only to ease this transition, and is considered deprecated; aside from any necessary security fixes, it will not be actively maintained, and it will be removed in a future release of Django.

Also, note that some features, like the new `DATABASE_OPTIONS` setting (see the [databases documentation](#) for details), are only available on the “mysql” backend, and will not be made available for “mysql_old”.

Database constraint names changed

The format of the constraint names Django generates for foreign key references have changed slightly. These names are generally only used when it is not possible to put the reference directly on the affected column, so they are not always visible.

The effect of this change is that running `manage.py reset` and similar commands against an existing database may generate SQL with the new form of constraint name, while the database itself contains constraints named in the old form; this will cause the database server to raise an error message about modifying non-existent constraints.

If you need to work around this, there are two methods available:

1. Redirect the output of `manage.py` to a file, and edit the generated SQL to use the correct constraint names before executing it.
2. Examine the output of `manage.py sqlall` to see the new-style constraint names, and use that as a guide to rename existing constraints in your database.

Name changes in `manage.py`

A few of the options to `manage.py` have changed with the addition of fixture support:

- There are new `dumpdata` and `loaddata` commands which, as you might expect, will dump and load data to/from the database. These commands can operate against any of Django’s supported serialization formats.
- The `sqlinitialdata` command has been renamed to `sqlcustom` to emphasize that `loaddata` should be used for data (and `sqlcustom` for other custom SQL – views, stored procedures, etc.).
- The vestigial `install` command has been removed. Use `syncdb`.

Backslash escaping changed

The Django database API now escapes backslashes given as query parameters. If you have any database API code that matches backslashes, and it was working before (despite the lack of escaping), you’ll have to change your code to “unescape” the slashes one level.

For example, this used to work:

```
# Find text containing a single backslash
MyModel.objects.filter(text__contains='\\\\')
```

The above is now incorrect, and should be rewritten as:

```
# Find text containing a single backslash
MyModel.objects.filter(text__contains='\\')
```

Removed `ENABLE_PSYCO` setting

The `ENABLE_PSYCO` setting no longer exists. If your settings file includes `ENABLE_PSYCO` it will have no effect; to use `Psyco`, we recommend writing a middleware class to activate it.

What's new in 0.96?

This revision represents over a thousand source commits and over four hundred bug fixes, so we can't possibly catalog all the changes. Here, we describe the most notable changes in this release.

New forms library

`django.newforms` is Django's new form-handling library. It's a replacement for `django.forms`, the old form/manipulator/validation framework. Both APIs are available in 0.96, but over the next two releases we plan to switch completely to the new forms system, and deprecate and remove the old system.

There are three elements to this transition:

- We've copied the current `django.forms` to `django.oldforms`. This allows you to upgrade your code *now* rather than waiting for the backwards-incompatible change and rushing to fix your code after the fact. Just change your import statements like this:

```
from django import forms           # 0.95-style
from django import oldforms as forms # 0.96-style
```

- The next official release of Django will move the current `django.newforms` to `django.forms`. This will be a backwards-incompatible change, and anyone still using the old version of `django.forms` at that time will need to change their import statements as described above.
- The next release after that will completely remove `django.oldforms`.

Although the `newforms` library will continue to evolve, it's ready for use for most common cases. We recommend that anyone new to form handling skip the old forms system and start with the new.

For more information about `django.newforms`, read the [newforms documentation](#).

URLconf improvements

You can now use any callable as the callback in URLconfs (previously, only strings that referred to callables were allowed). This allows a much more natural use of URLconfs. For example, this URLconf:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    ('^myview/$', 'mysite.myapp.views.myview')
)
```

can now be rewritten as:

```
from django.conf.urls.defaults import *
from mysite.myapp.views import myview

urlpatterns = patterns('',
    ('^myview/$', myview)
)
```

One useful application of this can be seen when using decorators; this change allows you to apply decorators to views *in your URLconf*. Thus, you can make a generic view require login very easily:

```
from django.conf.urls.defaults import *
from django.contrib.auth.decorators import login_required
from django.views.generic.list_detail import object_list
from mysite.myapp.models import MyModel

info = {
```



```
"queryset" : MyModel.objects.all(),
}

urlpatterns = patterns('',
    ('^myview/$', login_required(object_list), info)
)
```

Note that both syntaxes (strings and callables) are valid, and will continue to be valid for the foreseeable future.

The test framework

Django now includes a test framework so you can start transmuting fear into boredom (with apologies to Kent Beck). You can write tests based on [doctest](#) or [unittest](#) and test your views with a simple test client.

There is also new support for “fixtures” – initial data, stored in any of the supported [serialization formats](#), that will be loaded into your database at the start of your tests. This makes testing with real data much easier.

See [the testing documentation](#) for the full details.

Improvements to the admin interface

A small change, but a very nice one: dedicated views for adding and updating users have been added to the admin interface, so you no longer need to worry about working with hashed passwords in the admin.

Thanks

Since 0.95, a number of people have stepped forward and taken a major new role in Django’s development. We’d like to thank these people for all their hard work:

- Russell Keith-Magee and Malcolm Tredinnick for their major code contributions. This release wouldn’t have been possible without them.
- Our new release manager, James Bennett, for his work in getting out 0.95.1, 0.96, and (hopefully) future release.
- Our ticket managers Chris Beaven (aka SmileyChris), Simon Greenhill, Michael Radziej, and Gary Wilson. They agreed to take on the monumental task of wrangling our tickets into nicely cataloged submission. Figuring out what to work on is now about a million times easier; thanks again, guys.
- Everyone who submitted a bug report, patch or ticket comment. We can’t possibly thank everyone by name – over 200 developers submitted patches that went into 0.96 – but everyone who’s contributed to Django is listed in [AUTHORS](#).

Django 1.0 alpha notas de lançamento

Bem-vindo ao Django 1.0 alpha!

Este é o primeiro de uma série de releases de previsão/desenvolvimento conducentes à eventual liberação do Django 1.0, atualmente agendado para o início de setembro de 2008. Este release a princípio atinge desenvolvedores que estão interessados no código do Django e a ajudar a identificar e resolver bugs prioritários para o release 1.0 final.

Dessa forma, este release *não* é destinado para uso em produção, e qualquer uso neste sentido é desencorajado.

O que há de novo no Django 1.0 alpha

O trunk do desenvolvimento do Django tem sido local de atividade quase constante durante o último ano, com a criação de várias novas funcionalidades desde o release 0.96. Alguns dos destaques incluem:

Aplicação admin refatorada (newforms-admin) A interface administrativa do Django (`django.contrib.admin`) foi completamente refatorada; definições do admin agora são completamente dissociadas das definições dos modelos (nada de declaração `class Admin` nos seus modelos!), reescrito para usar a nova biblioteca form-handling do Django (introduzido no release 0.96 como `django.newforms`, e agora disponível simplesmente como `django.forms`) e replanejado tendo em mente extensibilidade e customização. A documentação completa para a aplicação admin está disponível online na documentação oficial do Django:

[referência da administração](#)

Manipulação de Unicode melhorada Internamente o Django foi completamente refatorado para usar Unicode; isso simplifica drasticamente a tarefa de lidar com conteúdos e dados não-Western-European no Django. Adicionalmente, funções utilitárias foram providas para facilitar interoperabilidade de bibliotecas e sistemas de terceiros que podem ou não manipular Unicode graciosamente. Detalhes estão disponíveis na documentação Manipulando Unicode:

[unicode reference](#)

Uma melhoria no ORM do Django O mapeador objeto-relacional do Django – o componente que provê o mapeamento entre as classes de modelo do Django e seu banco de dados, e que faz a mediação de suas consultas de banco de dados – tem sido dramaticamente melhorado por meio de uma massiva refatoração. Para a maioria dos usuários do Django isso é compatível com versões anteriores; a parte pública da API para consultas do banco de dados sofreu mudanças menores, e a maior parte das atualizações foi feita nas partes

internas do ORM. Um guia das mudanças, incluindo as modificações incompatíveis com versões anteriores e as menções às novas funcionalidades introduzidas por tal refatoramento, está disponível no wiki do Django:

<http://code.djangoproject.com/wiki/QuerysetRefactorBranch>

Escape automático de variáveis de template Para prover melhorias de segurança contra vulnerabilidades de cross-site scripting (XSS), o sistema de templates do Django agora escapa a saída de variáveis automaticamente. Esse comportamento é configurável, e permite ambos, variáveis e grandes construções de templates serem marcados como seguros (não necessitando de escape) ou não-seguro (necessitando de escape). Um guia completo para essa funcionalidade está na documentação do sistema da tag `autoescape`.

Há muitas outras funcionalidades, muitas correções de bugs e muitos aprimoramentos para funcionalidades existentes em releases anteriores. A biblioteca `newforms`, por exemplo, tem sofrido massivas melhorias incluindo vários add-ons usuais no `django.contrib` que complementam e constroem novas capacidades de manipulação de formulários no Django, e os manipuladores de upload de arquivos do Django têm sido refatorados para permitir um controle refinado sobre o processo de transferência, bem como upload stream de arquivos grandes.

Junto com essas melhorias e acréscimos, nós fizemos um número de mudanças incompatíveis com versões anteriores do framework, à medida que novas funcionalidades tomaram forma e APIs foram finalizadas para o release 1.0. Um guia completo para essas mudanças estará disponível como parte do release final Django 1.0, e uma lista completa das mudanças que trouxeram incompatibilidades com versões anteriores também está disponível no wiki do Django para aqueles que desejam começar a desenvolver e testar seu processo de atualização:

<http://code.djangoproject.com/wiki/BackwardsIncompatibleChanges>

Roadmap do Django 1.0

Um dos primeiros objetivos desta versão alpha é focar a atenção nos elementos a serem implementados para o Django 1.0, e sobre bugs que precisam ser resolvidos antes da versão final. Acompanhando esta versão, nós realizaremos uma série de sprints para concretizar uma série de versões beta e atingir um estágio release-candidate, logo em seguida virá o Django 1.0. A linha do tempo é projetada para ser:

- 1 de agosto de 2008: Sprint (localizado em Washington, DC, e online).
- 5 de agosto de 2008: Versão beta do Django 1.0. É quando acontecerá o congelamento de novidades para o 1.0. Qualquer funcionalidade que almeje ser incluída no 1.0 deve ser completada no trunk neste prazo.
- 8 de agosto de 2008: Sprint (localizado em Lawrence, KS, e online).
- 12 de agosto de 2008: Versão beta 2 do Django 1.0.
- 15 de agosto de 2008: Sprint (localizado em Austin, TX, e online).
- 19 de agosto de 2008: Release-candidate 1 do Django 1.0.
- 22 de agosto de 2008: Sprint (localizado em Portland, OR e online).
- 26 de agosto de 2008: Release-candidate 2 do Django 1.0
- 2 de setembro de 2008: Versão final do Django 1.0. A festa da versão oficial do Django 1.0 será durante a primeira DjangoCon, a ser realizada em Mountain View, CA, nos dias 6 e 7 Setembro.

É claro, como qualquer linha de tempo estimada, ela está sujeita a mudanças. A última informação sempre será disponibilizada no wiki do projeto Django:

<http://code.djangoproject.com/wiki/VersionOneRoadmap>

O que você pode fazer para ajudar

A fim de proporcionar uma versão 1.0 de alta qualidade, nós precisamos de ajuda. Embora esta versão alpha seja, novamente, *não-recomendada* para uso em produção, você pode ajudar o time do Django testando o código

alpha em um ambiente seguro de testes e reportando qualquer bug ou problema que encontrar. O ticket tracker do Django é o lugar central para procurar por problemas não resolvidos:

<http://code.djangoproject.com/timeline>

Por favor, crie novos tickets se não existir um ticket correspondente ao seu problema.

Adicionalmente, discussões do desenvolvimento do Django, incluindo progressos rumo à versão 1.0, ocorrem diariamente na lista de desenvolvedores:

<http://groups.google.com/group/django-developers>

...e no canal do IRC #django-dev na rede `irc.freenode.net`. Se você está interessado em ajudar com o desenvolvimento do Django, sintá-se livre para se juntar às discussões.

A documentação online do Django também inclui pontos sobre como contribuir para o Django:

[contribuindo para o Django](#)

Contribuições de qualquer nível – desenvolvendo código, escrevendo documentação ou simplesmente criando tickets e ajudando a testar correções de bugs propostos – são sempre bem-vindas e apreciadas.

Django 1.0 alpha 2 release notes

Welcome to Django 1.0 alpha 2!

This is the second in a series of preview/development releases leading up to the eventual release of Django 1.0, currently scheduled to take place in early September 2008. This release is primarily targeted at developers who are interested in testing the Django codebase and helping to identify and resolve bugs prior to the final 1.0 release.

As such, this release is *not* intended for production use, and any such use is strongly discouraged.

What's new in Django 1.0 alpha 2

Django's development trunk has been the site of nearly constant activity over the past year, with several major new features landing since the 0.96 release. For features which were new as of Django 1.0 alpha 1, see the 1.0 alpha 1 release notes <releases-1.0-alpha-1. Since the 1.0 alpha 1 release several new features have landed, including:

django.contrib.gis (GeoDjango) A project over a year in the making, this adds world-class GIS (Geographic Information Systems) support to Django, in the form of a `contrib` application. Its documentation is currently being maintained externally, and will be merged into the main Django documentation prior to the final 1.0 release. Huge thanks go to Justin Bronn, Jeremy Dunck, Brett Hoerner and Travis Pinney for their efforts in creating and completing this feature.

Pluggable file storage Django's built-in `FileField` and `ImageField` now can take advantage of pluggable file-storage backends, allowing extensive customization of where and how uploaded files get stored by Django. For details, see [the files documentation](#); big thanks go to Marty Alchin for putting in the hard work to get this completed.

Jython compatibility Thanks to a lot of work from Leo Soto during a Google Summer of Code project, Django's codebase has been refactored to remove incompatibilities with **Jython**, an implementation of Python written in Java, which runs Python code on the Java Virtual Machine. Django is now compatible with the forthcoming Jython 2.5 release.

There are many other new features and improvements in this release, including two major performance boosts: strings marked for translation using [Django's internationalization system](#) now consume far less memory, and Django's internal dispatcher – which is invoked frequently during request/response processing and when working with Django's object-relational mapper – is now significantly faster.

The Django 1.0 roadmap

One of the primary goals of this alpha release is to focus attention on the remaining features to be implemented for Django 1.0, and on the bugs that need to be resolved before the final release. Following this release, we'll be conducting a series of development sprints building up to the beta and release-candidate stages, followed soon after by Django 1.0. The timeline is projected to be:

- **August 14, 2008: Django 1.0 beta release.** Past this point Django will be in a “feature freeze” for the 1.0 release; after Django 1.0 beta, the development focus will be solely on bug fixes and stabilization.
- August 15, 2008: Sprint (based in Austin, Texas, USA, and online).
- August 17, 2008: Sprint (based in Tel Aviv, Israel, and online).
- **August 21, 2008: Django 1.0 release candidate 1.** At this point, all strings marked for translation within Django's codebase will be frozen, to provide contributors time to check and finalize all of Django's bundled translation files prior to the final 1.0 release.
- August 22, 2008: Sprint (based in Portland, Oregon, USA, and online).
- **August 26, 2008: Django 1.0 release candidate 2.**
- August 30, 2008: Sprint (based in London, England, UK, and online).
- **September 2, 2008: Django 1.0 final release.** The official Django 1.0 release party will take place during the first-ever DjangoCon, to be held in Mountain View, California, USA, September 6-7.

Of course, like any estimated timeline, this is subject to change as requirements dictate. The latest information will always be available on the Django project wiki:

<http://code.djangoproject.com/wiki/VersionOneRoadmap>

What you can do to help

In order to provide a high-quality 1.0 release, we need your help. Although this alpha release is, again, *not* intended for production use, you can help the Django team by trying out the alpha codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

<http://code.djangoproject.com/timeline>

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.0 release, takes place daily on the django-developers mailing list:

<http://groups.google.com/group/django-developers>

...and in the #django-dev IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

[contributing to Django](#)

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

Django 1.0 beta 1 release notes

Welcome to Django 1.0 beta 1!

This is the third in a series of preview/development releases leading up to the eventual release of Django 1.0, currently scheduled to take place in early September 2008. This releases is primarily targeted at developers who are interested in testing the Django codebase and helping to identify and resolve bugs prior to the final 1.0 release.

As such, this release is *not* intended for production use, and any such use is discouraged.

What's new in Django 1.0 beta 1

Django's development trunk has been the site of nearly constant activity over the past year, with several major new features landing since the 0.96 release. For features which were new as of Django 1.0 alpha 1, see [the 1.0 alpha 1 release notes](#). For features which were new as of Django 1.0 alpha 2, see [the 1.0 alpha 2 release notes](#).

This beta release does not contain any major new features, but does include several smaller updates and improvements to Django:

Generic relations in forms and admin Classes are now included in `django.contrib.contenttypes` which can be used to support generic relations in both the admin interface and in end-user forms. See [the documentation for generic relations](#) for details.

Improved flexibility in the admin Following up on the refactoring of Django's administrative interface (`django.contrib.admin`), introduced in Django 1.0 alpha 1, two new hooks have been added to allow customized pre- and post-save handling of model instances in the admin. Full details are in [the admin documentation](#).

INSERT/UPDATE distinction Although Django's default behavior of having a model's `save()` method automatically determine whether to perform an INSERT or an UPDATE at the SQL level is suitable for the majority of cases, there are occasional situations where forcing one or the other is useful. As a result, models can now support an additional parameter to `save()` which can force a specific operation. Consult the database API documentation for details and important notes about appropriate use of this parameter.

Split CacheMiddleware Django's `CacheMiddleware` has been split into three classes: `CacheMiddleware` itself still exists and retains all of its previous functionality, but it is now built from two separate middleware classes which handle the two parts of caching (inserting into and reading from the cache) separately, offering additional flexibility for situations where combining these functions into a single middleware posed problems. Full details, including updated notes on appropriate use, are in [the caching documentation](#).

Removal of deprecated features A number of features and methods which had previously been marked as deprecated, and which were scheduled for removal prior to the 1.0 release, are no longer present in Django. These include imports of the form library from `django.newforms` (now located simply at `django.forms`), the `form_for_model` and `form_for_instance` helper functions (which have been replaced by `ModelForm`) and a number of deprecated features which were replaced by the dispatcher, file-uploading and file-storage refactorings introduced in the Django 1.0 alpha releases. A full list of these and all other backwards-incompatible changes is available on [the Django wiki](#).

A number of other improvements and bugfixes have also been included: some tricky cases involving case-sensitivity in differing MySQL collations have been resolved, Windows packaging and installation has been improved and the method by which Django generates unique session identifiers has been made much more robust.

The Django 1.0 roadmap

One of the primary goals of this beta release is to focus attention on the remaining features to be implemented for Django 1.0, and on the bugs that need to be resolved before the final release. Following this release, we'll be conducting a series of development sprints building up to the release-candidate stage, followed soon after by Django 1.0. The timeline is projected to be:

- August 15, 2008: Sprint (based in Austin, Texas, USA, and online).
- August 17, 2008: Sprint (based in Tel Aviv, Israel, and online).
- **August 21, 2008: Django 1.0 release candidate 1.** At this point, all strings marked for translation within Django's codebase will be frozen, to provide contributors time to check and finalize all of Django's bundled translation files prior to the final 1.0 release.
- August 22, 2008: Sprint (based in Portland, Oregon, USA, and online).
- **August 26, 2008: Django 1.0 release candidate 2.**
- August 30, 2008: Sprint (based in London, England, UK, and online).
- **September 2, 2008: Django 1.0 final release.** The official Django 1.0 release party will take place during the first-ever DjangoCon, to be held in Mountain View, California, USA, September 6-7.

Of course, like any estimated timeline, this is subject to change as requirements dictate. The latest information will always be available on the Django project wiki:

<http://code.djangoproject.com/wiki/VersionOneRoadmap>

What you can do to help

In order to provide a high-quality 1.0 release, we need your help. Although this beta release is, again, *not* intended for production use, you can help the Django team by trying out the beta codebase in a safe test environment and reporting any bugs or issues you encounter. The Django ticket tracker is the central place to search for open issues:

<http://code.djangoproject.com/timeline>

Please open new tickets if no existing ticket corresponds to a problem you're running into.

Additionally, discussion of Django development, including progress toward the 1.0 release, takes place daily on the django-developers mailing list:

<http://groups.google.com/group/django-developers>

...and in the `#django-dev` IRC channel on `irc.freenode.net`. If you're interested in helping out with Django's development, feel free to join the discussions there.

Django's online documentation also includes pointers on how to contribute to Django:

[contributing to Django](#)

Contributions on any level – developing code, writing documentation or simply triaging tickets and helping to test proposed bugfixes – are always welcome and appreciated.

Notas de lançamento do Django 1.0 beta 2

Bem vindo ao Django 1.0 beta 2!

Esse é o quarto em uma série de versões de preview/desenvolvimento que eventualmente levará ao 1.0, atualmente agendado para acontecer no início de Setembro de 2008. Esse lançamento visa primeiramente desenvolvedores que estão interessados em testar a base de código do Django e ajudar a identificar e resolver bugs antes da versão 1.0 final.

Como sempre, essa versão *não* deve tem a intenção de ser usada em produção, e tal uso é desencorajado.

O que há de novo no Django 1.0 beta 2

O trunk do desenvolvimento do Django tem sido o lugar de desenvolvimento quase constante durante o ano passado, com algumas novas funcionalidades desde a versão. Para funcionalidades que eram novas no Django 1.0 alpha 1, veja *as notas de lançamento do 1.0 alpha 1*. Para funcionalidades que eram novas no Django 1.0 alpha 2, veja *as notas de lançamento do 1.0 alpha 2*. Para funcionalidades que eram novas no Django 1.0 beta 1, veja *as notas de lançamento do 1.0 beta 1*.

Essa versão beta inclui duas funcionalidades principais:

django.contrib.comments refatorado Como parte de um projeto do Google Summer of Code, Thejaswi Puthraya executou uma reescrita e refatoração no sistema de comentários incluído com o Django, aumentando grandemente sua flexibilidade e personalização. *Documentação completa* está disponível, assim como um *guia de atualização* se você estiver usando a encarnação anterior da aplicação de comentários..

Documentação refatorada A documentação incluída com o Django é a documentação online foram refatoradas significativamente; o novo sistema de documentação usa o *Sphinx* para construir documentos e lidar com coisas bacanas como índices de tópicos, documentação referenciada e inter referenciada dentro dos documentos. Você pode ver a nova documentação *online* ou, se você tiver o Sphinx instalado, gerar você mesmo o HTML dos arquivos de documentação incluídos com o Django.

Junto com essas novas funcionalidades, o time do Django tem trabalhado duro em esmiralhar a base de código do Django para a versão 1.0 final; essa versão beta contém um número grande de pequenas melhorias e correções de bugs a partir da versão anterior, a caminho da versão 1.0.

Também, como parte do seu processo de obsolescência, o sistema antigo de processamento de formulários do Django foi removido; isso significa que `django.oldforms` não existe mais, e seus vários ganchos de API (como manipuladores automáticos) não existem mais no Django. Esse sistema foi completamente substituído pelo *novo sistema de manipulação de formulários* em `django.forms`.

O caminho para o Django 1.0

Um dos alvos principais com essa versão beta é focar a atenção nas funcionalidades restantes a serem implementadas no Django 1.0, e nos bugs que precisam ser resolvidos antes da versão final. A partir dessa versão beta, o Django está em seu “feature freeze” final para o 1.0; pedidos de funcionalidades novas ficarão para lançamentos posteriores, e o esforço de desenvolvimento será todo concentrado em correção de bugs e estabilidade. O Django está também em um “string freeze”; strings traduzíveis (rótulos, mensagens de erro, etc.) na base de código do Django não serão mudadas antes do lançamento, para que nossos tradutores possam produzir a versão final 1.0 dos arquivos de tradução do Django.

Para esse lançamento, estaremos conduzindo um sprint de desenvolvimento final em 30 de agosto de 2008, baseado em Londres e coordenado online; o alvo desse sprint é corrigir tantos bugs quantos forem possíveis em antecipação ao lançamento do 1.0 final, cuja data está agendada para **2 de setembro de 2008**. A festa de lançamento oficial do Django 1.0 irá acontecer durante a primeira DjangoCon, que acontecerá em Mountain View, California, USA, nos dias 6 e 7 de setembro.

O que você pode fazer para ajudar

Para produzir uma versão 1.0 de alta qualidade, precisamos de sua ajuda. Mesmo que essa versão beta *não* deva ser usada em produção, você pode ajudar a equipe do Django testando a versão beta em um ambiente seguro de testes e informando qualquer bug ou problema que você encontrar. O gerenciador de tíquetes do Django é o lugar central para procurar por problemas em aberto:

<http://code.djangoproject.com/timeline>

Por favor, abra novos tíquetes apenas se você não encontrar um tíquete existente que corresponda ao problema que você está enfrentando.

Adicionalmente, discussão sobre o desenvolvimento do Django, incluindo o processo até a versão 1.0, acontece diariamente na lista de e-mails `django-developers`:

<http://groups.google.com/group/django-developers>

...e no canal de IRC `#django-dev` na `irc.freenode.net`. Se você está interessado em ajudar o desenvolvimento do Django, sintase livre para entrar nas discussões.

A documentação online do Django também inclui instruções de como contribuir com o Django:

[contribuindo para o Django](#)

Contribuições de qualquer nível – desenvolvimento de código, escrita de documentação ou simplesmente a triagem de tíquetes para ajudar a testar as correções de bugs propostas – são sempre bem vindas e apreciadas.

Django 1.0 release notes

Welcome to Django 1.0!

We've been looking forward to this moment for over three years, and it's finally here. Django 1.0 represents a the largest milestone in Django's development to date: a web framework that a group of perfectionists can truly be proud of.

Django 1.0 represents over three years of community development as an Open Source project. Django's received contributions from hundreds of developers, been translated into fifty languages, and today is used by developers on every continent and in every kind of job.

An interesting historical note: when Django was first released in July 2005, the initial released version of Django came from an internal repository at revision number 8825. Django 1.0 represents revision 8961 of our public repository. It seems fitting that our 1.0 release comes at the moment where community contributions overtake those made privately.

Stability and forwards-compatibility

The release of Django 1.0 comes with a promise of API stability and forwards-compatibility. In a nutshell, this means that code you develop against Django 1.0 will continue to work against 1.1 unchanged, and you should need to make only minor changes for any 1.X release.

See the *API stability guide* for full details.

Backwards-incompatible changes

Django 1.0 has a number of backwards-incompatible changes from Django 0.96. If you have apps written against Django 0.96 that you need to port, see our detailed porting guide:

Porting your apps from Django 0.96 to 1.0

Django 1.0 breaks compatibility with 0.96 in some areas.

This guide will help you port 0.96 projects and apps to 1.0. The first part of this document includes the common changes needed to run with 1.0. If after going through the first part your code still breaks, check the section *Less-common Changes* for a list of a bunch of less-common compatibility issues.

See also:

The [1.0 release notes](#). That document explains the new features in 1.0 more deeply; the porting guide is more concerned with helping you quickly update your code.

Common changes

This section describes the changes between 0.96 and 1.0 that most users will need to make.

Use Unicode

Change string literals (`'foo'`) into Unicode literals (`u'foo'`). Django now uses Unicode strings throughout. In most places, raw strings will continue to work, but updating to use Unicode literals will prevent some obscure problems.

See [Dados Unicode no Django](#) for full details.

Models

Common changes to your models file:

Rename `maxlength` to `max_length`

Rename your `maxlength` argument to `max_length` (this was changed to be consistent with form fields):

Replace `__str__` with `__unicode__`

Replace your model's `__str__` function with a `__unicode__` method, and make sure you *use Unicode* (`u'foo'`) in that method.

Remove `prepopulated_from`

Remove the `prepopulated_from` argument on model fields. It's no longer valid and has been moved to the `ModelAdmin` class in `admin.py`. See [the admin](#), below, for more details about changes to the admin.

Remove `core`

Remove the `core` argument from your model fields. It is no longer necessary, since the equivalent functionality (part of *inline editing*) is handled differently by the admin interface now. You don't have to worry about inline editing until you get to [the admin](#) section, below. For now, remove all references to `core`.

Replace `class Admin:` with `admin.py`

Remove all your inner `class Admin` declarations from your models. They won't break anything if you leave them, but they also won't do anything. To register apps with the admin you'll move those declarations to an `admin.py` file; see [the admin](#) below for more details.

See also:

A contributor to [djangosnippets](#) has written a script that'll [scan your models.py and generate a corresponding admin.py](#).

Example

Below is an example `models.py` file with all the changes you'll need to make:

Old (0.96) `models.py`:

```
class Author(models.Model):
    first_name = models.CharField(maxlength=30)
    last_name = models.CharField(maxlength=30)
    slug = models.CharField(maxlength=60, populate_from=('first_name', 'last_
↪name'))

    class Admin:
        list_display = ['first_name', 'last_name']

    def __str__(self):
        return '%s %s' % (self.first_name, self.last_name)
```

New (1.0) `models.py`:

```
class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    slug = models.CharField(max_length=60)

    def __unicode__(self):
        return u'%s %s' % (self.first_name, self.last_name)
```

New (1.0) `admin.py`:

```
from django.contrib import admin
from models import Author

class AuthorAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name']
    prepopulated_fields = {
        'slug': ('first_name', 'last_name')
    }

admin.site.register(Author, AuthorAdmin)
```

The Admin

One of the biggest changes in 1.0 is the new admin. The Django administrative interface (`django.contrib.admin`) has been completely refactored; admin definitions are now completely decoupled from model definitions, the framework has been rewritten to use Django's new form-handling library and redesigned with extensibility and customization in mind.

Practically, this means you'll need to rewrite all of your `class Admin` declarations. You've already seen in [models](#) above how to replace your `class Admin` with a `admin.site.register()` call in an `admin.py` file. Below are some more details on how to rewrite that Admin declaration into the new syntax.

Use new inline syntax

The new `edit_inline` options have all been moved to `admin.py`. Here's an example:

Old (0.96):


```
class Parent(models.Model):
    ...

class Child(models.Model):
    parent = models.ForeignKey(Parent, edit_inline=models.STACKED, num_in_admin=3)
```

New (1.0):

```
class ChildInline(admin.StackedInline):
    model = Child
    extra = 3

class ParentAdmin(admin.ModelAdmin):
    model = Parent
    inlines = [ChildInline]

admin.site.register(Parent, ParentAdmin)
```

See *Objetos InlineModelAdmin* for more details.

Simplify fields, or use fieldsets

The old `fields` syntax was quite confusing, and has been simplified. The old syntax still works, but you'll need to use `fieldsets` instead.

Old (0.96):

```
class ModelOne(models.Model):
    ...

    class Admin:
        fields = (
            (None, {'fields': ('foo', 'bar')}),
        )

class ModelTwo(models.Model):
    ...

    class Admin:
        fields = (
            ('group1', {'fields': ('foo', 'bar'), 'classes': 'collapse'}),
            ('group2', {'fields': ('spam', 'eggs'), 'classes': 'collapse wide'}),
        )
```

New (1.0):

```
class ModelOneAdmin(admin.ModelAdmin):
    fields = ('foo', 'bar')

class ModelTwoAdmin(admin.ModelAdmin):
    fieldsets = (
        ('group1', {'fields': ('foo', 'bar'), 'classes': 'collapse'}),
        ('group2', {'fields': ('spam', 'eggs'), 'classes': 'collapse wide'}),
    )
```

See also:

- More detailed information about the changes and the reasons behind them can be found on the [NewformsAdminBranch](#) wiki page
- The new admin comes with a ton of new features; you can read about them in the [admin documentation](#).

URLs

Update your root `urls.py`

If you're using the admin site, you need to update your root `urls.py`.

Old (0.96) `urls.py`:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^admin/', include('django.contrib.admin.urls')),

    # ... the rest of your URLs here ...
)
```

New (1.0) `urls.py`:

```
from django.conf.urls.defaults import *

# The next two lines enable the admin and load each admin.py file:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/(.*)', admin.site.root),

    # ... the rest of your URLs here ...
)
```

Views

Use `django.forms` instead of `newforms`

Replace `django.newforms` with `django.forms` – Django 1.0 renamed the `newforms` module (introduced in 0.96) to plain old `forms`. The `oldforms` module was also removed.

If you're already using the `newforms` library, and you used our recommended `import` statement syntax, all you have to do is change your import statements.

Old:

```
from django import newforms as forms
```

New:

```
from django import forms
```

If you're using the old forms system (formerly known as `django.forms` and `django.oldforms`), you'll have to rewrite your forms. A good place to start is the [forms documentation](#)

Handle uploaded files using the new API

Replace use of uploaded files – that is, entries in `request.FILES` – as simple dictionaries with the new [UploadedFile](#). The old dictionary syntax no longer works.

Thus, in a view like:

```
def my_view(request):
    f = request.FILES['file_field_name']
    ...
```

...you'd need to make the following changes:

Old (0.96)	New (1.0)
<code>f['content']</code>	<code>f.read()</code>
<code>f['filename']</code>	<code>f.name</code>
<code>f['content-type']</code>	<code>f.content_type</code>

Work with file fields using the new API

The internal implementation of `django.db.models.FileField` have changed. A visible result of this is that the way you access special attributes (URL, filename, image size, etc) of these model fields has changed. You will need to make the following changes, assuming your model's `FileField` is called `myfile`:

Old (0.96)	New (1.0)
<code>myfile.get_content_filename()</code>	<code>myfile.content.path</code>
<code>myfile.get_content_url()</code>	<code>myfile.content.url</code>
<code>myfile.get_content_size()</code>	<code>myfile.content.size</code>
<code>myfile.save_content_file()</code>	<code>myfile.content.save()</code>
<code>myfile.get_content_width()</code>	<code>myfile.content.width</code>
<code>myfile.get_content_height()</code>	<code>myfile.content.height</code>

Note that the width and height attributes only make sense for `ImageField` fields. More details can be found in the [model API](#) documentation.

Use Paginator instead of ObjectPaginator

The `ObjectPaginator` in 0.96 has been removed and replaced with an improved version, `django.core.paginator.Paginator`.

Templates

Learn to love autoescaping

By default, the template system now automatically HTML-escapes the output of every variable. To learn more, see [Automatic HTML escaping](#).

To disable auto-escaping for an individual variable, use the `safe` filter:

```
This will be escaped: {{ data }}
This will not be escaped: {{ data|safe }}
```

To disable auto-escaping for an entire template, wrap the template (or just a particular section of the template) in the `autoescape` tag:

```
{% autoescape off %}
... unescaped template content here ...
{% endautoescape %}
```

Less-common changes

The following changes are smaller, more localized changes. They should only affect more advanced users, but it's probably worth reading through the list and checking your code for these things.

Signals

- Add `**kwargs` to any registered signal handlers.
- Connect, disconnect, and send signals via methods on the `Signal` object instead of through module methods in `django.dispatch.dispatcher`.
- Remove any use of the `Anonymous` and `Any` sender options; they no longer exist. You can still receive signals sent by any sender by using `sender=None`
- Make any custom signals you've declared into instances of `django.dispatch.Signal` instead of anonymous objects.

Here's quick summary of the code changes you'll need to make:

Old (0.96)	New (1.0)
<code>def callback(sender)</code>	<code>def callback(sender, **kwargs)</code>
<code>sig = object()</code>	<code>sig = django.dispatch.Signal()</code>
<code>dispatcher.connect(callback, sig)</code>	<code>sig.connect(callback)</code>
<code>dispatcher.send(sig, sender)</code>	<code>sig.send(sender)</code>
<code>dispatcher.connect(callback, sig, sender=Any)</code>	<code>sig.connect(callback, sender=None)</code>

Comments

If you were using Django 0.96's `django.contrib.comments` app, you'll need to upgrade to the new comments app introduced in 1.0. See *Atualizando o sistema de comentários de um Django anterior* for details.

Template tags

spaceless tag

The spaceless template tag now removes *all* spaces between HTML tags, instead of preserving a single space.

Local flavors

U.S. local flavor

`django.contrib.localflavor.usa` has been renamed to `django.contrib.localflavor.us`. This change was made to match the naming scheme of other local flavors. To migrate your code, all you need to do is change the imports.

Sessions

Getting a new session key

`SessionBase.get_new_session_key()` has been renamed to `_get_new_session_key()`. `get_new_session_object()` no longer exists.

Fixtures

Loading a row no longer calls `save()`

Previously, loading a row automatically ran the model's `save()` method. This is no longer the case, so any fields (for example: timestamps) that were auto-populated by a `save()` now need explicit values in any fixture.

Settings

Better exceptions

The old `EnvironmentError` has split into an `ImportError` when Django fails to find the settings module and a `RuntimeError` when you try to reconfigure settings after having already used them

`LOGIN_URL` has moved

The `LOGIN_URL` constant moved from `django.contrib.auth` into the settings module. Instead of using `from django.contrib.auth import LOGIN_URL` refer to `settings.LOGIN_URL`.

`APPEND_SLASH` behavior has been updated

In 0.96, if a URL didn't end in a slash or have a period in the final component of its path, and `APPEND_SLASH` was `True`, Django would redirect to the same URL, but with a slash appended to the end. Now, Django checks to see whether the pattern without the trailing slash would be matched by something in your URL patterns. If so, no redirection takes place, because it is assumed you deliberately wanted to catch that pattern.

For most people, this won't require any changes. Some people, though, have URL patterns that look like this:

```
r'/some_prefix/(.*)$'
```

Previously, those patterns would have been redirected to have a trailing slash. If you always want a slash on such URLs, rewrite the pattern as:

```
r'/some_prefix/(.*)/'
```

Smaller model changes

Different exception from `get()`

Managers now return a `MultipleObjectsReturned` exception instead of `AssertionError`:

Old (0.96):

```
try:
    Model.objects.get(...)
except AssertionError:
    handle_the_error()
```

New (1.0):

```
try:
    Model.objects.get(...)
except Model.MultipleObjectsReturned:
    handle_the_error()
```

LazyDate has been fired

The `LazyDate` helper class no longer exists.

Default field values and query arguments can both be callable objects, so instances of `LazyDate` can be replaced with a reference to `datetime.datetime.now`:

Old (0.96):

```
class Article(models.Model):
    title = models.CharField(maxlength=100)
    published = models.DateField(default=LazyDate())
```

New (1.0):

```
import datetime

class Article(models.Model):
    title = models.CharField(max_length=100)
    published = models.DateField(default=datetime.datetime.now)
```

DecimalField is new, and FloatField is now a proper float

Old (0.96):

```
class MyModel(models.Model):
    field_name = models.FloatField(max_digits=10, decimal_places=3)
    ...
```

New (1.0):

```
class MyModel(models.Model):
    field_name = models.DecimalField(max_digits=10, decimal_places=3)
    ...
```

If you forget to make this change, you will see errors about `FloatField` not taking a `max_digits` attribute in `__init__`, because the new `FloatField` takes no precision-related arguments.

If you're using MySQL or PostgreSQL, no further changes are needed. The database column types for `DecimalField` are the same as for the old `FloatField`.

If you're using SQLite, you need to force the database to view the appropriate columns as decimal types, rather than floats. To do this, you'll need to reload your data. Do this after you have made the change to using `DecimalField` in your code and updated the Django code.

Warning: Back up your database first!

For SQLite, this means making a copy of the single file that stores the database (the name of that file is the `DATABASE_NAME` in your settings.py file).

To upgrade each application to use a `DecimalField`, you can do the following, replacing `<app>` in the code below with each app's name:

```
$ ./manage.py dumpdata --format=xml <app> > data-dump.xml
$ ./manage.py reset <app>
$ ./manage.py loaddata data-dump.xml
```

Notes:

1. It's important that you remember to use XML format in the first step of this process. We are exploiting a feature of the XML data dumps that makes porting floats to decimals with SQLite possible.

2. In the second step you will be asked to confirm that you are prepared to lose the data for the application(s) in question. Say yes; we'll restore this data in the third step, of course.
3. `DecimalField` is not used in any of the apps shipped with Django prior to this change being made, so you do not need to worry about performing this procedure for any of the standard Django models.

If something goes wrong in the above process, just copy your backed up database file over the original file and start again.

Internationalization

`django.views.i18n.set_language()` now requires a POST request

Previously, a GET request was used. The old behavior meant that state (the locale used to display the site) could be changed by a GET request, which is against the HTTP specification's recommendations. Code calling this view must ensure that a POST request is now made, instead of a GET. This means you can no longer use a link to access the view, but must use a form submission of some kind (e.g. a button).

`_()` is no longer in builtins

`_()` (the callable object whose name is a single underscore) is no longer monkeypatched into builtins – that is, it's no longer available magically in every module.

If you were previously relying on `_()` always being present, you should now explicitly import `gettext` or `gettext_lazy`, if appropriate, and alias it to `_` yourself:

```
from django.utils.translation import gettext as _
```

HTTP request/response objects

Dictionary access to `HttpRequest`

`HttpRequest` objects no longer directly support dictionary-style access; previously, both GET and POST data were directly available on the `HttpRequest` object (e.g., you could check for a piece of form data by using `if 'some_form_key' in request` or by reading `request['some_form_key']`). This is no longer supported; if you need access to the combined GET and POST data, use `request.REQUEST` instead.

It is strongly suggested, however, that you always explicitly look in the appropriate dictionary for the type of request you expect to receive (`request.GET` or `request.POST`); relying on the combined `request.REQUEST` dictionary can mask the origin of incoming data.

Accessing `HttpResponse` headers

`django.http.HttpResponse.headers` has been renamed to `_headers` and `HttpResponse` now supports containment checking directly. So use `if header in response:` instead of `if header in response.headers:`.

Generic relations

Generic relations have been moved out of core

The generic relation classes – `GenericForeignKey` and `GenericRelation` – have moved into the `django.contrib.contenttypes` module.

Testing

`django.test.Client.login()` has changed

Old (0.96):

```
from django.test import Client
c = Client()
c.login('/path/to/login', 'myuser', 'mypassword')
```

New (1.0):

```
# ... same as above, but then:
c.login(username='myuser', password='mypassword')
```

Management commands

Running management commands from your code

`django.core.management`` has been greatly refactored.

Calls to management services in your code now need to use `call_command`. For example, if you have some test code that calls `flush` and `load_data`:

```
from django.core import management
management.flush(verbosity=0, interactive=False)
management.load_data(['test_data'], verbosity=0)
```

...you'll need to change this code to read:

```
from django.core import management
management.call_command('flush', verbosity=0, interactive=False)
management.call_command('loaddata', 'test_data', verbosity=0)
```

Subcommands must now precede options

`django-admin.py` and `manage.py` now require subcommands to precede options. So:

```
$ django-admin.py --settings=foo.bar runserver
```

...no longer works and should be changed to:

```
$ django-admin.py runserver --settings=foo.bar
```

Syndication

`Feed.__init__` has changed

The `__init__()` method of the syndication framework's `Feed` class now takes an `HttpRequest` object as its second parameter, instead of the feed's URL. This allows the syndication framework to work without requiring the sites framework. This only affects code that subclasses `Feed` and overrides the `__init__()` method, and code that calls `Feed.__init__()` directly.

Data structures

SortedDictFromList is gone

`django.newforms.forms.SortedDictFromList` was removed. `django.utils.datastructures.SortedDict` can now be instantiated with a sequence of tuples.

To update your code:

1. Use `django.utils.datastructures.SortedDict` wherever you were using `django.newforms.forms.SortedDictFromList`.
2. Because `django.utils.datastructures.SortedDict.copy()` doesn't return a deepcopy as `SortedDictFromList.copy()` did, you will need to update your code if you were relying on a deep-copy. Do this by using `copy.deepcopy` directly.

Database backend functions

Database backend functions have been renamed

Almost *all* of the database backend-level functions have been renamed and/or relocated. None of these were documented, but you'll need to change your code if you're using any of these functions, all of which are in `django.db`:

Old (0.96)	New (1.0)
<code>backend.get_autoinc_sql</code>	<code>connection.ops.autoinc_sql</code>
<code>backend.get_date_extract_sql</code>	<code>connection.ops.date_extract_sql</code>
<code>backend.get_date_trunc_sql</code>	<code>connection.ops.date_trunc_sql</code>
<code>backend.get_datetime_cast_sql</code>	<code>connection.ops.datetime_cast_sql</code>
<code>backend.get_deferrable_sql</code>	<code>connection.ops.deferrable_sql</code>
<code>backend.get_drop_foreignkey_sql</code>	<code>connection.ops.drop_foreignkey_sql</code>
<code>backend.get_fulltext_search_sql</code>	<code>connection.ops.fulltext_search_sql</code>
<code>backend.get_last_insert_id</code>	<code>connection.ops.last_insert_id</code>
<code>backend.get_limit_offset_sql</code>	<code>connection.ops.limit_offset_sql</code>
<code>backend.get_max_name_length</code>	<code>connection.ops.max_name_length</code>
<code>backend.get_pk_default_value</code>	<code>connection.ops.pk_default_value</code>
<code>backend.get_random_function_sql</code>	<code>connection.ops.random_function_sql</code>
<code>backend.get_sql_flush</code>	<code>connection.ops.sql_flush</code>
<code>backend.get_sql_sequence_reset</code>	<code>connection.ops.sequence_reset_sql</code>
<code>backend.get_start_transaction_sql</code>	<code>connection.ops.start_transaction_sql</code>
<code>backend.get_tablespace_sql</code>	<code>connection.ops.tablespace_sql</code>
<code>backend.quote_name</code>	<code>connection.ops.quote_name</code>
<code>backend.get_query_set_class</code>	<code>connection.ops.query_set_class</code>
<code>backend.get_field_cast_sql</code>	<code>connection.ops.field_cast_sql</code>
<code>backend.get_drop_sequence</code>	<code>connection.ops.drop_sequence_sql</code>
<code>backend.OPERATOR_MAPPING</code>	<code>connection.operators</code>
<code>backend.allows_group_by_ordinal</code>	<code>connection.features.allows_group_by_ordinal</code>
<code>backend.allows_unique_and_pk</code>	<code>connection.features.allows_unique_and_pk</code>
<code>backend.autoindexes_primary_keys</code>	<code>connection.features.autoindexes_primary_keys</code>
<code>backend.needs_datetime_string_cast</code>	<code>connection.features.needs_datetime_string_cast</code>
<code>backend.needs_upper_for_iops</code>	<code>connection.features.needs_upper_for_iops</code>
<code>backend.supports_constraints</code>	<code>connection.features.supports_constraints</code>
<code>backend.supports_tablespace</code>	<code>connection.features.supports_tablespace</code>
<code>backend.uses_case_insensitive_names</code>	<code>connection.features.uses_case_insensitive_names</code>
<code>backend.uses_custom_queryset</code>	<code>connection.features.uses_custom_queryset</code>

A complete list of backwards-incompatible changes can be found at <http://code.djangoproject.com/wiki/BackwardsIncompatibleChanges>.

What's new in Django 1.0

A lot!

Since Django 0.96, we've made over 4,000 code commits, fixed more than 2,000 bugs, and edited, added, or removed around 350,000 lines of code. We've also added 40,000 lines of new documentation, and greatly improved what was already there.

In fact, new documentation is one of our favorite features of Django 1.0, so we might as well start there. First, there's a new documentation site:

<http://docs.djangoproject.com/>

The documentation has been greatly improved, cleaned up, and generally made awesome. There's now dedicated search, indexes, and more.

We can't possibly document everything that's new in 1.0, but the documentation will be your definitive guide. Anywhere you see something like: This feature is new in Django 1.0 You'll know that you're looking at something new or changed.

The other major highlights of Django 1.0 are:

Re-factored admin application

The Django administrative interface (`django.contrib.admin`) has been completely refactored; admin definitions are now completely decoupled from model definitions (no more `class Admin` declaration in models!), rewritten to use Django's new form-handling library (introduced in the 0.96 release as `django.newforms`, and now available as simply `django.forms`) and redesigned with extensibility and customization in mind. Full documentation for the admin application is available online in the official Django documentation:

See the [admin reference](#) for details

Improved Unicode handling

Django's internals have been refactored to use Unicode throughout; this drastically simplifies the task of dealing with non-Western-European content and data in Django. Additionally, utility functions have been provided to ease interoperability with third-party libraries and systems which may or may not handle Unicode gracefully. Details are available in Django's Unicode-handling documentation.

See *Dados Unicode no Django*.

An improved ORM

Django's object-relational mapper – the component which provides the mapping between Django model classes and your database, and which mediates your database queries – has been dramatically improved by a massive refactoring. For most users of Django this is backwards-compatible; the public-facing API for database querying underwent a few minor changes, but most of the updates took place in the ORM's internals. A guide to the changes, including backwards-incompatible modifications and mentions of new features opened up by this refactoring, is [available on the Django wiki](#).

Automatic escaping of template variables

To provide improved security against cross-site scripting (XSS) vulnerabilities, Django's template system now automatically escapes the output of variables. This behavior is configurable, and allows both variables and larger template constructs to be marked as safe (requiring no escaping) or unsafe (requiring escaping). A full guide to this feature is in the documentation for the `autoescape` tag.

`django.contrib.gis` (GeoDjango)

A project over a year in the making, this adds world-class GIS (Geographic Information Systems) support to Django, in the form of a `contrib` application. Its documentation is currently being maintained externally, and will be merged into the main Django documentation shortly. Huge thanks go to Justin Bronn, Jeremy Dunck, Brett Hoerner and Travis Pinney for their efforts in creating and completing this feature.

See <http://geodjango.org/> for details.

Pluggable file storage

Django's built-in `FileField` and `ImageField` now can take advantage of pluggable file-storage backends, allowing extensive customization of where and how uploaded files get stored by Django. For details, see [the files documentation](#); big thanks go to Marty Alchin for putting in the hard work to get this completed.

Jython compatibility

Thanks to a lot of work from Leo Soto during a Google Summer of Code project, Django's codebase has been refactored to remove incompatibilities with `Jython`, an implementation of Python written in Java, which runs Python code on the Java Virtual Machine. Django is now compatible with the forthcoming Jython 2.5 release.

See *Rodando Django no Jython*.

Generic relations in forms and admin

Classes are now included in `django.contrib.contenttypes` which can be used to support generic relations in both the admin interface and in end-user forms. See [the documentation for generic relations](#) for details.

INSERT/UPDATE distinction

Although Django's default behavior of having a model's `save()` method automatically determine whether to perform an `INSERT` or an `UPDATE` at the SQL level is suitable for the majority of cases, there are occasional situations where forcing one or the other is useful. As a result, models can now support an additional parameter to `save()` which can force a specific operation.

See *Forçando um INSERT ou UPDATE* for details.

Split CacheMiddleware

Django's `CacheMiddleware` has been split into three classes: `CacheMiddleware` itself still exists and retains all of its previous functionality, but it is now built from two separate middleware classes which handle the two parts of caching (inserting into and reading from the cache) separately, offering additional flexibility for situations where combining these functions into a single middleware posed problems.

Full details, including updated notes on appropriate use, are in [the caching documentation](#).

Refactored `django.contrib.comments`

As part of a Google Summer of Code project, Thejaswi Puthraya carried out a major rewrite and refactoring of Django's bundled comment system, greatly increasing its flexibility and customizability. [Full documentation](#) is available, as well as [an upgrade guide](#) if you were using the previous incarnation of the comments application.

Removal of deprecated features

A number of features and methods which had previously been marked as deprecated, and which were scheduled for removal prior to the 1.0 release, are no longer present in Django. These include imports of the form library from `django.newforms` (now located simply at `django.forms`), the `form_for_model` and `form_for_instance` helper functions (which have been replaced by `ModelForm`) and a number of deprecated features which were replaced by the dispatcher, file-uploading and file-storage refactorings introduced in the Django 1.0 alpha releases.

Known issues

We've done our best to make Django 1.0 as solid as possible, but unfortunately there are a couple of issues that we know about in the release.

Multi-table model inheritance with `to_field`

If you're using [multiple table model inheritance](#), be aware of this caveat: child models using a custom `parent_link` and `to_field` will cause database integrity errors. A set of models like the following are **not valid**:

```
class Parent(models.Model):
    name = models.CharField(max_length=10)
    other_value = models.IntegerField(unique=True)

class Child(Parent):
    father = models.OneToOneField(Parent, primary_key=True, to_field="other_value",
    ↪ parent_link=True)
    value = models.IntegerField()
```

This bug will be fixed in the next release of Django.

Caveats with support of certain databases

Django attempts to support as many features as possible on all database backends. However, not all database backends are alike, and in particular many of the supported database differ greatly from version to version. It's a good idea to checkout our [notes on supported database](#):

- [MySQL notes](#)
- [SQLite notes](#)
- [Oracle notes](#)

Notas de lançamento do Django 1.0.1

Bem-vindo ao Django 1.0.1!

Este é o primeiro release de correção de bugs na série Django 1.0, melhorando a estabilidade e performance do código de base do Django 1.0. Como tal, o Django 1.0.1 não contém novas funcionalidade (e, conforme [nossa política de compatibilidade](#), mantêm compatibilidade com o Django 1.0), mas contém uma série de correções e outras melhorias. É recomendado a atualização para o Django 1.0.1 para qualquer desenvolvimento ou instalação atualmente em uso ou segmentação do Django 1.0.

Correções e melhorias no Django 1.0.1

O Django 1.0.1 contém mais de 200 correções no código de base do Django 1.0; a lista detalhada de todas as correções estão disponíveis no [log do subversion do branch 1.0.X](#), mas aqui estão algumas das principais:

- Várias correções no `django.contrib.comments`, pertencentes ao feed RSS de comentários, ordenação padrão de comentários e ao XHTML e internacionalização dos templates padrão dos comentários.
- Múltiplas correções para suporte do Django a bancos de dados Oracle, incluindo suporte a paginação por GIS QuerySets, uma separação mais eficiente de resultados e melhoria na introspecção em bancos de dados existentes.
- Várias correções para suporte de consultas no mapeador objeto-relacional do Django, incluindo configuração repetida e redefinição de ordenação e correções no trabalho com consultas `INSERT-only`.
- Múltiplas correções nos formulários inline e formsets.
- Múltiplas correções nas restrições de `models.unique` e `unique_together` em formulários automaticamente gerados.
- Corrigido o suporte a declarações `upload_to` quando se manipula envio de arquivos através de formulários automaticamente gerados.
- Corrigido suporte para ordenar uma lista de mudanças do admin, baseado em atributos `callable` no `list_display`.
- Uma correção na aplicação de *autoscaping* em strings literais passadas para o filtro de template `join`. Anteriormente, strings literais passadas para o `join` eram automaticamente escapadas, ao contrário do [comportamento documentado para autoescaping e strings literais](#). Strings literais passadas para o `join` já não são automaticamente escapados, significando que você deve escapá-las manualmente agora; isto é uma incompatibilidade se você contou com este bug, mas não se você fez como está na documentação.

- Melhorado e expandido os arquivos de traduções para muitas das linguagens que o Django suporta por padrão.
- E como sempre, um grande número de melhorias para a documentação do Django, incluindo correções para ambos documentos existentes e expansão com documentação nova.

Notas de lançamento do Django 1.0.2

Bem vindo ao Django 1.0.2!

Essa é a segunda versão de “bugfixes” na série 1.0 do Django, melhorando a estabilidade e performance da base de código do Django 1.0. Assim, o Django 1.0.2 não contém novas features (e, segundo a nossa [política de compatibilidade](#), mantém retro-compatibilidade com o Django 1.0.0), mas contém um número de correções e outras melhorias. O Django 1.0.2 é uma atualização recomendada para qualquer desenvolvimento ou deployment atualmente usando ou planejando usar o Django 1.0.

Correções e melhorias no Django 1.0.2

A principal razão por trás desse release é corrigir um problema no recentemente lançado Django 1.0.1; os scripts de empacotamento usados para o Django 1.0.1 omitiram alguns diretórios do pacote final, incluindo um diretório requerido pelo `django.contrib.gis` e parte da suíte de testes unitários do Django.

O Django 1.0.2 contém scripts de empacotamento atualizados, e o pacote contém os diretórios omitidos no Django 1.0.1. Assim, esse release contém todas as correções e melhorias do Django 1.0.1; veja [as notas de lançamento do Django 1.0.1](#) para detalhes.

Adicionamento, desde o Django 1.0.1 foram lançados:

- Atualizadas traduções para Hebreu e Dinamarquês foram adicionadas.
- O método padrão `__repr__` dos models do Django estão mais robustos ao lidar com dados Unicode mal formados vindos dos métodos `__unicode__`; ao invés de lançar uma exceção em alguns casos, `repr()` irá conter uma string “[Bad Unicode data]” no lugar do Unicode inválido.
- Um bug envolvendo a interação da classe do Django `SafeUnicode` e o adaptador MySQL foi resolvido; Instâncias do `SafeUnicode` (gerados, por exemplo, por uma renderização de template) podem agora ser atribuídos a atributos de modelos e salvos no MySQL sem a necessidade de intermediação explícita de transformações para `unicode`.
- Um bug que afetava a filtragem de `DateField` anulável no SQLite foi resolvido.
- Várias atualizações e melhorias foram feitas na documentação do Django.

See also:

A lista de [mudanças com quebra de compatibilidade](#) feitas no desenvolvimento corrent, “trunk”. Se você estiver executando versões do Django mais recentes que a de um lançamento oficial, você deve manter um registo das novas peças que apontam por lá. Também é divertido ler se você estiver ansioso para novas versões do Django.

Part IX

Django internamente

A documentação para quem quer *hackear* o Django. Este é o lugar se você tem interesse em melhorar o Django, aprender mais sobre ele ou aprender sobre como ele funciona “por baixo dos panos”.

Warning: Em outros locais da documentação do Django, a cobertura de um recurso é uma espécie de um contrato: uma vez que a API está na documentação oficial, nós consideramos “stable” e não a mudamos sem uma boa razão. As APIs cobertas aqui, no entanto, são consideradas “apenas internas”: nós reservamos o direito de mudar estas APIs internas se for necessário.

Contributing to Django

If you think working *with* Django is fun, wait until you start working *on* it. We're passionate about helping Django users make the jump to contributing members of the community, so there are many ways you can help Django's development:

- Blog about Django. We syndicate all the Django blogs we know about on the [community page](#); contact jacob@jacobian.org if you've got a blog you'd like to see on that page.
- Report bugs and request features in our [ticket tracker](#). Please read [Reporting bugs](#), below, for the details on how we like our bug reports served up.
- Submit patches for new and/or fixed behavior. Please read [Submitting patches](#), below, for details on how to submit a patch.
- Join the [django-developers](#) mailing list and share your ideas for how to improve Django. We're always open to suggestions, although we're likely to be skeptical of large-scale suggestions without some code to back it up.
- Triage patches that have been submitted by other users. Please read [Ticket triage](#) below, for details on the triage process.

That's all you need to know if you'd like to join the Django development community. The rest of this document describes the details of how our community works and how it handles bugs, mailing lists, and all the other minutiae of Django development.

Reporting bugs

Well-written bug reports are *incredibly* helpful. However, there's a certain amount of overhead involved in working with any bug tracking system, so your help in keeping our ticket tracker as useful as possible is appreciated. In particular:

- **Do** read the [FAQ](#) to see if your issue might be a well-known question.
- **Do** [search the tracker](#) to see if your issue has already been filed.
- **Do** ask on [django-users](#) *first* if you're not sure if what you're seeing is a bug.
- **Do** write complete, reproducible, specific bug reports. Include as much information as you possibly can, complete with code snippets, test cases, etc. This means including a clear, concise description of the problem, and a clear set of instructions for replicating the problem. A minimal example that illustrates the bug in a nice small test case is the best possible bug report.

- **Don't** use the ticket system to ask support questions. Use the [django-users](#) list, or the #django IRC channel for that.
- **Don't** use the ticket system to make large-scale feature requests. We like to discuss any big changes to Django's core on the [django-developers](#) list before actually working on them.
- **Don't** reopen issues that have been marked "wontfix". This mark means that the decision has been made that we can't or won't fix this particular issue. If you're not sure why, please ask on [django-developers](#).
- **Don't** use the ticket tracker for lengthy discussions, because they're likely to get lost. If a particular ticket is controversial, please move discussion to [django-developers](#).
- **Don't** post to django-developers just to announce that you have filed a bug report. All the tickets are mailed to another list ([django-updates](#)), which is tracked by developers and triagers, so we see them as they are filed.

Reporting security issues

Report security issues to security@djangoproject.com. This is a private list only open to long-time, highly trusted Django developers, and its archives are not publicly readable.

In the event of a confirmed vulnerability in Django itself, we will take the following actions:

- Acknowledge to the reporter that we've received the report and that a fix is forthcoming. We'll give a rough timeline and ask the reporter to keep the issue confidential until we announce it.
- Halt all other development as long as is needed to develop a fix, including patches against the current and two previous releases.
- Determine a go-public date for announcing the vulnerability and the fix. To try to mitigate a possible "arms race" between those applying the patch and those trying to exploit the hole, we will not announce security problems immediately.
- Pre-notify everyone we know to be running the affected version(s) of Django. We will send these notifications through private e-mail which will include documentation of the vulnerability, links to the relevant patch(es), and a request to keep the vulnerability confidential until the official go-public date.
- Publicly announce the vulnerability and the fix on the pre-determined go-public date. This will probably mean a new release of Django, but in some cases it may simply be patches against current releases.

Submitting patches

We're always grateful for patches to Django's code. Indeed, bug reports with associated patches will get fixed *far* more quickly than those without patches.

"Claiming" tickets

In an open-source project with hundreds of contributors around the world, it's important to manage communication efficiently so that work doesn't get duplicated and contributors can be as effective as possible. Hence, our policy is for contributors to "claim" tickets in order to let other developers know that a particular bug or feature is being worked on.

If you have identified a contribution you want to make and you're capable of fixing it (as measured by your coding ability, knowledge of Django internals and time availability), claim it by following these steps:

- [Create an account](#) to use in our ticket system.
- If a ticket for this issue doesn't exist yet, create one in our [ticket tracker](#).

- If a ticket for this issue already exists, make sure nobody else has claimed it. To do this, look at the “Assigned to” section of the ticket. If it’s assigned to “nobody,” then it’s available to be claimed. Otherwise, somebody else is working on this ticket, and you either find another bug/feature to work on, or contact the developer working on the ticket to offer your help.
- Log into your account, if you haven’t already, by clicking “Login” in the upper right of the ticket page.
- Claim the ticket by clicking the radio button next to “Accept ticket” near the bottom of the page, then clicking “Submit changes.”

Ticket claimers’ responsibility

Once you’ve claimed a ticket, you have a responsibility to work on that ticket in a reasonably timely fashion. If you don’t have time to work on it, either unclaim it or don’t claim it in the first place!

Ticket triagers go through the list of claimed tickets from time to time, checking whether any progress has been made. If there’s no sign of progress on a particular claimed ticket for a week or two, a triager may ask you to relinquish the ticket claim so that it’s no longer monopolized and somebody else can claim it.

If you’ve claimed a ticket and it’s taking a long time (days or weeks) to code, keep everybody updated by posting comments on the ticket. If you don’t provide regular updates, and you don’t respond to a request for a progress report, your claim on the ticket may be revoked. As always, more communication is better than less communication!

Which tickets should be claimed?

Of course, going through the steps of claiming tickets is overkill in some cases. In the case of small changes, such as typos in the documentation or small bugs that will only take a few minutes to fix, you don’t need to jump through the hoops of claiming tickets. Just submit your patch and be done with it.

Patch style

- Make sure your code matches our *coding style*.
- Submit patches in the format returned by the `svn diff` command. An exception is for code changes that are described more clearly in plain English than in code. Indentation is the most common example; it’s hard to read patches when the only difference in code is that it’s indented.

Patches in `git diff` format are also acceptable.

- When creating patches, always run `svn diff` from the top-level `trunk` directory – i.e., the one that contains `django`, `docs`, `tests`, `AUTHORS`, etc. This makes it easy for other people to apply your patches.
- Attach patches to a ticket in the [ticket tracker](#), using the “attach file” button. Please *don’t* put the patch in the ticket description or comment unless it’s a single line patch.
- Name the patch file with a `.diff` extension; this will let the ticket tracker apply correct syntax highlighting, which is quite helpful.
- Check the “Has patch” box on the ticket details. This will make it obvious that the ticket includes a patch, and it will add the ticket to the [list of tickets with patches](#).
- The code required to fix a problem or add a feature is an essential part of a patch, but it is not the only part. A good patch should also include a regression test to validate the behavior that has been fixed (and prevent the problem from arising again).
- If the code associated with a patch adds a new feature, or modifies behavior of an existing feature, the patch should also contain documentation.

Non-trivial patches

A “non-trivial” patch is one that is more than a simple bug fix. It’s a patch that introduces Django functionality and makes some sort of design decision.

If you provide a non-trivial patch, include evidence that alternatives have been discussed on [django-developers](#). If you’re not sure whether your patch should be considered non-trivial, just ask.

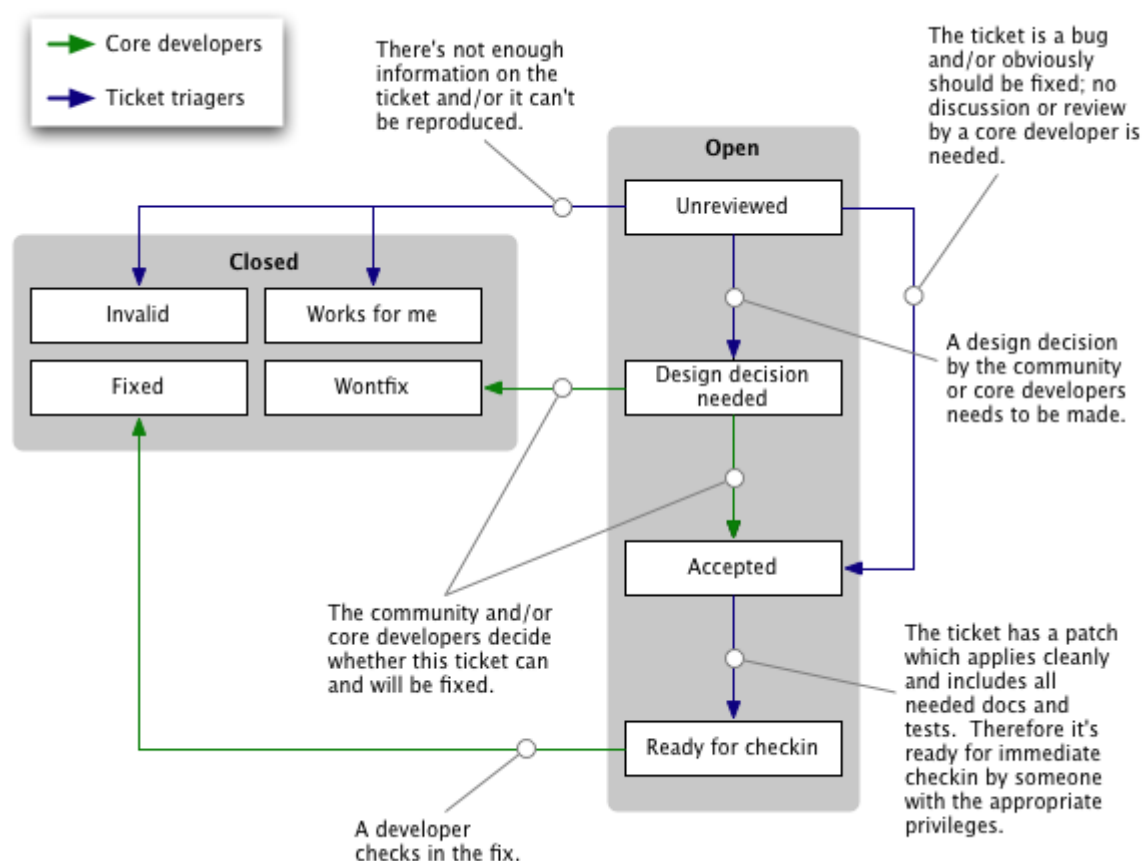
Ticket triage

Unfortunately, not all bug reports in the [ticket tracker](#) provide all the *required details*. A number of tickets have patches, but those patches don’t meet all the requirements of a *good patch*.

One way to help out is to *triage* bugs that have been reported by other users. A couple of dedicated volunteers work on this regularly, but more help is always appreciated.

Most of the workflow is based around the concept of a ticket’s “triage stage”. This stage describes where in its lifetime a given ticket is at any time. Along with a handful of flags, this field easily tells us what and who each ticket is waiting on.

Since a picture is worth a thousand words, let’s start there:



We’ve got two official roles here:

- Core developers: people with commit access who make the big decisions and write the bulk of the code.
- Ticket triagers: trusted community members with a proven history of working with the Django community. As a result of this history, they have been entrusted by the core developers to make some of the smaller decisions about tickets.

Second, note the five triage stages:

1. A ticket starts as “Unreviewed”, meaning that nobody has examined the ticket.
2. “Design decision needed” means “this concept requires a design decision,” which should be discussed either in the ticket comments or on [django-developers](#). The “Design decision needed” step will generally only be used for feature requests. It can also be used for issues that *might* be bugs, depending on opinion or interpretation. Obvious bugs (such as crashes, incorrect query results, or non-compliance with a standard) skip this step and move straight to “Accepted”.
3. Once a ticket is ruled to be approved for fixing, it’s moved into the “Accepted” stage. This stage is where all the real work gets done.
4. In some cases, a ticket might get moved to the “Someday/Maybe” state. This means the ticket is an enhancement request that we might consider adding to the framework if an excellent patch is submitted. These tickets are not a high priority.
5. If a ticket has an associated patch (see below), a triager will review the patch. If the patch is complete, it’ll be marked as “ready for checkin” so that a core developer knows to review and check in the patches.

The second part of this workflow involves a set of flags that describe what the ticket has or needs in order to be “ready for checkin”:

“Has patch” This means the ticket has an associated [patch](#). These will be reviewed by the triage team to see if the patch is “good”.

“Needs documentation” This flag is used for tickets with patches that need associated documentation. Complete documentation of features is a prerequisite before we can check a fix into the codebase.

“Needs tests” This flags the patch as needing associated unit tests. Again, this is a required part of a valid patch.

“Patch needs improvement” This flag means that although the ticket *has* a patch, it’s not quite ready for checkin. This could mean the patch no longer applies cleanly, or that the code doesn’t live up to our standards.

A ticket can be resolved in a number of ways:

“fixed” Used by one of the core developers once a patch has been rolled into Django and the issue is fixed.

“invalid” Used if the ticket is found to be incorrect. This means that the issue in the ticket is actually the result of a user error, or describes a problem with something other than Django, or isn’t a bug report or feature request at all (for example, some new users submit support queries as tickets).

“wontfix” Used when a core developer decides that this request is not appropriate for consideration in Django. This is usually chosen after discussion in the [django-developers](#) mailing list, and you should feel free to join in when it’s something you care about.

“duplicate” Used when another ticket covers the same issue. By closing duplicate tickets, we keep all the discussion in one place, which helps everyone.

“worksforme” Used when the ticket doesn’t contain enough detail to replicate the original bug.

If you believe that the ticket was closed in error – because you’re still having the issue, or it’s popped up somewhere else, or the triagers have – made a mistake, please reopen the ticket and tell us why. Please do not reopen tickets that have been marked as “wontfix” by core developers.

Triage by the general community

Although the core developers and ticket triagers make the big decisions in the ticket triage process, there’s also a lot that general community members can do to help the triage process. In particular, you can help out by:

- Closing “Unreviewed” tickets as “invalid”, “worksforme” or “duplicate.”
- Promoting “Unreviewed” tickets to “Design decision needed” if a design decision needs to be made, or “Accepted” in case of obvious bugs.
- Correcting the “Needs tests”, “Needs documentation”, or “Has patch” flags for tickets where they are incorrectly set.

- Checking that old tickets are still valid. If a ticket hasn't seen any activity in a long time, it's possible that the problem has been fixed but the ticket hasn't yet been closed.
- Contacting the owners of tickets that have been claimed but have not seen any recent activity. If the owner doesn't respond after a week or so, remove the owner's claim on the ticket.
- Identifying trends and themes in the tickets. If there a lot of bug reports about a particular part of Django, it may indicate we should consider refactoring that part of the code. If a trend is emerging, you should raise it for discussion (referencing the relevant tickets) on [django-developers](#).

However, we do ask the following of all general community members working in the ticket database:

- Please **don't** close tickets as "wontfix." The core developers will make the final determination of the fate of a ticket, usually after consultation with the community.
- Please **don't** promote tickets to "Ready for checkin" unless they are *trivial* changes – for example, spelling mistakes or broken links in documentation.
- Please **don't** reverse a decision that has been made by a core developer. If you disagree with a discussion that has been made, please post a message to [django-developers](#).
- Please be conservative in your actions. If you're unsure if you should be making a change, don't make the change – leave a comment with your concerns on the ticket, or post a message to [django-developers](#).

Submitting and maintaining translations

Various parts of Django, such as the admin site and validation error messages, are internationalized. This means they display different text depending on a user's language setting. For this, Django uses the same internationalization infrastructure that is available to Django applications that is described in the [i18n documentation](#).

These translations are contributed by Django users worldwide. If you find an incorrect translation, or if you'd like to add a language that isn't yet translated, here's what to do:

- Join the [Django i18n mailing list](#) and introduce yourself.
- Create translations using the methods described in the [i18n documentation](#). For this you will use the `django-admin.py makemessages` tool. In this particular case it should be run from the top-level `django` directory of the Django source tree.

The script runs over the entire Django source tree and pulls out all strings marked for translation. It creates (or updates) a message file in the directory `conf/locale` (for example for `pt-BR`, the file will be `conf/locale/pt-br/LC_MESSAGES/django.po`).

- Make sure that `django-admin.py compilemessages -l <lang>` runs without producing any warnings.
- Repeat the last two steps for the `djangojs` domain (by appending the `-d djangojs` command line option to the `django-admin.py` invocations).
- Create a diff of the `.po` file(s) against the current Subversion trunk.
- Open a ticket in Django's ticket system, set its `Component` field to `Translations`, and attach the patch to it.

Coding style

Please follow these coding standards when writing code for inclusion in Django:

- Unless otherwise specified, follow [PEP 8](#).
You could use a tool like [pep8.py](#) to check for some problems in this area, but remember that PEP 8 is only a guide, so respect the style of the surrounding code as a primary goal.
- Use four spaces for indentation.

- Use underscores, not camelCase, for variable, function and method names (i.e. `poll.get_unique_voters()`, not `poll.getUniqueVoters`).
- Use `InitialCaps` for class names (or for factory functions that return classes).
- Mark all strings for internationalization; see the *118n documentation* for details.
- In docstrings, use “action words” such as:

```
def foo():
    """
    Calculates something and returns the result.
    """
    pass
```

Here’s an example of what not to do:

```
def foo():
    """
    Calculate something and return the result.
    """
    pass
```

- Please don’t put your name in the code you contribute. Our policy is to keep contributors’ names in the `AUTHORS` file distributed with Django – not scattered throughout the codebase itself. Feel free to include a change to the `AUTHORS` file in your patch if you make more than a single trivial change.

Template style

- In Django template code, put one (and only one) space between the curly brackets and the tag contents.

Do this:

```
{{ foo }}
```

Don’t do this:

```
{{foo}}
```

View style

- In Django views, the first parameter in a view function should be called `request`.

Do this:

```
def my_view(request, foo):
    # ...
```

Don’t do this:

```
def my_view(req, foo):
    # ...
```

Model style

- Field names should be all lowercase, using underscores instead of camelCase.

Do this:

```
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)
```

Don't do this:

```
class Person(models.Model):
    First_Name = models.CharField(max_length=20)
    Last_Name = models.CharField(max_length=40)
```

- The class `Meta` should appear *after* the fields are defined, with a single blank line separating the fields and the class definition.

Do this:

```
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)

    class Meta:
        verbose_name_plural = 'people'
```

Don't do this:

```
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)
    class Meta:
        verbose_name_plural = 'people'
```

Don't do this, either:

```
class Person(models.Model):
    class Meta:
        verbose_name_plural = 'people'

    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)
```

- The order of model inner classes and standard methods should be as follows (noting that these are not all required):
 - All database fields
 - Custom manager attributes
 - class `Meta`
 - def `__unicode__()`
 - def `__str__()`
 - def `save()`
 - def `get_absolute_url()`
 - Any custom methods
- If `choices` is defined for a given model field, define the choices as a tuple of tuples, with an all-uppercase name, either near the top of the model module or just above the model class. Example:

```
GENDER_CHOICES = (
    ('M', 'Male'),
    ('F', 'Female'),
)
```

Documentation style

We place a high importance on consistency and readability of documentation. (After all, Django was created in a journalism environment!)

How to document new features

We treat our documentation like we treat our code: we aim to improve it as often as possible. This section explains how writers can craft their documentation changes in the most useful and least error-prone ways.

Documentation changes come in two forms:

- General improvements – Typo corrections, error fixes and better explanations through clearer writing and more examples.
- New features – Documentation of features that have been added to the framework since the last release.

Our policy is:

All documentation of new features should be written in a way that clearly designates the features are only available in the Django development version. Assume documentation readers are using the latest release, not the development version.

Our preferred way for marking new features is by prefacing the features' documentation with: `“.. versionadded:: X.Y”`, followed by an optional one line comment and a mandatory blank line.

General improvements, or other changes to the APIs that should be emphasized should use the `“.. versionchanged:: X.Y”` directive (with the same format as the `versionadded` mentioned above).

There's a full page of information about the [Django documentation system](#) that you should read prior to working on the documentation.

Guidelines for ReST files

These guidelines regulate the format of our ReST documentation:

- In section titles, capitalize only initial words and proper nouns.
- Wrap the documentation at 80 characters wide, unless a code example is significantly less readable when split over two lines, or for another good reason.

Commonly used terms

Here are some style guidelines on commonly used terms throughout the documentation:

- **Django** – when referring to the framework, capitalize Django. It is lowercase only in Python code and in the `django`project.com logo.
- **e-mail** – it has a hyphen.
- **MySQL**
- **PostgreSQL**
- **Python** – when referring to the language, capitalize Python.
- **realize**, **customize**, **initialize**, etc. – use the American “ize” suffix, not “ise.”
- **SQLite**
- **subclass** – it's a single word without a hyphen, both as a verb (“subclass that model”) and as a noun (“create a subclass”).
- **Web**, **World Wide Web**, **the Web** – note Web is always capitalized when referring to the World Wide Web.

- **Web site** – use two words, with Web capitalized.

Django-specific terminology

- **model** – it’s not capitalized.
- **template** – it’s not capitalized.
- **URLconf** – use three capitalized letters, with no space before “conf.”
- **view** – it’s not capitalized.

Committing code

Please follow these guidelines when committing code to Django’s Subversion repository:

- For any medium-to-big changes, where “medium-to-big” is according to your judgment, please bring things up on the [django-developers](#) mailing list before making the change.

If you bring something up on [django-developers](#) and nobody responds, please don’t take that to mean your idea is great and should be implemented immediately because nobody contested it. Django’s lead developers don’t have a lot of time to read mailing-list discussions immediately, so you may have to wait a couple of days before getting a response.

- Write detailed commit messages in the past tense, not present tense.
 - Good: “Fixed Unicode bug in RSS API.”
 - Bad: “Fixes Unicode bug in RSS API.”
 - Bad: “Fixing Unicode bug in RSS API.”
- For commits to a branch, prefix the commit message with the branch name. For example: “magic-removal: Added support for mind reading.”
- Limit commits to the most granular change that makes sense. This means, use frequent small commits rather than infrequent large commits. For example, if implementing feature X requires a small change to library Y, first commit the change to library Y, then commit feature X in a separate commit. This goes a *long way* in helping all core Django developers follow your changes.
- Separate bug fixes from feature changes.

Bug fixes need to be added to the current bugfix branch (e.g. the 1.0.X branch) as well as the current trunk.

- If your commit closes a ticket in the Django [ticket tracker](#), begin your commit message with the text “Fixed #abc”, where “abc” is the number of the ticket your commit fixes. Example: “Fixed #123 – Added support for foo”. We’ve rigged Subversion and Trac so that any commit message in that format will automatically close the referenced ticket and post a comment to it with the full commit message.

If your commit closes a ticket and is in a branch, use the branch name first, then the “Fixed #abc.” For example: “magic-removal: Fixed #123 – Added whizbang feature.”

For the curious: We’re using a [Trac post-commit hook](#) for this.

- If your commit references a ticket in the Django [ticket tracker](#) but does *not* close the ticket, include the phrase “Refs #abc”, where “abc” is the number of the ticket your commit references. We’ve rigged Subversion and Trac so that any commit message in that format will automatically post a comment to the appropriate ticket.

Unit tests

Django comes with a test suite of its own, in the `tests` directory of the Django tarball. It's our policy to make sure all tests pass at all times.

The tests cover:

- Models and the database API (`tests/modeltests/`).
- Everything else in core Django code (`tests/regressiontests`)
- Contrib apps (`django/contrib/<contribapp>/tests`, see below)

We appreciate any and all contributions to the test suite!

The Django tests all use the testing infrastructure that ships with Django for testing applications. See [Testing Django applications](#) for an explanation of how to write new tests.

Running the unit tests

To run the tests, `cd` to the `tests/` directory and type:

```
./runtests.py --settings=path.to.django.settings
```

Yes, the unit tests need a settings module, but only for database connection info, with the `DATABASE_ENGINE` setting.

If you're using the `sqlite3` database backend, no further settings are needed. A temporary database will be created in memory when running the tests.

If you're using another backend:

- Your `DATABASE_USER` setting needs to specify an existing user account for the database engine.
- The `DATABASE_NAME` setting must be the name of an existing database to which the given user has permission to connect. The unit tests will not touch this database; the test runner creates a new database whose name is `DATABASE_NAME` prefixed with `test_`, and this test database is deleted when the tests are finished. This means your user account needs permission to execute `CREATE DATABASE`.

You will also need to ensure that your database uses UTF-8 as the default character set. If your database server doesn't use UTF-8 as a default charset, you will need to include a value for `TEST_DATABASE_CHARSET` in your settings file.

If you want to run the full suite of tests, you'll need to install a number of dependencies:

- [PyYAML](#)
- [Markdown](#)
- [Textile](#)
- [Docutils](#)
- [setuptools](#)
- [memcached](#), plus the either the [python-memcached](#) or [cmemcached](#) Python binding

If you want to test the memcached cache backend, you will also need to define a `CACHE_BACKEND` setting that points at your memcached instance.

Each of these dependencies is optional. If you're missing any of them, the associated tests will be skipped.

To run a subset of the unit tests, append the names of the test modules to the `runtests.py` command line. See the list of directories in `tests/modeltests` and `tests/regressiontests` for module names.

As an example, if Django is not in your `PYTHONPATH`, you placed `settings.py` in the `tests/` directory, and you'd like to only run tests for generic relations and internationalization, type:


```
PYTHONPATH=..
./runtests.py --settings=settings generic_relations i18n
```

Contrib apps

Tests for apps in `django/contrib/` go in their respective directories under `django/contrib/`, in a `tests.py` file. (You can split the tests over multiple modules by using a `tests` directory in the normal Python way.)

For the tests to be found, a `models.py` file must exist (it doesn't have to have anything in it). If you have URLs that need to be mapped, put them in `tests/urls.py`.

To run tests for just one contrib app (e.g. `markup`), use the same method as above:

```
./runtests.py --settings=settings markup
```

Requesting features

We're always trying to make Django better, and your feature requests are a key part of that. Here are some tips on how to most effectively make a request:

- Request the feature on [django-developers](#), not in the ticket tracker; it'll get read more closely if it's on the mailing list.
- Describe clearly and concisely what the missing feature is and how you'd like to see it implemented. Include example code (non-functional is OK) if possible.
- Explain *why* you'd like the feature. In some cases this is obvious, but since Django is designed to help real developers get real work done, you'll need to explain it, if it isn't obvious why the feature would be useful.

As with most open-source projects, code talks. If you are willing to write the code for the feature yourself or if (even better) you've already written it, it's much more likely to be accepted. If it's a large feature that might need multiple developers we're always happy to give you an experimental branch in our repository; see below.

Branch policy

In general, the trunk must be kept stable. People should be able to run production sites against the trunk at any time. Additionally, commits to trunk ought to be as atomic as possible – smaller changes are better. Thus, large feature changes – that is, changes too large to be encapsulated in a single patch, or changes that need multiple eyes on them – must happen on dedicated branches.

This means that if you want to work on a large feature – anything that would take more than a single patch, or requires large-scale refactoring – you need to do it on a feature branch. Our development process recognizes two options for feature branches:

1. Feature branches using a distributed revision control system like [Git](#), [Mercurial](#), [Bazaar](#), etc.

If you're familiar with one of these tools, this is probably your best option since it doesn't require any support or buy-in from the Django core developers.

However, do keep in mind that Django will continue to use Subversion for the foreseeable future, and this will naturally limit the recognition of your branch. Further, if your branch becomes eligible for merging to trunk you'll need to find a core developer familiar with your DVCS of choice who'll actually perform the merge.

If you do decide to start a distributed branch of Django and choose to make it public, please add the branch to the [Django branches](#) wiki page.

2. Feature branches using SVN have a higher bar. If you want a branch in SVN itself, you'll need a "mentor" among the *core committers*. This person is responsible for actually creating the branch, monitoring your process (see below), and ultimately merging the branch into trunk.

If you want a feature branch in SVN, you'll need to ask in [django-developers](#) for a mentor.

Branch rules

We've got a few rules for branches born out of experience with what makes a successful Django branch.

DVCS branches are obviously not under central control, so we have no way of enforcing these rules. However, if you're using a DVCS, following these rules will give you the best chance of having a successful branch (read: merged back to trunk).

Developers with branches in SVN, however, **must** follow these rules. The branch mentor will keep an eye on the branch and **will delete it** if these rules are broken.

- Only branch entire copies of the Django tree, even if work is only happening on part of that tree. This makes it painless to switch to a branch.
- Merge changes from trunk no less than once a week, and preferably every couple-three days.

In our experience, doing regular trunk merges is often the difference between a successful branch and one that fizzles and dies.

If you're working on an SVN branch, you should be using [svnmerge.py](#) to track merges from trunk.

- Keep tests passing and documentation up-to-date. As with patches, we'll only merge a branch that comes with tests and documentation.

Once the branch is stable and ready to be merged into the trunk, alert [django-developers](#).

After a branch has been merged, it should be considered "dead"; write access to it will be disabled, and old branches will be periodically "trimmed." To keep our SVN wrangling to a minimum, we won't be merging from a given branch into the trunk more than once.

Using branches

To use a branch, you'll need to do two things:

- Get the branch's code through Subversion.
- Point your Python `site-packages` directory at the branch's version of the `django` package rather than the version you already have installed.

Getting the code from Subversion

To get the latest version of a branch's code, check it out using Subversion:

```
svn co http://code.djangoproject.com/svn/django/branches/<branch>/
```

...where `<branch>` is the branch's name. See the [list of branch names](#).

Alternatively, you can automatically convert an existing directory of the Django source code as long as you've checked it out via Subversion. To do the conversion, execute this command from within your `django` directory:

```
svn switch http://code.djangoproject.com/svn/django/branches/<branch>/
```

The advantage of using `svn switch` instead of `svn co` is that the `switch` command retains any changes you might have made to your local copy of the code. It attempts to merge those changes into the "switched" code. The disadvantage is that it may cause conflicts with your local changes if the "switched" code has altered the same lines of code.

(Note that if you use `svn switch`, you don't need to point Python at the new version, as explained in the next section.)

Pointing Python at the new Django version

Once you've retrieved the branch's code, you'll need to change your Python `site-packages` directory so that it points to the branch version of the `django` directory. (The `site-packages` directory is somewhere such as `/usr/lib/python2.4/site-packages` or `/usr/local/lib/python2.4/site-packages` or `C:\Python\site-packages`.)

The simplest way to do this is by renaming the old `django` directory to `django.OLD` and moving the trunk version of the code into the directory and calling it `django`.

Alternatively, you can use a symlink called `django` that points to the location of the branch's `django` package. If you want to switch back, just change the symlink to point to the old code.

A third option is to use a [path file](#) (`<something>.pth`) which should work on all systems (including Windows, which doesn't have symlinks available). First, make sure there are no files, directories or symlinks named `django` in your `site-packages` directory. Then create a text file named `django.pth` and save it to your `site-packages` directory. That file should contain a path to your copy of Django on a single line and optional comments. Here is an example that points to multiple branches. Just uncomment the line for the branch you want to use ('Trunk' in this example) and make sure all other lines are commented:

```
# Trunk is a svn checkout of:
# http://code.djangoproject.com/svn/django/trunk/
#
/path/to/trunk

# <branch> is a svn checkout of:
# http://code.djangoproject.com/svn/django/branches/<branch>/
#
/path/to/<branch>

# On windows a path may look like this:
# C:/path/to/<branch>
```

If you're using Django 0.95 or earlier and installed it using `python setup.py install`, you'll have a directory called something like `Django-0.95-py2.4.egg` instead of `django`. In this case, edit the file `setuptools.pth` and remove the line that references the Django `.egg` file. Then copy the branch's version of the `django` directory into `site-packages`.

Deciding on features

Once a feature's been requested and discussed, eventually we'll have a decision about whether to include the feature or drop it.

Whenever possible, we strive for a rough consensus. To that end, we'll often have informal votes on [django-developers](#) about a feature. In these votes we follow the voting style invented by Apache and used on Python itself, where votes are given as +1, +0, -0, or -1. Roughly translated, these votes mean:

- +1: "I love the idea and I'm strongly committed to it."
- +0: "Sounds OK to me."
- -0: "I'm not thrilled, but I won't stand in the way."
- -1: "I strongly disagree and would be very unhappy to see the idea turn into reality."

Although these votes on `django-developers` are informal, they'll be taken very seriously. After a suitable voting period, if an obvious consensus arises we'll follow the votes.

However, consensus is not always possible. Tough decisions will be discussed by all full committers and finally decided by the Benevolent Dictators for Life, Adrian and Jacob.

Commit access

Django has two types of committers:

Full committers These are people who have a long history of contributions to Django’s codebase, a solid track record of being polite and helpful on the mailing lists, and a proven desire to dedicate serious time to Django’s development.

The bar is very high for full commit access. It will only be granted by unanimous approval of all existing full committers, and the decision will err on the side of rejection.

Partial committers These are people who are “domain experts.” They have direct check-in access to the subsystems that fall under their jurisdiction, and they’re given a formal vote in questions that involve their subsystems. This type of access is likely to be given to someone who contributes a large subframework to Django and wants to continue to maintain it.

Like full committers, partial commit access is by unanimous approval of all full committers (and any other partial committers in the same area). However, the bar is set lower; proven expertise in the area in question is likely to be sufficient.

To request commit access, please contact an existing committer privately. Public requests for commit access are potential flame-war starters, and will be ignored.

Como a documentação do Django funciona

... e como contribuir.

A documentação do Django usa o sistema de documentação [Sphinx](#), que por sua vez é baseado no [docutils](#). A idéia básica é ter uma documentação leve em formato de texto-plano, que seja transformada em HTML, PDF, ou qualquer outro formato.

Na verdade para compilar a documentação localmente, você precisará instalar o `Sphinx – easy_install`. `Sphinx` deve fazer a mágica.

Somente então, transforme tudo em html, para isso é só executar `make html` dentro do diretório `docs`.

Para começar a contribuir, você precisará ler a [ReStructuredText Primer](#). Depois disso, você poderá ler sobre o [Sphinx-specific markup](#) que é utilizado para manipular os metadados, indexação, e cruzamento de referências.

A principal coisa que você deve manter em mente quando escrever ou editar a documentação é, quanto mais semântica você puder colocar, melhor. Então:

```
Adicione ``django.contrib.auth`` no seu ``INSTALLED_APPS``...
```

Não é tão útil quanto:

```
Adicione :mod:`django.contrib.auth` no seu :setting:`INSTALLED_APPS`...
```

Isto porque o `Sphinx` irá gerar links durante a compilação, que ajudarão muito os leitores. Basicamente não há limites para a quantidade de markup que você pode adicionar.

Markup específica do Django

Além das [markups built-in do Sphinx](#), a documentação do Django define algumas unidades de descrição extras:

- Settings:

```
.. setting:: INSTALLED_APPS
```

Para linkar o setting, use `:setting:`INSTALLED_APPS``.

- Tags de template:

```
.. templatetag:: regroup
```

Para linkar, use `:ttag:`regroup``.

- Filtros de template:

```
.. templatefilter:: linebreaksbr
```

Para linkar, use `:tfilter:`linebreaksbr``.

- Lookups de campos (i.e. `Foo.objects.filter(bar__exact=qualquercoisa)`):

```
.. fieldlookup:: exact
```

Para linkar, use `:lookup:`exact``.

- Comandos do `django-admin`:

```
.. django-admin:: syncdb
```

Para linkar, use `:dadmin:`syncdb``.

- Opções de linha de comando do `django-admin`:

```
.. django-admin-option:: --traceback
```

Para linkar, use `:dadminopt:`--traceback``.

Um Exemplo

Para um exemplo rápido de como tudo se agrupa, dá uma olhada nisso:

- Primeiro, o documentno `ref/settings.txt` começa, dessa forma:

```
.. _ref-settings:

Configurações disponíveis
=====

...
```

- Em seguida se olharmos para o documento `topics/settings`, você pode ver como um link para ``ref/settings`` funciona:

```
Configurações disponíveis
=====

Para uma lista completa das configurações disponíveis, veja
:ref:`referencia do settings <ref-settings>`.
```

- Depois, note como o `settings` (bem agora, no início) está anotado:

```
.. setting:: ADMIN_FOR

ADMIN_FOR
-----

Padrão: ``()`` (Tupla vazia)

Usado para definir módulos do site de administração. este deve ser uma
tupla de configurações de módulos (no formato ``'foo.bar.baz'``) para
que este site tenha uma amdinistração.
```

```
O site de administração usa isto em sua instrospecção automática da
documentação de models, views e tags de template.
```

Isto marca o seguinte cabeçalho como um alvo “canonico” para a configuração `ADMIN_FOR`. Isto significa que toda vez que você fores mencionar `ADMIN_FOR`, podes referenciá-lo utilizando `:setting:`ADMIN_FOR``.

Basicamente, é desta forma que tudo se encaixa.

TODO

O trabalho está na maioria feito, mas aqui tem o que falta, em ordem de prioridade.

- Corrigir a documentação das generic views: adaptar o capítulo 9 do Django Book (considerar esse item de TODO minha permissão e licença) em `topics/generic-views.txt`; remover o material de introdução de `ref/generic-views.txt` e apenas deixar a referência de funções.
- Mudar o “Added/changed na versão de desenvolvimento” chamar a diretiva adequada do Sphinx `.. versionadded::` ou `.. versionchanged::`.
- Checar links mal formados. Fazer isso rodando o `make linkcheck` e concertar todos os mais 300 errors/warnings.

Particularmente, olhar todos os links relativos; estes precisam ser modificados para as referências apropriadas.

- A maior parte dos documentos `index.txt` têm textos de introdução *muito* curtos, ou quase inexistem. Cada um destes documentos precisar de de um bom conteúdo de introdução.
- O glossário é muito superficial. Ele precisa ser preenchido.
- E mais alvos de metadados: há vários lugares que se parecem com isso:

```
`File.close()``
~~~~~
```

... e deveria ser:

```
.. method:: File.close()
```

Que é, usar os metadados no lugar de títulos.

- Adicionar mais links – quase tudo o que um código inline literal, provavelmente pode ser transformado em um xref.

Veja o arquivo `literals_to_xrefs.py` no `_ext` – este é um script shell para ajudar no trabalho.

Este será, ou provavelmente continuará sendo, um projeto sem fim.

- Adicionar [listas de informações de campos](#) onde for apropriado.
- Adicionar `.. code-block:: <lang>` para blocos literais para que eles sejam destacados.

Dicas

Algumas dicas para fazer as coisas parecerem melhor:

- Sempre que possível, use links. Então, use `:setting:`ADMIN_FOR`` ao invés de ``ADMIN_FOR``.

- Algumas diretivas (`.. setting::`, por exemplo) são diretivas de estilo prefixado; elas vão *antes* da unidade que estão descrevendo. Estas são conhecidas como diretivas “crossref”. Já outras (`.. class::`, ex.) geram suas próprias marcações; elas devem ir dentro da seção que estão descrevendo. São chamadas “unidades de descrição”.

Você pode dizer quem é quem olhando no `_ext/djangodocs.py`; que registra as regras tanto para uma quanto para outra.

- Quando estiver referindo-se a classes/funções/módulos, etc., você terá que usar um nome completo para o alvo (`:class:`django.contrib.contenttypes.models.ContentType``).

Uma vez que isto não pareça ser uma saída incrível – ela mostra o caminho completo para o objeto – você pode prefixar o alto com um `~` para usar somente o final do caminho. Então `:class:`~django.contrib.contenttypes.models.ContentType`` mostrará somente um link com o título “ContentType”.

Django committers

The original team

Django originally started at World Online, the Web department of the [Lawrence Journal-World](#) of Lawrence, Kansas, USA.

Adrian Holovaty Adrian is a Web developer with a background in journalism. He’s known in journalism circles as one of the pioneers of “journalism via computer programming”, and in technical circles as “the guy who invented Django.”

He was lead developer at World Online for 2.5 years, during which time Django was developed and implemented on World Online’s sites. He’s now the leader and founder of [EveryBlock](#), a “news feed for your block”.

Adrian lives in Chicago, USA.

Simon Willison Simon is a well-respected web developer from England. He had a one-year internship at World Online, during which time he and Adrian developed Django from scratch. The most enthusiastic Brit you’ll ever meet, he’s passionate about best practices in web development and maintains a well-read [web-development blog](#).

Simon lives in Brighton, England.

Jacob Kaplan-Moss Jacob is a partner at [Revolution Systems](#) which provides support services around Django and related open source technologies. A good deal of Jacob’s work time is devoted to working on Django. Jacob previously worked at World Online, where Django was invented, where he was the lead developer of Ellington, a commercial web publishing platform for media companies.

Jacob lives in Lawrence, Kansas, USA.

Wilson Miner Wilson’s design-fu is what makes Django look so nice. He designed the website you’re looking at right now, as well as Django’s acclaimed admin interface. Wilson is the designer for [EveryBlock](#).

Wilson lives in San Francisco, USA.

Current developers

Currently, Django is led by a team of volunteers from around the globe.

BDFLs

Adrian and Jacob are the Co-Benevolent Dictators for Life of Django. When “rough consensus and working code” fails, they’re the ones who make the tough decisions.

Core developers

These are the folks who have a long history of contributions, a solid track record of being helpful on the mailing lists, and a proven desire to dedicate serious time to Django. In return, they’ve been granted the coveted commit bit, and have free rein to hack on all parts of Django.

Malcolm Tredinnick Malcolm originally wanted to be a mathematician, somehow ended up a software developer. He’s contributed to many Open Source projects, has served on the board of the GNOME foundation, and will kick your ass at chess.

When he’s not busy being an International Man of Mystery, Malcolm lives in Sydney, Australia.

Russell Keith-Magee Russell studied physics as an undergraduate, and studied neural networks for his PhD. His first job was with a startup in the defense industry developing simulation frameworks. Over time, mostly through work with Django, he’s become more involved in web development.

Russell has helped with several major aspects of Django, including a couple major internal refactorings, creation of the test system, and more.

Russell lives in the most isolated capital city in the world — Perth, Australia.

Joseph Kocherhans Joseph is currently a developer at [EveryBlock](#), and previously worked for the Lawrence Journal-World where he built most of the backend for their Marketplace site. He often disappears for several days into the woods, attempts to teach himself computational linguistics, and annoys his neighbors with his [Charango](#) playing.

Joseph’s first contribution to Django was a series of improvements to the authorization system leading up to support for pluggable authorization. Since then, he’s worked on the new forms system, its use in the admin, and many other smaller improvements.

Joseph lives in Chicago, USA.

Luke Plant At University Luke studied physics and Materials Science and also met [Michael Meeks](#) who introduced him to Linux and Open Source, re-igniting an interest in programming. Since then he has contributed to a number of Open Source projects and worked professionally as a developer.

Luke has contributed many excellent improvements to Django, including database-level improvements, the CSRF middleware and many unit tests.

Luke currently works for a church in Bradford, UK, and part-time as a freelance developer.

Brian Rosner Brian is currently a web developer working on an e-commerce system in Django. He spends his free time contributing to Django and enjoys to learn more about programming languages and system architectures. Brian is the co-host of the weekly podcast, [This Week in Django](#).

Brian helped immensely in getting Django’s “newforms-admin” branch finished in time for Django 1.0; he’s now a full committer, continuing to improve on the admin and forms system.

Brian lives in Denver, USA.

Gary Wilson Gary starting contributing patches to Django in 2006 while developing Web applications for [The University of Texas](#) (UT). Since, he has made contributions to the e-mail and forms systems, as well as many other improvements and code cleanups throughout the code base.

Gary is currently a developer and software engineering graduate student at UT, where his dedication to spreading the ways of Python and Django never ceases.

Gary lives in Austin, Texas, USA.

Justin Bronn Justin Bronn is a computer scientist and attorney specializing in legal topics related to intellectual property and spatial law.

In 2007, Justin began developing `django.contrib.gis` in a branch, a.k.a. [GeoDjango](#), which was merged in time for Django 1.0. While implementing GeoDjango, Justin obtained a deep knowledge of Django's internals including the ORM, the admin, and Oracle support.

Justin lives in Houston, Texas.

Karen Tracey Karen has a background in distributed operating systems (graduate school), communications software (industry) and crossword puzzle construction (freelance). The last of these brought her to Django, in late 2006, when she set out to put a web front-end on her crossword puzzle database. That done, she stuck around in the community answering questions, debugging problems, etc. – because coding puzzles are as much fun as word puzzles.

Karen lives in Apex, NC, USA.

Specialists

James Bennett James is Django's release manager; he also contributes to the documentation.

James came to web development from philosophy when he discovered that programmers get to argue just as much while collecting much better pay. He lives in Lawrence, Kansas, where he works for the Journal-World developing Ellington. He [keeps a blog](#), has written a [book on Django](#), and enjoys fine port and talking to his car.

Ian Kelly Ian is responsible for Django's support for Oracle.

Matt Boersma Matt is also responsible for Django's Oracle support.

Jeremy Dunck Jeremy the lead developer of Pegasus News, a personalized local site based in Dallas, Texas. An early contributor to Greasemonkey and Django, he sees technology as a tool for communication and access to knowledge.

Jeremy helped kick off GeoDjango development, and is mostly responsible for the serious speed improvements that signals received in Django 1.0.

Jeremy lives in Dallas, Texas, USA.

Developers Emeritus

Georg “Hugo” Bauer Georg created Django's internationalization system, managed i18n contributions and made a ton of excellent tweaks, feature additions and bug fixes.

Robert Wittams Robert was responsible for the *first* refactoring of Django's admin application to allow for easier reuse and has made a ton of excellent tweaks, feature additions and bug fixes.

Django's release process

Official releases

Django's release numbering works as follows:

- Versions are numbered in the form `A.B` or `A.B.C`.
- `A` is the *major version* number, which is only incremented for major changes to Django, and these changes are not necessarily backwards-compatible. That is, code you wrote for Django 6.0 may break when we release Django 7.0.
- `B` is the *minor version* number, which is incremented for large yet backwards compatible changes. Code written for Django 6.4 will continue to work under Django 6.5.
- `C` is the *micro version* number which, is incremented for bug and security fixes. A new micro-release will always be 100% backwards-compatible with the previous micro-release.
- In some cases, we'll make alpha, beta, or release candidate releases. These are of the form `A.B alpha/beta/rc N`, which means the `N`th alpha/beta/release candidate of version `A.B`.

An exception to this version numbering scheme is the pre-1.0 Django code. There's no guarantee of backwards-compatibility until the 1.0 release.

In Subversion, each Django release will be tagged under `tags/releases`. If it's necessary to release a bug fix release or a security release that doesn't come from the trunk, we'll copy that tag to `branches/releases` to make the bug fix release.

Major releases

Major releases (1.0, 2.0, etc.) will happen very infrequently (think “years”, not “months”), and will probably represent major, sweeping changes to Django.

Minor releases

Minor release (1.1, 1.2, etc.) will happen roughly every six months – see [release process](#), below for details. These releases will contain new features, improvements to existing features, and such. A minor release may deprecate certain features from previous releases. If a feature in version `A.B` is deprecated, it will continue to work in

version A.B+1. In version A.B+2, use of the feature will raise a `PendingDeprecationWarning` but will continue to work. Version A.B+3 will remove the feature entirely.

So, for example, if we decided to remove a function that existed in Django 1.0:

- Django 1.1 will contain a backwards-compatible replica of the function which will raise a `PendingDeprecationWarning`. This warning is silent by default; you need to explicitly turn on display of these warnings.
- Django 1.2 will contain the backwards-compatible replica, but the warning will be promoted to a full-fledged `DeprecationWarning`. This warning is *loud* by default, and will likely be quite annoying.
- Django 1.3 will remove the feature outright.

Micro releases

Micro releases (1.0.1, 1.0.2, 1.1.1, etc.) will be issued at least once half-way between minor releases, and probably more often as needed.

These releases will always be 100% compatible with the associated minor release – the answer to “should I upgrade to the latest micro release?” will always be “yes.”

Each minor release of Django will have a “release maintainer” appointed. This person will be responsible for making sure that bug fixes are applied to both trunk and the maintained micro-release branch. This person will also work with the release manager to decide when to release the micro releases.

Supported versions

At any moment in time, Django’s developer team will support a set of releases to varying levels:

- The current development trunk will get new features and bug fixes requiring major refactoring.
- All bug fixes applied to the trunk will also be applied to the last minor release, to be released as the next micro release.
- Security fixes will be applied to the current trunk and the previous two minor releases.

As a concrete example, consider a moment in time halfway between the release of Django 1.3 and 1.4. At this point in time:

- Features will be added to development trunk, to be released as Django 1.4.
- Bug fixes will be applied to a `1.3.X` branch, and released as 1.3.1, 1.3.2, etc.
- Security releases will be applied to trunk, a `1.3.X` branch and a `1.2.X` branch. Security fixes will trigger the release of `1.3.1`, `1.2.1`, etc.

Release process

Django uses a time-based release schedule, with minor (i.e. 1.1, 1.2, etc.) releases every six months, or more, depending on features.

After each previous release (and after a suitable cooling-off period of a week or two), the core development team will examine the landscape and announce a timeline for the next release. Most releases will be scheduled in the 6-9 month range, but if we have bigger features to development we might schedule a longer period to allow for more ambitious work.

Release cycle

Each release cycle will be split into three periods, each lasting roughly one-third of the cycle:

Phase one: feature proposal

The first phase of the release process will be devoted to figuring out what features to include in the next version. This should include a good deal of preliminary work on those features – working code trumps grand design.

At the end of part one, the core developers will propose a feature list for the upcoming release. This will be broken into:

- “Must-have”: critical features that will delay the release if not finished
- “Maybe” features: that will be pushed to the next release if not finished
- “Not going to happen”: features explicitly deferred to a later release.

Anything that hasn’t got at least some work done by the end of the first third isn’t eligible for the next release; a design alone isn’t sufficient.

Phase two: development

The second third of the release schedule is the “heads-down” working period. Using the roadmap produced at the end of phase one, we’ll all work very hard to get everything on it done.

Longer release schedules will likely spend more than a third of the time in this phase.

At the end of phase two, any unfinished “maybe” features will be postponed until the next release. Though it shouldn’t happen, any “must-have” features will extend phase two, and thus postpone the final release.

Phase two will culminate with an alpha release.

Phase three: bugfixes

The last third of a release is spent fixing bugs – no new features will be accepted during this time. We’ll release a beta release about halfway through, and an rc complete with string freeze two weeks before the end of the schedule.

Bug-fix releases

After a minor release (i.e 1.1), the previous release will go into bug-fix mode.

A branch will be created of the form `branches/releases/1.0.X` to track bug-fixes to the previous release. When possible, bugs fixed on trunk must *also* be fixed on the bug-fix branch; this means that commits need to cleanly separate bug fixes from feature additions. The developer who commits a fix to trunk will be responsible for also applying the fix to the current bug-fix branch. Each bug-fix branch will have a maintainer who will work with the committers to keep them honest on backporting bug fixes.

How this all fits together

Let’s look at a hypothetical example for how this all fits together. Imagine, if you will, a point about halfway between 1.1 and 1.2. At this point, development will be happening in a bunch of places:

- On trunk, development towards 1.2 proceeds with small additions, bugs fixes, etc. being checked in daily.
- On the branch “`branches/releases/1.1.X`”, bug fixes found in the 1.1 release are checked in as needed. At some point, this branch will be released as “1.1.1”, “1.1.2”, etc.
- On the branch “`branches/releases/1.0.X`”, security fixes are made if needed and released as “1.0.2”, “1.0.3”, etc.
- On feature branches, development of major features is done. These branches will be merged into trunk before the end of phase two.

Agenda de Obsolescência do Django

Esse documento descreve quando várias partes do serão removidas, seguindo sua obsolescência, como descrito na *Política de obsolescência do Django*

- **1.3**

- `AdminSite.root()`. Essa versão irá remover o método antigo para habilitar URLs administrativas. Isso foi tornado obsoleto desde a versão 1.1.

- **2.0**

- `django.views.defaults.shortcut()`. Essa função foi movida para `django.contrib.contenttypes.views.shortcut()` como parte do plano de remover todas as referências a `django.contrib` do núcleo do Django. O atalho antigo será removido na versão 2.0.

Part X

Índices, glossário e tabelas

- [genindex](#)
- [modindex](#)
- [Glossário](#)

Part XI

Documentação depreciada/obsoleta

A documentação a seguir cobre funcionalidades que estão entrando em desuso ou que serão substituídas nas novas versões do Django.

Documentação depreciada/obsoleto

Estes documentos são sobre funcionalidades que são depreciadas ou que serão substituídas nas novas versões do Django. Elas são preservadas aqui para quem estiver usando versões mais antigas do Django, ou que ainda usa APIs depreciadas. Nenhum código novo baseado nestas APIs deverá ser escrito

Customizando a interface de administração do Django

Warning: O design da administração mudou um pouco desde que a documentação foi escrita, e algumas partes podem não se aplicar mais. Este documento não é mais mantido, uma vez que uma API para customização da interface de administração do Django está em desenvolvimento.

A interface de administração dinâmica do Django oferece uma administração gratuita, plenamente funcional, sem a necessidade de escrita de código. A administração dinâmica é projetada para estar pronta para produção, não somente um ponto de partida, então você pode usá-la em um site real. Embora o formato subjacente do admin seja construído em páginas do Django, você pode customizar o visual editando os estilos e imagens.

Eis uma breve visão sobre alguns dos principais estilos e classes utilizadas no CSS do admin do Django.

Módulos

A classe `.module` é um bloco de construção básico para agrupar conteúdos no admin. Ela geralmente é aplicada em um `div` ou um `fieldset`, envolvendo os conteúdos do grupo em uma caixa e aplicando certos estilos sobre os elementos internos. Um `h2` dentro de um `div.module` irá alinhar-se ao topo do `div` como um cabeçalho para o grupo inteiro.

Core		
Groups		
Users		
Redirects		
Flat files		

Tipos de Coluna

Note: Todas as páginas do admin (exceto o dashboard) são largura fluída (se ajustam ao navegador). Todas as classes de largura fixa de versões anteriores do Django foram removidas.

O template base para cada página admin possui um bloco que define a estrutura de colunas da página. Ele configura uma classe na área de conteúdo da página (`div#content`) de modo que tudo dentro da página sabe quão largo deve ser. Existem três tipos de colunas disponíveis.

colM Esta é a configuração padrão para colunas para todas as páginas. O “M” vem de “Main”. Assume que todo conteúdo na página está em uma coluna principal (`div#content-main`).

colMS Esta é para páginas com uma coluna principal e uma “sidebar” (coluna lateral) à direita. O “S” vem de “sidebar”. Assume-se que o conteúdo principal está em `div#content-main` e o conteúdo da “sidebar” está em `div#content-related`. Isto é usado na página principal do admin.

colSM O mesmo descrito acima, com a “sidebar” à esquerda. A ordem das colunas não importa.

Por exemplo, você pode colocar isto em um template para fazer uma página de duas colunas, com uma coluna lateral à direita:

```
.. code-block:: html+django
```

```
{% block coltype %}colMS{% endblock %}
```

Estilos de Texto

Tamanhos de Font

A maioria dos elementos HTML (cabecinhos, listas, etc.) possui um tamanho de fonte básico, de acordo com o contexto de estilos. Existem três classes disponíveis para forçar um texto a ter um certo tamanho em qualquer contexto.

small 11px

tiny 10px

mini 9px (use com sabedoria)

Estilos de Fontes e Alinhamento

Existem também alguns poucos para estilizar textos.

.quiet Configura a cor da fonte para um cinza claro. Bom para notas laterais em instruções. Combine com `.small` ou `.tiny` por pura emoção.

.help Esta é uma classe customizada para blocos de ajuda em linha, como o texto que explica a função dos elementos dos formulários. Ela faz o texto ficar pequeno e cinza, e quando aplicado para `p` dentro de um elemento `.form-row` (veja estilos de formulário abaixo), ele será alinhado com o campo de formulário. Use isto para texto de ajuda, em vez de `small quiet`. Esta classe funciona em outros elementos, mas tente colocá-la em elementos `p` sempre que puder.

.align-left Esta alinha o texto à esquerda. Somente funciona em elementos de bloco contendo elementos em linha.

.align-right Você está prestando atenção?

.nowrap Mantém o texto e objetos em linha sem quebra de linha. É útil para cabecinhos de tabelas, que você deseja manter em uma linha somente.

Floats e Clears

float-left Flutua à esquerda

float-right Flutua à direita

clear Limpa tudo

Ferramentas de Objetos

Algumas ações que se aplicam diretamente a objetos são usadas em páginas de formulário e de listagens. Estas aparecem em uma “barra de ferramentas” acima do formulário ou listagem, à direita da página. As ferramentas ficam envolvidos em um `ul` com a classe `object-tools`. Há dois tipos de ferramentas customizadas que podem ser definidas com uma classe adicional sobre o `a` para a ferramenta. São elas `.addlink` e `.viewsitelink`.

Exemplo de uma página de listagem:

```
<ul class="object-tools">
  <li><a href="/stories/add/" class="addlink">Adicionar redirecionador</a></li>
</ul>
```



and from a form page:

```
<ul class="object-tools">
  <li><a href="/history/303/152383/">History</a></li>
  <li><a href="/r/303/152383/" class="viewsitelink">View on site</a></li>
</ul>
```



Estilos de formulário

Fieldsets

Os formulários do admin são separados em grupos por meio de elementos `fieldset`. Cada `fieldset` de formulário deve ter uma classe `.module`. Cada `fieldset` deve também ter um cabeçalho `h2` dentro e no topo (exceto o primeiro grupo do formulário, e em alguns casos onde o grupo de campos não possui uma nomenclatura lógica).

Cada `fieldset` pode ainda receber classes extras, além de `.module` para aplicar a formatação apropriada para o grupo de campos.

.aligned Isto irá alinhar as labels e inputs lado a lado na mesma linha.

.wide Usado na combinação com `.aligned` para ampliar o espaço disponível para as labels.

Linhas de formulário

Cada linha do formulário (dentro de um `fieldset`) deve estar em um `div` com a classe `form-row`. Se o campo na linha é obrigatório, a classe `required` também deverá ser adicionada ao `div.form-row`.

Site:

Redirect from: **div class="form-row required"**
This should be an absolute path, excluding the domain name. Example: "/events/search/".

Redirect to:
This can be either an absolute path (as above) or a full URL starting with "http://".

[x Delete](#) [Save and add another](#) [Save and continue editing](#) [Save](#)

Labels

As labels de formulários devem sempre preceder o campo, exceto no caso de ser um checkbox ou radio, onde o `input` deve vir primeiro. Qualquer explicação ou texto de ajuda deve seguir o `label` em um `p` com a classe `.help`.

d

- `django.contrib.admin`, 357
- `django.contrib.auth`, 195
- `django.contrib.auth.forms`, 206
- `django.contrib.auth.middleware`, 509
- `django.contrib.comments`, 375
- `django.contrib.comments.models`, 378
- `django.contrib.comments.signals`, 379
- `django.contrib.contenttypes`, 381
- `django.contrib.csrf`, 386
- `django.contrib.csrf.middleware`, 509
- `django.contrib.databrowse`, 386
- `django.contrib.flatpages`, 388
- `django.contrib.formtools`, 390
- `django.contrib.formtools.wizard`, 391
- `django.contrib.humanize`, 395
- `django.contrib.localflavor`, 397
- `django.contrib.redirects`, 404
- `django.contrib.sessions`, 122
- `django.contrib.sessions.middleware`, 509
- `django.contrib.sitemaps`, 405
- `django.contrib.sites`, 410
- `django.contrib.syndication`, 415
- `django.contrib.webdesign`, 429
- `django.core.files`, 455
- `django.core.mail`, 227
- `django.core.paginator`, 249
- `django.core.signals`, 582
- `django.core.urlresolvers`, 99
- `django.db.models`, 61
- `django.db.models.fields`, 511
- `django.db.models.fields.related`, 521
- `django.db.models.signals`, 579
- `django.dispatch`, 263
- `django.forms.fields`, 471
- `django.forms.widgets`, 483
- `django.http`, 553
- `django.middleware`, 507
- `django.middleware.cache`, 507
- `django.middleware.common`, 507
- `django.middleware.doc`, 508
- `django.middleware.gzip`, 508
- `django.middleware.http`, 508
- `django.middleware.locale`, 508
- `django.middleware.transaction`, 509
- `django.test`, 179
- `django.test.client`, 184
- `django.test.signals`, 582
- `django.test.utils`, 193
- `django.views.static`, 331

Symbols

-adminmedia
 django-admin command-line option, 448
 -email
 django-admin command-line option, 443
 -exclude
 django-admin command-line option, 444
 -format
 django-admin command-line option, 444
 -help
 django-admin command-line option, 442
 -indent
 django-admin command-line option, 444
 -noinput
 django-admin command-line option, 444
 -noreload
 django-admin command-line option, 448
 -username
 django-admin command-line option, 443
 -verbosity
 django-admin command-line option, 442
 -version
 django-admin command-line option, 442
 __contains__() (QueryDict method), 556
 __delitem__() (HttpResponse method), 559
 __getitem__() (HttpResponse method), 559
 __getitem__() (QueryDict method), 556
 __init__() (HttpResponse method), 559
 __init__() (SyndicationFeed method), 427
 __iter__() (File method), 455
 __setitem__() (HttpResponse method), 559
 __setitem__() (QueryDict method), 556
 __str__() (Model method), 530
 __unicode__() (Model method), 531

A

ABSOLUTE_URL_OVERRIDES
 setting, 561
 abstract (Options attribute), 525
 add
 template filter, 594
 add() (QuerySet method), 524
 add_item() (SyndicationFeed method), 427

add_view() (ModelAdmin method), 367
 addslashes
 template filter, 594
 ADMIN_FOR
 setting, 561
 ADMIN_MEDIA_PREFIX
 setting, 561
 AdminPasswordChangeForm (class in
 django.contrib.auth.forms), 206
 ADMINS
 setting, 562
 ALLOWED_INCLUDE_ROOTS
 setting, 562
 app_label (models.ContentType attribute), 382
 APPEND_SLASH
 setting, 562
 appendlist() (QueryDict method), 557
 ar.forms.ARCUITField (class in
 django.contrib.localflavor), 398
 ar.forms.ARDNIField (class in
 django.contrib.localflavor), 398
 ar.forms.ARPostalCodeField (class in
 django.contrib.localflavor), 398
 ar.forms.ARProvinceSelect (class in
 django.contrib.localflavor), 398
 assertContains() (TestCase method), 191
 assertFormError() (TestCase method), 191
 assertNotContains() (TestCase method), 191
 assertRedirects() (TestCase method), 191
 assertTemplateNotUsed() (TestCase method), 191
 assertTemplateUsed() (TestCase method), 191
 at.forms.ATSocialSecurityNumberField (class in
 django.contrib.localflavor), 398
 at.forms.ATStateSelect (class in
 django.contrib.localflavor), 398
 at.forms.ATZipCodeField (class in
 django.contrib.localflavor), 398
 attrs (Widget attribute), 484
 au.forms.AUPhoneNumberField (class in
 django.contrib.localflavor), 398
 au.forms.AUPostCodeField (class in
 django.contrib.localflavor), 398
 au.forms.AUStateSelect (class in
 django.contrib.localflavor), 398

AUTH_PROFILE_MODULE

setting, 562

authenticate() (in module django.contrib.auth), 201

AUTHENTICATION_BACKENDS

setting, 562

AuthenticationForm (class
django.contrib.auth.forms), 206

auto_now (DateField attribute), 516

auto_now_add (DateField attribute), 516

autoescape
template tag, 583

AutoField (class in django.db.models), 515

B

blank (Field attribute), 512

block
template tag, 583

BooleanField (class in django.db.models), 515

BooleanField (class in django.forms), 475

br.forms.BRPhoneNumberField (class
django.contrib.localflavor), 399br.forms.BRStateSelect (class
django.contrib.localflavor), 399br.forms.BRZipCodeField (class
django.contrib.localflavor), 399

build_absolute_uri() (HttpRequest method), 555

Cca.forms.CAPhoneNumberField (class
django.contrib.localflavor), 399ca.forms.CAPostalCodeField (class
django.contrib.localflavor), 399ca.forms.CAProvinceField (class
django.contrib.localflavor), 399ca.forms.CAProvinceSelect (class
django.contrib.localflavor), 399ca.forms.CASocialInsuranceNumberField (class
django.contrib.localflavor), 399

CACHE_BACKEND

setting, 562

CACHE_MIDDLEWARE_KEY_PREFIX

setting, 563

CACHE_MIDDLEWARE_SECONDS

setting, 563

capfirst
template filter, 594center
template filter, 594ch.forms.CHIdentityCardNumberField (class
django.contrib.localflavor), 403ch.forms.CHPhoneNumberField (class
django.contrib.localflavor), 403ch.forms.CHStateSelect (class
django.contrib.localflavor), 403ch.forms.CHZipCodeField (class
django.contrib.localflavor), 403

change_form_template (ModelAdmin attribute), 366

change_list_template (ModelAdmin attribute), 366

change_view() (ModelAdmin method), 367

changefreq (Sitemap attribute), 407

changelist_view() (ModelAdmin method), 367

CharField (class in django.db.models), 515

CharField (class in django.forms), 475

in check_password() (in module django.contrib.auth), 201

check_password() (models.User method), 197

CheckboxInput (class in django.forms), 483

CheckboxSelectMultiple (class in django.forms), 483

ChoiceField (class in django.forms), 476

choices (ChoiceField attribute), 476

choices (Field attribute), 512

chunks() (File method), 455

cl.forms.CLRRegionSelect (class
django.contrib.localflavor), 399cl.forms.CLRutField (class
django.contrib.localflavor), 399

clean() (Field method), 471

clear() (QuerySet method), 525

in clear_cache() (models.ContentTypeManager method),
383

in Client (class in django.test.client), 185

client (Response attribute), 187

in client (TestCase attribute), 189

close() (File method), 456

codename (models.Permission attribute), 209

coerce (TypedChoiceField attribute), 476

ComboField (class in django.forms), 481

in CommaSeparatedIntegerField (class
django.db.models), 516in comment
template tag, 583in Comment (class in django.contrib.comments.models),
378

in comment (Comment attribute), 378

comment_form_target

template tag, 377

in COMMENT_MAX_LENGTH

setting, 379

COMMENTS_APP

setting, 379

COMMENTS_HIDE_REMOVED

setting, 379

content (HttpResponse attribute), 559

content (Response attribute), 187

content_object (Comment attribute), 378

content_type (Comment attribute), 378

content_type (models.Permission attribute), 209

context (Response attribute), 187

cookies (Client attribute), 188

COOKIES (HttpRequest attribute), 554

copy() (QueryDict method), 557

count (Paginator attribute), 251

in coupling
loose, 625

create() (models.User.message_set method), 211

create() (QuerySet method), 524

create_test_db() (in module django.test.utils), 193

- create_user() (models.UserManager method), 198
- createcachetable <tablename>
 - django-admin command, 443
- createsuperuser
 - django-admin command, 443
- ct_field (generic.GenericInlineModelAdmin attribute), 385
- ct_fk_field (generic.GenericInlineModelAdmin attribute), 385
- cut
 - template filter, 594
- cycle
 - template tag, 583
- D**
- DATABASE_ENGINE
 - setting, 563
- DATABASE_HOST
 - setting, 563
- DATABASE_NAME
 - setting, 563
- DATABASE_OPTIONS
 - setting, 564
- DATABASE_PASSWORD
 - setting, 564
- DATABASE_PORT
 - setting, 564
- DATABASE_USER
 - setting, 564
- date
 - template filter, 594
- DATE_FORMAT
 - setting, 564
- date_hierarchy (ModelAdmin attribute), 358
- date_joined (models.User attribute), 196
- DateField (class in django.db.models), 516
- DateField (class in django.forms), 476
- DATETIME_FORMAT
 - setting, 564
- DateTimeField (class in django.db.models), 516
- DateTimeField (class in django.forms), 477
- DateTimeInput (class in django.forms), 483
- db_column (Field attribute), 513
- db_index (Field attribute), 513
- db_table (ManyToManyField attribute), 523
- db_table (Options attribute), 525
- db_tablespace (Field attribute), 513
- db_tablespace (Options attribute), 526
- db_type(), 278
- dbshell
 - django-admin command, 443
- de.forms.DEIdentityCardNumberField (class in django.contrib.localflavor), 400
- de.forms.DEStateSelect (class in django.contrib.localflavor), 400
- de.forms.DEZipCodeField (class in django.contrib.localflavor), 400
- DEBUG
 - setting, 564
- debug
 - template tag, 584
- decimal_places (DecimalField attribute), 478, 517
- DecimalField (class in django.db.models), 517
- DecimalField (class in django.forms), 477
- decorators.login_required() (in module django.contrib.auth), 202
- decorators.permission_required() (in module django.contrib.auth), 207
- decorators.user_passes_test() (in module django.contrib.auth), 206
- default
 - template filter, 594
- default (Field attribute), 513
- DEFAULT_CHARSET
 - setting, 565
- DEFAULT_CONTENT_TYPE
 - setting, 565
- DEFAULT_FROM_EMAIL
 - setting, 565
- default_if_none
 - template filter, 595
- DEFAULT_INDEX_TABLESPACE
 - setting, 566
- DEFAULT_TABLESPACE
 - setting, 565
- delete() (File method), 456
- delete() (Model method), 530
- delete_confirmation_template (ModelAdmin attribute), 366
- delete_cookie() (HttpResponse method), 560
- delete_view() (ModelAdmin method), 367
- destroy_test_db() (in module django.test.utils), 194
- dictsort
 - template filter, 595
- dictsortreversed
 - template filter, 595
- diffsettings
 - django-admin command, 443
- DISALLOWED_USER_AGENTS
 - setting, 566
- divisibleby
 - template filter, 595
- django-admin command
 - createcachetable <tablename>, 443
 - createsuperuser, 443
 - dbshell, 443
 - diffsettings, 443
 - dumpdata <appname appname ...>, 444
 - runserver [port or ipaddr:port], 448
- django-admin command-line option
 - adminmedia, 448
 - email, 443
 - exclude, 444
 - format, 444
 - help, 442
 - indent, 444

- `--noinput`, 444
 - `--noreload`, 448
 - `--username`, 443
 - `--verbosity`, 442
 - `--version`, 442
- `django.conf.settings.configure()` (built-in function), 261
- `django.contrib.admin` (module), 357
- `django.contrib.auth` (module), 195
- `django.contrib.auth.forms` (module), 206
- `django.contrib.auth.middleware` (module), 509
- `django.contrib.auth.middleware.AuthenticationMiddleware` (class in `django.contrib.auth.middleware`), 509
- `django.contrib.comments` (module), 375
- `django.contrib.comments.models` (module), 378
- `django.contrib.comments.signals` (module), 379
- `django.contrib.comments.signals.comment_was_flagged` (built-in variable), 380
- `django.contrib.comments.signals.comment_was_posted` (built-in variable), 380
- `django.contrib.comments.signals.comment_will_be_posted` (built-in variable), 379
- `django.contrib.contenttypes` (module), 381
- `django.contrib.csrf` (module), 386
- `django.contrib.csrf.middleware` (module), 509
- `django.contrib.csrf.middleware.CsrfMiddleware` (class in `django.contrib.csrf.middleware`), 509
- `django.contrib.databrowse` (module), 386
- `django.contrib.flatpages` (module), 388
- `django.contrib.formtools` (module), 390
- `django.contrib.formtools.wizard` (module), 391
- `django.contrib.humanize` (module), 395
- `django.contrib.localflavor` (module), 397
- `django.contrib.redirects` (module), 404
- `django.contrib.sessions` (module), 122
- `django.contrib.sessions.middleware` (module), 509
- `django.contrib.sessions.middleware.SessionMiddleware` (class in `django.contrib.sessions.middleware`), 509
- `django.contrib.sitemaps` (module), 405
- `django.contrib.sites` (module), 410
- `django.contrib.sites.managers.CurrentSiteManager` (class in `django.contrib.sites`), 413
- `django.contrib.sites.models.Site` (class in `django.contrib.sites`), 410
- `django.contrib.syndication` (module), 415
- `django.contrib.syndication.feeds.Feed` (class in `django.contrib.syndication`), 420
- `django.contrib.webdesign` (module), 429
- `django.core.files` (module), 455
- `django.core.mail` (module), 227
- `django.core.mail.outbox` (in module `django.core.mail`), 191
- `django.core.paginator` (module), 249
- `django.core.signals` (module), 582
- `django.core.signals.got_request_exception` (built-in variable), 582
- `django.core.signals.request_finished` (built-in variable), 582
- `django.core.signals.request_started` (built-in variable), 582
- `django.core.urlresolvers` (module), 99
- `django.db.models` (module), 61
- `django.db.models.fields` (module), 511
- `django.db.models.fields.related` (module), 521
- `django.db.models.signals` (module), 579
- `django.db.models.signals.class_prepared` (built-in variable), 581
- `django.db.models.signals.post_delete` (built-in variable), 581
- `django.db.models.signals.post_init` (built-in variable), 580
- `django.db.models.signals.post_save` (built-in variable), 580
- `django.db.models.signals.post_syncdb` (built-in variable), 581
- `django.db.models.signals.pre_delete` (built-in variable), 580
- `django.db.models.signals.pre_save` (built-in variable), 580
- `django.db.models.SubfieldBase` (built-in class), 278
- `django.dispatch` (module), 263
- `django.forms.fields` (module), 471
- `django.forms.widgets` (module), 483
- `django.http` (module), 553
- `django.middleware` (module), 507
- `django.middleware.cache` (module), 507
- `django.middleware.cache.FetchFromCacheMiddleware` (class in `django.middleware.cache`), 507
- `django.middleware.cache.UpdateCacheMiddleware` (class in `django.middleware.cache`), 507
- `django.middleware.common` (module), 507
- `django.middleware.common.CommonMiddleware` (class in `django.middleware.common`), 507
- `django.middleware.doc` (module), 508
- `django.middleware.doc.XViewMiddleware` (class in `django.middleware.doc`), 508
- `django.middleware.gzip` (module), 508
- `django.middleware.gzip.GZipMiddleware` (class in `django.middleware.gzip`), 508
- `django.middleware.http` (module), 508
- `django.middleware.http.ConditionalGetMiddleware` (class in `django.middleware.http`), 508
- `django.middleware.http.SetRemoteAddrFromForwardedFor` (class in `django.middleware.http`), 508
- `django.middleware.locale` (module), 508
- `django.middleware.locale.LocaleMiddleware` (class in `django.middleware.locale`), 508
- `django.middleware.transaction` (module), 509
- `django.middleware.transaction.TransactionMiddleware` (class in `django.middleware.transaction`), 509
- `django.test` (module), 179
- `django.test.client` (module), 184
- `django.test.signals` (module), 582
- `django.test.signals.template_rendered` (built-in variable), 582

able), 582
 django.test.utils (module), 193
 django.utils.feedgenerator.Atom1Feed (class in
 django.contrib.syndication), 427
 django.utils.feedgenerator.Rss201rev2Feed (class in
 django.contrib.syndication), 427
 django.utils.feedgenerator.RssUserland091Feed (class
 in django.contrib.syndication), 427
 django.utils.feedgenerator.SyndicationFeed (class in
 django.contrib.syndication), 427
 django.views.static (module), 331
 DJANGO_SETTINGS_MODULE, 441
 Don't repeat yourself, 625
 DRY, 625
 dumpdata <appname appname ...>
 django-admin command, 444

E

editable (Field attribute), 514
 email (models.User attribute), 196
 EMAIL_HOST
 setting, 566
 EMAIL_HOST_PASSWORD
 setting, 566
 EMAIL_HOST_USER
 setting, 566
 EMAIL_PORT
 setting, 566
 EMAIL_SUBJECT_PREFIX
 setting, 566
 EMAIL_USE_TLS
 setting, 567
 email_user() (models.User method), 197
 EmailField (class in django.db.models), 517
 EmailField (class in django.forms), 478
 EmailMessage (class in django.core.mail), 230
 empty_label (ModelChoiceField attribute), 482
 empty_value (TypedChoiceField attribute), 476
 encoding (HttpRequest attribute), 553
 end_index() (Page method), 252
 environment variable
 DJANGO_SETTINGS_MODULE, 441
 error_messages (Field attribute), 474
 errors (Form attribute), 460
 es.forms.ESCCCFld (class
 in django.contrib.localflavor), 403
 es.forms.ESIdentityCardNumberField (class
 in django.contrib.localflavor), 403
 es.forms.ESPhoneNumberField (class
 in django.contrib.localflavor), 403
 es.forms.ESPostalCodeField (class
 in django.contrib.localflavor), 403
 es.forms.ESProvinceSelect (class
 in django.contrib.localflavor), 403
 es.forms.ESRegionSelect (class
 in django.contrib.localflavor), 403
 escape
 template filter, 596

escapejs
 template filter, 596
 exclude (ModelAdmin attribute), 360
 extends
 template tag, 584

F

fi.forms.FIMunicipalitySelect (class in
 django.contrib.localflavor), 399
 fi.forms.FISocialSecurityNumber (class in
 django.contrib.localflavor), 399
 fi.forms.FIZipCodeField (class in
 django.contrib.localflavor), 399
 Field, 129
 field, 639
 Field (class in django.forms), 471
 fields (ModelAdmin attribute), 360
 fieldsets (ModelAdmin attribute), 359
 File (class in django.core.files), 455
 FILE_CHARSET
 setting, 567
 FILE_UPLOAD_HANDLERS
 setting, 567
 FILE_UPLOAD_MAX_MEMORY_SIZE
 setting, 567
 FILE_UPLOAD_PERMISSIONS
 setting, 567
 FILE_UPLOAD_TEMP_DIR
 setting, 567
 FileField (class in django.db.models), 517
 FileField (class in django.forms), 478
 FileInput (class in django.forms), 483
 FilePathField (class in django.db.models), 518
 FilePathField (class in django.forms), 478
 FILES (HttpRequest attribute), 554
 filesizeformat
 template filter, 596
 filter
 template tag, 585
 filter_horizontal (ModelAdmin attribute), 361
 filter_vertical (ModelAdmin attribute), 361
 first
 template filter, 596
 first_name (models.User attribute), 196
 in firstof
 template tag, 585
 in fix_ampersands
 template filter, 596
 in FIXTURE_DIRS
 setting, 568
 in fixtures (TestCase attribute), 189
 FlatPageSitemap (class in django.contrib.sitemaps),
 408
 FloatField (class in django.db.models), 519
 floatformat
 template filter, 597
 flush() (HttpResponse method), 560
 for

template tag, 585
 force_escape
 template filter, 597
 ForeignKey (class in `django.db.models`), 521
 Form, 129
 form (ModelAdmin attribute), 358
 Form Media, 129
 formfield(), 282
 FormWizard (class in `django.contrib.formtools.wizard`), 394
 fr.forms.FRDepartmentSelect (class in `django.contrib.localflavor`), 399
 fr.forms.FRPhoneNumberField (class in `django.contrib.localflavor`), 399
 fr.forms.FRZipCodeField (class in `django.contrib.localflavor`), 399

G

generic view, 639
 generic.GenericInlineModelAdmin (class in `django.contrib.contenttypes`), 385
 GenericSitemap (class in `django.contrib.sitemaps`), 408
 GET (HttpRequest attribute), 554
 get() (Client method), 185
 get() (QueryDict method), 556
 get_absolute_url() (Model method), 531
 get_all_permissions() (models.User method), 197
 get_and_delete_messages() (models.User method), 197
 get_comment_count
 template tag, 376
 get_comment_form
 template tag, 377
 get_comment_list
 template tag, 376
 get_db_prep_lookup(), 281
 get_db_prep_save(), 281
 get_db_prep_value(), 280
 get_digit
 template filter, 597
 get_FOO_display() (Model method), 533
 get_for_model() (models.ContentTypeManager method), 383
 get_full_name() (models.User method), 197
 get_full_path() (HttpRequest method), 555
 get_group_permissions() (models.User method), 197
 get_host() (HttpRequest method), 555
 get_internal_type(), 282
 get_latest_by (Options attribute), 526
 get_next_by_FOO() (Model method), 533
 get_object_for_this_type() (models.ContentType method), 382
 get_previous_by_FOO() (Model method), 533
 get_profile() (models.User method), 197
 get_template() (FormWizard method), 394
 getlist() (QueryDict method), 557

H

handler404 (in module `django.core.urlresolvers`), 102

handler500 (in module `django.core.urlresolvers`), 103
 has_header() (HttpResponse method), 559
 has_module_perms() (models.User method), 197
 has_next() (Page method), 252
 has_other_pages() (Page method), 252
 has_perm() (models.User method), 197
 has_perms() (models.User method), 197
 has_previous() (Page method), 252
 has_usable_password() (models.User method), 197
 height (File attribute), 456
 height_field (ImageField attribute), 519
 help_text (Field attribute), 474, 514
 HiddenInput (class in `django.forms`), 483
 history_view() (ModelAdmin method), 367
 HttpRequest (class in `django.http`), 553
 HttpResponse (class in `django.http`), 558
 HttpResponseBadRequest (class in `django.http`), 560
 HttpResponseForbidden (class in `django.http`), 560
 HttpResponseGone (class in `django.http`), 560
 HttpResponseNotAllowed (class in `django.http`), 560
 HttpResponseNotFound (class in `django.http`), 560
 HttpResponseNotModified (class in `django.http`), 560
 HttpResponsePermanentRedirect (class in `django.http`), 560
 HttpResponseRedirect (class in `django.http`), 560
 HttpResponseServerError (class in `django.http`), 560

I

if
 template tag, 586
 ifchanged
 template tag, 587
 ifequal
 template tag, 588
 ifnotequal
 template tag, 588
 IGNOREABLE_404_ENDS
 setting, 568
 IGNOREABLE_404_STARTS
 setting, 568
 ImageField (class in `django.db.models`), 519
 ImageField (class in `django.forms`), 479
 in.forms.INStateField (class in `django.contrib.localflavor`), 400
 in.forms.INStateSelect (class in `django.contrib.localflavor`), 400
 in.forms.INZipCodeField (class in `django.contrib.localflavor`), 400
 include
 template tag, 588
 include() (in module `django.core.urlresolvers`), 103
 index_template (AdminSite attribute), 374
 initial (Field attribute), 473
 initial (Form attribute), 461
 inlines (ModelAdmin attribute), 364
 input_formats (DateField attribute), 476
 input_formats (DateTimeField attribute), 477
 input_formats (TimeField attribute), 481

INSTALLED_APPS

setting, 568

IntegerField (class in django.db.models), 519

IntegerField (class in django.forms), 479

INTERNAL_IPS

setting, 568

ip_address (Comment attribute), 378

IPAddressField (class in django.db.models), 520

IPAddressField (class in django.forms), 480

iriendcode

template filter, 597

is_.forms.ISIdNumberField (class
django.contrib.localflavor), 400

is_.forms.ISPhoneNumberField (class
django.contrib.localflavor), 400

is_.forms.ISPostalCodeSelect (class
django.contrib.localflavor), 400

is_active (models.User attribute), 196

is_ajax() (HttpRequest method), 556

is_anonymous() (models.User method), 197

is_authenticated() (models.User method), 197

is_bound (Form attribute), 459

is_public (Comment attribute), 378

is_removed (Comment attribute), 378

is_secure() (HttpRequest method), 556

is_staff (models.User attribute), 196

is_superuser (models.User attribute), 196

is_valid() (Form method), 460

it.forms.ITProvinceSelect (class
django.contrib.localflavor), 401

it.forms.ITRegionSelect (class
django.contrib.localflavor), 401

it.forms.ITSocialSecurityNumberField (class
django.contrib.localflavor), 400

it.forms.ITVatNumberField (class
django.contrib.localflavor), 400

it.forms.ITZipCodeField (class
django.contrib.localflavor), 400

items (Sitemap attribute), 407

items() (QueryDict method), 557

iteritems() (QueryDict method), 557

iterlists() (QueryDict method), 557

itervalues() (QueryDict method), 557

J

Java, 319

JING_PATH

setting, 569

join

template filter, 597

jp.forms.JPPostalCodeField (class
django.contrib.localflavor), 401

jp.forms.JPPrefectureSelect (class
django.contrib.localflavor), 401

JVM, 319

Jython, 319

JYTHONPATH, 320

L

label (Field attribute), 472

LANGUAGE_CODE

setting, 569

LANGUAGE_COOKIE_NAME

setting, 569

LANGUAGES

setting, 569

last

template filter, 598

last_login (models.User attribute), 196

in last_name (models.User attribute), 196

lastmod (Sitemap attribute), 407

in length

template filter, 598

in length_is

template filter, 598

limit_choices_to (ForeignKey attribute), 522

limit_choices_to (ManyToManyField attribute), 523

linebreaks

template filter, 598

linebreaksbr

template filter, 598

linenumbers

template filter, 598

list_display (ModelAdmin attribute), 361

list_display_links (ModelAdmin attribute), 363

list_filter (ModelAdmin attribute), 363

in list_per_page (ModelAdmin attribute), 364

in list_select_related (ModelAdmin attribute), 364

in lists() (QueryDict method), 558

ljust

in template filter, 599

load

in template tag, 589

LOCALE_PATHS

in setting, 570

location (Sitemap attribute), 407

login() (Client method), 186

login() (in module django.contrib.auth), 201

LOGIN_REDIRECT_URL

setting, 570

login_template (AdminSite attribute), 374

LOGIN_URL

setting, 570

logout() (Client method), 187

logout() (in module django.contrib.auth), 202

LOGOUT_URL

setting, 570

lower

in template filter, 599

M

mail_admins() (in module django.core.mail), 228

mail_managers() (in module django.core.mail), 228

make_list

template filter, 599

- `make_random_password()` (models.UserManager method), 198
 - `Manager` (class in `django.db.models`), 88
 - `MANAGERS`
 - setting, 570
 - `ManyToManyField` (class in `django.db.models`), 522
 - `match` (`FilePathField` attribute), 479, 518
 - `max_digits` (`DecimalField` attribute), 478, 517
 - `max_length` (`CharField` attribute), 476, 516
 - `max_length` (`URLField` attribute), 481
 - `max_value` (`DecimalField` attribute), 477
 - `max_value` (`IntegerField` attribute), 479
 - `MEDIA_ROOT`
 - setting, 570
 - `MEDIA_URL`
 - setting, 570
 - `META` (`HttpRequest` attribute), 554
 - `method` (`HttpRequest` attribute), 553
 - `MIDDLEWARE_CLASSES`
 - setting, 571
 - `min_length` (`CharField` attribute), 476
 - `min_length` (`URLField` attribute), 481
 - `min_value` (`DecimalField` attribute), 477
 - `min_value` (`IntegerField` attribute), 479
 - `model`, 639
 - `Model` (class in `django.db.models`), 528
 - `model` (models.ContentType attribute), 382
 - `model_class()` (models.ContentType method), 382
 - `ModelAdmin` (class in `django.contrib.admin`), 358
 - `ModelChoiceField` (class in `django.forms`), 482
 - `ModelMultipleChoiceField` (class in `django.forms`), 482
 - `models.AnonymousUser` (class in `django.contrib.auth`), 199
 - `models.ContentType` (class in `django.contrib.contenttypes`), 382
 - `models.ContentTypeManager` (class in `django.contrib.contenttypes`), 383
 - `models.FlatPage` (class in `django.contrib.flatpages`), 389
 - `models.Permission` (class in `django.contrib.auth`), 209
 - `models.Redirect` (class in `django.contrib.redirects`), 405
 - `models.User` (class in `django.contrib.auth`), 196
 - `models.UserManager` (class in `django.contrib.auth`), 198
 - `MONTH_DAY_FORMAT`
 - setting, 571
 - `MTV`, 639
 - `multiple_chunks()` (File method), 456
 - `MultipleChoiceField` (class in `django.forms`), 480
 - `MultipleHiddenInput` (class in `django.forms`), 483
 - `MultiValueField` (class in `django.forms`), 481
 - `MultiWidget` (class in `django.forms`), 483
 - `MVC`, 639
 - `mx.forms.MXStateSelect` (class in `django.contrib.localflavor`), 401
 - `name` (File attribute), 455
 - `name` (models.ContentType attribute), 382
 - `name` (models.Permission attribute), 209
 - `next_page_number()` (Page method), 252
 - `nl.forms.NLPhoneNumberField` (class in `django.contrib.localflavor`), 400
 - `nl.forms.NLProvinceSelect` (class in `django.contrib.localflavor`), 400
 - `nl.forms.NLSofNumberField` (class in `django.contrib.localflavor`), 400
 - `nl.forms.NLZipCodeField` (class in `django.contrib.localflavor`), 400
 - `no.forms.NOMunicipalitySelect` (class in `django.contrib.localflavor`), 401
 - `no.forms.NOSocialSecurityNumber` (class in `django.contrib.localflavor`), 401
 - `no.forms.NOZipCodeField` (class in `django.contrib.localflavor`), 401
 - `now`
 - template tag, 589
 - `null` (Field attribute), 511
 - `NullBooleanField` (class in `django.db.models`), 520
 - `NullBooleanField` (class in `django.forms`), 480
 - `NullBooleanSelect` (class in `django.forms`), 483
 - `num_pages` (Paginator attribute), 252
 - `number` (Page attribute), 253
- ## O
- `object_history_template` (ModelAdmin attribute), 366
 - `object_list` (Page attribute), 253
 - `object_pk` (Comment attribute), 378
 - `OneToOneField` (class in `django.db.models`), 523
 - `open()` (File method), 455
 - `order_with_respect_to` (Options attribute), 526
 - `ordering` (ModelAdmin attribute), 364
 - `ordering` (Options attribute), 526
- ## P
- `page()` (Paginator method), 251
 - `page_range` (Paginator attribute), 252
 - `Paginator` (class in `django.core.paginator`), 251
 - `paginator` (Page attribute), 253
 - `parent_link` (OneToOneField attribute), 524
 - `parse_params()` (FormWizard method), 394
 - `password` (models.User attribute), 196
 - `password_reset_complete()` (in module `django.contrib.auth`), 206
 - `password_reset_confirm()` (in module `django.contrib.auth`), 205
 - `PasswordChangeForm` (class in `django.contrib.auth.forms`), 206
 - `PasswordInput` (class in `django.forms`), 483
 - `PasswordResetForm` (class in `django.contrib.auth.forms`), 206
 - `path` (File attribute), 455
 - `path` (`FilePathField` attribute), 478, 518
 - `path` (`HttpRequest` attribute), 553
 - `patterns()` (in module `django.core.urlresolvers`), 101
 - `permalink()` (in module `django.db.models`), 532

- permissions (Options attribute), 527
 - phone2numeric
 - template filter, 599
 - ping_google() (in module django.contrib.sitemaps), 409
 - pk (Model attribute), 529
 - pl.forms.PLAdministrativeUnitSelect (class in django.contrib.localflavor), 402
 - pl.forms.PLNationalBusinessRegisterField (class in django.contrib.localflavor), 401
 - pl.forms.PLNationalIdentificationNumberField (class in django.contrib.localflavor), 401
 - pl.forms.PLPostalCodeField (class in django.contrib.localflavor), 401
 - pl.forms.PLTaxNumberField (class in django.contrib.localflavor), 401
 - pl.forms.PLVoivodeshipSelect (class in django.contrib.localflavor), 402
 - pluralize
 - template filter, 599
 - PositiveIntegerField (class in django.db.models), 520
 - PositiveSmallIntegerField (class in django.db.models), 520
 - POST (HttpRequest attribute), 554
 - post() (Client method), 186
 - pprint
 - template filter, 600
 - pre_init (django.db.models.signals attribute), 579
 - pre_save(), 281
 - prefix (Form attribute), 471
 - prefix_for_step() (FormWizard method), 394
 - PREPEND_WWW
 - setting, 571
 - prepopulated_fields (ModelAdmin attribute), 364
 - previous_page_number() (Page method), 252
 - primary_key (Field attribute), 514
 - priority() (Sitemap method), 408
 - process_exception(), 121
 - process_request(), 120
 - process_response(), 121
 - process_step() (FormWizard method), 395
 - process_view(), 120
 - PROFANITIES_LIST
 - setting, 571
 - projeto, 639
 - property, 639
 - pt.forms.PEDepartmentSelect (class in django.contrib.localflavor), 401
 - pt.forms.PEDNIField (class in django.contrib.localflavor), 401
 - pt.forms.PERUCField (class in django.contrib.localflavor), 401
 - Python Enhancement Proposals
 - PEP 257, 180
 - PEP 8, 692
 - QuerySet (class in django.db.models), 535
 - queryset (ModelChoiceField attribute), 482
- ## R
- radio_fields (ModelAdmin attribute), 364
 - RadioSelect (class in django.forms), 483
 - random
 - template filter, 600
 - raw_id_fields (ModelAdmin attribute), 365
 - raw_post_data (HttpRequest attribute), 555
 - read() (File method), 455
 - recursive (FilePathField attribute), 479, 519
 - regex (RegexField attribute), 480
 - RegexField (class in django.forms), 480
 - regroup
 - template tag, 590
 - related_name (ForeignKey attribute), 522
 - related_name (ManyToManyField attribute), 523
 - remove() (QuerySet method), 525
 - removetags
 - template filter, 600
 - render_comment_form
 - template tag, 376
 - render_hash_failure() (FormWizard method), 394
 - render_template() (FormWizard method), 395
 - REQUEST (HttpRequest attribute), 554
 - request (Response attribute), 187
 - required (Field attribute), 472
 - resolve() (in module django.core.urlresolvers), 108
 - Response (class in django.test.client), 187
 - reverse() (in module django.core.urlresolvers), 108
 - rjust
 - template filter, 600
 - ro.forms.ROCIFField (class in django.contrib.localflavor), 402
 - ro.forms.ROCNPField (class in django.contrib.localflavor), 402
 - ro.forms.ROCountyField (class in django.contrib.localflavor), 402
 - ro.forms.ROCountySelect (class in django.contrib.localflavor), 402
 - ro.forms.ROIBANField (class in django.contrib.localflavor), 402
 - ro.forms.ROPhoneNumberField (class in django.contrib.localflavor), 402
 - ro.forms.ROPostalCodeField (class in django.contrib.localflavor), 402
 - ROOT_URLCONF
 - setting, 571
 - run_tests() (in module django.test.simple), 193
 - runserver [port or ipaddr:port]
 - django-admin command, 448
- ## S
- safe
 - template filter, 600
 - save() (File method), 456
 - save() (Model method), 528

- save_as (ModelAdmin attribute), 365
- save_formset() (ModelAdmin method), 367
- save_model() (ModelAdmin method), 367
- save_on_top (ModelAdmin attribute), 365
- savepoint() (transaction method), 97
- savepoint_commit() (transaction method), 97
- savepoint_rollback() (transaction method), 97
- schema_path (in module django.db.models), 521
- search_fields (ModelAdmin attribute), 365
- SECRET_KEY
 - setting, 572
- security_hash() (FormWizard method), 394
- Select (class in django.forms), 483
- SelectMultiple (class in django.forms), 483
- send() (Signal method), 265
- SEND_BROKEN_LINK_EMAILS
 - setting, 572
- send_mail() (in module django.core.mail), 227
- send_mass_mail() (in module django.core.mail), 228
- SERIALIZATION_MODULES
 - setting, 572
- SERVER_EMAIL
 - setting, 572
- session (Client attribute), 188
- session (HttpRequest attribute), 555
- SESSION_COOKIE_AGE
 - setting, 572
- SESSION_COOKIE_DOMAIN
 - setting, 573
- SESSION_COOKIE_NAME
 - setting, 573
- SESSION_COOKIE_PATH
 - setting, 573
- SESSION_COOKIE_SECURE
 - setting, 573
- SESSION_ENGINE
 - setting, 572
- SESSION_EXPIRE_AT_BROWSER_CLOSE
 - setting, 573
- SESSION_FILE_PATH
 - setting, 573
- SESSION_SAVE_EVERY_REQUEST
 - setting, 573
- set_cookie() (HttpResponse method), 559
- set_password() (models.User method), 197
- set_unusable_password() (models.User method), 197
- setdefault() (QueryDict method), 557
- setlist() (QueryDict method), 557
- setlistdefault() (QueryDict method), 557
- SetPasswordForm (class in django.contrib.auth.forms), 206
- setting
 - ABSOLUTE_URL_OVERRIDES, 561
 - ADMIN_FOR, 561
 - ADMIN_MEDIA_PREFIX, 561
 - ADMINS, 562
 - ALLOWED_INCLUDE_ROOTS, 562
 - APPEND_SLASH, 562
 - AUTH_PROFILE_MODULE, 562
 - AUTHENTICATION_BACKENDS, 562
 - CACHE_BACKEND, 562
 - CACHE_MIDDLEWARE_KEY_PREFIX, 563
 - CACHE_MIDDLEWARE_SECONDS, 563
 - COMMENT_MAX_LENGTH, 379
 - COMMENTS_APP, 379
 - COMMENTS_HIDE_REMOVED, 379
 - DATABASE_ENGINE, 563
 - DATABASE_HOST, 563
 - DATABASE_NAME, 563
 - DATABASE_OPTIONS, 564
 - DATABASE_PASSWORD, 564
 - DATABASE_PORT, 564
 - DATABASE_USER, 564
 - DATE_FORMAT, 564
 - DATETIME_FORMAT, 564
 - DEBUG, 564
 - DEFAULT_CHARSET, 565
 - DEFAULT_CONTENT_TYPE, 565
 - DEFAULT_FROM_EMAIL, 565
 - DEFAULT_INDEX_TABLESPACE, 566
 - DEFAULT_TABLESPACE, 565
 - DISALLOWED_USER_AGENTS, 566
 - EMAIL_HOST, 566
 - EMAIL_HOST_PASSWORD, 566
 - EMAIL_HOST_USER, 566
 - EMAIL_PORT, 566
 - EMAIL_SUBJECT_PREFIX, 566
 - EMAIL_USE_TLS, 567
 - FILE_CHARSET, 567
 - FILE_UPLOAD_HANDLERS, 567
 - FILE_UPLOAD_MAX_MEMORY_SIZE, 567
 - FILE_UPLOAD_PERMISSIONS, 567
 - FILE_UPLOAD_TEMP_DIR, 567
 - FIXTURE_DIRS, 568
 - IGNORABLE_404_ENDS, 568
 - IGNORABLE_404_STARTS, 568
 - INSTALLED_APPS, 568
 - INTERNAL_IPS, 568
 - JING_PATH, 569
 - LANGUAGE_CODE, 569
 - LANGUAGE_COOKIE_NAME, 569
 - LANGUAGES, 569
 - LOCALE_PATHS, 570
 - LOGIN_REDIRECT_URL, 570
 - LOGIN_URL, 570
 - LOGOUT_URL, 570
 - MANAGERS, 570
 - MEDIA_ROOT, 570
 - MEDIA_URL, 570
 - MIDDLEWARE_CLASSES, 571
 - MONTH_DAY_FORMAT, 571
 - PREPEND_WWW, 571
 - PROFANITIES_LIST, 571
 - ROOT_URLCONF, 571
 - SECRET_KEY, 572
 - SEND_BROKEN_LINK_EMAILS, 572

- SERIALIZATION_MODULES, 572
 - SERVER_EMAIL, 572
 - SESSION_COOKIE_AGE, 572
 - SESSION_COOKIE_DOMAIN, 573
 - SESSION_COOKIE_NAME, 573
 - SESSION_COOKIE_PATH, 573
 - SESSION_COOKIE_SECURE, 573
 - SESSION_ENGINE, 572
 - SESSION_EXPIRE_AT_BROWSER_CLOSE, 573
 - SESSION_FILE_PATH, 573
 - SESSION_SAVE_EVERY_REQUEST, 573
 - SITE_ID, 574
 - TEMPLATE_CONTEXT_PROCESSORS, 574
 - TEMPLATE_DEBUG, 574
 - TEMPLATE_DIRS, 574
 - TEMPLATE_LOADERS, 574
 - TEMPLATE_STRING_IF_INVALID, 575
 - TEST_DATABASE_CHARSET, 575
 - TEST_DATABASE_COLLATION, 575
 - TEST_DATABASE_NAME, 575
 - TEST_RUNNER, 575
 - TIME_FORMAT, 575
 - TIME_ZONE, 576
 - URL_VALIDATOR_USER_AGENT, 576
 - USE_ETAGS, 576
 - USE_I18N, 576
 - YEAR_MONTH_FORMAT, 576
 - setup_test_environment() (in module django.test.utils), 193
 - Signal (class in django.dispatch), 265
 - site (Comment attribute), 378
 - SITE_ID
 - setting, 574
 - Sitemap (class in django.contrib.sitemaps), 407
 - size (File attribute), 455
 - sk.forms.SKDistrictSelect (class in django.contrib.localflavor), 402
 - sk.forms.SKPostalCodeField (class in django.contrib.localflavor), 402
 - sk.forms.SKRegionSelect (class in django.contrib.localflavor), 402
 - slice
 - template filter, 600
 - slug, 639
 - SlugField (class in django.db.models), 520
 - slugify
 - template filter, 600
 - SmallIntegerField (class in django.db.models), 520
 - SMTPConnection (class in django.core.mail), 231
 - spaceless
 - template tag, 591
 - SplitDateTimeField (class in django.forms), 482
 - SplitDateTimeWidget (class in django.forms), 483
 - ssi
 - template tag, 592
 - start_index() (Page method), 252
 - status_code (Response attribute), 187
 - storage (FileField attribute), 518
 - stringformat
 - template filter, 601
 - striptags
 - template filter, 601
 - submit_date (Comment attribute), 378
 - symmetrical (ManyToManyField attribute), 523
- ## T
- teardown_test_environment() (in module django.test.utils), 193
 - tell() (HttpResponse method), 560
 - template, 639
 - template (Response attribute), 187
 - template filter
 - add, 594
 - addslashes, 594
 - capfirst, 594
 - center, 594
 - cut, 594
 - date, 594
 - default, 594
 - default_if_none, 595
 - dictsort, 595
 - dictsortreversed, 595
 - divisibleby, 595
 - escape, 596
 - escapejs, 596
 - filesizeformat, 596
 - first, 596
 - fix_ampersands, 596
 - floatformat, 597
 - force_escape, 597
 - get_digit, 597
 - iriendcode, 597
 - join, 597
 - last, 598
 - length, 598
 - length_is, 598
 - linebreaks, 598
 - linebreaksbr, 598
 - linenumbers, 598
 - ljust, 599
 - lower, 599
 - make_list, 599
 - phone2numeric, 599
 - pluralize, 599
 - pprint, 600
 - random, 600
 - removetags, 600
 - rjust, 600
 - safe, 600
 - slice, 600
 - slugify, 600
 - stringformat, 601
 - striptags, 601
 - time, 601
 - timesince, 601

- timeuntil, 602
 - title, 602
 - truncatewords, 602
 - truncatewords_html, 602
 - unordered_list, 602
 - upper, 603
 - urlencode, 603
 - urlize, 603
 - urlizetrunc, 603
 - wordcount, 603
 - wordwrap, 604
 - yesno, 604
 - template tag
 - autoescape, 583
 - block, 583
 - comment, 583
 - comment_form_target, 377
 - cycle, 583
 - debug, 584
 - extends, 584
 - filter, 585
 - firstof, 585
 - for, 585
 - get_comment_count, 376
 - get_comment_form, 377
 - get_comment_list, 376
 - if, 586
 - ifchanged, 587
 - ifequal, 588
 - ifnotequal, 588
 - include, 588
 - load, 589
 - now, 589
 - regroup, 590
 - render_comment_form, 376
 - spaceless, 591
 - ssi, 592
 - templatetag, 592
 - url, 592
 - widthratio, 593
 - with, 593
 - TEMPLATE_CONTEXT_PROCESSORS
 - setting, 574
 - TEMPLATE_DEBUG
 - setting, 574
 - TEMPLATE_DIRS
 - setting, 574
 - TEMPLATE_LOADERS
 - setting, 574
 - TEMPLATE_STRING_IF_INVALID
 - setting, 575
 - templatetag
 - template tag, 592
 - TEST_DATABASE_CHARSET
 - setting, 575
 - TEST_DATABASE_COLLATION
 - setting, 575
 - TEST_DATABASE_NAME
 - setting, 575
 - TEST_RUNNER
 - setting, 575
 - TestCase (class in django.test), 189
 - Textarea (class in django.forms), 483
 - TextField (class in django.db.models), 520
 - TextInput (class in django.forms), 483
 - through (ManyToManyField attribute), 523
 - time
 - template filter, 601
 - TIME_FORMAT
 - setting, 575
 - TIME_ZONE
 - setting, 576
 - TimeField (class in django.db.models), 521
 - TimeField (class in django.forms), 481
 - timesince
 - template filter, 601
 - timeuntil
 - template filter, 602
 - title
 - template filter, 602
 - to_field (ForeignKey attribute), 522
 - to_python(), 280
 - truncatewords
 - template filter, 602
 - truncatewords_html
 - template filter, 602
 - TypedChoiceField (class in django.forms), 476
- ## U
- uk.forms.UKCountySelect (class in django.contrib.localflavor), 403
 - uk.forms.UKNationSelect (class in django.contrib.localflavor), 403
 - uk.forms.UKPostcodeField (class in django.contrib.localflavor), 403
 - unique (Field attribute), 514
 - unique_for_date (Field attribute), 514
 - unique_for_month (Field attribute), 515
 - unique_for_year (Field attribute), 515
 - unique_together (Options attribute), 527
 - unordered_list
 - template filter, 602
 - update() (QueryDict method), 557
 - upload_to (FileField attribute), 517
 - UploadedFile (class in django.core.files), 114
 - upper
 - template filter, 603
 - url
 - template tag, 592
 - url (File attribute), 455
 - url() (in module django.core.urlresolvers), 102
 - URL_VALIDATOR_USER_AGENT
 - setting, 576
 - urlconf (HttpRequest attribute), 555
 - urlencode
 - template filter, 603

- urlencode() (QueryDict method), 558
 - URLField (class in django.db.models), 521
 - URLField (class in django.forms), 481
 - urlize
 - template filter, 603
 - urlizetrunc
 - template filter, 603
 - urls
 - definitive, 627
 - urls (TestCase attribute), 190
 - us.forms.USPhoneNumberField (class in django.contrib.localflavor), 403
 - us.forms.USSocialSecurityNumberField (class in django.contrib.localflavor), 403
 - us.forms.USStateField (class in django.contrib.localflavor), 404
 - us.forms.USStateSelect (class in django.contrib.localflavor), 404
 - us.forms.USZipCodeField (class in django.contrib.localflavor), 404
 - us.models.PhoneNumberField (class in django.contrib.localflavor), 404
 - us.models.USStateField (class in django.contrib.localflavor), 404
 - USE_ETAGS
 - setting, 576
 - USE_I18N
 - setting, 576
 - user (Comment attribute), 378
 - user (HttpRequest attribute), 555
 - user_email (Comment attribute), 378
 - user_name (Comment attribute), 378
 - user_url (Comment attribute), 378
 - UserChangeForm (class in django.contrib.auth.forms), 206
 - UserCreationForm (class in django.contrib.auth.forms), 206
 - username (models.User attribute), 196
- V**
- validator_user_agent (URLField attribute), 481
 - value_to_string(), 283
 - values() (QueryDict method), 557
 - verbose_name (Field attribute), 515
 - verbose_name (Options attribute), 527
 - verbose_name_plural (Options attribute), 528
 - verify_exists (URLField attribute), 481, 521
 - view, 639
 - views.login() (in module django.contrib.auth), 203
 - views.logout() (in module django.contrib.auth), 204
 - views.logout_then_login() (in module django.contrib.auth), 204
 - views.password_change() (in module django.contrib.auth), 204
 - views.password_change_done() (in module django.contrib.auth), 205
 - views.password_reset() (in module django.contrib.auth), 205
 - views.password_reset_done() (in module django.contrib.auth), 205
 - views.redirect_to_login() (in module django.contrib.auth), 205
- W**
- Widget, 129
 - widget (Field attribute), 474
 - widget (Form attribute), 484
 - width (File attribute), 456
 - width_field (ImageField attribute), 519
 - widthratio
 - template tag, 593
 - with
 - template tag, 593
 - wordcount
 - template filter, 603
 - wordwrap
 - template filter, 604
 - write() (File method), 456
 - write() (HttpResponse method), 560
 - write() (SyndicationFeed method), 428
 - writeString() (SyndicationFeed method), 428
- X**
- xml
 - suckiness of, 628
 - XMLField (class in django.db.models), 521
- Y**
- YEAR_MONTH_FORMAT
 - setting, 576
 - yesno
 - template filter, 604
- Z**
- za.forms.ZAIDField (class in django.contrib.localflavor), 402
 - za.forms.ZAPostCodeField (class in django.contrib.localflavor), 402