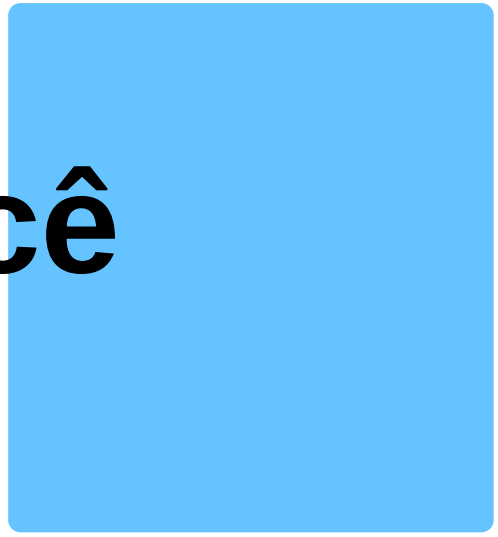
03 o django e você



Alguns fatos sobre o Django

é o framework favorito do Guido van Rossum, criador de Python

é largamente utilizado internamente no Google

é suportado pelo Google App Engine (serviço de hospedagem)

foi criado num site de notícias, e ciclo de produção reflete isso

Principais vantagens do Django

O banco é gerado a partir do código Python, e não o contrário

fácil de aprender, porém robusto, flexível e extensível

interface administrativa auto-gerada feita para usar

Estrutura básica de um projeto

Exemplo: projeto *pizza*

Sistemas web para uma pizzaria de bairro

Estrutura

```
pizza
|-- entrega
|    \ ...
|-- portal
|    \ ...
|-- __init__.py
|-- manage.py
|-- static
|-- templates
\-- urls.py
```

formado por duas aplicações:

- `entrega`: sistema interno para receber despachar pedidos
- `portal`: site público da pizzaria

`pizza/` e demais arquivos marcados com o selo foram criados pelo comando

```
$ django-admin.py startproject pizza
```

diretorios

`pizza/`: o diretório raiz do projeto

- `entrega/`: o diretório da aplicação `entrega`
- `portal/`: diretório da aplicação `portal`
- `static/`: diretório de arquivos estáticos do projeto (.css, .gif)
- `templates/`: diretório de templates do projeto
- `__init__.py`: módulo vazio para marcar o diretório `pizza/` como um *package*
- `manage.py`: script administrativo

`settings.py`: configurações do projeto

`urls.py`: mapeamento de cada tipo de URL para a *view* correspondente

applicação pizza/entrega/

diretório de uma das aplicações que formam o sistema

`pizza/entrega/` e demais arquivos marcados com a estrela foram criados via

```
$ ./manage.py startapp entrega
```

Estrutura básica de uma aplicação

```
entrega
|-- admin.py
|-- __init__.py
|-- models.py
|-- templates
|   \-- ...
\-- views.py
```

arquivos da aplicação

`__init__.py`: módulo vazio para marcar o diretório `entrega/` como um *package*

`admin.py`: configurações da interface administrativa

`models.py`: modelos de dados (classes de persistência)

`views.py`: funções de tratamento de requisições

Modelos de dados: o básico

por convenção cada aplicação tem o seu `models.py`

o `models.py` determina o esquema de dados (e não o BD)

este arquivo contém praticamente só definições de classes derivadas de `models.Model`

Exemplo de Model

```
class Livro(models.Model):
    titulo = models.CharField(max_length=256)
    isbn = models.CharField(max_length=16, blank=True)
    edicao = models.CharField(max_length=64, blank=True)
    qt_paginas = models.PositiveIntegerField(default=0)
    dt_catalogacao = models.DateField(auto_now_add=True)
    editora = models.ForeignKey('Editora')
    categoria = models.CharField(max_length=8, blank=True,
                                 choices=CATEGORIAS)

    class Meta:
        ordering = ('titulo', 'isbn', 'id')

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

Tipos de campos primitivos

campos que emulam tipos básicos de SQL

`CharField`, `TextField`, `BooleanField`, `NullBooleanField`

`DateField`, `DateTimeField`, `TimeField`

`IntegerField`, `SmallIntegerField`, `AutoField`

`DecimalField`, `FloatField`

Campos derivados

campos que acrescentam validações sobre tipos básicos

`EmailField`, `URLField`, `IPAddressField`, `SlugField`, `XMLField`

`PositiveIntegerField`, `PositiveSmallIntegerField`, `CommaSeparatedIntegerField`

Campos para armazenar arquivos

`FileField`, `FilePathField`, `ImageField`
nos três casos os dados são armazenados no sistema de arquivos e o campo no banco de dados registra apenas o nome do arquivo ou o caminho

Campo de referência: `ForeignKey`

`ForeignKey`: referência a objeto (chave estrangeira)

relação muitos-para-um

```
class Livro(models.Model):
    titulo = models.CharField(max_length=256)
    editora = models.ForeignKey('Editora')

class Editora(models.Model):
    nome = models.CharField(max_length=128)
    cidade = models.CharField(max_length=128)
```

Relacionamentos automáticos

objeto referente (editora) ganha um atributo dinâmico `«modelo»_set` onde `«modelo»` é o nome do modelo relacionado em caixa baixa (livro).

Exemplo

Ex: objeto `ed` instância de `Editora` ganha `ed.livro_set`)

```
>>> ed = Editora.objects.get(nome__icontains='norton')
>>> ed
<Editora: W. W. Norton & Company>
>>> for l in ed.livro_set.all(): print l
...
Colors of the World
The Annotated Alice
```

OneToOneField

OneToOneField: referência a objeto (chave estrangeira) relação um-para-um

```
class Criador(models.Model):
    nome = models.CharField(max_length=128)

class Biografia(models.Model):
    sobre = models.OneToOneField(Criador)
    texto = models.TextField()
```

Relacionamentos automáticos

objeto referente (criador) ganha um atributo dinâmico com o nome do modelo relacionado em caixa baixa

Exemplo

ex: instância `c` de `Criador` ganha `c.biografia`

```
>>> lc = Criador.objects.get(id=1)
>>> print lc.biografia.texto
Charles Lutwidge Dodgson, ou Lewis Carrol
foi um escritor e um matemático britânico...
```

ManyToManyField

`ManyToManyField`: referência a múltiplos objetos via tabela de ligação

relação muitos-para-muitos

objeto referente ganha um atributo dinâmico «`modelo`»`_set` (ver *fk-intro*)

a tabela de ligação pode ser implícita ou explícita via parâmetro `through`

Muitos para muitos

```
class Livro(models.Model):
    titulo = models.CharField(max_length=256)
    assuntos = models.ManyToManyField('Assunto')
    criadores = models.ManyToManyField('Criador',
                                       through='Credito')

class Credito(models.Model):
    livro = models.ForeignKey(Livro)
    criador = models.ForeignKey('Criador')
    papel = models.CharField(max_length=64)

class Criador(models.Model):
    nome = models.CharField(max_length=128)
    dt_nascimento = models.DateField(null=True,
                                     blank=True)
```

Parâmetros comuns para campos

fonte:
`django/db/models/fields/__init__.py`

```
class Field(object):
    '...'
    def __init__(self, verbose_name=None, name=None, primary_key=False,
            max_length=None, unique=False, blank=False, null=False,
            db_index=False, rel=None, default=NOT_PROVIDED, editable=True,
            serialize=True, unique_for_date=None, unique_for_month=None,
            unique_for_year=None, choices=None, help_text='',
            db_column=None, db_tablespace=None, auto_created=False):
        '...'
```

Banco de dados

parâmetros que definem o esquema no banco de dados:

primary_key

unique

null

db_index

db_column

db_tablespace

Validacoes

parâmetros que definem a validação e a apresentação do campo para o usuário:

verbose_name

blank

default

unique_for_date

unique_for_month

unique_for_year

choices

help_text

Parâmetros que definem o esquema

Em ordem de utilidade (subjetiva).

`null=False`

Determina se o campo aceitará valores nulos (`NULL` em SQL; `None` em Python). O default implica em `NOT NULL`.

`unique=False`

Determina se o campo terá uma restrição de unicidade. Caso `True` implica também na criação de um índice.

`db_index=False`

Determina se o campo será indexado. O default é `False` para a maioria dos tipos de campos, mas é `True` em alguns casos (ex. `SlugField`).

`primary_key=False`

Determina se o campo é a chave primária. Pouco usado, porque a melhor prática é deixar o Django criar um `AutoField` com o nome `id`. Implica na criação de um índice.

`db_column=None`

Determina o nome da coluna no banco de dados SQL. O default `None` implica que a coluna terá o mesmo nome do campo, exceto no caso dos campos referenciais que ganham o sufixo `_id` (ex. `editora_id`).

`db_tablespace=None`

Em servidores Oracle, determina o *tablespace* a ser usado para os índices do campo. O parâmetro não tem efeito no PostgreSQL, no MySQL e no SQLite.

Parâmetros que definem a apresentação

Em ordem de utilidade (subjetiva).

`verbose_name=None`

Rótulo (*label*) do campo em formulários gerados pelo Django. Usado principalmente para associar rótulos acentuados (ex. `u"edição"`). Recomendável usar `unicode`.

`help_text=''`

Texto de ajuda do campo. Usado em formulário gerados pelo Django. Útil para exibir exemplo de preenchimento (ex. `help_text=u'ex. (11)8432-0333'`). Recomendável usar `unicode`.

`default=NOT_PROVIDED`

Valor default do campo. Se for um valor simples, pode ser implementado na DDL. Mas também pode ser um *callable*, que será invocado sempre que o objeto for instanciado.

Parâmetros que definem a validação

Em ordem de utilidade (subjetiva).

`max_length=None`

Tamanho máximo do conteúdo do campo para validação. Parâmetro obrigatório em campos `CharField` e derivados; não usado em vários tipos de campos. Pode ser implementado na DDL como o tamanho do `VARCHAR`.

`blank=False`

Determina se o campo pode ser validado com seu conteúdo vazio "". Os autores do Django sempre preferem usar campos tipo caractere que aceitam brancos em vez de nulos.

`choices`

Conjunto de valores válidos para o campo. Veja como em Parâmetro choices.

`unique_for_date=None`
`unique_for_month=None`
`unique_for_year=None`

Determina que o valor deste campo deve ser único em relação ao campo data especificado.

Parâmetro `choices`

O parâmetro deve ser um iterável (*iterable*) que produz duplas `(valor,legenda)` onde o valor será o conteúdo da escolha (ex. `'cafe'`) e legenda é o que será exibido para o usuário (ex. `u'Café expresso'`))

```
BEBIDAS = (('cafe',u'Café expresso'),
           ('mate',u'Chá mate'),
           ('chocolate',u'Chocolate quente'),
           )

class Pedido(models.Model):
    bebida = models.CharField(max_length=16, choices=BEBIDAS)
```

html

Em HTML, as opções acima podem ser exibidas assim:

```
<select name="bebidas">
    <option value="cafe">Café expresso</option>
    <option value="mate">Chá mate</option>
    <option value="chocolate">Chocolate quente</option>
</select>
```

Para cada campo `x` com parâmetro `choices`, o modelo ganha dinamicamente um método `get_x_display(v)` para obter a legenda corresponende a um valor.

Meta-opções para modelos

Em ordem de utilidade (subjetiva).

`ordering`

Estabelece a ordenação padrão dos resultados consultas a este modelo. O valor deste atributo é uma sequência de nomes de campos. Use `-` como prefixo de um campo para definir ordem descendente:

```
ordering = ['-dt_publicacao', 'editoria']
```

`unique_together`

Estabelece a restrição de unicidade para conjuntos de campos.

`verbose_name`, `verbose_name_plural`

Define o nome do modelo (singular e plural) para apresentação na interface administrativa.

`get_latest_by`

Estabelece o campo `DateTime` a ser usado como critério para o método de consulta `latest`.

`order_with_respect_to`

Estabelece qual campo `ForeignKey` determina a ordem relativa dos itens. Ver *ordenar-relacionados*.

`abstract`

Define que este é um modelo abstrato (*abstract model*), que não será persistido em uma tabela mas será usado para definir um esquema reutilizável por herança.

`db_table`

Define o nome da tabela que corresponde ao modelo. Quando esta opção não é usada o nome da tabela é `aplicacao_modelo` (ex.: `catalogo_livro` é o modelo `Livro` da aplicação `catalogo`.

`db_tablespace`

Estabelece o *tablespace* que será usado para armazenar os dados deste modelo. Não tem efeito na maioria dos bancos de dados suportados pelo Django 1.0.

Métodos especiais

Os seguintes métodos, se definidos em um modelo, são utilizados pelo Django:

`__unicode__`

Devolve a representação em unicode do objeto; por exemplo, para um livro esta representação pode ser o seu título. Usado em várias partes do admin do Django para representar o objeto em listagens e combos.

`get_absolute_url`

Devolve o camiho a partir da raiz do site até o objeto. Usado pelo admin do Django para exibir um botão **View on site** com link para a página pública do objeto. Essencial para qualquer view que precisa gerar links para objetos, por exemplo, uma página de resultados de busca. Veja exemplo em *primeiro-template*.

Django ORM: o básico

O que o ORM oferece

- independência em relação ao banco de dados SQL
- acesso direto a objetos relacionados
- implementação fácil e flexível de operações *CRUD*
- validação de campos
- transações *ACID*

API do ORM: exemplo de interação

Os modelos ganham por default um atributo `«Modelo».objects` que é um *manager*, através do qual você acessa toda a coleção de objetos do modelo (ou seja, operações no banco de dados a nível de tabela, e não registro).

por baixo dos panos

A maioria dos métodos de managers na verdade são delegados para um `QuerySet`, e devolvem instâncias de `QuerySet`. Por exemplo, a chamada `Livro.objects.all()` devolve um `QuerySet` que engloba todos os registros da tabela de livros.

usando o shell do django

```
$ ./manage.py shell

>>> from biblio.catalogo.models import *
>>> alice = Livro.objects.get(isbn='9780393048476')
>>> for c in alice.criador_set.all(): print c
...
Lewis Carroll
Martin Gardner
John Tenniel
>>> lc = alice.criador_set.get(nome__contains='Carroll')
>>> print lc.biografia.texto
Charles Lutwidge Dodgson, ou Lewis Carrol (Cheshire,
27 de janeiro de 1832 – Guildford, 14 de Janeiro de
1898) foi um escritor e matemático britânico
>>>
```

Métodos de Managers e QuerySets

Os mais usados são:

listar todos

«qs».all()

Devolve um QuerySet com todos os objetos do modelo (isto é, todos os registros da tabela correspondente).

```
>>> noticias.objects.all()
[<noticia1>,<noticia2>...]
```

filtrar

`«qs».filter(«critério1», «critério2», ...)`

Devolve um `QuerySet` com todos os objetos do modelo selecionados pelo critério, ou seja, gerando uma em SQL uma cláusula `WHERE` com os critérios combinados por `AND`. Ver *criterios*.

```
>>> noticias.objects.filter(tipo=1)
```

pegando um registro

`«qs».get(«critério1», «critério2», ...)`

Devolve **o único** objeto do modelo selecionado pelos critérios.

```
>>> noticias.objects.get(id=15)
<noticia15>
```

Exceptions

Se nenhum objeto é encontrado lança «modelo».DoesNotExist.

Se mais de um objeto é encontrado lança «modelo».MultipleObjectsReturned.

ordenação

«qs».order_by(«campo1», «campo2», ...)
Determina a ordenação do resultado pelos campos indicados. Se o nome de um campo for precedido de **-** então a ordem é descendente.

Ex. para obter as 5 notícias mais recentes:

```
>>> noticias.objects.order_by('-dt_public')[:5]
```

Seleção de objetos referentes

`«qs».select_related(«campo1», «campo2», ..., depth=0)`

Força o ORM a realizar *joins* para buscar os objetos referentes e evitar acessos posteriores ao banco de dados.

Os «campos» são nomes de campos de referência (`ForeignKey` etc.). Pode-se usar a sintaxe `referente__campo`.

O único parâmetro nomeado aceito é `depth`, e serve para limitar a extensão dos relacionamentos a serem recuperados. `*fields` e `depth` não podem ser usados ao mesmo tempo.

```
>>> noticias.objects.all().select_releated(depth=1)
```

Critérios para buscar objetos

Os critérios de busca usados em métodos de `QuerySet` são *argumentos nomeados <keyword argument>*, com nomes formados por atributos do modelo e operadores como `contains`, `in` ou `isnull`, unidos por `__` (dois underscores)

```
>>> lc = alice.criador_set.get(nome__icontains='Carroll')
# operador __icontains
```

Exemplos

Alguns exemplos de critérios:

exata

«campo»__exact=«valor»

Corresponde ao SQL SELECT ... WHERE «campo» = «valor». Por conveniência, o operador __exact pode ser omitido, ou seja, a busca exata pode ser escrita assim

```
>>> alice = Livro.objects.get(isbn='9780393048476')
# busca exata
```

like

`«campo»__icontains=«valor»`

Corresponde ao SQL `SELECT ... WHERE «campo» LIKE '%«valor»%'`. O prefixo `i` significa que este operador é indiferente a caixa alta ou baixa (*case insensitive*).

comparações

«campo»__lt=«valor»

Operador *menor que* (*less than*). Corresponde ao SQL SELECT ... WHERE «campo» < '%«valor»%'. O operador lte é *menor ou igual que* (*less than or equal*). Há também os operadores gt e gte.

```
>>> livros_curtos = Livro.objects.filter(qt_paginas__lt=100)
# <100 pgs.
```

Atributos dinâmicos

O ORM do Django cria dinamicamente os seguintes atributos em cada instância `i` de um *model*:

chave

`i.pk`

Nome alternativo para o campo `id`. Útil para acessar um campo de chave primária com outro nome, criado com o parâmetro `primary_key`.

relacionamentos

`i.«relacionado»_set`

Um *manager* para acessar o conjunto de objetos relacionados que fazem referência a este, através de campos `ForeignKeyField` ou `ManyToManyField`.

O nome deste atributo pode ser configurado pelo parâmetro `related_name` na definição do campo `ForeignKeyField` ou `ManyToManyField`.

one-to-one

`i.«relacionado»`

Acesso direto ao objeto que faz referência a este através de um `OneToOneField`.

ids

`i.«referente»_id`

Valor da chave estrangeira de um campo `ForeignKeyField`, `ManyToManyField` ou `OneToOneField`.

Para acessar diretamente o objeto apontado pelo campo, use `i.«referente»`.

Métodos dinâmicos

O ORM do Django cria dinamicamente os seguintes métodos em cada instância `i` de um *model*:

choices

`i.get_«opção»_display(valor)`
Devolve a legenda que corresponde ao valor em um campo «opção» criado com o parâmetro `choices`.

ids dos relacionados

`i.get_«objeto»_order()`

Devolve uma lista com as chaves primárias dos objetos relacionados, em ordem.

inverso

`i.set_«objeto»_order(lista)`

Dada de uma lista de chaves primárias, redefine a ordem dos objetos relacionados.

proximo

`i.get_next_by_«datahora»()`

Devolve a próxima instância em ordem cronológica de acordo com o campo «datahora».

anterior

`i.get_previous_by_«datahora»()`

Devolve a instância anterior em ordem cronológica de acordo com o campo «datahora».

Ordenação de objetos relacionados

Às vezes a ordem dos objetos em um `«relacionado»_set` é importante (por exemplo, os autores de um livro devem ser citados na ordem correta).

O parâmetro `order_with_respect_to` estabelece que os objetos relacionados devem manter sua ordem em relação aos seus referentes (ex. créditos em relação a livros).

```
class Credito(models.Model):
    livro = models.ForeignKey(Livro)
    criador = models.ForeignKey('Criador')
    papel = models.CharField(max_length=64, blank=True)

    class Meta:
        order_with_respect_to = 'livro'
```

A ordem é mantida através de um campo `_order` (integer) criado automaticamente na tabela deste modelo.

Ordenação de objetos relacionados (cont.)

O modelo referente (apontado pela `ForeignKey`) ganha os métodos dinâmicos `get_«item»_order` e `set_«item»_order` que permitem ler e alterar a ordem relativa dos itens relacionados.

exemplo

```
>>> from biblio.catalogo.models import *
>>> livro = Livro.objects.get(isbn='9780393048476')
>>> livro
<Livro: The Annotated Alice>
>>> livro.get_credito_order()
[1, 2, 3]
>>> for c in livro.credito_set.all(): print c
The Annotated Alice: Lewis Carroll (autor)
The Annotated Alice: Martin Gardner (editor)
The Annotated Alice: John Tenniel (ilustrador)
>>> livro.set_credito_order([1,3,2])
>>> for c in alice.credito_set.all(): print c
The Annotated Alice: Lewis Carroll (autor)
The Annotated Alice: John Tenniel (ilustrador)
The Annotated Alice: Martin Gardner (editor)
```

Configuração da interface administrativa

O mínimo necessário

Para habilitar a interface administrativa do Django:

- em `settings.py`, instale a aplicação `django.contrib.admin`
- em `urls.py`, descomente as linhas ligadas ao admin
- execute o comando `./manage.py syncdb` para que o Django crie as tabelas administrativas

instalando

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.admin',  # <----
)
```

urls

```
from django.contrib import admin  # <----
admin.autodiscover()              # <----

urlpatterns = patterns('',
    '...'
    (r'^admin/', include(admin.site.urls)),   # <----
    '...'
)
```

urls

```
$ ./manage.py syncdb
```

admin.py

```
from django.contrib import admin
from pizza.entrega.models import Pedido, Pizza, Entregador

class PizzaInline(admin.TabularInline):
    model = Pizza

class PedidoAdmin(admin.ModelAdmin):
    inlines = [PizzaInline]
    list_display = ('entrou', 'cliente', 'nome_entregador', 'partiu', 'despachado')
    list_display_links = ('entrou', 'cliente')

class PizzaAdmin(admin.ModelAdmin):
    list_display = ('pedido', '__unicode__')

admin.site.register(Pedido, PedidoAdmin)
admin.site.register(Pizza, PizzaAdmin)
admin.site.register(Entregador)
```

Opções na definição do ModelAdmin

Na instância de ModelAdmin:

```
class ClienteAdmin(admin.ModelAdmin):
    list_display = ('fone', 'contato', 'endereco')
    list_display_links = ('fone', 'contato')
    search_fields = ('fone', 'contato', 'logradouro', 'numero')
```

Formatação de listas

`list_display=«tupla-de-atributos»`

Transforma a listagem em uma tabela onde cada atributo é uma coluna. Os atributos podem ser campos ou métodos do `Model`, métodos do `ModelAdmin` ou simples funções que aceitam um objeto como argumento e devolvem o valor a ser exibido. É comum colocar um atributo `short_description` em tais métodos e funções para rotular o cabeçalho da coluna. Ver *admin-model-ops*.

`list_display_links=«tupla-de-atributos»`

Determina quais campos na listagem ganham links para o form de edição do item. Por default, apenas o campo da primeira coluna ganha link.

`list_per_page=«int»`

Determina o número máximo de itens por página na listagem. O default é 100.

`list_select_related=«bool»`

Determina se o Django ORM deve buscar os objetos relacionados ao modelo da listagem, realizando *joins* para reduzir o número de consultas ao banco de dados. O default é `False`. Ver *select-related*.

`ordering=«tupla-de-campos»`

Determina o critério de ordenação padrão da listagem. No admin do Django 1.0x, apenas o primeiro item é levado em conta.

Filtros e listas hierárquicas

`search_fields=«tupla-de-campos»`

Faz surgir no topo da listagem uma caixa de busca para selecionar os resultados buscando nos campos indicados na «tupla-de-campos»

`date_hierarchy=«campo-data»`

Quebra a listagem por uma hierarquia de datas (ano, dia, mês...)

`list_filter=«tupla-de-campos»`

Faz surgir uma barra lateral esquerda que permite a filtrar os resultados segundo o valor dos campos indicados na «tupla-de-campos». Os campos podem ser `BooleanField`, `CharField`, `DateField`, `DateTimeField`, `IntegerField` ou `ForeignKey`.

Opções na definição do `Model`

Alguns metadados aplicados a métodos no modelo ou funções em `admin.py` alteram a exibição de resultados no admin.

marcador

`«func».boolean`

Se `True`, o admin exibe um marcador verde se o resultado for verdadeiro, ou vermelho se não for.

exemplo

```
class Pedido(models.Model):
    '...'
    def despachado(self):
        return self.entregador and self.partida
    despachado.boolean = True
```

html no admin

`«func».allow_tags`

Se `True`, os tags HTML contidos no resutado ficam intactos; do contrário, eles são suprimidos (suprimir tags é o comportamento padrão, por motivos de segurança).

exemplo

```
class Tarefa(models.Model):
    '...'
    def rotulo(self):
        fmt = '''<span style="color: #%s;">%s</span>'''
        return fmt % (self.cor(), self.prioridade)
    rotulo.allow_tags = True
    rotulo.short_description = u'rótulo'
    rotulo.admin_order_field = 'prioridade'
```

abreviatura

`«func».short_description`

Define o nome da coluna onde o resultado será exibido nas listagens do admin. Ver *admin-lists*.

ordem dos campos

`«func».admin_order_field`

Define o campo do modelo a ser usado para ordenar os resultados quando o usuário pedir a ordenação por esta coluna no admin. Sem este atributo, colunas geradas por métodos não podem ser usadas para ordenação, pois o admin utiliza o banco de dados para fazer a ordenação.

Opções na definição do `Model` (cont.)

```
class Pedido(models.Model):
    inclusao = models.DateTimeField(auto_now_add=True)
    cliente = models.ForeignKey(Cliente)
    entregador = models.ForeignKey('Entregador', null=True, blank=True)
    partida = models.TimeField(null=True, blank=True)

    class Meta:
        ordering = ['-inclusao']

    def despachado(self):
        return (self.entregador is not None) and (self.partida is not None)
    despachado.boolean = True
```

Atenção

no admin, apenas o primeiro critério de ordenação defindo em `Meta.ordering` é usado

```
http://docs.djangoproject.com/en/dev/ref/models/options/#ordering
```

Views, URLs e templates: o básico

Views genéricas

Vamos começar o tema das views apresentando as views genéricas que vêm prontas com o Django. A documentação do Django considera as views genéricas um tópico avançado, mas temos três ótimos motivos para começar por elas:

1

usando as views genéricas não precisamos escrever código Python para tratar *requests*, e podemos praticar rapidamente a configuração de URLs e a programação de templates, que são as principais novidades deste capítulo

2

conhecendo bem as views genéricas você evita "reinventar a roda" e escrever código desnecessariamente, seguindo os princípios *DRY* e *KISS*

3

mesmo quando as views genéricas incluídas no Django não resolverem o seu problema, você poderá se inspirar em suas convenções para criar as suas próprias views parametrizadas, tornando mais flexível a sua aplicação e seguindo o princípio *DRY*

referências

A melhor referência para views genéricas ainda é o **Apêndice D** do **Django Book (primeira edição)**:

```
http://djangobook.com/en/1.0/appendixD/
```

A referência oficial é a mais atualizada mas não tem os exemplos do Django Book, por isso é mais difícil de ler:

```
http://docs.djangoproject.com/en/dev/ref/generic-views/
```

Localização dos templates

a busca por templates no sistema de arquivos é feita por funções configuradas em `settings.py`

```
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.load_template_source',
    'django.template.loaders.app_directories.load_template_source',
    # 'django.template.loaders.eggs.load_template_source',
)
```

por app

```
loaders.app_directories.load_template_source
```

permite que cada aplicação tenha seu próprio diretório de templates

default

as *generic views* por convenção procuram templates em locais como:

```
«aplicação»/«modelo»_detail.html
```

assim, a melhor forma de organizar os templates no sistema de arquivos é em diretórios como segue (sim, «aplicação» aparece duas vezes)

```
«projeto»/«aplicação»/«templates»/«aplicação»/*.html
```

Configuração das URLs

Django usa expressões regulares configuradas no módulo `urls.py` para analisar as URLs das requisições e invocar a *view* apropriada para cada padrão de URL

Modularidade

em um projeto modular, recomenda-se que cada aplicação tenha seu próprio módulo `«aplicação»/urls.py`, estes são incluídos no `urls.py` principal na raiz do projeto

exemplo

```
urlpatterns = patterns('',
    (r'^cat/', include('biblio.catalogo.urls')),
    (r'^admin/doc/', include('django.contrib.admindocs.urls')),
    (r'^admin/', include(admin.site.urls)),
    (r'^db/(.*)', databrowse.site.root),
)
```

por app

em «aplicação»/urls.py a análise dos caminhos de URLs continua

```
urlpatterns = patterns('',
    url(r'^$', list_detail.object_list, livros_info),
    url(r'^livro/(?P<object_id>\d+)/$', list_detail.object_detail, livros_info),
)
```

http://exemplo.com/cat/ aciona a *view* object_list

http://exemplo.com/cat/livro/3/ aciona object_detail

Configuração de *views* genéricas

`urls.py` é o único código Python necessário para uma *generic view* funcionar; por exemplo, veja o módulo `biblio/catalogo/urls.py`:

```
 1 from django.conf.urls.defaults import *
 2 from django.views.generic import list_detail
 3
 4 from biblio.catalogo.models import Livro
 5
 6 livros_info = {
 7     'queryset' : Livro.objects.all(),
 8 }
 9
10 urlpatterns = patterns('',
11     url(r'^$', list_detail.object_list, livros_info),
12     url(r'^livro/(?P<object_id>\d+)/$', list_detail.object_detail, livros_info),
13 )
```

linha 2: importação do módulo `views.generic.list_detail`

linhas 6 a 8: dicionário com parâmetro para as *generic views*

linhas 10 a 13: configuração das *generic views*

linha 12: o grupo nomeado `(?P<object_id>\d+)` é passado para a *view* como um parâmetro de mesmo nome

Primeiro template: `livro_list.html`

o caminho do template para a view genérica `list_detail.object_list` segue a convenção `«aplicação»/«modelo»_list.html`, em caixa baixa; os nomes da aplicação e do modelo são obtidos por introspecção do parâmetro `queryset`

o contexto do template inclui a variável `object_list`, referência ao parâmetro `queryset`

```
<h1>Livros</h1>

<table border="1">
  <tr><th>ISBN</th><th>Título</th></tr>
  {% for livro in object_list %}
    <tr>
      <td>{{ livro.isbn }}</td>
      <td>
        <a href="{{ livro.get_absolute_url }}">{{ livro.titulo }}</a>
      </td>
    </tr>
  {% endfor %}
</table>
```

Segundo template: `livro_detail.html`

o nome do template para a view genérica `list_detail.object_detail` segue a convenção `«aplicação»/«modelo»_detail.html`, sempre em caixa baixa

o contexto do template inclui a variável `object`, referência ao objeto localizado através de `queryset.get(id=object_id)`

```
1 <h1>Ficha catalográfica</h1>
2
3 <dl>
4     <dt>Título</dt>
5         <dd>{{ object.titulo }}</dd>
6     <dt>ISBN</dt>
7         <dd>{{ object.isbn }}</dd>
8 </dl>
```

O problema do caminho da aplicação nas URLs

O funcionamento das *views* genéricas de listagem/detalhe dependem do método `get_absolute_url` para produzir os links da listagem para a página de detalhe. Eis uma implementação fácil de entender:

```
class Livro(models.Model):
    '...'
    def get_absolute_url(self):
        return '/cat/livro/%s/' % self.id
```

Este código é simples, mas viola o princípio *DRY*, pois o prefixo *cat/* da URL está definido no módulo `urls.py` do projeto:

```
urlpatterns = patterns('',
    '...'
    (r'^cat/', include('biblio.catalogo.urls')),
    '...'
)
```

Isto significa que se um administrador decidir mudar o prefixo das URLs da aplicação `catalogo`, o método `get_absolute_url` do livro deixará de funcionar.

Solução: views nomeadas e o *decorator* `permalink`

A solução do problema envolve duas alterações, ambas dentro da aplicação `catalogo`:

urls.py

1. no módulo `urls.py` da aplicação, a configuração da view de detalhe recebe um nome (último argumento na linha 4 do trecho abaixo):

```
1 urlpatterns = patterns('',
2     url(r'^$', list_detail.object_list, livros_info),
3     url(r'^livro/(?P<object_id>\d+)/$', list_detail.object_detail,
4         livros_info, 'catalogo-livro-detalhe'),
5 )
```

models.py

2. no módulo `models.py` da aplicação, o método `get_absolute_url` recebe o *decorator* `permalink` e é alterado para devolver uma tupla no formato

```
(«nome-da-view-url»,
«parâmetros-posicionais»,
«parâmetros-nomeados»)
```

```python
class Livro(models.Model):
    '...'
    @models.permalink
    def get_absolute_url(self):
        #return '/cat/livro/%s/' % self.id
        return ('catalogo-livro-detalhe', (), {'object_id':self.id})
```

Views genéricas incluídas com o Django

todas são submodulos de

```
django.views.generic
```

por exemplo

```
django.views.generic.simple.redirect_to
```

Simples

```
simple.direct_to_template
simple.redirect_to
```

Exemplo

```
from django.views.generic.simple import direct_to_template
from django.views.generic.simple import redirect_to
from django.conf.urls.defaults import *

urlpatterns = patterns('',
  (r'^$', direct_to_template, {'template': "home/index.html"}),
  (r'^cadastro/$',redirect_to, {'url':'usuario/novo/'}),
)
```

listagem/detalhe

```
list_detail.object_list
list_detail.object_detail
```

Exemplo

```
from django.views.generic.list_detail import object_detail

foos = {'queryset':Foo.objects.order_by('nome'),"paginate_by":50}
foos_slug = dict(foos, slug_field='slug')
...

(r'^$','django.views.generic.list_detail.object_list', foos),
(r'^foo/(?P<object_id>\d+)/$',object_detail,foos),
(r'^foo/(?P<slug>[-\w]+)/$',object_detail,foos_slug),
```

criar/alterar/deletar objetos

```
create_update.create_object
create_update.update_object
create_update.delete_object
```

navegar por arquivos cronológicos

```
date_based.archive_index
date_based.archive_year
date_based.archive_month
date_based.archive_week
date_based.archive_day
date_based.archive_today
date_based.object_detail
```

Principais funções para configuração de URLs

Usadas em `urls.py`:

urls patterns

```
patterns(prefixo, url1, url2, ...)
```

Define uma sequência de padrões de URLs. O prefixo serve para abreviar as referências às views em forma de strings, sendo pre-pendado a todas as views do conjunto. Não tem utilidade quando se usa referências diretas às views.

Os demais argumentos são chamadas de `url`, ou tuplas formadas por item na ordem exata dos parâmetros da função `url` (ver abaixo).

Sequências de padrões de URLs podem ser concatenadas.

Exemplo

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template

from pizza import settings

urlpatterns = patterns('',
  (r'^$', direct_to_template, {'template': "home/index.html"}),
)
if settings.DEBUG:
    urlpatterns += patterns('',
        (r'deb/^$', direct_to_template, {'template': "debug.html"}),
    )
```

url(regex, ref_view, extra_dict=None, name='')

Define um padrão de URL vinculado a uma view. Os parâmetros são:

`regex`

Expressão regular que será aplicada à URL. Grup
anônimos (ex. `(+\d)`) são passados para a view co
parâmetros posicionais, em ordem. Grupos nomeac
(ex. `(?P<object_id>\d+)`) são passados co
parâmetros nomeados. A melhor prática é u
sempre grupos nomeados para reduzir o acoplame
da configuração com a definição da view.

`ref_view`
Referência a uma view. Pode ser uma string ou u
referência real à função da view. No segundo caso
preciso importar a função no topo do mód
`urls.py`.

`extra_dict`
Dicionário com valores adicionais a serem passado
view. Opcional.

`name`
Nome da view, para referência reversa.

Enquanto isso no template

Os templates de django por padrao tem a terminação `html` e existem dois tipos de marcadores de tags

`{% ALGO %}` que executam coisas

`{{ ALGO }}` que imprimem o resultado

Existem dois tipos de funções que podem ser chamadas nos templates

Filtros e Templatetags

Parecem python, mas não sao exatamente.

Metodos não tem () por exemplo

```
{% foo.bar %}
```

ele na verdade tenta

```
foo.bar()
foo[bar]
```

Blocos

São a forma do django conseguir heranca com templates

base.html

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

no seu template

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

{% block content %}
    <h1>pizzas para {{cliente.nome}}</h1>
{% endblock %}
```

renderiza

```
<html>
    <head>
        <title>Minha pizzaria</title>
    </head>
    <body>
        <h1>pizzas para Fulano</h1>
    </body>
</html>
```

Template tags

Ver na documentação do django

```
{{csrf_token}}
#protecao contra XSS

{%cycle "claro" "escuro"%}

{% firstof var1 var2 "padrao" %}
```

For

```
{% for livro in livros %}
    {{ livro.nome }}<br/>
{% empty %}
    Alguem queimou os livros
{% endfor %}
```

For(2)

Dentro de um loop você ganha variaveis novas

```
forloop.counter # contador base 1
forloop.counter0 # contador base 0
forloop.revcounter # quantas interacoes faltam base 0
forloop.revcounter0 # base 0
forloop.first # booleano se é o primeiro
forloop.last # booleano se é o último
forloop.parentloop # o loop pai para nested loops
```

Exemplo

```
{% for livro in livros %}
    {% if forloop.first %}
        <table>
    {% endif %}
    <tr><td>{{forloop.count}}</td><td>{{ livro.nome }}</td></tr>
    {% if forloop.last %}
        </table>
    {% endif %}
{% empty %}
    <p>Alguem queimou os livros</p>
{% endfor %}
```

Exemplo

```
<table>
    <tr><td>1</td><td>Foo o livro</td></tr>
    <tr><td>2</td><td>A volta de Foo</td></tr>
    <tr><td>3</td><td>O Foo e eu</td></tr>
</table>
```

ou

```
<p>Alguem queimou os livros</p>
```

if

Mudou no 1.2 e agora suporta Operadores booleanos complexos

```
== operator
!= operator
< operator
> operator
<= operator
>= operator
(not) in operator
```

Exemplo

```
{% if a == b or c == d and e %}
```

é equivalente em python a

```
if (a == b) or ((c == d) and e)
```

django < 1.1

usa

```
{% ifequal username "adriano"}
    sou eu
{% endifequal%}
{% ifnotequal username "adriano"}
    voce nao tem o meu username
{% endifnotequal %}
```

Template tags cont.

```
{% include "foo.html" %}

# comparar com

{% ssi /home/html/ljworld.com/includes/right_generic.html %}

{% now "jS F Y H:i" %} #usa a sintaxe do PHP

{{ url myapp:view-name }}
```

Filtros

Filtros modificam a saida de uma variavel ver:

```
http://docs.djangoproject.com/en/dev/ref/templates/builtins/
```

por exemplo

```
{{ foo.dt|date:"Y/m/d" }}
```

```
2010/09/25
```

Mais do que voce queria saber sobre Queries

Fazendo OR, usa o operador Q

```
from django.db.models import Q

Foo.objects.filter( Q(nome="joe")|Q(nome="ze") )
# WHERE nome = 'joe' OR nome = 'ze'

Foo.objects.filter( Q(nome="joe"),
                    Q(dt="20100101") | Q(dt="20091231"))
# WHERE nome = 'joe' AND (dt = '20100101" OR dt="20091231)
```

CUIDADO

Mas não pode misturar com os filtros normais

```
#ERRADO ERRADO ERRADO
Foo.objects.filter( nome="joe",
                    Q(dt="20100101") | Q(dt="20091231"))
# ERRADO ERRADO ERRADO
```

Um erro comum

```
#ERRADO ERRADO ERRADO
Foo.objects.filter(nome!="joe")
#ERRADO ERRADO ERRADO



#Certo
Foo.objects.exclude(nome="joe")
```

Views não genéricas

Uma view no django e' tudo que recebe um request e devolve uma response

Sao mapeadas pelas urls.py

Exemplo mais simples

urls.py

```
(r'^$', "pizza.entrega.views.index"),
```

views.py

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("home")
```

Usando templates em views

Facilitador render_to_response

```
from django.shortcuts import render_to_response

def index(request):
    return render_to_response("home.html", {} )
```

Pegando parametros

3 formas:

na url

GET

POST

na url

urls.py

```
(r'^foo/(?P<object_id>\d+)/$',show_foo),
```

views.py

```
def show_foo(request, object_id):
    pass
```

POST GET

/?pagina=5

```
def index(request):
    # existe tambem o request.POST e request.REQUEST
    pagina = request.GET.get("pagina", 1)
    return render_to_response("home.html", {"pagina":pagina} )
```

Testando se teve POST

```
if request.POST:
    pass

# ou

if request.method == 'POST':
    pass
```

Mais facilitadores

```
from django.shortcuts import get_object_or_404
from pizza.entrega.models import Cliente

def show_cliente(request, object_id):
    cliente = get_object_or_404(Cliente, id=object_id)
    ...
```

Forms

```
from django import forms

class SearchForm(forms.Form):
    q = forms.CharField()
    dt = forms.DateField()

def listar(request):
    form = SearchForm()
    if request.POST:
        form = SearchForm(request.POST)
        if form.is_valid():
            q = form.cleaned_data["q"]
            # faz algo com q
    return render_to_response('avulso.html', {'form':form})
```

No template

```
<form method="post" action=".">
<table>
 {{form}}
</table>
</form>
```

Model form

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

class ClienteForm(forms.ModelForm):
    class Meta:
        model = Cliente

def criar(request):
    form = ClienteForm()
    if request.POST:
        form = ClienteForm(request.POST)
        if form.is_valid():
            cliente = form.save()
            return HttpResponseRedirect("/show/%s/"% cliente.id)
    return render_to_response('avulso.html', {'form':form})
```

Mais facilitadores ainda

Decoradores

```
from django.contrib.auth.decorators import login_required

@login_required
def foo(request):
    pass
```

Pre-requisitos

se nao logado ele redireciona para

```
settings.LOGIN_URL # por padrao /accounts/login/
```

e voce precisa ter essa url ou usar a do contrib. Incluir essa linha no urls.py

```
(r'^accounts/login/$', 'django.contrib.auth.views.log
```

Form de login

criar um template em registration/login.html

```
{% extends "base.html" %}
{% block content %}
{% if form.errors %}
<p>Senha ou Usuario incorretos, tente novamente</p>
{% endif %}
<form method="post" action="{% url django.contrib.auth.views.login %}">
{% csrf_token %}
<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 %}
```

Glossário

abstract model

Em Django um *abstract model* (modelo abstrato) é um *model* que não pode ser instanciado e não tem uma tabela correspodente no banco de dados. Sua utilidade é definir um conjunto de atributos e métodos comuns a vários modelos que serão suas subclasses. Um modelo é definido como abstrato quando tem o atributo `abstract=True` em sua classe interna `Meta`.

application

Em Django uma *application* (aplicação) é um dos subsistemas que compõe um projeto (*project*). Para criar uma aplicação usa-se o comando `./manage.py startapp «nome-da-aplicação»`.

ACID

Atomicity, Consistency, Isolation, Durability (atomicidade, consistência, isolação e durabilidae): propriedades que asseguram a confiabilidade do processamento de transações.

callable

Em Python, um *callable* (invocável) é um objeto que pode ser acionado com o operador de invocação `()`. Isso inclui funções, métodos, classes e qualquer objeto que implemente um método `__call__`.

CRUD

Create, Read, Update, Delete (criar, ler, atualizar, apagar), as quatro operações básicas da persistência de dados.

decorator

Em Python, um *decorator* é uma função que modifica o comportamento de outra função; por exemplo, um *decorator* pode ser usado para logar todas as chamadas de uma função, ou cachear seus resultados.

DRY

Don't Repeat Yourself (não se repita): princípio de engenharia de software segundo o qual cada função, dado ou configuração deve aparecer uma e apenas uma vez em um sistema, pois cada duplicação torna muito mais difícil a manutenção e evolução futura do sistema.

iterable

Em Python um *iterable* (iterável) é uma coleção que pode ser percorrida item a item. Sequências, como listas e tuplas, são iteráveis, mas existem também iteráveis *preguiçosos* que geram seus valores sob demanda, como as expressões geradoras a partir do Python 2.4, ou as instâncias de `QuerySet` no Django.

keyword argument

Em Python um *keyword argument* (argumento nomeado) é um argumento de função passado no formato `nome=valor` no momento da invocação. Python vincula tal argumento ao parâmetro de mesmo nome declarado na definição da função. Se não existe parâmetro com este nome, mas existe um parâmetro com prefixo `**` (convencionalmente chamado de `**kwargs`), o argumento nomeado é passado para este parâmetro na forma de um item de dicionário. Ou seja, tipicamente o parâmetro ``kwargs recebe algo como `{'nome1':valor1,`

'nome2', valor2}.

KISS

Keep It Simple, Stupid (preserve a simplicidade, colega [tradução gentil]): princípio de engenharia de software segundo o qual a solução deve ser a mais simples possível capaz de atender aos requisitos do sistema (e não a mais elegante, ou a mais otimizada, ou aquela capaz de resolver um problema que um dia talvez exista). Eistein disse algo como "Things should be as simple as possible, but no simpler" ("As coisas devem ser tão simples quanto possível, mas não simples demais"). http://c2.com/cgi/wiki?EinsteinPrinciple

manager

Em Django um *manager* é um objeto presente em cada *model* que permite consultar ou alterar a coleção de instâncias do modelo no banco de dados através de métodos como `all()`, `filter()`, `delete()` etc. Por default, cada modelo tem um manager chamado `objects`, mas o programador pode criar modelos adicionais (por exemplo, um modelo chamado `ativos` pode limitar as consultas aos objetos considerados ativos em uma dada aplicação). Managers são instâncias de `django.db.models.manager.Manager` ou de subclasses desta.

model

Em Django um *model* (modelo) é uma classe derivada de `django.db.models.Model` que representa um tipo de objeto armazenado em uma tabela no banco de dados (exceto quando se trata de um *abstract model*). Por convenção, dentro de uma aplicação (*application*) Django as *views* são criadas em arquivos `models.py`.

package

Em Python um *package* (pacote) é um diretório que contém módulos que podem ser importados. Para ser reconhecido como um *package*, o diretório precisa conter um módulo chamado `__init__.py`, que pode ser um arquivo vazio.

project

Em Django um *project* (projeto) é um *package* que contém na sua raiz um arquivo *settings.py* com as configurações globais de várias *aplicações*<*application*>.

template

Um *template* (gabarito) é um arquivo que representa genericamente um tipo de página com conteúdo variável. Normalmente o *template* é formado por código HTML com marcações especiais da linguagem de tags do Django. Os *templates* podem ser renderizados, processo pelo qual as marcações do Django são processadas e substituidas por valores específicos, produzindo código HTML puro (sem tags do Django).

view

No Django, uma *view* (visão) é uma função que aceita como primeiro parâmetro um objeto `request` que representa uma requisição Web (além de outros parâmetros), e trata esta requisição, normalmente produzindo um *template* HTML renderizado. Por convenção, dentro de uma aplicação (*application*) Django as *views* são criadas em arquivos `views.py`.