Community Experience Distilled

Django By Example

Create your own line of successful web applications with Django

Antonio Melé

[PACKT] PUBLISHING open source* community experience distilled

《Django By Example》中文版

目录

第一章 创建一个 blog 应用

在这本书中，你将学习如何创建完整的 Django 项目，可以在生产环境中使用。假如你还没有安装 Django，在本章的第一部分你将学习如何安装。本章会覆盖如何使用 Django 去创建一个简单的 blog 应用。本章的目的是使你对该框架的工作有个基本概念，了解不同的组件之间是如何产生交互，并且教你一些技能通过使用一些基本功能方便地创建 Djang 项目。你会被引导创建一个完整的项目但是不会对所有的细节都进行详细说明。不同的框架组件将在本书接下来的章节中进行介绍。
本章会覆盖以下几点：

- 安装 Django 并创建你的第一个项目
- 设计模型（models）并且生成模型（model）数据库迁移
- 给你的模型（models）创建一个管理站点
- 使用查询集（QuerySet）和管理器（managers）
- 创建视图（views），模板（templates）和 URLs
- 给列表视图（views）添加页码
- 使用 Django 内置的视图（views）

安装 Django

如果你已经安装好了 Django，你可以直接略过这部分跳到*创建你的第一个项目*。Django 是一个 Python 包因此可以安装在任何的 Python 的环境中。如果你还没有安装 Django，这里有一个快速的指南帮助你安装 Django 用来本地开发。
Django 需要在 Python2.7 或者 3 版本上才能更好的工作。在本书的例子中，我们将使用 Python 3。如果你使用 Linux 或者 Max OSX，你可能已经有安装好的 Python。如果你不确定你的计算机中是否安装了 Python，你可以在终端中输入 *python* 来确定。如果你看到以下类似的提示，说明你的计算机中已经安装好了 Python:

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

如果你计算机中安装的 Python 版本低于 3，或者没有安装，下载并安装 Python 3.5.0 从 http://www.python.org/download/（**译者注：最新已经是 3.6.0 了，Django2.0 将不再支持 pytyon2.7，所以大家都从 3 版本以上开始学习吧**）。
由于你使用的是 Python3，所以你没必要再安装一个数据库。这个 Python 版本自带 SQLite 数据库。SQLLite 是一个轻量级的数据库，你可以在 Django 中进行使用用来开发。如果你准备在生产环境中部署你的应用，你应该使用一个更高级的数据库，比如 PostgreSQL，MySQL 或 Oracle。你能获取到更多的信息关于数据库和 Django 的集成通过访问 https://docs.djangoproject.com/en/1.8/topics/install/#database-installation 。

创建一个独立的 Python 环境

强烈建议你使用 virtualenv 来创建独立的 Python 环境，这样你可以使用不同的包版本对应不同的项目，这比直接在真实系统中安装 Python 包更加的实用。另一个高级之处在于当你使用 virtualenv 你不需要任何管理员权限来安装 Python 包。在终端中运行以下命令来安装 virtualenv：

```
pip install virtualenv
```

（译者注：如果你本地有多个 python 版本，注意 Python3 的 pip 命令可能是 pip3）
当你安装好 virtualenv 之后，通过以下命令来创建一个独立的环境：

```
virtualenv my_env
```

以上命令会创建一个包含你的 Python 环境的 my_env/目录。当你的 virtualenv 被激活的时候所有已经安装的 Python 库都会带入 my_env/lib/python3.5/site-packages 目录中。
如果你的系统自带 Python2.X 然后你又安装了 Python3.X，你必须告诉 virtualenv 使用后者 Python3.X。
通过以下命令你可以定位 Python3 的安装路径然后使用该安装路径来创建 virtualenv：

```
zenx\$ *which python3*
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3
zenx\$ *virtualenv my_env -p
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3*
```

通过以下命令来激活你的 virtualenv：

```
source my_env/bin/activate
```

shell 提示将会附上激活的 virtualenv 名，被包含在括号中，如下所示：

```
(my_evn)laptop:~ zenx$
```

你可以使用 *deactivate* 命令随时停用你的 virtualenv。
你可以获取更多的信息关于 virtualenv 通过访问 https://virtualenv.pypa.io/en/latest/ 。
在 virtualenv 之上，你可以使用 virtualenvwrapper 工具。这个工具提供一些封装用来方便的创建和管理你的虚拟环境。你可以在 http://virtualenvwrapper.readthedocs.org/en/latest/ 下载该工具。

使用 pip 安装 Django

（译者注：请注意以下的操作都在激活的虚拟环境中使用）
pip 是安装 Django 的第一选择。Python3.5 自带预安装的 pip，你可以找到 pip 的安装指令通过访问 https://pip.pypa.io/en/stable/installing/ 。运行以下命令通过 pip 安装 Django：

```
pip install Django==1.8.6
```

Django 将会被安装在你的虚拟环境的 Python 的 *site-packages/*目录下。
现在检查 Django 是否成功安装。在终端中运行 *python* 并且导入 Django 来检查它的版本：

```
>>> import django
>>> django.VERSION
DjangoVERSION（1, 8, 5, 'final', 0)
```

如果你获得了以上输出，Django 已经成功安装在你的机器中。
Django 也可以使用其他方式来安装。你可以找到更多的信息通过访问
https://docs.djangoproject.com/en/1.8/topics/install/ 。

创建你的第一个项目

我们的第一个项目将会是一个完整的 blog 站点。Django 提供了一个命令允许你方便的创建一个初始化的项目文件结构。在终端中运行以下命令：

```
django-admin startproject mysite
```

该命令将会创建一个名为 *mysite* 的项目。
让我们来看下生成的项目结构：

```
mysite/
    manage.py
    mysite/
        __init__.py
```

```
        settings.py
        urls.py
        wsgi.py
```

让我们来了解一下这些文件：

- *manage.py*:一个实用的命令行，用来与你的项目进行交互。它是一个对 *django-admin.py* 工具的简单封装。你不需要编辑这个文件。
- *mysite/*:你的项目目录，由以下的文件组成：
  - **init**.py:一个空文件用来告诉 Python 这个 *mysite* 目录是一个 Python 模块。
  - settings.py:你的项目的设置和配置。里面包含一些初始化的设置。
  - urls.py:你的 URL 模式存放的地方。这里定义的每一个 URL 都映射一个视图（view）。
  - wsgi.py:配置你的项目运行如同一个 WSGI 应用。

默认生成的 *settings.py* 文件包含一个使用一个 SQLite 数据库的基础配置以及一个 Django 应用列表，这些应用会默认添加到你的项目中。我们需要为这些初始应用在数据库中创建表。
打开终端执行以下命令：

```
cd mysite
python manage.py migrate
```

你将会看到以下的类似输出：

```
Rendering model states... DONE
Applying contenttypes.ooo1_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length...OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying sessions.0001_initial... OK
```

这些初始应用表将会在数据库中创建。过一会儿你就会学习到一些关于 *migrate* 的管理命令。

运行开发服务器

Django 自带一个轻量级的 web 服务器来快速运行你的代码，不需要花费额外的时间来配置一个生产服务器。当你运行 Django 的开发服务器，它会一直检查你的代码变化。当代码有改变，它会自动重启，将你从手动重启中解放出来。但是，它可能无法注意到一些操作，例如在项目中添加了一个新文件，所以你在某些场景下还是需要手动重启。
打开终端，在你的项目主目录下运行以下代码来开启开发服务器：

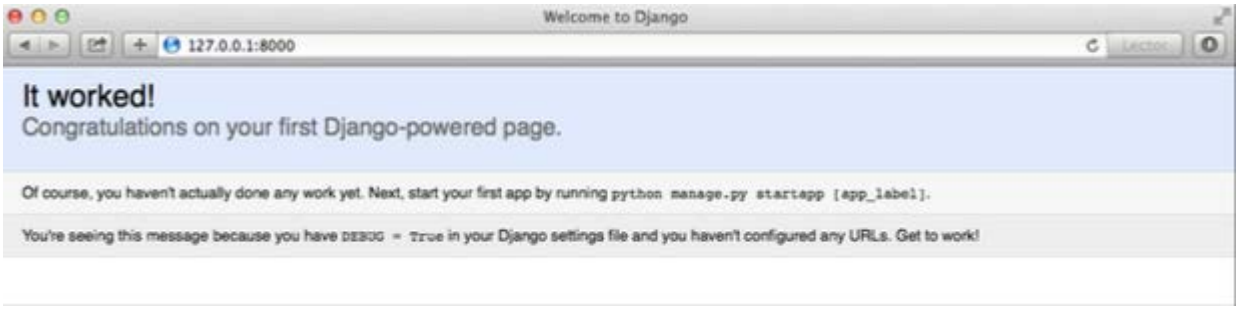
```
python manage.py runserver
```

你会看到以下类似的输出：

```
Performing system checks...

System check identified no issues (0 silenced).
November 5, 2015 - 19:10:54
Django version 1.8.6, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
```

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

现在，在浏览器中打开 http://127.0.0.1:8000/ ，你会看到一个告诉你项目成功运行的页面，如下图所示：



django-1-1

你可以指定 Django 在定制的 host 和端口上运行开发服务，或者告诉它你想要运行你的项目通过读取一个不同的配置文件。例如：你可以运行以下 manage.py 命令：

```
python manage.py runserver 127.0.0.1:8001 \
--settings=mysite.settings
```

这个命令迟早会对处理需要不同设置的多套环境启到作用。记住，这个服务器只是单纯用来开发，不适合在生产环境中使用。为了在生产环境中部署 Django，你需要使用真实的 web 服务让它运行成一个 WSGI 应用例如 Apache，Gunicorn 或者 uWSGI（**译者注：强烈推荐 nginx+uwsgi+Django**）。你能够获取到更多关于如何在不同的 web 服务中部署 Django 的信息，访问 https://docs.djangoproject.com/en/1.8/howto/deployment/wsgi/ 。
本书外额外的需要下载的章节*第十三章,Going Live* 包含为你的 Django 项目设置一个生产环境。

项目设置

让我们打开 settings.py 文件来看看你的项目的配置。在该文件中有许多设置是 Django 内置的，但这些只是所有 Django 可用配置的一部分。你可以通过访问 https://docs.djangoproject.com/en/1.8/ref/settings/ 看到所有的设置和它们默认的值。
以下列出的设置非常值得一看：

- DEBUG 一个布尔型用来开启或关闭项目的 debug 模式。如果设置为 True，当你的应用抛出一个未被捕获的异常时 Django 将会显示一个详细的错误页面。当你准备部署项目到生产环境，请记住一定要关闭 debug 模式。永远不要在生产环境中部署一个打开 debug 模式的站点因为那会暴露你的项目中的敏感数据。
- ALLOWED_HOSTS 当 debug 模式开启或者运行测试的时候不会起作用（**译者注：最新的 Django 版本中，不管有没有开启 debug 模式该设置都会启作用**）。一旦你准备部署你的项目到生产环境并且关闭了 debug 模式，为了允许访问你的 Django 项目你就必须添加你的域或 host 在这个设置中。
- INSTALLED_APPS 这个设置你在所有的项目中都需要编辑。这个设置告诉 Django 有哪些应用会在这个项目中激活。默认的，Django 包含以下应用：
  - o django.contrib.admin：这是一个管理站点。
  - o django.contrib.auth：这是一个权限框架。
  - o django.contrib.contenttypes：这是一个内容类型的框架。
  - o django.contrib.sessions：这是一个会话（session）框架。
  - o django.contrib.messages：这是一个消息框架。
  - o django.contrib.staticfiles：这是一个用来管理静态文件的框架
- MIDDLEWARE_CLASSES 是一个包含可执行中间件的元组。
- ROOT_URLCONF 指明你的应用定义的主 URL 模式存放在哪个 Python 模块中。
- DATABASES 是一个包含了所有在项目中使用的数据库的设置的字典。里面一定有一个默认的数据库。默认的配置使用的是 SQLite3 数据库。

- LANGUAGE_CODE 定义 Django 站点的默认语言编码。

不要担心你目前还看不懂这些设置的含义。你将会在之后的章节中熟悉这些设置。

项目和应用

贯穿全书，你会反复的读到项目和应用的地位。在 Django 中，一个项目被认为是一个安装了一些设置的 Django；一个应用是一个包含模型（models），视图（views），模板（templates）以及 URLs 的组合。应用之间的交互通过 Django 框架提供的一些特定功能，并且应用可能被各种各样的项目重复使用。你可以认为项目就是你的网站，这个网站包含多个应用，例如 blog，wiki 或者论坛，这些应用都可以被其他的项目使用。**（译者注：我去，我竟然漏翻了这一节- -|||，罪过罪过，阿米头发）**

创建一个应用

现在让我们创建你的第一个 Django 应用。我们将要创建一个勉强凑合的 blog 应用。在你的项目主目录下，运行以下命令：

```
python manage.py startapp blog
```

这个命令会创建 blog 应用的基本目录结构，如下所示：

```
blog/
    __init__.py
    admin.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

这些文件的含义：

- admin.py: 在这儿你可以注册你的模型（models）并将它们包含到 Django 的管理页面中。使用 Django 的管理页面是可选的。
- migrations: 这个目录将会包含你的应用的数据库迁移。Migrations 允许 Django 跟踪你的模型（model）变化并因此来同步数据库。
- models.py: 你的应用的数据模型（models）。所有的 Django 应用都需要拥有一个 *models.py* 文件，但是这个文件可以是空的。
- tests.py：在这儿你可以为你的应用创建测试。
- views.py：你的应用逻辑将会放在这儿。每一个视图（view）都会接受一个 HTTP 请求，处理该请求，最后返回一个响应。

设计 blog 数据架构

我们将要开始为你的 blog 设计初始的数据模型（models）。一个模型（model）就是一个 Python 类，该类继承了 *django.db.models.model*,在其中的每一个属性表示一个数据库字段。Django 将会为 *models.py* 中的每一个定义的模型（model）创建一张表。当你创建好一个模型（model），Django 会提供一个非常实用的 API 来方便的查询数据库。

首先，我们定义一个 *POST* 模型（model）。在 blog 应用下的 *models.py* 文件中添加以下内容：

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User


class Post(models.Model):
    STATUS_CHOICES = (
```

```
        ('draft', 'Draft'),
        ('published', 'Published'),
    )
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250,
                            unique_for_date='publish')
    author = models.ForeignKey(User,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10,
                              choices=STATUS_CHOICES,
                              default='draft')

    class Meta:
        ordering = ('-publish',)

    def __str__(self):
        return self.title
```

这就是我们给 blog 帖子使用的基础模型（model）。让我们来看下刚才在这个模型（model）中定义的各个字段含义：

- title： 这个字段对应帖子的标题。它是 *CharField*，在 SQL 数据库中会被转化成 VARCHAR。
- slug：这个字段将会在 URLs 中使用。slug 就是一个短标签，该标签只包含字母，数字，下划线或连接线。我们将通过使用 slug 字段给我们的 blog 帖子构建漂亮的，友好的 URLs。我们给该字段添加了 *unique_for_date* 参数，这样我们就可以使用日期和帖子的 slug 来为所有帖子构建 URLs。在相同的日期中 Django 会阻止多篇帖子拥有相同的 slug。
- author：这是一个 *ForeignKey*。这个字段定义了一个多对一（many-to-one）的关系。我们告诉 Django 一篇帖子只能由一名用户编写，一名用户能编写多篇帖子。根据这个字段，Django 将会在数据库中通过有关联的模型（model）主键来创建一个外键。在这个场景中，我们关联上了 Django 权限系统的 *User* 模型（model）。我们通过 *related_name* 属性指定了从 *User* 到 *Post* 的反向关系名。我们将会在之后学习到更多关于这方面的内容。
- body：这是帖子的主体。它是 *TextField*，在 SQL 数据库中被转化成 *TEXT*。
- publish：这个日期表明帖子什么时间发布。我们使用 Djnago 的 *timezone* 的 *now* 方法来设定默认值。This is just a timezone-aware datetime.now（**译者注：这句该咋翻译好呢**）。
- created：这个日期表明帖子什么时间创建。因为我们在这儿使用了 *auto_now_add*，当一个对象被创建的时候这个字段会自动保存当前日期。
- updated：这个日期表明帖子什么时候更新。因为我们在这儿使用了 *auto_now*，当我们更新保存一个对象的时候这个字段将会自动更新到当前日期。
- status：这个字段表示当前帖子的展示状态。我们使用了一个 *choices* 参数，这样这个字段的值只能是给予的选择参数中的某一个值。**（译者注：传入元组，比如`(1,2)`，那么该字段只能选择 1 或者 2，没有其他值可以选择）**

就像你所看到的的，Django 内置了许多不同的字段类型给你使用，这样你就能够定义你自己的模型（models）。通过访问 https://docs.djangoproject.com/en/1.8/ref/models/fields/ 你可以找到所有的字段类型。

在模型（model）中的类 *Meta* 包含元数据。我们告诉 Django 查询数据库的时候默认返回的是根据 *publish* 字段进行降序排列过的结果。我们使用负号来指定进行降序排列。

str()方法是当前对象默认的可读表现。Django 将会在很多地方用到它例如管理站点中。

如果你之前使用过 Python2.X，请注意在 Python3 中所有的 strings 都使用 unicode，因此我们只使用 ***str()***方法。***unicode()***方法已经废弃。**（译者注：Python3 大法好，Python2 别再学了，直接学 Python3 吧）**

在我们处理日期之前，我们需要下载 *pytz* 模块。这个模块给 Python 提供时区的定义并且 SQLite 也需要它来对日期进行操作。在终端中输入以下命令来安装 *pytz*：

```
pip install pytz
```

Django 内置对时区日期处理的支持。你可以在你的项目中的 *settings.py* 文件中通过 *USE_TZ* 来设置激活或停用对时区的支持。当你通过 *startproject* 命令来创建一个新项目的时候这个设置默认为 *True*。

激活你的应用

为了让 Django 能保持跟踪你的应用并且根据你的应用中的模型（models）来创建数据库表，我们必须激活你的应用。因此，编辑 *settings.py* 文件，在 *INSTALLED_APPS* 设置中添加 *blog*。看上去如下所示：

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',
 )
```

（译者注：该设置中应用的排列顺序也会对项目的某些方面产生影响，具体情况后几章会有介绍，这里提醒下）

现在 Django 已经知道在项目中的我们的应用是激活状态并且将会对其中的模型（models）进行自审。

创建和进行数据库迁移

让我们为我们的模型（model）在数据库中创建一张数据表格。Django 自带一个数据库迁移（migration）系统来跟踪你对模型（models）的修改，然后会同步到数据库。*migrate* 命令会应用到所有在 *INSTALLED_APPS* 中的应用，它会根据当前的模型（models）和数据库迁移（migrations）来同步数据库。

首先，我们需要为我们刚才创建的新模型（model）创建一个数据库迁移（migration）。在你的项目主目录下，执行以下命令：

```
python manage.py makemigrations blog
```

你会看到以下输出:

```
Migrations for 'blog':
    0001_initial.py;
        - Create model Post
```

Django 在 blog 应用下的 migrations 目录中创建了一个 *0001——initial.py* 文件。你可以打开这个文件来看下一个数据库迁移的内容。

让我们来看下 Django 根据我们的模型（model）将会为在数据库中创建的表而执行的 SQL 代码。*sqlmigrate* 命令带上数据库迁移（migration）的名字将会返回它们的 SQL，但不会立即去执行。运行以下命令来看下输出：

```
python manage.py sqlmigrate blog 0001
```

输出类似如下：

```
BEGIN;
CREATE TABLE "blog_post" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "title" varchar(250) NOT NULL, "slug"
varchar(250) NOT NULL, "body" text NOT NULL, "publish" datetime NOT NULL, "created" datetime NOT NULL, "updated"
datetime NOT NULL, "status" varchar(10) NOT NULL, "author_id" integer NOT NULL REFERENCES "auth_user" ("id"));
CREATE INDEX "blog_post_2dbcba41" ON "blog_post" ("slug");
CREATE INDEX "blog_post_4f331e2f" ON "blog_post" ("author_id");
COMMIT;
```

Django 会根据你正在使用的数据库进行以上精准的输出。以上 SQL 语句是为 SQLite 数据库准备的。如你所见，Django 生成的表名前缀为应用名之后跟上模型（model）的小写（blog_post），但是你也可以通过在模型（models）的 *Meta* 类中使用 *db_table* 属性来指定表名。Django 会自动为每个模型（model）创建一个主键，但是你也可以通过在模型（model）中的某个字段上设置 *primarry_key=True* 来指定主键。
让我们根据新模型（model）来同步数据库。运行以下的命令来应用已存在的数据迁移（migrations）：

```
python manage.py migrate
```

你应该会看到以下行跟在输出的末尾：

```
Applying blog.0001_initial... OK
```

我们刚刚为 *INSTALLED_APPS* 中所有的应用进行了数据库迁移（migrations），包括我们的 *blog* 应用。在进行了数据库迁移（migrations）之后，数据库会反映我们模型的当前状态。
如果为了添加，删除，或是改变了存在的模型（models）中字段，或者你添加了新的模型（models）而编辑了 *models.py* 文件，你都需要通过使用 *makemigrations* 命令做一次新的数据库迁移（migration）。数据库迁移（migration）允许 Django 来保持对模型（model）改变的跟踪。之后你必须通过 *migrate* 命令来保持数据库与我们的模型（models）同步。

为你的模型（models）创建一个管理站点

现在我们已经定义好了 *Post* 模型（model），我们将要创建一个简单的管理站点来管理 blog 帖子。Django 内置了一个管理接口，该接口对编辑内容非常的有用。这个 Django 管理站点会根据你的模型（model）元数据进行动态构建并且提供一个可读的接口来编辑内容。你可以对这个站点进行自由的定制，配置你的模型（models）在其中如何进行显示。
请记住，*django.contrib.admin* 已经被包含在我们项目的 *INSTALLED_APPS* 设置中，我们不需要再额外添加。

创建一个超级用户

首先，我们需要创建一名用户来管理这个管理站点。运行以下的命令：

```
python manage.py createsuperuser
```

你会看下以下输出。输入你想要的用户名，邮箱和密码：

```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: ********
Password (again): ********
```

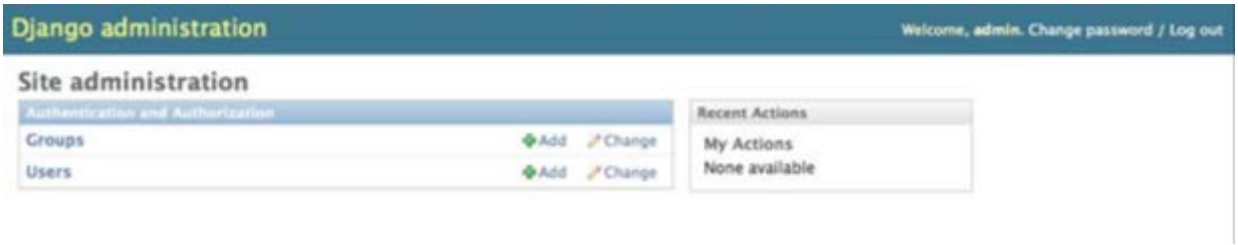
```
Superuser created successfully.
```

Django 管理站点

现在，通过 `python manage.py runserver` 命令来启动开发服务器，之后在浏览器中打开 http://127.0.0.1:8000/admin/ 。你会看到管理站点的登录页面，如下所示：

django-1-2

使用你在上一步中创建的超级用户信息进行登录。你将会看到管理站点的首页，如下所示：



django-1-3

Group 和 *User* 模型（models） 位于 *django.contrib.auth*，是 Django 权限管理框架的一部分。如果你点击 *Users*，你将会看到你之前创建的用户信息。你的 blog 应用的 *Post* 模型（model）和 *User*（model）关联在了一起。记住，它们是通过 *author* 字段进行关联的。

在管理站点中添加你的模型（models）

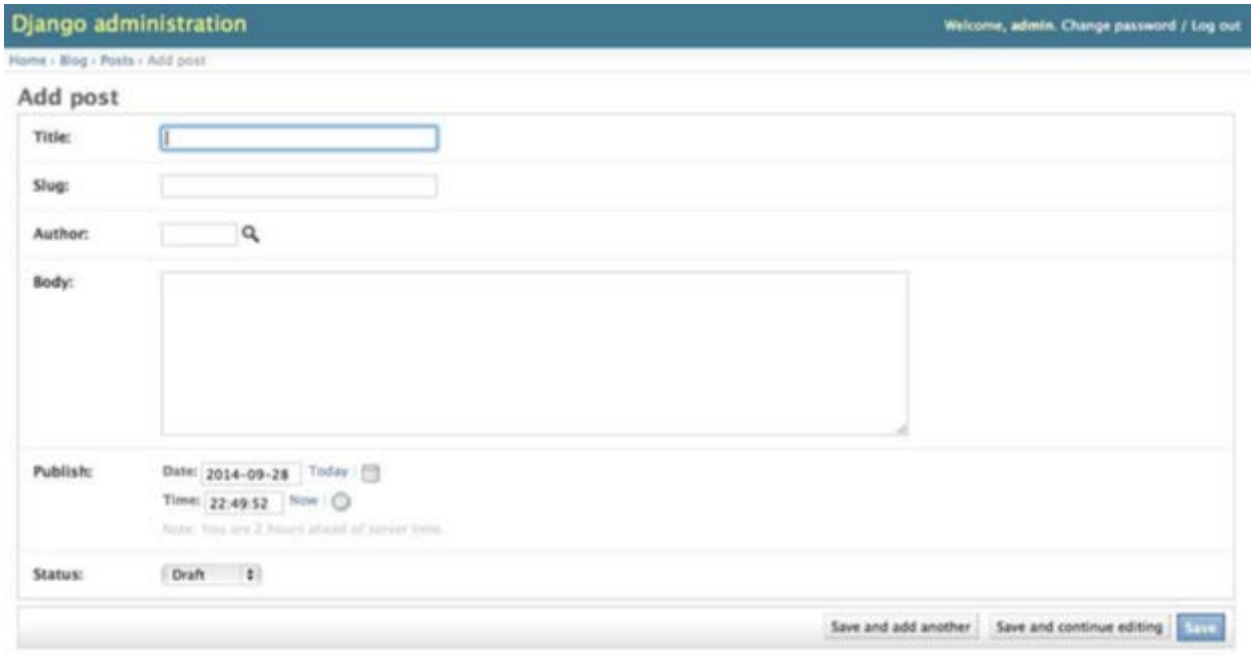
让我们在管理站点中添加你的 blog 模型（models）。编辑 blog 应用下的 *admin.py* 文件，如下所示：

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

现在，在浏览器中刷新管理站点。你会看到你的 *Post* 模型（model）已经在页面中展示，如下所示：

django-1-4

这很简单，对吧？当你在 Django 的管理页面注册了一个模型（model），Django 会通过对你的模型（models）进行内省然后提供给你一个非常友好有用的接口，这个接口允许你非常方便的排列，编辑，创建，以及删除对象。
点击 *Posts* 右侧的 *Add* 链接来添加一篇新帖子。你将会看到 Django 根据你的模型（model）动态生成了一个表单，如下所示：



django-1-5

Django 给不同类型的字段使用了不同的表单控件。即使是复杂的字段例如 *DateTimeField* 也被展示成一个简单的接口类似一个 JavaScript 日期选择器。
填写好表单然后点击 *Save* 按钮。你会被重定向到帖子列页面并且得到一条帖子成功创建的提示，如下所示：

django-1-6

定制 models 的展示形式

现在我们来看下如何定制管理站点。编辑 blog 应用下的 *admin.py* 文件，使之如下所示：

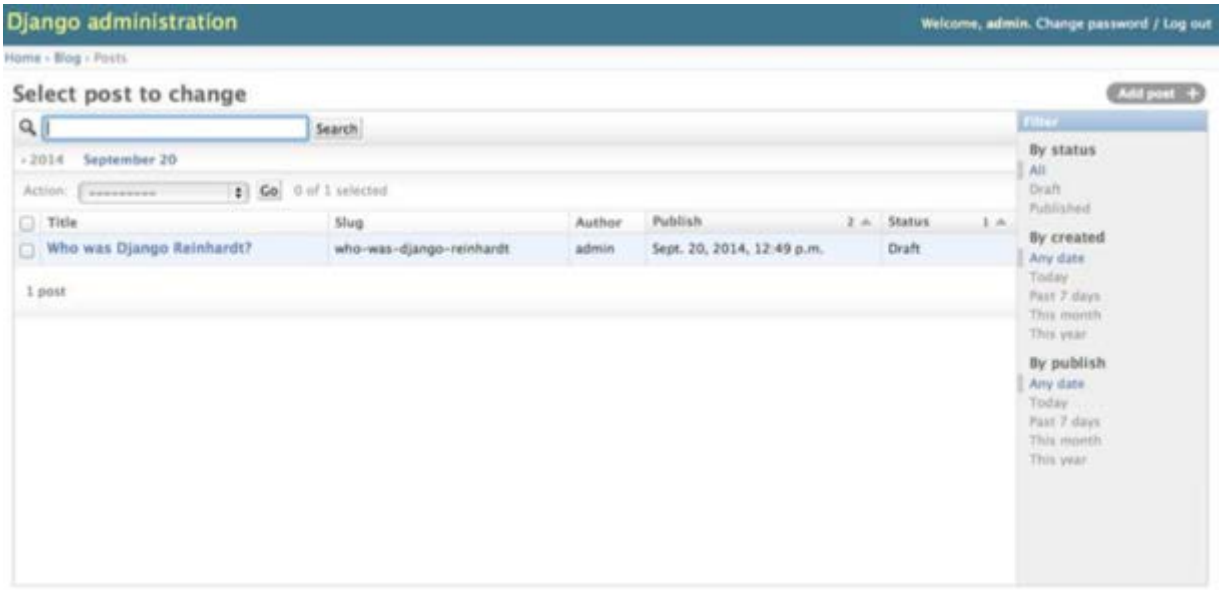
```
from django.contrib import admin
from .models import Post

class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'author', 'publish',
                    'status')
admin.site.register(Post, PostAdmin)
```

我们使用继承了 *ModelAdmin* 的定制类来告诉 Django 管理站点中需要注册我们自己的模型（model）。在这个类中，我们可以包含一些关于如何在管理站点中展示模型（model）的信息以及如何与该模型

（model）进行交互。*list_display* 属性允许你在设置一些你想要在管理对象列表页面显示的模型（model）字段。
让我们通过更多的选项来定制管理模型（model），如使用以下代码：

```
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'author', 'publish',
                    'status')
    list_filter = ('status', 'created', 'publish', 'author')
    search_fields = ('title', 'body')
    prepopulated_fields = {'slug': ('title',)}
    raw_id_fields = ('author',)
    date_hierarchy = 'publish'
    ordering = ['status', 'publish']
```

回到浏览器刷新管理站点页面，现在应该如下所示：



django-1-7

你可以看到帖子列页面中展示的字段都是你在 *list-dispaly* 属性中指定的。这个列页面现在包含了一个右侧边栏允许你根据 *list_filter* 属性中指定的字段来过滤返回结果。一个搜索框也应用在页面中。这是因为我们还通过使用 *search_fields* 属性定义了一个搜索字段列。在搜索框的下方，有个可以通过时间层快速导航的栏，该栏通过定义 *date_hierarchy* 属性出现。你还能看到这些帖子默认的通过 *Status* 和 *Publish* 列进行排序。这是因为你通过使用 *ordering* 属性指定了默认排序。
现在，点击 *Add post* 链接。你还会在这儿看到一些改变。当你输入完成新帖子的标题，*slug* 字段将会自动填充。我们通过使用 *prepoupulated_fields* 属性告诉 Django 通过输入的标题来填充 *slug* 字段。同时，现在的 *author* 字段展示显示为了一个搜索控件，这样当你的用户量达到成千上万级别的时候比再使用下拉框进行选择更加的人性化，如下图所示：

django-1-8

通过短短的几行代码，我们就在管理站点中自定义了我们的模型（model）的展示形式。还有更多的方式可以用来定制 Django 的管理站点。在这本书的后面，我们还会进一步讲述。

使用查询集（QuerySet）和管理器（managers）

现在，你已经有了一个完整功能的管理站点来管理你的 blog 内容，是时候学习如何从数据库中检索信息并且与这些信息进行交互了。Django 自带了一个强大的数据库抽象 API 可以让你轻松的创建，检索，更新以及删除对象。Django 的 *Object-relational Mapper(ORM)*可以兼容 MySQL,PostgreSQL,SQLite

以及 Oracle。请记住你可以在你项目下的 *setting.py* 中编辑 *DATABASES* 设置来指定数据库。Django 可以同时与多个数据库进行工作，这样你可以编写数据库路由通过任何你喜欢的方式来操作数据。
一旦你创建好了你的数据模型（models），Django 会提供你一个 API 来与它们进行交互。你可以找到数据模型（model）的官方参考文档通过访问 https://docs.djangoproject.com/en/1.8/ref/models/ 。

创建对象

打开终端运行以下命令来打开 Python shell：

```
python manage.py shell
```

然后依次输入以下内容：

```
>>> from django.contrib.auth.models import User
>>> from blog.models import Post
>>> user = User.objects.get(username='admin')
>>> post = Post.objects.create(title='One more post',
                       slug='one-more-post',
                       body='Post body.',
                       author=user)
>>> post.save()
```

让我们来研究下这些代码做了什么。首先，我们取回了一个 username 是 *admin* 的用户对象：

```
user = User.objects.get(username='admin')
```

`get()`方法允许你从数据库取回一个单独的对象。注意这个方法只希望在查询中有唯一的一个匹配。如果在数据库中没有返回结果，这个方法会抛出一个 *DoesNotExist* 异常，如果数据库返回多个匹配结果，将会抛出一个 *MultipleObjectsReturned* 异常。当查询执行的时候，所有的异常都是模型（model）类的属性。
接着，我们来创建一个拥有定制标题标题，slug 和内容的 *Post* 实例，然后我们设置之前取回的 user 胃这篇帖子的作者如下所示：

```
post = Post(title='Another post', slug='another-post', body='Postbody.', author=user)
```

这个对象只是存在内存中不会执行到数据库中
最后，我们通过使用 *save()*方法来保存该对象到数据库中：

```
post.save()
```

这步操作将会执行一段 SQL 的插入语句。我们已经知道如何在内存中创建一个对象并且之后才在数据库中进行插入，但是我们也可以通过使用 *create()*方法直接在数据库中创建对象，如下所示：

```
Post.objects.create(title='One more post', slug='one-more-post',body='Post body.', author=user)
```

更新对象

现在，改变这篇帖子的标题并且再次保存对象：

```
>>> post.title = 'New title'
>>> post.save()
```

这一次，*save()*方法执行了一条更新语句。
你对对象的改变一直存在内存中直到你执行到 *save()*方法。

取回对象

Django 的 *Object-relational mapping(ORM)*是基于查询集（QuerySet）。查询集（QuerySet）是从你的数据库中根据一些过滤条件范围取回的结果对象进行的采集。你已经知道如何通过 *get()*方法从数据库中取回单独的对象。如你所见：我们通过`Post.objects.get()`来使用这个方法。每一个 Django 模型（model）至少有一个管理器（manager），默认管理器（manager）叫做 *objects*。你通过使用你的模型（models）的管理器（manager）就能获得一个查询集（QuerySet）对象。获取一张表中的所有对象，你只需要在默认的 *objects* 管理器（manager）上使用 *all()*方法即可，如下所示：

```
>>> all_posts = Post.objects.all()
```

这就是我们如何创建一个用于返回数据库中所有对象的查询集（QuerySet）。注意这个查询集（QuerySet）并还没有执行。Django 的查询集（QuerySets）是惰性（lazy）的，它们只会被动的去执行。这样的行为可以保证查询集（QuerySet）非常有效率。如果我们没有把查询集（QuerySet）设置给一个变量，而是直接在 Python shell 中编写，因为我们迫使它输出结果，这样查询集（QuerySet）的 SQL 语句将立马执行：

```
>>> Post.objects.all()
```

使用 filter()方法

为了过滤查询集（QuerySet），你可以在管理器（manager）上使用`filter()`方法。例如，我们可以返回所有在 2015 年发布的帖子，如下所示：

```
Post.objects.filter(publish__year=2015)
```

你也可以使用多个字段来进行过滤。例如，我们可以返回 2015 年发布的所有作者用户名为 *admin* 的帖子，如下所示：

```
Post.objects.filter(publish__year=2015, author__username='admin')
```

上面的写法和下面的写法产生的结果是一致的：

```
Post.objects.filter(publish__year=2015).filter(author__username='admin')
```

我们构建了字段的查找方法，通过使用两个下划线`(publish__year)`来查询，除此以外我们也可以通过使用两个下划线`(author__username)`访问关联的模型（model）字段。

使用 exclude()

你可以在管理器（manager）上使用 *exclude()*方法来排除某些返回结果。例如：我们可以返回所有 2015 年发布的帖子但是这些帖子的题目开头不能是 *Why*:

```
Post.objects.filter(publish__year=2015).exclude(title__startswith='Why')
```

使用 order_by()

通过在管理器（manager）上使用 *order_by()*方法来对不同的字段进行排序，你可以对结果进行排序。例如：你可以取回所有对象并通过它们的标题进行排序：

```
Post.objects.order_by('title')
```

默认是升序。你可以通过负号来指定使用降序，如下所示：

```
Post.objects.order_by('-title')
```

删除对象

如果你想删除一个对象，你可以对对象实例进行下面的操作：

```
post = Post.objects.get(id=1)
post.delete()
```

请注意，删除对象也将删除任何的依赖关系

查询集（QuerySet）什么时候会执行

只要你喜欢，你可以连接许多的过滤给查询集（QuerySet）而且不会立马在数据库中执行直到这个查询集（QuerySet）被执行。查询集（QuerySet）只有在以下情况中才会执行：

```
* 在你第一次迭代它们的时候
* 当你对它们的实例进行切片：例如`Post.objects.all()[:3]`
* 当你对它们进行了打包或缓存
* 当你对它们调用了`repr()`或`len()`方法
* 当你明确的对它们调用了`list()`方法
* 当你在一个声明中测试它，例如*bool()*, or, and, or if
```

创建 model manager

我们之前提到过, *objects* 是每一个模型（models）的默认管理器（manager），它会返回数据库中所有的对象。但是我们也可以为我们的模型（models）定义一些定制的管理器（manager）。我们准备创建一个定制的管理器（manager）来返回所有状态为已发布的帖子。
有两种方式可以为你的模型(models)添加管理器(managers)：你可以添加额外的管理器(manager)方法或者继承管理器（manager）的查询集（QuerySets）进行修改。第一种方法类似 `Post.objects.my_manager()`,第二种方法类似 `Post.my_manager.all()`。我们的管理器（manager）将会允许我们返回所有帖子通过使用 `Post.published`。
编辑你的 blog 应用下的 *models.py* 文件添加如下代码来创建一个管理器（manager）：

```
class PublishedManager(models.Manager):
    def get_queryset(self):
        return super(PublishedManager,
                     self).get_queryset().filter(status='published')

class Post(models.Model):
    # ...
    objects = models.Manager() # The default manager.
    published = PublishedManager() # Our custom manager.
```

`get_queryset()`是返回执行过的查询集（QuerySet）的方法。我们通过使用它来包含我们定制的过滤到完整的查询集(QuerySet)中。我们定义我们定制的管理器(manager)然后添加它到 *Post* 模型(model)中。我们现在可以来执行它。例如，我们可以返回所有标题开头为 *Who* 的并且是已经发布的帖子:

```
Post.published.filter(title__startswith='Who')
```

构建列和详情视图（views）

现在你已经学会了一些如何使用 ORM 的基本知识，你已经准备好为 blog 应用创建视图（views）了。一个 Django 视图（view）就是一个 Python 方法，它可以接收一个 web 请求然后返回一个 web 响应。在视图（views）中通过所有的逻辑处理返回期望的响应。
首先我们会创建我们的应用视图（views），然后我们将会为每个视图（view）定义一个 URL 模式，我们将会创建 HTML 模板（templates）来渲染这些视图（views）生成的数据。每一个视图（view）都会渲染模板（template）传递变量给它然后会返回一个经过渲染输出的 HTTP 响应。

创建列和详情 views

让我们开始创建一个视图（view）来展示帖子列。编辑你的 blog 应用下中 *views.py* 文件，如下所示：

```
from django.shortcuts import render, get_object_or_404
from .models import Post
def post_list(request):
    posts = Post.published.all()
    return render(request,
                  'blog/post/list.html',
                  {'posts': posts})
```

你刚创建了你的第一个 Django 视图（view）。*post_list* 视图（view）将 *request* 对象作为唯一的参数。记住所有的的视图（views）都有需要这个参数。在这个视图（view）中，我们获取到了所有状态为已发布的帖子通过使用我们之前创建的 *published* 管理器（manager）。
最后，我们使用 Django 提供的快捷方法 *render()*通过给予的模板（template）来渲染帖子列。这个函数将 *request* 对象作为参数，模板（template）路径以及变量来渲染的给予的模板（template）。它返回一个渲染文本（一般是 HTML 代码）*HttpResponse* 对象。*render()*方法考虑到了请求内容，这样任何模板（template）内容处理器设置的变量都可以带入给予的模板（template）中。你会在*第三章，扩展你的 blog 应用*学习到如何使用它们。
让我们创建第二个视图（view）来展示一篇单独的帖子。添加如下代码到 *views.py* 文件中：

```
def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post, slug=post,
                                   status='published',
                                   publish__year=year,
                                   publish__month=month,
                                   publish__day=day)
    return render(request,
                  'blog/post/detail.html',
                  {'post': post})
```

这是一个帖子详情视图（view）。这个视图（view）使用 *year，month，day* 以及 *post* 作为参数通过给予 slug 和日期来获取到一篇已经发布的帖子。请注意，当我们创建 *Post* 模型（model）的时候，我们给 slgu 字段添加了 *unique_for_date* 参数。这样我们可以确保在给予的日期中只有一个帖子会带有一个 slug，因此，我们能通过日期和 slug 取回单独的帖子。在这个详情视图（view）中，我们通过使用 *get_object_or_404()*快捷方法来检索期望的 *Post*。这个函数能取回匹配给予的参数的对象，或者当没有匹配的对象时返回一个 HTTP 404（Not found）异常。最后，我们使用 *render()*快捷方法来使用一个模板（template）去渲染取回的帖子。

为你的视图（views）添加 URL 模式

一个 URL 模式是由一个 Python 正则表达，一个视图（view），一个全项目范围内的命名组成。Django 在运行中会遍历所有 URL 模式直到第一个匹配的请求 URL 才停止。之后，Django 导入匹配的 URL 模式中的视图（view）并执行它，使用关键字或指定参数来执行一个 *HttpRequest* 类的实例。

如果你之前没有接触过正则表达式，你需要去稍微了解下，通过访问 https://docs.python.org/3/howto/regex.html 。
在 blog 应用目录下创建一个 *urls.py* 文件，输入以下代码：

```
from django.conf.urls import url
from . import views
urlpatterns = [
    # post views
    url(r'^$', views.post_list, name='post_list'),
    url(r'^(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/(?P<post>[-\w]+)/$',
        views.post_detail,
        name='post_detail'),
]
```

第一条 URL 模式没有带入任何参数，它映射到 *post_list* 视图（view）。第二条 URL 模式带上了以下 4 个参数映射到 *post_detail* 视图（view）中。让我们看下这个 URL 模式中的正则表达式：

- year：需要四位数
- month：需要两位数。不及两位数，开头带上 0，比如 01，02
- day：需要两位数。不及两位数开头带上 0
- post：可以由单词和连字符组成

为每一个应用创建单独的 *urls.py* 文件是最好的方法，可以保证你的应用能给别的项目再度使用。
现在你需要将你 blog 中的 URL 模式包含到项目的主 URL 模式中。编辑你的项目中的 *mysite* 文件夹中的 *urls.py* 文件，如下所示：

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

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^blog/', include('blog.urls',
        namespace='blog',
        app_name='blog')),
]
```

通过这样的方式，你告诉 Django 在 *blog/*路径下包含了 blog 应用中的 *urls.py* 定义的 URL 模式。你可以给它们一个命名空间叫做 *blog*，这样你可以方便的引用这个 URLs 组。

模型（models）的标准 URLs

你可以使用之前定义的 *post_detail* URL 给 *Post* 对象构建标准 URL。Django 的惯例是给模型（model）添加 *get_absolute_url()*方法用来返回一个对象的标准 URL。在这个方法中，我们使用 *reverse()*方法允许你通过它们的名字和可选的参数来构建 URLS。编辑你的 *models.py* 文件添加如下代码：

```
from django.core.urlresolvers import reverse
Class Post(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('blog:post_detail',
                       args=[self.publish.year,
                             self.publish.strftime('%m'),
                             self.publish.strftime('%d'),
                             self.slug])
```

请注意，我们通过使用 *strftime()*方法来保证个位数的月份和日期需要带上 0 来构建 URL（**译者注：也就是 01,02,03**）。我们将会在我们的模板（templates）中使用 *get_absolute_url()*方法。

为你的视图（views）创建模板（templates）

我们为我们的应用创建了视图（views）和 URL 模式。现在该添加模板（templates）来展示界面友好的帖子了。

在你的 blog 应用目录下创建以下目录结构和文件：

```
templates/
    blog/
        base.html
        post/
            list.html
            detail.html
```

以上就是我们的模板（templates）的文件目录结构。*base.html* 文件将会包含站点主要的 HTML 结构以及分割内容区域和一个导航栏。*list.html* 和 *detail.html* 文件会继承 *base.html* 文件来渲染各自的 blog 帖子列和详情视图（view）。

Django 有一个强大的模板（templates）语言允许你指定数据的如何进行展示。它基于模板标签（templates tags），例如 `{% tag %}`, `{{ variable }}`以及模板过滤器（templates filters），可以对变量进行过滤，例如 `{{ variable|filter }}`。你可以通过访问 https://docs.djangoproject.com/en/1.8/ ref/templates/builtins/ 找到所有的内置模板标签（templates tags）和过滤器（filters）。

让我们来编辑 *base.html* 文件并添加如下代码：

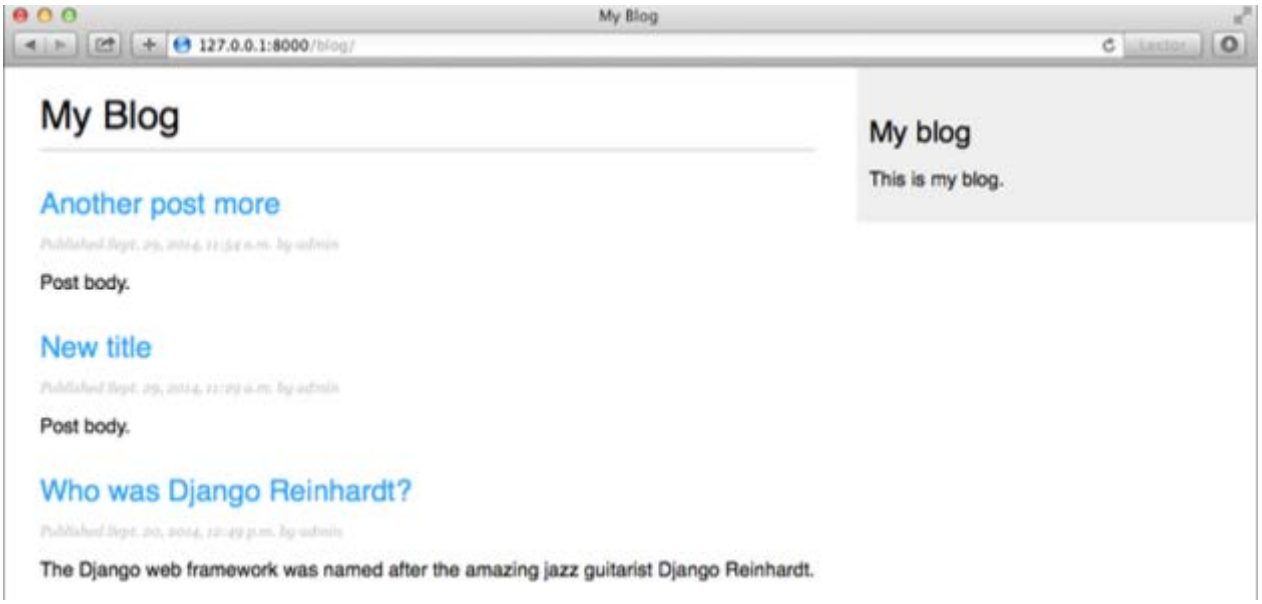
```
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
  <div id="sidebar">
    <h2>My blog</h2>
      <p>This is my blog.</p>
  </div>
</body>
</html>
```

`{% load staticfiles %}`告诉 Django 去加载 *django.contrib.staticfiles* 应用提供的 *staticfiles* 模板标签（temaplate tags）。通过加载它，你可以在这个模板（template）中使用`{% static %}`模板过滤器（template filter）。通过使用这个模板过滤器（template filter），你可以包含一些静态文件比如说 *blog.css* 文件，你可以在本书的范例代码例子中找到该文件，在 blog 应用的 *static/*目录中（**译者注：给大家个地址去拷贝 https://github.com/levelksk/django-by-example-book** ）拷贝这个目录到你的项目下的相同路径来使用这些静态文件。

你可以看到有两个`{% block %}`标签（tags）。这些是用来告诉 Django 我们想在这个区域中定义一个区块（block）。继承这个模板（template）的其他模板（templates）可以使用自定义的内容来填充区块（block）。我们定义了一个区块（block）叫做 *title*，另一个区块（block）叫做 *content*。
让我们编辑 *post/list.html* 文件使它如下所示：

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

{% block title %}My Blog{% endblock %}

{% block content %}
  <h1>My Blog</h1>
  {% for post in posts %}
    <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
    </h2>
    <p class="date">
      Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
{% endblock %}
```

通过`{% extends %}`模板标签（template tag），我们告诉 Django 需要继承 *blog/base.html* 模板（template）。然后我们在 *title* 和 *content* 区块（blocks）中填充内容。我们通过循环迭代帖子来展示它们的标题，日期，作者和内容，在标题中还集成了帖子的标准 URL 链接。在帖子的内容中，我们应用了两个模板过滤器（template filters）： *truncatewords* 用来缩短内容限制在一定的字数内，*linebreaks* 用来转换内容中的换行符为 HTML 的换行符。只要你喜欢你可以连接许多模板标签（tempalte filters），每一个都会应用到上个输出生成的结果上。
打开终端执行命令 `python manage.py runserver` 来启动开发服务器。在浏览器中打开 http://127.0.0.1:8000/blog/ 你会看到运行结果。注意，你需要添加一些发布状态的帖子才能在这儿看到它们。你会看到如下图所示：



django-1-9

这之后，让我们来编辑 *post/detail.html* 文件使它如下所示：

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

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
  <h1>{{ post.title }}</h1>
  <p class="date">
    Published {{ post.publish }} by {{ post.author }}
  </p>
  {{ post.body|linebreaks }}
{% endblock %}
```

现在，你可以在浏览器中点击其中一篇帖子的标题来看帖子的详细视图（view）。你会看到类似以下页面：

django-1-10

添加页码

当你开始给你的 blog 添加内容，你很快会意识到你需要将帖子分页显示。Django 有一个内置的 *Paginator* 类允许你方便的管理分页。
编辑 blog 应用下的 *views.py* 文件导入 Django 的页码类修改 *post_list* 如下所示：

```
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

def post_list(request):
    object_list = Post.published.all()
    paginator = Paginator(object_list, 3) # 3 posts in each page
    page = request.GET.get('page')
    try:
        posts = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer deliver the first page
        posts = paginator.page(1)
    except EmptyPage:
        # If page is out of range deliver last page of results
        posts = paginator.page(paginator.num_pages)
    return render(request,
                  'blog/post/list.html',
                  {'page': page,
                   'posts': posts})
```

Paginator 是如何工作的：

- 我们使用希望在每页中显示的对象的数量来实例化 *Paginator* 类。
- 我们获取到 *page* GET 参数来指明页数

- 我们通过调用 *Paginator* 的 *page()*方法在期望的页面中获得了对象。
- 如果 *page* 参数不是一个整数，我们就返回第一页的结果。如果这个参数数字超出了最大的页数，我们就展示最后一页的结果。
- 我们传递页数并且获取对象给这个模板（template）。

现在，我们必须创建一个模板（template）来展示分页处理，它可以被任意的模板（template）包含来使用分页。在 blog 应用的 *templates* 文件夹下创建一个新文件命名为 *pagination.html*。在该文件中添加如下 HTML 代码：

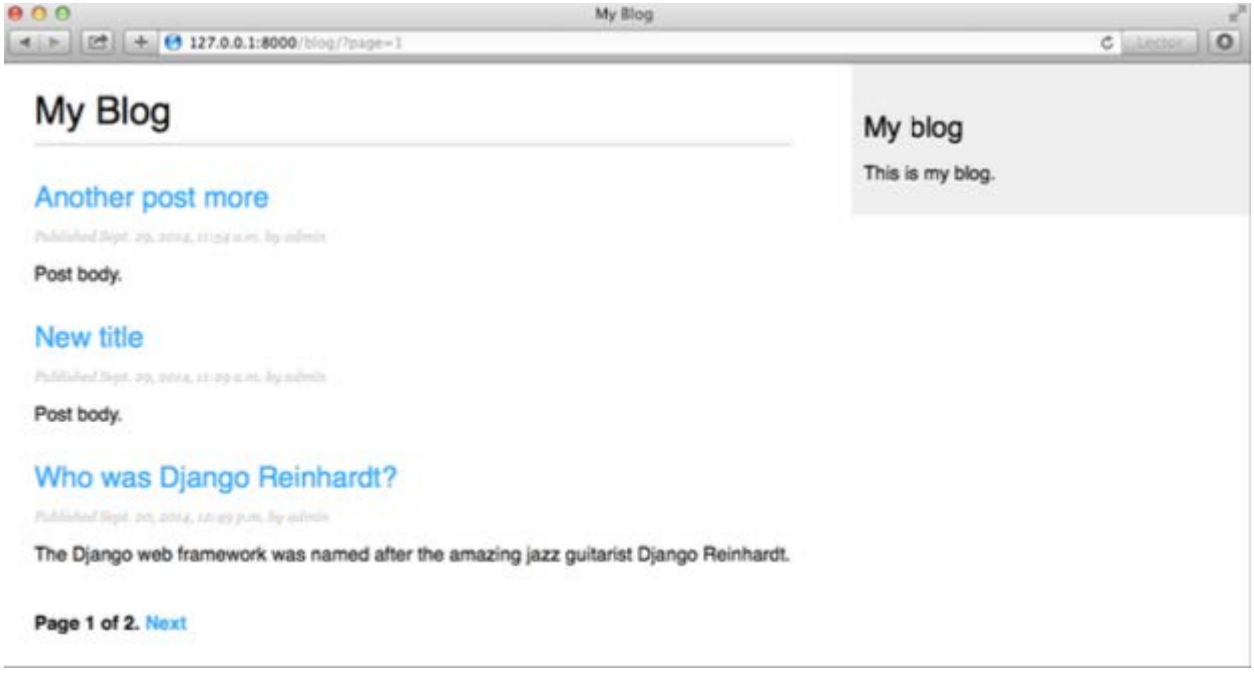
```
<div class="pagination">
  <span class="step-links">
    {% if page.has_previous %}
      <a href="?page={{ page.previous_page_number }}">Previous</a>
    {% endif %}
    <span class="current">
      Page {{ page.number }} of {{ page.paginator.num_pages }}.
    </span>
      {% if page.has_next %}
        <a href="?page={{ page.next_page_number }}">Next</a>
      {% endif %}
  </span>
</div>
```

为了渲染上一页与下一页的链接并且展示当前页面和所有页面的结果，这个分页模板（template）期望一个 *Page* 对象。让我们回到 *blog/post/list.html* 模板（tempalte）中将 *pagination.html* 模板（template）包含在`{% content %}`区块（block）中，如下所示：

```
{% block content %}
  ...
  {% include "pagination.html" with page=posts %}
{% endblock %}
```

我们传递给模板（template）的 *Page* 对象叫做 *posts*，我们将分页模板（tempalte）包含在帖子列模板（template）中指定参数来对它进行正确的渲染。这种方法你可以反复使用，用你的分页模板（template）对不同的模型（models）视图（views）进行分页处理。

现在，在你的浏览器中打开 http://127.0.0.1:8000/blog/。 你会看到帖子列的底部已经有分页处理：



使用基于类的视图（views）

因为一个视图(view)的调用就是得到一个 web 请求并且返回一个 web 响应，你可以将你的视图(views)定义成类方法。Django 为此定义了基础的视图（view）类。它们都从 *View* 类继承而来，*View* 类可以操控 HTTP 方法调度以及其他的功能。这是一个可替代的方法来创建你的视图（views）。
我们准备通过使用 Django 提供的通用 *ListView* 使我们的 *post_list* 视图（view）转变为一个基于类的视图。这个基础视图（view）允许你对任意的对象进行排列。
编辑你的 blog 应用下的 *views.py* 文件，如下所示：

```
from django.views.generic import ListView
class PostListView(ListView):
    queryset = Post.published.all()
    context_object_name = 'posts'
    paginate_by = 3
    template_name = 'blog/post/list.html'
```

这个基于类的的视图（view）类似与之前的 *post_list* 视图（view）。在这儿，我们告诉 *ListView* 做了以下操作：

- 使用一个特定的查询集（QuerySet）代替取回所有的对象。代替定义一个 *queryset* 属性，我们可以指定 `model = Post` 然后 Django 将会构建 *Post.objects.all()* 查询集（QuerySet）给我们。
- 使用环境变量 *posts* 给查询结果。如果我们不指定任意的 *context_object_name* 默认的变量将会是 *object_list*。
- 对结果进行分页处理每页只显示 3 个对象。
- 使用定制的模板（template）来渲染页面。如果我们不设置默认的模板（template），*ListView* 将会使用 `blog/post_list.html`。

现在，打开你的 blog 应用下的 *urls.py* 文件，注释到之前的 *post_list*URL 模式，在之后添加一个新的 URL 模式来使用 *PostlistView* 类，如下所示：

```
urlpatterns = [
    # post views
    # url(r'^$', views.post_list, name='post_list'),
    url(r'^$', views.PostListView.as_view(),name='post_list'),
    url(r'^(?P<year>\d{4})/(?P<month>\d{2})/(?P<day>\d{2})/'\
        r'(?P<post>[-\w]+)/$',
        views.post_detail,
        name='post_detail'),
]
```

为了保持分页处理能工作，我们必须将正确的页面对象传递给模板（tempalte）。Django 的 *ListView* 通过叫做 *page_obj* 的变量来传递被选择的页面，所以你必须编辑你的 *post_list_html* 模板(template)去包含使用了正确的变量的分页处理，如下所示：

```
{% include "pagination.html" with page=page_obj %}
```

在你的浏览器中打开 http://127.0.0.1:8000/blog/ 然后检查每一样功能是否都和之前的 *post_list* 视图（view）一样工作。这是一个简单的，通过使用 Django 提供的通用类的基于类视图（view）的例子。你将在*第十章，创建一个在线学习平台*以及相关的章节中学到更多的基于类的视图（views）。

总结

在本章中，你通过创建一个基础的 blog 应用学习了 Django web 框架的基础。你为你的项目设计了数据模型（models）并且进行了数据库迁移。你为你的 blog 创建了视图（views），模板（templates）以及 URLs，还包括对象分页。
在下一章中，你会学习到如何增强你的 blog 应用，例如评论系统，标签（tag）功能，并且允许你的用户通过邮件来分享帖子。

译者总结

终于将第一章勉强翻译完成了，很多翻译的句子我自己都读不懂 - -|||
大家看到有错误有歧义的地方请帮忙指出，之后还会随时进行修改保证基本能读懂。
按照第一章的翻译速度，全书都翻译下来估计要 2，3 个月，这是非常非常乐观的估计，每天只有中午休息和下班后大概有两三小时的翻译时间。

第二章 用高级特性来增强你的 blog

在上一章中，你创建了一个基础的博客应用。现在你将利用一些高级的特性例如通过 email 来分享帖子，添加评论，给帖子打上 tag，检索出相似的帖子等将它改造成为一个功能更加齐全的博客。在本章中，你将会学习以下几点：

- 通过 Django 发送 email
- 在视图（views）中创建并操作表单
- 通过模型（models）创建表单
- 集成第三方应用
- 构建复杂的查询集（QuerySets)

通过 email 分享帖子

首先，我们会允许用户通过发送邮件来分享他们的帖子。让我们花费一小会时间来想下，根据在上一章中学到的知识，你该如何使用 views，urls 和 templates 来创建这个功能。现在，核对一下你需要哪些才能允许你的用户通过邮件来发送帖子。你需要做到以下几点：

- 给用户创建一个表单来填写他们的姓名，email，收件人以及评论，评论不是必选项。
- 在 *views.py* 文件中创建一个视图（view）来操作发布的数据和发送 email
- 在 blog 应用的 *urls.py* 中为新的视图（view）添加一个 URL 模式
- 创建一个模板（template）来展示这个表单

使用 Django 创建表单

让我们开始创建一个表单来分享帖子。Django 有一个内置的表单框架允许你通过简单的方式来创建表单。这个表单框架允许你定义你的表单字段，指定这些字段必须展示的方式，以及指定这些字段如何验证输入的数据。Django 表单框架还提供了一种灵活的方式来渲染表单以及操作数据。
Django 提供了两个可以创建表单的基本类：

- Form: 允许你创建一个标准表单
- ModelForm: 允许你创建一个可用于创建或者更新 model 实例的表单

首先，在你 blog 应用的目录下创建一个 *forms.py* 文件，输入以下代码：

```
from django import forms


class EmailPostForm(forms.Form):
    name = forms.CharField(max_length=25)
    email = forms.EmailField()
    to = forms.EmailField()
    comments = forms.CharField(required=False,
                               widget=forms.Textarea)
```

这是你的第一个 Django 表单。看下代码：我们已经创建了一个继承了基础 *Form* 类的表单。我们使用不同的字段类型以使 Django 有依据的来验证字段。
表单可以存在你的 Django 项目的任何地方，但按照惯例将它们放在每一个应用下面的 *forms.py* 文件中
name 字段是一个 *CharField*。这种类型的字段被渲染成`<input type="text">`HTML 元素。每种字段类型都有默认的控件来确定它在 HTML 中的展示形式。通过改变控件的属性可以重写默认的控件。在 comment 字段中，我们使用`<textarea></textarea>`HTML 元素而不是使用默认的`<input>`元素来显示它。字段验证取决于字段类型。例如，*email* 和 *to* 字段是 *EmailField*,这两个字段都需要一个有效的 email 地址，否则字段验证将会抛出一个 *forms.ValidationError* 异常导致表单验证不通过。在表单验证的时候其他的参数也会被考虑进来：我们将 name 字段定义为一个最大长度为 25 的字符串；通过设置`required=False`让 comments 的字段可选。所有这些也会被考虑到字段验证中去。目前我们在表单中使用的这些字段类型只是 Django 支持的表单字段的一部分。要查看更多可利用的表单字段，你可以访问：https://docs.djangoproject.com/en/1.8/ref/forms/fields/

在视图（views）中操作表单

当表单成功提交后你必须创建一个新的视图（views）来操作表单和发送 email。编辑 blog 应用下的 *views.py* 文件，添加以下代码：

```
from .forms import EmailPostForm

def post_share(request, post_id):
    # retrieve post by id
    post = get_object_or_404(Post, id=post_id, status='published')

    if request.method == 'POST':
        # Form was submitted
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Form fields passed validation
            cd = form.cleaned_data
            # ... send email
    else:
        form = EmailPostform()
    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form: form})
```

该视图（view）完成了以下工作：

- 我们定义了 *post_share* 视图，参数为 *request* 对象和 *post_id*。
- 我们使用 *get_object_or_404* 快捷方法通过 ID 获取对应的帖子，并且确保获取的帖子有一个 *published* 状态。
- 我们使用同一个视图（view）来展示初始表单和处理提交后的数据。我们会区别被提交的表单和不基于这次请求方法的表单。我们将使用 *POST* 来提交表单。如果我们得到一个 *GET* 请求，一个空的表单必须显示，而如果我们得到一个 *POST* 请求，则表单需要提交和处理。因此，我们使用`request.method == 'POST'`来区分这两种场景。

下面是展示和操作表单的过程：

- 1.通过 GET 请求视图（view）被初始加载后，我们创建一个新的表单实例，用来在模板（template）中显示一个空的表单：

```
  form = EmailPostForm()
```

- 2.当用户填写好了表单并通过 *POST* 提交表单。之后，我们会用保存在 `request.POST` 中提交的数据创建一个表单实例。

```
    if request.method == 'POST':
        # Form was submitted

        form = EmailPostForm(request.POST)
```

- 3.在以上步骤之后，我们使用表单的 *is_valid()*方法来验证提交的数据。这个方法会验证表单引进的数据，如果所有的字段都是有效数据，将会返回 *True*。一旦有任何一个字段是无效的数据，*is_valid()* 就会返回 *False*。你可以通过访问 `form.errors` 来查看所有验证错误的列表。
- 4 如果表单数据验证没有通过，我们会再次使用提交的数据在模板（template）中渲染表单。我们会在模板（template）中显示验证错误的提示。
- 5.如果表单数据验证通过，我们通过访问 `form.cleaned_data` 获取验证过的数据。这个属性是一个表单字段和值的字典。

如果你的表单数据没有通过验证，*cleaned_data* 只会包含验证通过的字段
现在，你需要学习如何使用 Django 来发送 email,把所有的事情结合起来。

使用 Django 发送 email

使用 Django 发送 email 非常简单。首先，你需要有一个本地的 SMTP 服务或者通过在你项目的 settings.py 文件中添加以下设置去定义一个外部 SMTP 服务器的配置：

- EMAIL_HOST: SMTP 服务地址。默认本地。
- EMAIL_POSR: SMATP 服务端口，默认 25。
- EMAIL_HOST_USER: SMTP 服务的用户名。
- EMAIL_HOST_PASSWORD: SMTP 服务的密码。
- EMAIL_USE_TLS: 是否使用 TLS 加密连接。
- EMAIL_USE_SSL: 是否使用隐式的 SSL 加密连接。

如果你没有本地 SMTP 服务，你可以使用你的 email 服务供应商提供的 SMTP 服务。下面提供了一个简单的例子展示如何通过使用 Google 账户的 Gmail 服务来发送 email：

```
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'your_account@gmail.com'
EMAIL_HOST_PASSWORD = 'your_password'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

运行命令 python manage.py shell 来打开 Python shell，发送一封 email 如下所示：

```
>>> from django.core.mail import send_mail
>>> send_mail('Django mail', 'This e-mail was sent with Django.','your_account@gmail.com',
['your_account@gmail.com'], fail_silently=False)
```

*send_mail()*方法需要这些参数：邮件主题，内容，发送人以及一个收件人的列表。通过设置可选参数 `fail_silently=False`，我们告诉这个方法如果 email 没有发送成功那么需要抛出一个异常。如果你看到输出是 *1*，证明你的 email 发送成功了。如果你使用之前的配置用 Gmail 来发送邮件，你可能需要去 https://www.google.com/settings/security/lesssecureapps 去开通一下低安全级别应用的权限。(译者注：练习时老老实实用 QQ 邮箱吧）
现在，我们要将以上代码添加到我们的视图（view）中。在 blog 应用下的 *views.py* 文件中编辑 *post_share* 视图（view）如下所示：

```
from django.core.mail import send_mail
```

```
def post_share(request, post_id):
    # Retrieve post by id
    post = get_object_or_404(Post, id=post_id, status='published')
    sent = False
    if request.method == 'POST':
        # Form was submitted
        form = EmailPostForm(request.POST)
        if form.is_valid():
            # Form fields passed validation
            cd = form.cleaned_data
            post_url = request.build_absolute_url(
                                post.get_absolute_url())
            subject = '{} ({}) recommends you reading "{}"'.format(cd['name'], cd['email'], post.title)
            message = 'Read "{}" at {}\n\n{}\'s comments: {}'.format(post.title, post_url, cd['name'],
cd['comments'])
            send_mail(subject, message, 'admin@myblog.com',[cd['to']])
            sent = True
    else:
        form = EmailPostForm()

    return render(request, 'blog/post/share.html', {'post': post,
                                                    'form': form,
                                                    'sent': sent})
```

请注意，我们声明了一个 *sent* 变量并且当帖子被成功发送时赋予它 *True*。当表单成功提交的时候，我们之后将在模板（template）中使用这个变量显示一条成功提示。由于我们需要在 email 中包含帖子的超链接，所以我们通过使用 `post.get_absolute_url()`方法来获取到帖子的绝对路径。我们将这个绝对路径作为 `request.build_absolute_uri()`的输入值来构建一个完整的包含了 HTTP schema 和主机名的 url。我们通过使用验证过的表单数据来构建 email 的主题和消息内容并最终给表单 to 字段中包含的所有 email 地址发送 email。

现在你的视图（view）已经完成了，别忘记为它去添加一个新的 URL 模式。打开你的 blog 应用下的 *urls.py* 文件添加 *post_share* 的 URL 模式如下所示：

```
urlpatterns = [
# ...
url(r'^(?P<post_id>\d+)/share/$', views.post_share,
    name='post_share'),
]
```

在模板（templates）中渲染表单

在通过创建表单，编写视图（view）以及添加 URL 模式后，我们就只剩下为这个视图（view）添加模板（tempalte）了。在 `blog/templates/blog/post/`目录下创建一个新的文件并命名为 *share.html*。在该文件中添加如下代码：

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

{% block title %}Share a post{% endblock %}

{% block content %}
  {% if sent %}
```

```
    <h1>E-mail successfully sent</h1>
    <p>
      "{{ post.title }}" was successfully sent to {{ cd.to }}.
    </p>
  {% else %}
    <h1>Share "{{ post.title }}" by e-mail</h1>
    <form action="." method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <input type="submit" value="Send e-mail">
    </form>
  {% endif %}
{% endblock %}
```

这个模板（tempalte）专门用来显示一个表单或一条成功提示信息。如你所见，我们创建的 HTML 表单元素里面表明了它必须通过 POST 方法提交：

```
<form action="." method="post">
```

接下来我们要包含真实的表单实例。我们告诉 Django 用 `as_p` 方法利用 HTML 的`<p>`元素来渲染它的字段。我们也可以使用 `as_ul` 利用无序列表来渲染表单或者使用 `as_table` 利用 HTML 表格来渲染。如果我们想要逐一渲染每一个字段，我们可以迭代字段。例如下方的例子：

```
{% for field in form %}
  <div>
    {{ field.errors }}
    {{ field.label_tag }} {{ field }}
  </div>
{% endfor %}
```

`{% csrf_token %}`模板（tempalte）标签（tag）引进了可以避开 *Cross-Site request forgery(CSRF)*攻击的自动生成的令牌，这是一个隐藏的字段。这些攻击由恶意的站点或者可以在你的站点中为用户执行恶意行为的程序组成。通过访问 https://en.wikipedia.org/wiki/Cross-site_request_forgery 你可以找到更多的信息 。

上述的标签（tag）生成的隐藏字段就像下面一样：

```
<input type='hidden' name='csrfmiddlewaretoken' value='26JjKo2lcEtYkGoV9z4XmJIEHLXN5LDR' />
```

默认情况下，Django 在所有的 POST 请求中都会检查 CSRF 标记（token）。请记住要在所有使用 POST 方法提交的表单中包含 *csrf_token* 标签。**（译者注：当然你也可以关闭这个检查，注释掉 app_list 中的 csrf 应用即可，我就是这么做的，因为我懒）**

编辑你的 *blog/post/detail.html* 模板（template），在`{{ post.body|linebreaks }}`变量后面添加如下的链接来分享帖子的 URL：

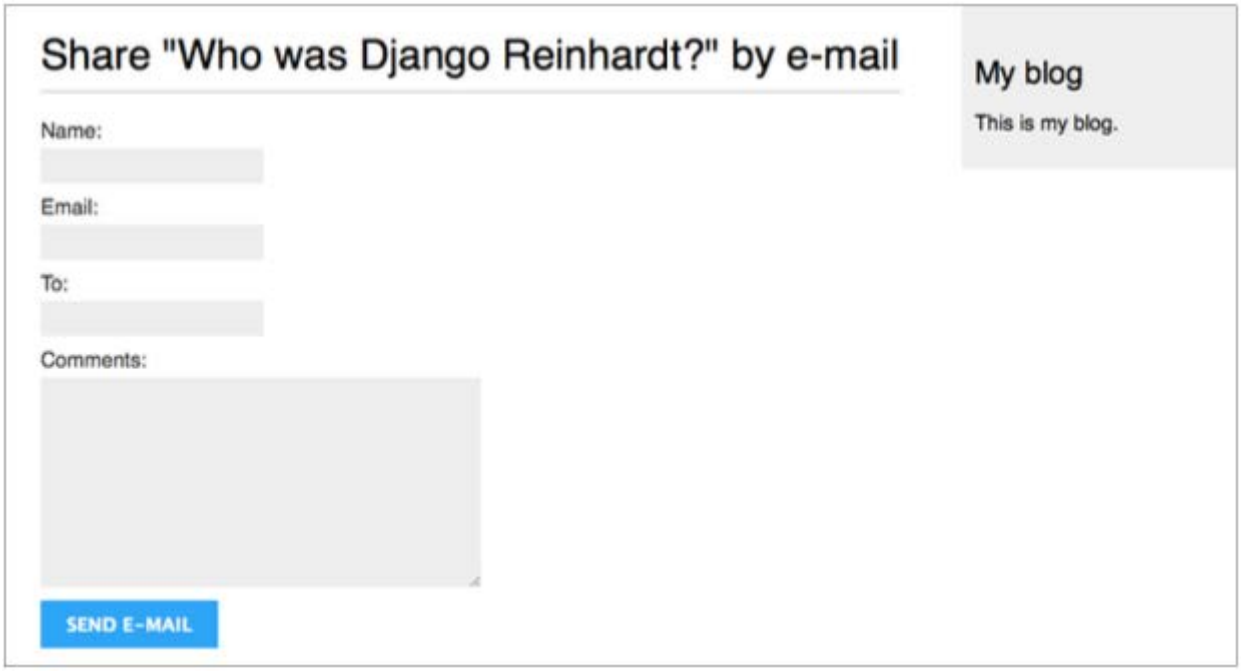
```
<p>
  <a href="{% url "blog:post_share" post.id %}">
    Share this post
  </a>
</p>
```

请记住，我们通过使用 Django 提供的`{% url %}`模板（template）标签（tag）来动态的生成 URL。我们以 *blog* 为命名空间，以 *post_share* 为 URL，同时传递帖子 ID 作为参数来构建绝对的 URL。

现在，通过 `python manage.py runserver` 命令来启动开发服务器，在浏览器中打开 http://127.0.0.1:8000/blog/ 。点击任意一个帖子标题查看详情页面。在帖子内容的下方，你会看到我们刚刚添加的链接，如下所示：

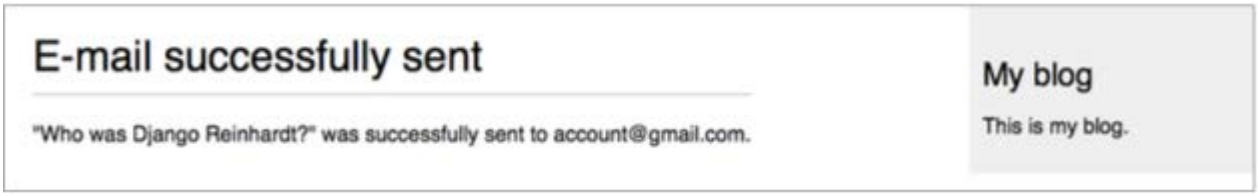
django-2-1

点击 **Share this post**,你会看到包含通过 email 分享帖子的表单的页面。看上去如下所示：



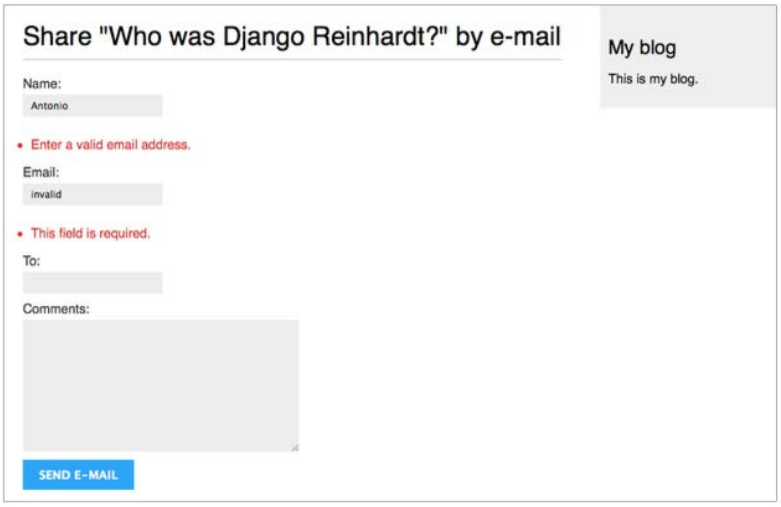
django-2-2

这个表单的 CSS 样式被包含在示例代码中的 static/css/blog.css 文件中。当你点击 **Send e-mail** 按钮，这个表单会提交并验证。如果所有的字段都通过了验证，你会得到一条成功信息如下所示：



django-2-3

如果你输入了无效数据，你会看到表单被再次渲染，并展示出验证错误信息，如下所示：



创建一个评论系统

现在我们准备为 blog 创建一个评论系统，这样用户可以在帖子上进行评论。需要做到以下几点来创建一个评论系统：

- 创建一个模型（model）用来保存评论
- 创建一个表单用来提交评论并且验证输入的数据
- 添加一个视图（view）来处理表单和保存新的评论到数据库中
- 编辑帖子详情模板（template）来展示评论列表以及用来添加新评论的表单

首先，让我们创建一个模型（model）来存储评论。打开你的 blog 应用下的 *models.py* 文件添加如下代码：

```
class Comment(models.Model):
    post = models.ForeignKey(Post, related_name='comments')
    name = models.CharField(max_length=80)
    email = models.EmailField()
    body = models.TextField()
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    active = models.BooleanField(default=True)

    class Meta:
        ordering = ('created',)

    def __str__(self):
        return 'Comment by {} on {}'.format(self.name, self.post)
```

以上就是我们的 *Comment* 模型（model）。它包含了一个外键将一个单独的帖子和评论关联起来。在 *Comment* 模型（model）中定义多对一（many-to-one）的关系是因为每一条评论只能在一个帖子下生成，而每一个帖子又可能包含多个评论。*related_name* 属性允许我们给这个属性命名，这样我们就可以利用这个关系从相关联的对象反向定位到这个对象。定义好这个之后，我们通过使用 `comment.post` 就可以从一条评论来取到对应的帖子，以及通过使用 `post.comments.all()`来取回一个帖子所有的评论。如果你没有定义 *related_name* 属性，Django 会使用这个模型（model）的名称加上_*set*（在这里是：comment_set）来命名从相关联的对象反向定位到这个对象的 manager。

访问 https://docs.djangoproject.com/en/1.8/topics/db/examples/many_to_one/，你可以学习更多关于多对一的关系。

我们用一个 *active* 布尔字段用来手动禁用那些不合适的评论。默认情况下，我们根据 *created* 字段，对评论按时间顺序进行排序。

你刚创建的这个新的 *Comment* 模型（model）并没有同步到数据库中。运行以下命令生成一个新的反映了新模型（model）创建的数据迁移：

```
python manage.py makemigrations blog
```

你会看到如下输出：

```
Migrations for 'blog':
  0002_comment.py:
    - Create model Comment
```

Django 在 blog 应用下的 *migrations/*目录中生成了一个 *0002_comment.py* 文件。现在你需要创建一个关联数据库模式并且将这些改变应用到数据库中。运行以下命令来执行已经存在的数据迁移：

```
python manage.py migrate
```

你会获取以下输出：

```
Applying blog.0002_comment... OK
```

我们刚刚创建的数据迁移已经被执行，现在一张 *blog_comment* 表已经存在数据库中。
现在，我们可以添加我们新的模型（model）到管理站点中并通过简单的接口来管理评论。打开 blog 应用下的 *admin.py* 文件，添加 comment model 的导入,添加如下内容：

```
from .models import Post, Comment

class CommentAdmin(admin.ModelAdmin):
    list_display = ('name', 'email', 'post', 'created', 'active')
    list_filter = ('active', 'created', 'updated')
    search_fields = ('name', 'email', 'body')
admin.site.register(Comment, CommentAdmin)
```

运行命令 `python manage.py runserver` 来启动开发服务器然后在浏览器中打开 http://127.0.0.1:8000/admin/ 。你会看到新的模型（model）在 **Blog** 区域中出现，如下所示：

django-2-5

我们的模型(model)现在已经被注册到了管理站点，这样我们就可以使用简单的接口来管理评论实例。

通过模型（models）创建表单

我们仍然需要构建一个表单让我们的用户在 blog 帖子下进行评论。请记住，Django 有两个用来创建表单的基础类：*Form* 和 *ModelForm*。你先前已经使用过第一个让用户通过 email 来分享帖子。在当前的例子中，你将需要使用 *ModelForm*，因为你必须通过你的 *Comment* 模型（model）动态的创建表单。编辑 blog 应用下的 *forms.py*，添加如下代码：

```
from .models import Comment

class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment
        fields = ('name', 'email', 'body')
```

根据模型（model）创建表单，我们只需要在这个表单的 *Meta* 类里表明使用哪个模型（model）来构建表单。Django 将会解析 model 并为我们动态的创建表单。每一种模型（model）字段类型都有对应的默认表单字段类型。表单验证时会考虑到我们定义模型(model)字段的方式。Django 为模型(model)中包含的每个字段都创建了表单字段。然而，使用 *fields* 列表你可以明确的告诉框架你想在你的表单中包含哪些字段，或者使用 *exclude* 列表定义你想排除在外的那些字段。对于我们的 *CommentForm* 来说,我们在表单中只需要 *name*,*email*,和 *body* 字段，因为我们只需要用到这 3 个字段让我们的用户来填写。

在视图（views）中操作 *ModelForms*

为了能更简单的处理它，我们会使用帖子的详情视图（view）来实例化表单。编辑 *views.py* 文件**(注：原文此处有错，应为 `views.py`)**，导入 *Comment* 模型（model）和 *CommentForm* 表单，并且修改 *post_detail* 视图（view）如下所示：

```
from .models import Post, Comment
from .forms import EmailPostForm, CommentForm

def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post, slug=post,
                                status='published',
                                publish__year=year,
                                publish__month=month,
                                publish__day=day)
    # List of active comments for this post
    comments = post.comments.filter(active=True)
    new_comment = None

    if request.method == 'POST':
        # A comment was posted
        comment_form = CommentForm(data=request.POST)
        if comment_form.is_valid():
            # Create Comment object but don't save to database yet
            new_comment = comment_form.save(commit=False)
            # Assign the current post to the comment
            new_comment.post = post
            # Save the comment to the database
            new_comment.save()
    else:
        comment_form = CommentForm()
    return render(request,
                  'blog/post/detail.html',
                  {'post': post,
                  'comments': comments,
                  'new_comment': new_comment,
                  'comment_form': comment_form})
```

让我们来回顾一下我们刚才对视图（view）添加了哪些操作。我们使用 *post_detail* 视图（view）来显示帖子和该帖子的评论。我们添加了一个查询集（QuerySet）来获取这个帖子所有有效的评论：

```
comments = post.comments.filter(active=True)
```

我们从 *post* 对象开始构建这个查询集（QuerySet）。我们使用关联对象的 manager，这个 manager 是我们在 *Comment* 模型（model）中使用 *related_name* 关系属性为 *comments* 定义的。
我们还在这个视图（view）中让我们的用户添加一条新的评论。因此，如果这个视图（view）是通过 GET 请求被加载的，那么我们用 `comment_fomr = commentForm()`来创建一个表单实例。如果是通过 POST 请求，我们使用提交的数据并且用 *is_valid()*方法验证这些数据去实例化表单。如果这个表单是无效的，我们会用验证错误信息渲染模板（template）。如果表单通过验证，我们会做以下的操作：

- 1.我们通过调用这个表单的 *save()*方法创建一个新的 *Comment* 对象，如下所示：

```
new_comment = comment_form.save(commit=False)
```

*Save()*方法创建了一个表单链接的 model 的实例，并将它保存到数据库中。如果你调用这个方法时设置 `comment=False`，你创建的模型（model）实例不会即时保存到数据库中。当你想在最终保存之前修改

这个 model 对象会非常方便，我们接下来将做这一步骤。*save()*方法是给 *ModelForm* 用的，而不是给 *Form* 实例用的，因为 *Form* 实例没有关联上任何模型（model）。

- 2.我们为我们刚创建的评论分配一个帖子：

```
new_comment.post = post
```

通过以上动作，我们指定新的评论是属于这篇给定的帖子。

- 3.最后，我们用下面的代码将新的评论保存到数据库中：

```
new_comment.save()
```

我们的视图（view）已经准备好显示和处理新的评论了。

在帖子详情模板（template）中添加评论

我们为帖子创建了一个管理评论的功能。现在我们需要修改我们的 *post_detail.html* 模板（template）来适应这个功能，通过做到以下步骤：

- 显示这篇帖子的评论总数
- 显示评论的列表
- 显示一个表单给用户来添加新的评论

首先，我们来添加评论的总数。打开 *views_detail.html* （**译者注：根据官网最新更正修改，原文是 blog_detail.html**）模板（template）在 *content* 区块中添加如下代码：

```
{% with comments.count as total_comments %}
  <h2>
    {{ total_comments }} comment{{ total_comments|pluralize }}
  </h2>
{% endwith %}
```

在模板（template）中我们使用 Django ORM 执行 `comments.count()` 查询集（QuerySet）。注意，Django 模板（template）语言中不使用圆括号来调用方法。`{% with %}` 标签（tag）允许我们分配一个值给新的变量，这个变量可以一直使用直到遇到`{% endwith %}`标签（tag）。

`{% with %}`模板（template）标签（tag）是非常有用的，可以避免直接操作数据库或避免多次调用花费较多的方法。

根据 *total_comments* 的值，我们使用 *pluralize* 模板（template）过滤器（filter）为单词 *comment* 显示复数后缀。模板（Template）过滤器（filters）获取到他们输入的变量值，返回计算后的值。我们将会在*第三章 扩展你的博客应用*中讨论更多的模板过滤器（tempalte filters）。

pluralize 模板（template）过滤器（filter）在值不为 1 时，会在值的末尾显示一个"s"。之前的文本将会被渲染成类似：*0 comments*, *1 comment* 或者 *N comments*。Django 内置大量的模板（template）标签（tags）和过滤器（filters）来帮助你以你想要的方式来显示信息。

现在，让我们加入评论列表。在模板（template）中之前的代码后面加入以下内容：

```
{% for comment in comments %}
  <div class="comment">
    <p class="info">
      Comment {{ forloop.counter }} by {{ comment.name }}
      {{ comment.created }}
    </p>
    {{ comment.body|linebreaks }}
  </div>
{% empty %}
  <p>There are no comments yet.</p>
```

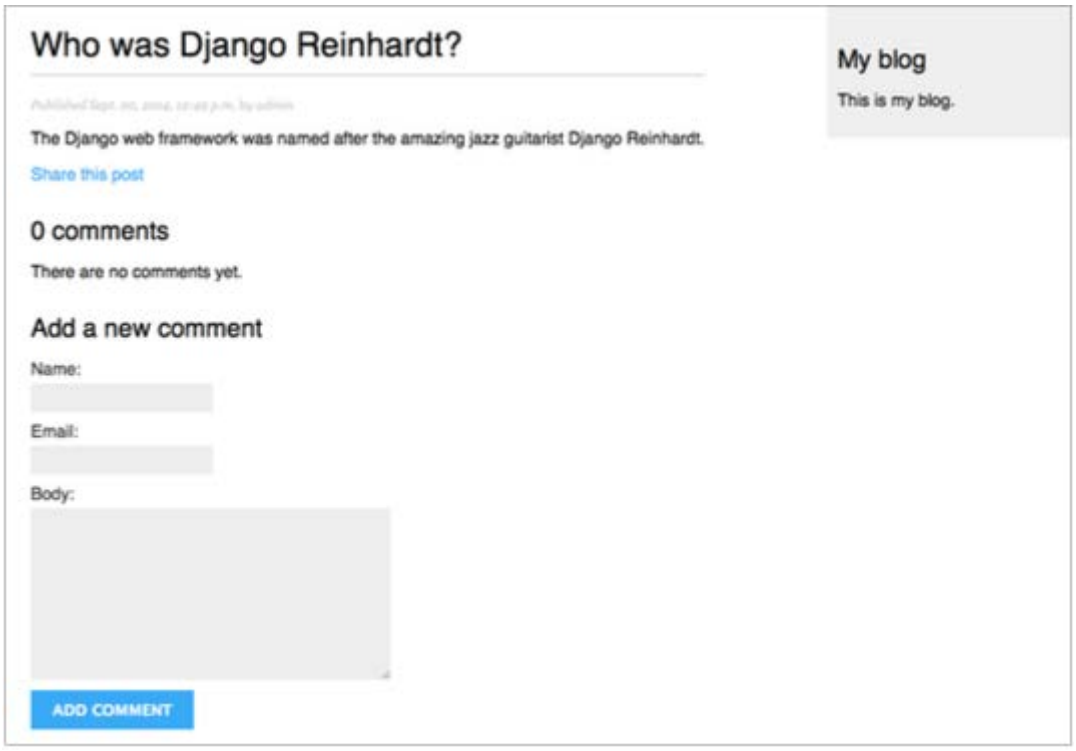
```
{% endfor %}
```

我们使用`{% for %}`模板（template）标签（tag）来循环所有的评论。如果 *comments* 列为空我们会显示一个默认的信息，告诉我们的用户这篇帖子还没有任何评论。我们使用 `{{ forloop.counter }}`变量来枚举所有的评论，在每次迭代中该变量都包含循环计数。之后我们显示发送评论的用户名，日期，和评论的内容。

最后，当表单提交成功后,你需要渲染表单或者显示一条成功的信息来代替之前的内容。在之前的代码后面添加如下内容：

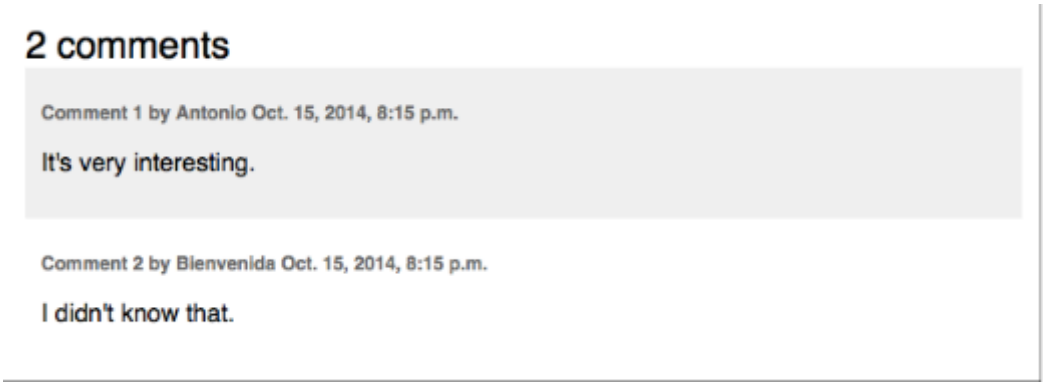
```
{% if new_comment %}
  <h2>Your comment has been added.</h2>
{% else %}
  <h2>Add a new comment</h2>
  <form action="." method="post">
    {{ comment_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Add comment"></p>
  </form>
{% endif %}
```

这段代码非常简洁明了：如果 *new_comment* 对象存在，我们会展示一条成功信息因为成功创建了一条新评论。否则，我们用段落`<p>`元素渲染表单中每一个字段，并且包含 *POST* 请求需要的 *CSRF* 令牌。在浏览器中打开 http://127.0.0.1:8000/blog/ 然后点击任意一篇帖子的标题查看它的详情页面。你会看到如下页面展示：



django-2-6

使用该表单添加数条评论。这些评论会在你的帖子下面根据时间排序来展示，类似下图：



django-2-7

在你的浏览器中打开 http://127.0.0.1:8000/admin/blog/comment/ 。你会在管理页面中看到你创建的评论列表。点击其中一个进行编辑，取消选择 **Active** 复选框，然后点击 **Save** 按钮。你会再次被重定向到评论列表页面，刚才编辑的评论 **Save** 列将会显示成一个没有激活的图标。类似下图的第一个评论：

django-2-8

增加标签（tagging）功能

在实现了我们的评论系统之后，我们准备创创建一个方法来给我们的帖子添加标签。我们将通过在我们的项目中集成第三方的 Django 标签应用来完成这个功能。*django-taggit* 是一个可复用的应用，它会提供给你一个 *Tag* 模型（model）和一个管理器（manager）来方便的给任何模型（model）添加标签。你可以在 https://github.com/alex/django-taggit 看到它的源码。

首先，你需要通过 pip 安装 django-taggit，运行以下命令：

```
pip install django-taggit==0.17.1**（译者注：根据@孤独狂饮 验证，直接 `pip install django-taggit` 安装最新版即可，原作者提供的版本过旧会有问题，感谢@孤独狂饮）**
```

之后打开 *mysite* 项目下的 *settings.py* 文件，在 *INSTALLED_APPS* 设置中设置如下：

```
INSTALLED_APPS = (
    # ...
    'blog',
    'taggit',
)
```

打开你的 blog 应用下的 *model.py* 文件，给 *Post* 模型（model）添加 *django-taggit* 提供的 *TaggableManager* 管理器（manager），使用如下代码：

```
from taggit.managers import TaggableManager
class Post(models.Model):
    # ...
    tags = TaggableManager()
```

这个 *tags* 管理器（manager）允许你给 *Post* 对象添加，获取以及移除标签。

运行以下命令为你的模型（model）改变创建一个数据库迁移：

```
python manage.py makemigrations blog
```

你会看到如下输出：

```
Migrations for 'blog':
  0003_post_tags.py:
    - Add field tags to post
```

现在，运行以下代码在数据库中生成 *django-taggit* 模型（model）对应的表以及同步你的模型（model）的改变：

```
python manage.py migrate
```

你会看到以下输出,：

```
Applying taggit.0001_initial... OK
Applying taggit.0002_auto_20150616_2121... OK
Applying blog.0003_post_tags... OK
```

你的数据库现在已经可以使用 *django-taggit* 模型（model）。打开终端运行命令 `python manage.py shell` 来学习如何使用 *tags* 管理器（manager）。首先，我们取回我们的其中一篇帖子（该帖子的 ID 为 1）：

```
>>> from blog.models import Post
>>> post = Post.objects.get(id=1)
```

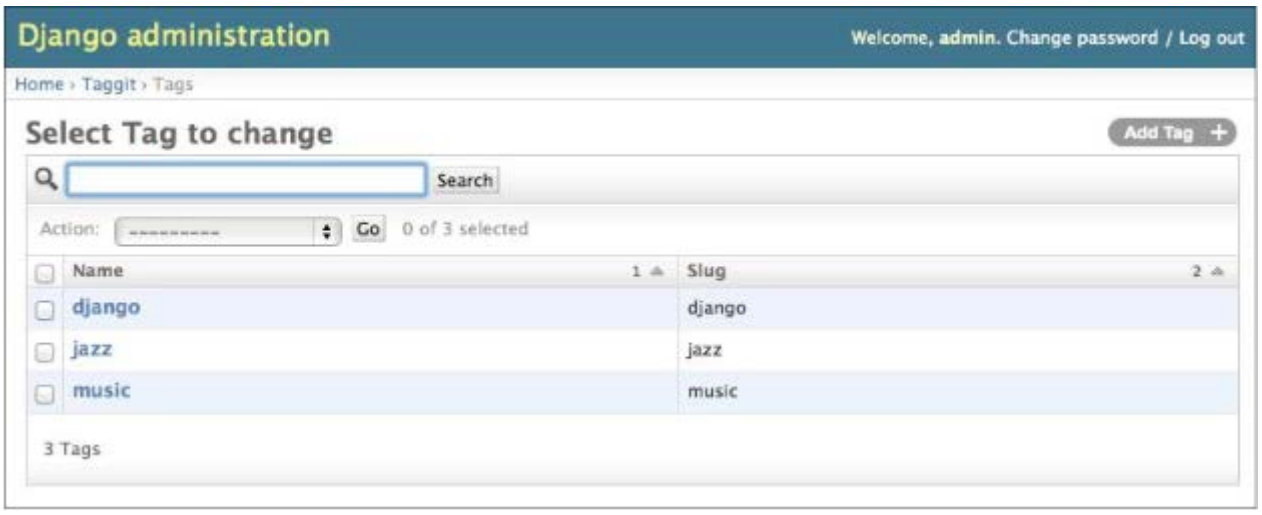
之后为它添加一些标签并且取回它的标签来检查标签是否添加成功：

```
>>> post.tags.add('music', 'jazz', 'django')
>>> post.tags.all()
[<Tag: jazz>, <Tag: django>, <Tag: music>]
```

最后，移除一个标签并且再次检查标签列表：

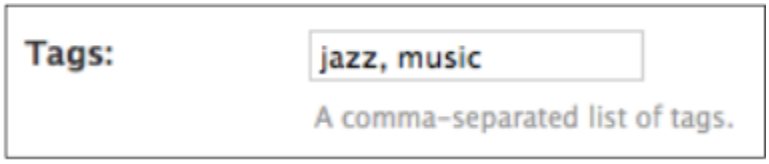
```
>>> post.tags.remove('django')
>>> post.tags.all()
[<Tag: jazz>, <Tag: music>]
```

非常简单，对吧？运行命令 `python manage.py runserver` 启动开发服务器，在浏览器中打开 http://127.0.0.1:8000/admin/taggit/tag/ 。你会看到管理页面包含了 *taggit* 应用的 *Tag* 对象列表：



django-2-9

转到 http://127.0.0.1:8000/admin/blog/post/ 并点击一篇帖子进行编辑。你会看到帖子中包含了一个新的 **Tags** 字段如下所示，你可以非常容易的编辑它：

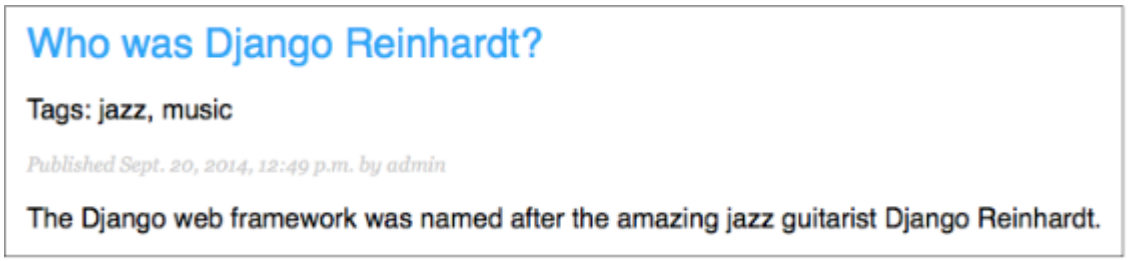


django-2-10

现在，我们准备编辑我们的 blog 帖子来显示这些标签。打开 *blog/post/list.html* 模板（template）在帖子标题下方添加如下 HTML 代码：

```
<p class="tags">Tags: {{ post.tags.all|join:", " }}</p>
```

join 模板（template）过滤器（filter）的功能类似 python 字符串的 join()方法，将给定的字符串连接起来。在浏览器中打开 http://127.0.0.1:8000/blog/ 。 你会看到每一个帖子的标题下面的标签列表：



django-2-11

现在，让我们来编辑我们的 *post_list* 视图（view）让用户可以列出打上了特定标签的所有帖子。打开 blog 应用下的 *views.py* 文件，从 *django-taggit* 中导入 *Tag* 模型（model），然后修改 *post_list* 视图（view）让它可以通过标签选择性的过滤，如下所示：

```
from taggit.models import Tag

def post_list(request, tag_slug=None):
    object_list = Post.published.all()
    tag = None

    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        object_list =   object_list.filter(tags__in=[tag])
        # ...
```

这个视图（view）做了以下工作：

- 1.视图（view）带有一个可选的 *tag_slug* 参数，默认是一个 *None* 值。这个参数会带进 URL 中。
- 2.视图（view）的内部，我们构建了初始的查询集（QuerySet），取回所有发布状态的帖子，假如给予一个标签 slug，我们通过 `get_object_or_404()`用给定的 slug 来获取标签对象。
- 3.之后我们过滤所有帖子只留下包含给定标签的帖子。因为有一个多对多（many-to-many）的关系，我们必须通过给定的标签列表来过滤，在我们的例子中标签列表只包含一个元素。

要记住查询集（QuerySets)是惰性的。这个查询集（QuerySets)只有当我们在模板（template）中循环渲染帖子列表时才会被执行。

最后，修改视图（view）最底部的 *render()*函数来，传递 *tag* 变量给模板（template）。这个视图（view）完成后如下所示：

```
def post_list(request, tag_slug=None):
   object_list = Post.published.all()
   tag = None
```

```
    if tag_slug:
        tag = get_object_or_404(Tag, slug=tag_slug)
        object_list = object_list.filter(tags__in=[tag])

    paginator = Paginator(object_list, 3) # 3 posts in each page
    page = request.GET.get('page')
    try:
        posts = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer deliver the first page
        posts = paginator.page(1)
    except EmptyPage:
        # If page is out of range deliver last page of results
        posts = paginator.page(paginator.num_pages)
    return render(request, 'blog/post/list.html', {'page': page,
                                                   'posts': posts,
                                                   'tag': tag})
```

打开 blog 应用下的 *url.py* 文件，注释基于类的 *PostListView* URL 模式，然后取消 *post_list* 视图（view）的注释，如下所示：

```
url(r'^$', views.post_list, name='post_list'),
# url(r'^$', views.PostListView.as_view(), name='post_list'),
```

添加下面额外的 URL pattern 到通过标签过滤过的帖子列表中：

```
url(r'^tag/(?P<tag_slug>[-\w]+)/$',views.post_list,
    name='post_list_by_tag'),
```

如你所见，两个模式都指向了相同的视图（view），但是我们可以给它们不同的命名。第一个模式会调用 *post_list* 视图（view）并且不带上任何可选参数。然而第二个模式会调用这个视图（view）带上 *tag_slug* 参数。

因为我们要使用 *post_list* 视图（view）,编辑 *blog/post/list.html* 模板（template），使用 *posts* 对象修改 pagination，如下所示：

```
{% include "pagination.html" with page=posts %}
```

在`{% for %}`循环上方添加如下代码：

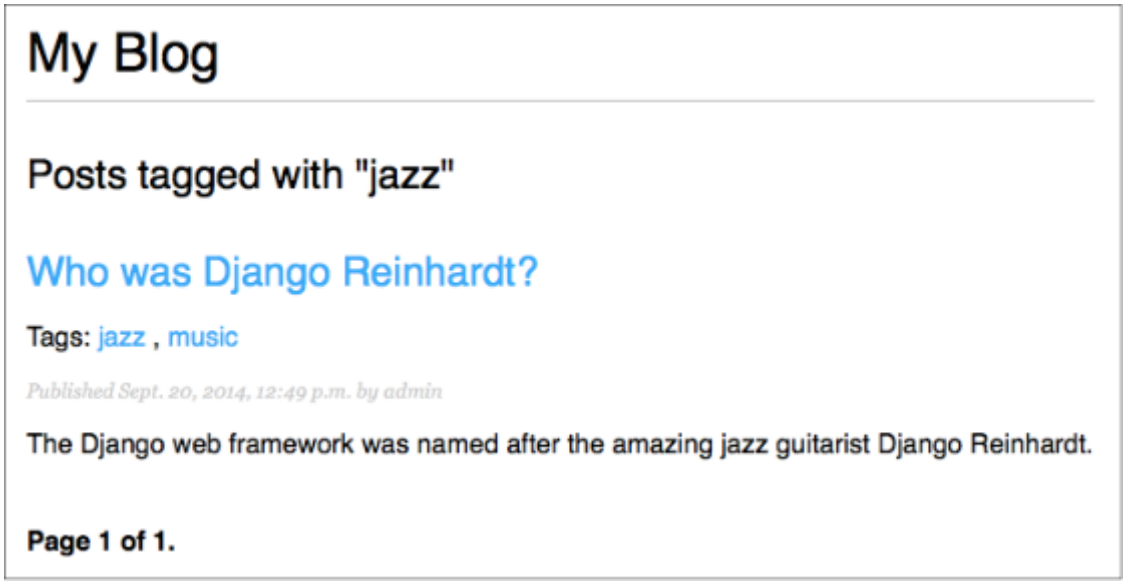
```
{% if tag %}
  <h2>Posts tagged with "{{ tag.name }}"</h2>
{% endif %}
```

如果用户正在访问 blog，他会看到所有帖子列表。如果他指定一个标签来过滤所有的帖子，他就会看到以上的信息。现在，修改标签的显示方式，如下所示：

```
<p class="tags">
  Tags:
  {% for tag in post.tags.all %}
    <a href="{% url "blog:post_list_by_tag" tag.slug %}">
      {{ tag.name }}
    </a>
```

```
    {% if not forloop.last %}, {% endif %}
  {% endfor %}
</p>
```

现在，我们循环一个帖子的所有标签，通过某一标签来显示一个自定义的链接 URL。我们通过`{% url "blog:post_list_by_tag" tag.slug %}`，用 URL 的名称以及标签 slug 作为参数来构建 URL。我们使用逗号分隔这些标签。
在浏览器中打开 http://127.0.0.1:8000/blog/ 然后点击任意的标签链接，你会看到通过该标签过滤过的帖子列表，如下所示：



django-2-12

检索类似的帖子

如今，我们已经可以给我们的 blog 帖子加上标签，我们可以通过它们做更多有意思的事情。通过使用标签，我们能够很好的分类我们的 blog 帖子。拥有类似主题的帖子一般会有几个共同的标签。我们准备创建一个功能：通过帖子共享的标签数量来显示类似的帖子。这样的话，当一个用户阅读一个帖子，我们可以建议他们去读其他有关联的帖子。
为了通过一个特定的帖子检索到类似的帖子，我们需要做到以下几点：

- 返回当前帖子的所有标签。
- 返回所有带有这些标签的帖子。
- 在返回的帖子列表中排除当前的帖子，避免推荐相同的帖子。
- 通过和当前帖子共享的标签数量来排序所有的返回结果。
- 假设有两个或多个帖子拥有相同数量的标签，推荐最近的帖子。
- 限制我们想要推荐的帖子数量。

这些步骤可以转换成一个复杂的查询集（QuerySet），该查询集（QuerySet）我们需要包含在我们的 *post_detail* 视图（view）中。打开 blog 应用中的 *view.py* 文件，在顶部添加如下导入：

```
from django.db.models import Count
```

这是 Django ORM 的 *Count* 聚合函数。这个函数允许我们处理聚合计算。然后在 *post_detail* 视图（view）的 *render()* 函数之前添加如下代码：

```
# List of similar posts
post_tags_ids = post.tags.values_list('id', flat=True)
similar_posts = Post.published.filter(tags__in=post_tags_ids)\
                              .exclude(id=post.id)
similar_posts = similar_posts.annotate(same_tags=Count('tags'))\
```

```
                    .order_by('-same_tags','-publish')[:4]
```

以上代码的解释如下：

- 1.我们取回了一个包含当前帖子所有标签的 ID 的 Python 列表。*values_list()* 查询集（QuerySet）返回包含给定的字段值的元祖。我们传给元祖 `flat=True` 来获取一个简单的列表类似`[1,2,3,...]`。
- 2.我们获取所有包含这些标签的帖子排除了当前的帖子。
- 3.我们使用 *Count* 聚合函数来生成一个计算字段 *same_tags*，该字段包含与查询到的所有 标签共享的标签数量。
- 4.我们通过共享的标签数量来排序（降序）结果并且通过 *publish* 字段来挑选拥有相同共享标签数量的帖子中的最近的一篇帖子。我们对返回的结果进行切片只保留最前面的 4 篇帖子。

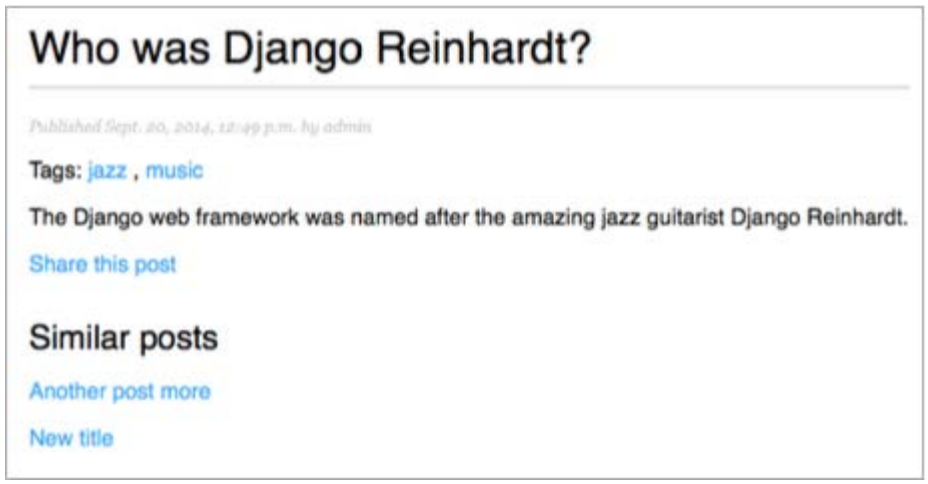
在 *render()*函数中给上下文字典增加 *similar_posts* 对象，如下所示：

```
return render(request,
              'blog/post/detail.html',
              {'post': post,
              'comments': comments,
              'comment_form': comment_form,
              'similar_posts': similar_posts})
```

现在，编辑 *blog/post/detail.html* 模板（template）在帖子评论列表前添加如下代码：

```
<h2>Similar posts</h2>
  {% for post in similar_posts %}
    <p>
      <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
    </p>
  {% empty %}
    There are no similar posts yet.
  {% endfor %}
```

推荐你在你的帖子详情模板中添加标签列表，就像我们在帖子列表模板所做的一样。现在，你的帖子详情页面看上去如下所示：



django-2-13

你已经成功的为你的用户推荐了类似的帖子。*django-taggit* 还内置了一个 `similar_objects()` 管理器（manager）使你可以通过共享的标签返回所有对象。你可以通过访问 http://django-taggit.readthedocs.org/en/latest/api.html 看到所有 django-taggit 管理器。

总结

在本章中，你学习了如何使用 Django 的表单和模型（model）表单。你创建了一个通过 email 分享你的站点内容的系统，还为你的博客创建了一个评论系统。通过集成一个可复用的应用，你为你的帖子增加了打标签的功能。同时，你还构建了一个复杂的查询集（QuerySets)用来返回类似的对象。
在下一章中，你会学习到如何创建自定义的模板（temaplate）标签（tags）和过滤器（filters）。你还会为你的博客应用构建一个自定义的站点地图，集成一个高级的搜索引擎。

第三章 扩展你的 blog 应用

在上一章中我们学习了表单的基础以及如何在项目中集成第三方的应用。本章将会包含以下内容：

- 创建自定义的模板标签（template tags)和过滤器（filters）
- 添加一个站点地图和帖子反馈（post feed）
- 使用 Solr 和 Haystack 构建一个搜索引擎

创建自定义的模板标签（template tags)和过滤器（filters）

Django 提供了很多内置的模板标签（tags），例如`{% if %}`或者`{% block %}`。你已经在你的模板（template）中使用过一些了。你可以在 https://docs.djangoproject.com/en/1.8/ ref/templates/builtins/ 中找到关于内置模板标签（template tags）以及过滤器（filter）的完整参考。
当然，Django 也允许你创建自己的模板标签（template tags）来执行自定义的动作。当你需要在你的模板中添加功能而 Django 模板标签（template tags)的核心设置无法提供此功能的时候，自定义模板标签会非常方便。

创建自定义的模板标签（template tags）

Django 提供了以下帮助函数（functions）来允许你以一种简单的方式创建自己的模板标签（template tags）：

- simple_tag：处理数据并返回一个字符串（string）
- inclusion_tag：处理数据并返回一个渲染过的模板（template）
- assignment_tag：处理数据并在上下文（context）中设置一个变量（variable）

模板标签（template tags）必须存在 Django 的应用中。
进入你的 blog 应用目录，创建一个新的目录命名为 *templatetags* 然后在该目录下创建一个空的 ***init.py*** 文件。接着在该目录下继续创建一个文件并命名为 *blog_tags.py*。到此，我们的 blog 应用文件结构应该如下所示：

```
blog/
    __init__.py
    models.py
    ...
    templatetags/
        __init__.py
        blog_tags.py
```

文件的命名是非常重要的。你将在模板（template）中使用这些模块的名字加载你的标签（tags)。
我们将要开始创建一个简单标签（simple tag）来获取 blog 中所有已发布的帖子。编辑你刚才创建的 *blog_tags.py* 文件，加入以下代码：

```
from django import template

register = template.Library()

from ..models import Post
```

```
@register.simple_tag
def total_posts():
    return Post.published.count()
```

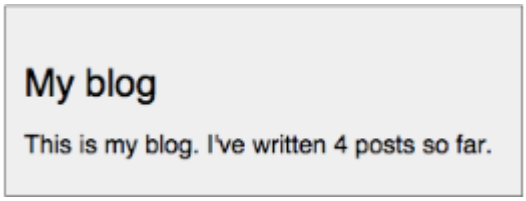
我们已经创建了一个简单的模板标签（template tag）用来取回目前为止所有已发布的帖子。每一个模板标签（template tags）都需要包含一个叫做 *register* 的变量来表明自己是一个有效的标签（tag）库。这个变量是 *template.Library* 的一个实例，它是用来注册你自己的模板标签（template tags）和过滤器（filter）的。我们用一个 Python 函数定义了一个名为 *total_posts* 的标签，并用`@register.simple-tag`装饰器定义此函数为一个简单标签（tag）并注册它。
Django 将会使用这个函数名作为标签（tag）名。如果你想使用别的名字来注册这个标签（tag），你可以指定装饰器的 *name* 属性，比如`@register.simple_tag(name='my_tag')`。
在添加了新的模板标签（template tags）模块后，你必须重启 Django 开发服务才能使用新的模板标签（template tags)和过滤器（filters)。
在使用自定义的模板标签（template tags)之前，你必须使用*{% load %}*标签在模板（template）中来加载它们才能有效。就像之前提到的，你需要使用包含了你的模板标签（template tags)和过滤器（filter)的 Python 模块的名字。打开 *blog/base.html* 模板（template）在顶部添加*{% load blog_tags %}*加载你自己的模板标签（template tags)模块。之后使用你创建的标签（tag）来显示你的帖子总数。只需要在你的模板（template）中添加*{% total_posts %}*。最新的模板（template）看上去如下所示：

```
{% load blog_tags %}
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
  <div id="content">
    {% block content %}
    {% endblock %}
  </div>
<div id="sidebar">
  <h2>My blog</h2>
  <p>This is my blog. I've written {% total_posts %} posts so far.</p>
  </div>
</body>
</html>
```

我们需要重启服务来保证新的文件被加载到项目中。使用 *Ctrl+C* 停止服务然后通过以下命令再次启动：

```
python manage.py runserver
```

在浏览器中打开 http://127.0.0.1:8000/blog/ 。你会在站点的侧边栏（sidebar）看到帖子的总数，如下所示：



django-3-1

自定义模板标签（template tags）的作用是你可以处理任何的数据并且在任何模板（template）中添加它而不用去关心视图（view）执行。你可以在你的模板（template）中运行查询集（QuerySets）或者处理任何数据展示结果。
现在，我们要创建另外一个标签（tag），可以在我们 blog 的侧边栏（sidebar）展示最新的几个帖子。这一次我们要使用一个包含标签（inclusion tag）。使用一个包含标签（inclusion tag），你就可以利用模板标签（template tags）返回的上下文变量（context variables）来渲染模板（template）。编辑 *blog_tags.py* 文件，添加如下代码：

```
@register.inclusion_tag('blog/post/latest_posts.html')
def show_latest_posts(count=5):
    latest_posts = Post.published.order_by('-publish')[:count]
    return {'latest_posts': latest_posts}
```

在以上代码中，我们通过装饰器*@register.inclusion_tag* 注册模板标签（template tag），指定模板（template）必须被 *blog/post/latest_posts.html* 返回的值渲染。我们的模板标签（template tag)将会接受一个可选的 *count* 参数（默认是 5）允许我们指定我们想要显示的帖子数量。我们使用这个变量来限制 `Post.published.order_by('-publish')[:count]`查询的结果。请注意，这个函数返回了一个字典变量而不是一个简单的值。包含标签（inclusion tags）必须返回一个字典值，作为上下文（context）来渲染特定的模板(template)。包含标签(inclusion tags)返回一个字典。这个我们刚创建的模板标签(template tag）可以通过传入可选的评论数量值来使用显示，类似*{% show_latest_posts 3 %}。
现在，在 *blog/post/*下创建一个新的模板（template）文件并且命名为 *latest_posts.html*。在该文件中添加如下代码：

```
<ul>
{% for post in latest_posts %}
  <li>
    <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
  </li>
{% endfor %}
</ul>
```

在这里，我们使用模板标签（template tag）返回的 *latest_posts* 变量展示了一个无序的帖子列表。现在，编辑 *blog/base.hmtl* 模板（template）添加这个新的模板标签（template tag）来展示最新的 3 条帖子。侧边栏（sidebar）区块（block）看上去应该如下所示：

```
<div id="sidebar">
  <h2>My blog</h2>
  <p>This is my blog. I've written {% total_posts %} posts so far.</p>
  <h3>Latest posts</h3>
  {% show_latest_posts 3 %}
</div>
```

这个模板标签(template tag)被调用而且传入了需要展示的帖子数(原文此处 number of comments，应该是写错了)。当前模板（template）在给予上下文（context）的位置会被渲染。
现在，回到浏览器刷新这个页面，侧边栏应该如下图所示：

django-3-2

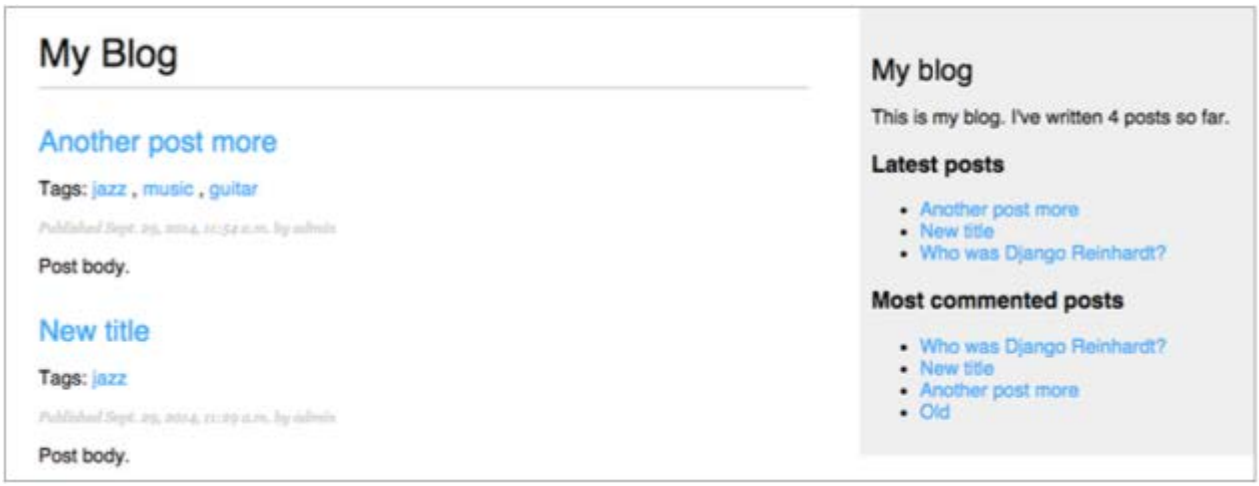
最后，我们来创建一个分配标签（assignment tag）。分配标签（assignment tag）类似简单标签（simple tags）但是他们将结果存储在给予的变量中。我们将会创建一个分配标签（assignment tag）来展示拥有最多评论的帖子。编辑 *blog_tags.py* 文件，在其中添加如下导入和模板标签：

```
from django.db.models import Count


@register.assignment_tag
def get_most_commented_posts(count=5):
    return Post.published.annotate(
                total_comments=Count('comments')
            ).order_by('-total_comments')[:count]
```

这个查询集（QuerySet）使用 *annotate()*函数，为了进行聚合查询，使用了 *Count* 聚合函数。我们构建了一个查询集（QuerySet），聚合了每一个帖子的评论总数并保存在 *total_comments* 字段中，接着我们通过这个字段对查询集（QuerySet）进行排序。我们还提供了一个可选的 *count* 变量，通过给定的值来限制返回的帖子数量。
除了 *Count* 以外，Django 还提供了不少聚合函数，例如 *Avg,Max,Min,Sum*.你可以在 https://docs.djangoproject.com/en/1.8/topics/db/aggregation/ 页面读到更多关于聚合方法的信息。
编辑 *blog/base.html* 模板（template），在侧边栏（sidebar）`<div>`元素中添加如下代码：

```
<h3>Most commented posts</h3>
{% get_most_commented_posts as most_commented_posts %}
<ul>
{% for post in most_commented_posts %}
  <li>
    <a href="{{ post.get_absolute_url }}">{{ post.title }}</a>
  </li>
{% endfor %}
</ul>
```

使用分配模板标签（assignment template tags）的方法是*{% template_tag as variable %}*。对于我们的模板标签（template tag）来说，我们使用*{% get_most_commented_posts as most_commented_posts %}*。 这样，我们可以存储这个模板标签（template tag）返回的结果到一个新的名为 *most_commented_posts* 变量中。之后，我们就可以用一个无序列表(unordered list)显示返回的帖子。
现在，打开浏览器刷新页面来看下最终的结果，应该如下图所示：



django-3-3

你可以在 https://docs.djangoproject.com/en/1.8/howto/custom-template-tags/ 页面得到更多的关于自定义模板标签（template tags）的信息。

创建自定义的模板过滤器（template filters）

Django 拥有多种内置的模板过滤器（template filters）允许你对模板（template）中的变量进行修改。这些过滤器其实就是 Python 函数并提供了一个或两个参数————一个是需要处理的变量值，一个是可选的参数。它们返回的值可以被展示或者被别的过滤器（filters）处理。一个过滤器（filter）类似 *{{ variable|my_filter }}*或者再带上一个参数，类似*{{ variable|my_filter:"foo" }}*。你可以对一个变量调用你想要的多次过滤器（filter），类似*{{ variable|filter1|filter2 }}*，每一个过滤器（filter）都会对上一个过滤器（filter）输出的结果进行过滤。

我们这就创建一个自定义的过滤器（filter），可以在我们的 blog 帖子中使用 markdown 语法，然后在模板（template）中将帖子内容转变为 HTML。Markdown 是一种非常容易使用的文本格式化语法并且它可以转变为 HTML。你可以在 http://daringfireball.net/projects/markdown/basics 页面学习这方面的知识。

首先，通过 pip 渠道安装 Python 的 markdown 模板：

```
pip install Markdown==2.6.2
```

之后编辑 *blog_tags.py* 文件，包含如下代码：

```
from django.utils.safestring import mark_safe
import markdown

@register.filter(name='markdown')
def markdown_format(text):
    return mark_safe(markdown.markdown(text))
```

我们使用和模板标签（template tags）一样的方法来注册我们自己的模板过滤器（template filter）。为了避免我们的函数名和 *markdown* 模板名起冲突，我们将我们的函数命名为 *markdown_format*，然后将过滤器（filter）命名为 *markdown*，在模板（template）中的使用方法类似*{{ variable|markdown }}*。Django 会转义过滤器（filter）生成的 HTML 代码。我们使用 Django 提供的 *mark_safe* 方法来标记结果，在模板（template）中作为安全的 HTML 被渲染。默认的，Django 不会信赖任何 HTML 代码并且在输出之前会进行转义。唯一的例外就是被标记为安全转义的变量。这样的操作可以阻止 Django 从输出中执行潜在的危险的 HTML，并且允许你创建一些例外情况只要你知道你正在运行的是安全的 HTML。

现在，在帖子列表和详情模板（template）中加载你的模板标签（template tags）模块。在 *post/list.html* 和 *post/detail.html* 模板（template）的顶部*{% extends %}*的后方添加如下代码：

```
{% load blog_tags %}
```

在 *post/detail.thml* 模板中，替换以下内容：

```
{{ post.body|linebreaks }}
```

替换成：

```
{{ post.body|markdown }}
```

之后，在 *post/list.html* 文件中，替换以下内容：

```
{{ post.body|truncatewords:30|linebreaks }}
```

替换成：

```
{{ post.body|markdown|truncatewords_html:30 }}
```

truncateword_html 过滤器（filter）会在一定数量的单词后截断字符串，避免没有关闭的 HTML 标签（tags）。
现在，打开 http://127.0.0.1:8000/admin/blog/post/add/ ,添加一个帖子，内容如下所示：

```
This is a post formatted with markdown
--------------------------------------
*This is emphasized* and **this is more emphasized**.
Here is a list:
* One
* Two
* Three
And a [link to the Django website](https://www.djangoproject.com/)
```

在浏览器中查看帖子的渲染情况，你会看到如下图所示：

django-3-4

就像你所看到的，自定义的模板过滤器（template filters）对于自定义的格式化是非常有用的。你可以在 https://docs.djangoproject.com/en/1.8/howto/custom-template-tags/#writing-custom-templatefilters 页面获取更多关于自定义过滤器（filter）的信息。

为你的站点添加一个站点地图（sitemap）

Django 自带一个站点地图（sitemap）框架，允许你为你的站点动态生成站点地图（sitemap）。一个站点地图（sitemap）是一个 xml 文件，它会告诉搜索引擎你的网站中存在的页面，它们的关联和它们

更新的频率。使用站点地图（sitemap），你可以帮助网络爬虫（crawlers）来对你的网站内容进行索引和标记。

Django 站点地图（sitemap）框架依赖 *django.contrib.sites* 模块，这个模块允许你将对象和正在你项目运行的特殊网址关联起来。当你想用一个单独 Django 项目运行多个网站时，这是非常方便的。为了安装站点地图（sitemap）框架，我们需要在项目中激活 *sites* 和 *sitemap* 这两个应用。编辑项目中的 *settings.py* 文件在 *INSTALLED_APPS* 中添加 *django.contrib.sites* 和 *django.contrib.sitemaps*。之后为站点 ID 定义一个新的设置，如下所示：

```
SITE_ID = 1
# Application definition
INSTALLED_APPS = (
# ...
'django.contrib.sites',
'django.contrib.sitemaps',
)
```

现在，运行以下命令在数据库中为 Django 的站点应用创建所需的表：

```
python manage.py migrate
```

你会看到如下的输出内容：

```
Applying sites.0001_initial... OK
```

sites 应用现在已经在数据库中进行了同步。现在，在你的 *blog* 应用目录下创建一个新的文件命名为 *sitemaps.py*。打开这个文件，输入以下代码：

```
from django.contrib.sitemaps import Sitemap
from .models import Post

class PostSitemap(Sitemap):
    changefreq = 'weekly'
    priority = 0.9
    def items(self):
        return Post.published.all()
    def lastmod(self, obj):
        return obj.publish
```

通过继承 *sitemaps* 模块提供的 *Sitemap* 类我们创建一个自定义的站点地图（sitemap）。*changefreq* 和 *priority* 属性表明了帖子页面修改的频率和它们在网站中的关联性（最大值是 1）。*items()*方法返回了在这个站点地图（sitemap）中所包含对象的查询集（QuerySet）。默认的，Django 在每个对象中调用 *get_absolute_url()*方法来获取它的 URL。请记住，这个方法是我们在第一章（创建一个 blog 应用）中创建的，用来获取每个帖子的标准 URL。如果你想为每个对象指定 URL，你可以在你的站点地图（sitemap）类中添加一个 *location* 方法。*lastmode* 方法接收 *items()*返回的每一个对象并且返回对象的最后修改时间。*changefreq* 和 *priority* 两个方法既可以是方法也可以是属性。你可以在 Django 的官方文档 https://docs.djangoproject.com/en/1.8/ref/contrib/sitemaps/ 页面中获取更多的站点地图（sitemap）参考。

最后，我们只需要添加我们的站点地图（sitemap）URL。编辑项目中的主*urls.py 文件，如下所示添加站点地图（sitemap）：

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

```
from blog.sitemaps import PostSitemap

sitemaps = {
    'posts': PostSitemap,
}

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^blog/',
        include('blog.urls'namespace='blog', app_name='blog')),
    url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps},
        name='django.contrib.sitemaps.views.sitemap'),
]
```

在这里，我们加入了必要的导入并定义了一个 *sitemaps* 的字典。我们定义了一个 URL 模式来匹配 *sitemap.xml* 并使用 *sitemap* 视图（view）。*sitemaps* 字典会被传入到 *sitemap* 视图（view）中。现在，在浏览器中打开 http://127.0.0.1:8000/sitemap.xml 。你会看到类似下方的 XML 代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
    <url>
        <loc>http://example.com/blog/2015/09/20/another-post/</loc>
        <lastmod>2015-09-29</lastmod>
        <changefreq>weekly</changefreq>
        <priority>0.9</priority>
    </url>
    <url>
        <loc>http://example.com/blog/2015/09/20/who-was-djangoreinhardt/</loc>
        <lastmod>2015-09-20</lastmod>
        <changefreq>weekly</changefreq>
        <priority>0.9</priority>
    </url>
</urlset>
```

调用 *get_absolute_url()*方法，每个帖子的 URL 已经被构建。如同我们之前在站点地图(sitemap)中所指定的，*lastmode* 属性对应该帖子的 *publish* 日期字段，*changefreq* 和 *priority* 属性也从我们的 *PostSitemap* 类中带入。你能看到被用来构建 URL 的域（domain）是 *example.com*。这个域（domain）来自存储在数据库中的一个 *Site* 对象。这个默认的对象是在我们之前同步 *sites* 框架数据库时创建的。在你的浏览器中打开 http://127.0.0.1:8000/admin/sites/site/ ，你会看到如下图所示：

django-3-5

这是 *sites* 框架管理视图（admin view）的列表显示。在这里，你可以设置 *sites* 框架使用的域（domain）或者主机（host），而且应用也依赖它们。为了生成能在我们本地环境可用的 URL，更改域（domain）名为 *127.0.0.1:8000*，如下图所示并保存：

django-3-6

为了开发需要我们指向了我们本地主机。在生产环境中，你必须使用你自己的 *sites* 框架域（domain）名。

为你的 blog 帖子创建 feeds

译者注：这节中有不少英文单词，例如 feed，syndication，Atom 等，没有比较好的翻译，很多地方也都是直接不翻译保留原文，所以我也保留原文）

Django 有一个内置的 syndication feed 框架，就像用 *sites* 框架创建站点地图（sitemap）一样，使用类似的方式（manner），你可以动态（dynamically）生成 RSS 或者 Atom feeds。

在 blog 应用的目录下创建一个新文件命名为 *feeds.py*。添加如下代码：

```
from django.contrib.syndication.views import Feed
from django.template.defaultfilters import truncatewords
from .models import Post

class LatestPostsFeed(Feed):
    title = 'My blog'
    link = '/blog/'
    description = 'New posts of my blog.'

    def items(self):
        return Post.published.all()[:5]

    def item_title(self, item):
        return item.title

    def item_description(self, item):
        return truncatewords(item.body, 30)
```

首先，我们继承了 syndication 框架的 *Feed* 类创建了一个子类。其中的 *title*，*link*，*description* 属性各自对应 RSS 中的`<title>`,`<link>`,`<description>`元素。

*items()*方法返回包含在 feed 中的对象。我们只给这个 feed 取回最新五个已发布的帖子。*item_title()* 和 *item_description()*方法接受 *items()*返回的每个对象然后返回每个 item 各自的标题和描述信息。我们使用内置的 *truncatewords* 模板过滤器（template filter）构建帖子的描述信息并只保留最前面的 30 个单词。

现在，编辑 blog 应用下的 *urls.py* 文件，导入你刚创建的 *LatestPostsFeed*，在新的 URL 模式（pattern）中实例化 feed：

```
from .feeds import LatestPostsFeed
```

```
urlpatterns = [
    # ...
    url(r'^feed/$', LatestPostsFeed(), name='post_feed'),
]
```

在浏览器中转到 http://127.0.0.1:8000/blog/feed/ 。你会看到最新的5个blog帖子的RSS feedincluding：

```
<?xml version="1.0" encoding="utf-8"?>

<rss xmlns:atom="http://www.w3.org/2005/Atom" version="2.0">
    <channel>
        <title>My blog</title>
        <link>http://127.0.0.1:8000/blog/</link>
        <description>New posts of my blog.</description>
        <atom:link href="http://127.0.0.1:8000/blog/feed/" rel="self"/>
        <language>en-us</language>
        <lastBuildDate>Sun, 20 Sep 2015 20:40:55 -0000</lastBuildDate>
        <item>
            <title>Who was Django Reinhardt?</title>
            <link>http://127.0.0.1:8000/blog/2015/09/20/who-was-django-reinhardt/</link>
            <description>The Django web framework was named after the amazing jazz guitarist Django
Reinhardt.</description>
            <guid>http://127.0.0.1:8000/blog/2015/09/20/who-was-django-reinhardt/</guid>
        </item>
        ...
    </channel>
</rss>
```

如果你在一个 RSS 客户端中打开相同的 URL，通过用户友好的接口你能看到你的 feed。
最后一步是在 blog 的侧边栏（sitebar）添加一个 feed 订阅（subscription）链接。打开 *blog/base.html* 模板（template），在侧边栏（sitebar）的 *div* 中的帖子总数下添加如下代码：

```
<p><a href="{% url "blog:post_feed" %}">Subscribe to my RSS feed</a></p>
```

现在，在浏览器中打开 http://127.0.0.1:8000/blog/ 看下侧边栏（sitebar）。这个新链接将会带你去 blog 的 feed：

django-3-7

使用 Solr 和 Haystack 添加一个搜索引擎

译者注:终于到了这一节，之前自己按照本节一步步操作的走下来添加了一个搜索引擎但是并没有达到像本节中所说的效果，期间还出现了很多莫名其妙的错误，可以说添加的搜索引擎功能失败了，而且本节中的提到的工具版本过低，官网最新版本的操作步骤已经和本节中描述的不一样，本节的翻译就怕会带来更多的错误，大家有需要尽量去阅读下英文原版。另外，一些单词没有好的翻译我还是保留原文。

现在，我们要为我们的 blog 添加搜索功能。Django ORM 允许你使用 *icontains* 过滤器（filter）执行对大小写不敏感的查询操作。举个例子，你可以使用以下的查询方式来找到内容中包含单词 *framework* 的帖子：

```
Post.objects.filter(body__icontains='framework')
```

但是，如果你想要更加强大的搜索功能，你需要使用合适的搜索引擎。我们准备使用 *Solr* 结合 Django 的方式为我们的 blog 构建一个搜索引擎。*Solr* 是一个非常流行的开源搜索平台，它提供了全文检索（full text search），term boosting，hit highlighting，分类搜索（faceted search）以及动态聚集（clustering），还有其他更多的高级搜索特性。
为了在我们的项目中集成 *Solr*，我们需要使用 *Haystack*。*Haystack* 是一个为多个搜索引擎提供抽象层工作的 Django 应用。它提供了一个非常类似于 Django 查询集（QuerySets）的简单的 API。让我们开始安装和配置 *Solr* 和 *Haystack*。

安装 Solr

你需要 1.7 或更高的 Java 运行环境来安装 Solr。你可以在终端中输入 `java -version` 来检查你的 java 版本。下方的输出和你的输出可能有所出入，但是你必须保证安装的版本至少也要是 1.7 的：

```
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b15)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)
```

如果你没有安装过 Java 或者版本低于要求版本，你可以在 http://www.oracle.com/technetwork/java/javase/downloads/index.html 下载 Java。
检查 Java 版本后，从 http://archive.apache.org/dist/lucene/solr/ 下载 **4.10.4** 版本的 Solr（译者注：请一定要下载这个版本，不然下面的操作无法进行！！）。解压下载的文件进入 *Solr* 安装路径下的 *example* 目录（也就是,`cd solr-4.10.4/example/`）。这个目录下包含了一个准备使用的 *Solr* 配置。在这个目录下，通过以下命令，用内置的 Jetty web 服务运行 *Solr*：

```
java -jar start.jar
```

打开你的浏览器，进入 URL:http://127.0.0.1:8983/solr/ 。你会看到如下图所示：

django-3-8

以上是 *Solr* 的管理控制台。这个控制台向你展示了数据统计，允许你管理你的搜索后端，检测索引数据，并且执行查询操作。

创建一个 Solr core

Solr 允许你隔离每一个 core 实例。每个 Solr **core** 是一个 **Lucene 全文搜索引擎**实例，连同一个 Solr 配置，一个数据架构（schema），以及其他要求的配置才能使用。Slor 允许你动态地创建和管理 cores。参考例子配置中包含了一个叫 *collection1* 的 core。如果你点击 **Core Admin** 菜单栏， 你可以看到这个 core 的信息，如下图所示：

django-3-9

我们要为我们的 blog 应用创建一个 core。首先，我们需要为我们的 core 创建文件结构。进入 *solr-4.10.4/example/*目录下，创建一个新的目录命名为 *blog*。然后在 *blog* 目录下创建空文件和目录，如下所示：

```
blog/
    data/
    conf/
    protwords.txt
    schema.xml
    solrconfig.xml
    stopwords.txt
    synonyms.txt
    lang/
    stopwords_en.txt
```

在 *solrconfig.xml* 文件中添加如下 XML 代码：

```
<?xml version="1.0" encoding="utf-8" ?>
<config>
    <luceneMatchVersion>LUCENE_36</luceneMatchVersion>
    <requestHandler name="/select" class="solr.StandardRequestHandler" default="true" />
    <requestHandler name="/update" class="solr.UpdateRequestHandler" />
    <requestHandler name="/admin" class="solr.admin.AdminHandlers" />
    <requestHandler name="/admin/ping" class="solr.PingRequestHandler">
        <lst name="invariants">
            <str name="qt">search</str>
            <str name="q">*:*</str>
        </lst>
    </requestHandler>
</config>
```

你还可以从本章的示例代码中拷贝该文件。这是一个最小的 Solr 配置。编辑 *schema.xml* 文件，加入如下 XML 代码：

```
<?xml version="1.0" ?>
<schema name="default" version="1.5">
</schema>
```

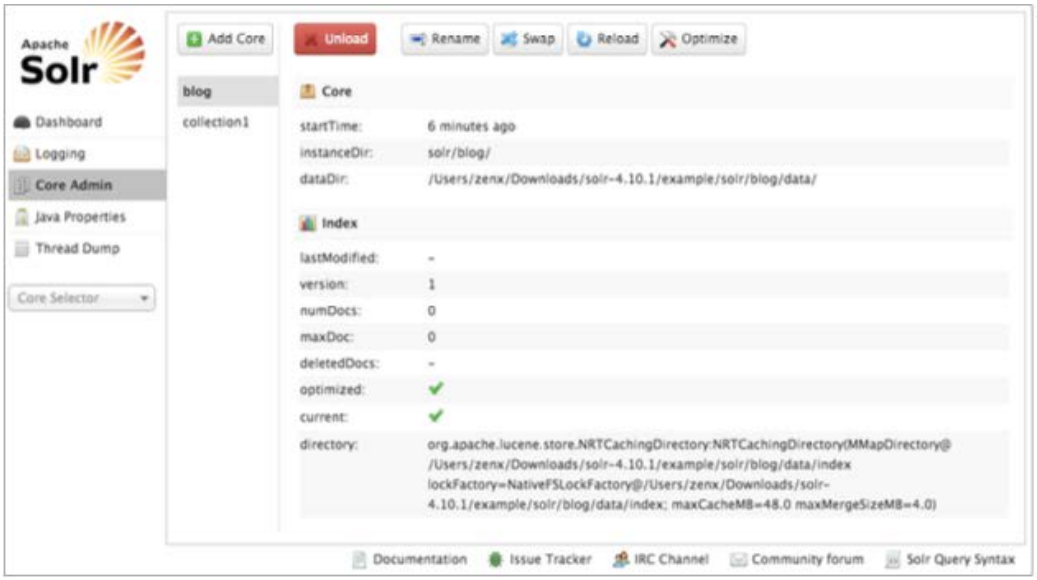
这是一个空的**架构（schema）**。这个架构（schema）定义了在搜索引擎中将被索引到的数据的字段和字段类型。之后我们要使用一个自定义的架构（schema）。
现在，点击 **Core Admin** 菜单栏再点击 **Add core** 按钮。你会看到如下图所示的一张表单，允许你为你的 core 指定信息：

django-3-10

用以下数据填写表单：

- name: blog
- instanceDir: blog
- dataDir: data
- config: solrconfig.xml
- schema: schema.xml

name 字段是你想给这个 core 起的名字。*instanceDir* 字段是你的 core 的目录。*dataDir* 是索引数据将要存放的目录，它位于 *instanceDir* 目录下面。*config* 字段是你的 *Solr* XML 配置文件名。*schema* 字段是你的 *Solr* XML 数据架构（schema)文件名。
现在，点击 **Add Core** 按钮。如果你看到下图所示，说明你新的 core 已经成功的添加到 Solr 中：



django-3-11

安装 Haystack

为了在 Django 中使用 Solr，我们还需要 Haystack。使用下面的命令，通过 pip 渠道安装 Haystack：

```
pip install django-haystack==2.4.0
```

Haystack 能和一些搜索引擎后台交互。要使用 Solr 后端，你还需要安装 *pysolr* 模块。运行如下命令安装它：

```
pip install pysolr==3.3.2
```

在 *django-haystack* 和 *pysolr* 完成安装后，你还需要在你的项目中激活 *Haystack*。打开 *settings.py* 文件，在 *INSTALLED_APPS* 设置中添加 *haystack*，如下所示：

```
INSTALLED_APPS = (
    # ...
    haystack',
)
```

你还需要为 haystack 定义搜索引擎后端。为此你要添加一个 *HAYSTACK_CONNECTIONS* 设置。在 *settings.py* 文件中添加如下内容：

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://127.0.0.1:8983/solr/blog'
    },
}
```

要注意 URL 要指向我们的 blog core。到此为止，Haystack 已经安装好并且已经为使用 Solr 做好了准备。

创建索引（indexex）

现在，我们必须将我们想要存储在搜索引擎中的模型进行注册。Haystack 的惯例是在你的应用中创建一个 *search_indexes.py* 文件，然后在该文件中注册你的模型（models）。在你的 blog 应用目录下创建一个新的文件命名为 *search_indexes.py*，添加如下代码：

```
from haystack import indexes
from .models import Post

class PostIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    publish = indexes.DateTimeField(model_attr='publish')

    def get_model(self):
        return Post

    def index_queryset(self, using=None):
        return self.get_model().published.all()
```

这是一个 *Post* 模型（model）的自定义 *SearchIndex*。通过这个索引（index），我们告诉 Haystack 这个模型（model）中的哪些数据必须被搜索引擎编入索引。这个索引（index）是通过继承 *indexes.SearchIndex* 和 *indexes.Indexable* 构建的。每一个 *SearchIndex* 都需要它的其中一个字段拥有 `document=True`。按照惯例，这个字段命名为 *text*。这个字段是一个主要的搜索字段。通过使用 `use_template=True`，我们告诉 Haystack 这个字段将会被渲染成一个数据模板（template）来构建 document，它会被搜索引擎编入索引（index）。*publish* 字段是一个日期字段也会被编入索引。我们通过 *model_attr*

参数来表明这个字段对应 *Post* 模型（model）的 *publish* 字段。这个字段将用 被索引的 *Post* 对象的 *publish* 字段的内容 索引。
额外的字段，像这个为搜索提供额外的过滤器（filters），是非常有用的。*get_model()*方法必须返回将储存在这个索引中的 documents 的模型（model）。*index_queryset()*方法返回将会被编入索引的对象的查询集（QuerySet）。请注意，我们只包含了发布状态的帖子。
现在，在 blog 应用的模板（templates）目录下创建目录和文件 *search/indexes/blog/post_text.txt*，然后添加如下代码：

```
{{ object.title }}
{{ object.tags.all|join:", " }}
{{ object.body }}
```

这是 document 模板（template）的默认路径，是给索引中的 *text* 字段使用的。Haystack 使用应用名和模型（model）名来动态构建这个路径。每一次我们要索引一个对象，Haystack 都会基于这个模板（template）构建一个 document，之后在 Solr 的搜索引擎中索引这个 document。
现在，我们已经有了一个自定义的搜索索引（index），我们需要创建合适的 Solr 架构（schema）。Solr 的配置基于 XML，所以我们必须为我们即将索引（index）的数据生成一个 XML 架构（schema）。非常幸运，haystack 提供了一个基于我们的搜索索引（indexes），动态生成架构（schema）的方法。打开终端，运行以下命令：

```
python manage.py build_solr_schema
```

你会看到一个 XML 输出。如果你看下生成的 XML 代码的底部，你会看到 Haystack 自动为你的 *PostIndex* 生成了字段：

```
<field name="text" type="text_en" indexed="true" stored="true" multiValued="false" />
<field name="publish" type="date" indexed="true" stored="true" multiValued="false" />
```

从`<?xml version="1.0"? >`开始拷贝所有输出的 XML 内容直到最后的标签（tag）`</schema>`，需要包含所有的标签（tags）。
这个 XML 架构（schema）是用来将数据做索引（index）到 Solr 中。粘贴这个新的架构（schema）到你的 Solr 安装路径下的 *example* 目录下的 *blog/conf/schema.xml* 文件中。*schema.xml* 文件也被包含在本章的示例代码中，所以你可以直接从示例代码中复制出来使用。
在你的浏览器中打开 http://127.0.0.1:8983/solr/ 然后点击 **Core Admin** 菜单栏，再点击 **blog** core，然后再点击 **Reload** 按钮：

django-3-12

我们重新载入这个 core 确保 *schema.xml* 的改变生效。当 core 重新载入完毕，新的架构（schema）准备好索引（index）新数据。

索引数据（Indexing data）

让我们 blog 中的帖子编辑索引（index）到 Solr 中。打开终端，执行以下命令：

```
python manage.py rebuild_index
```

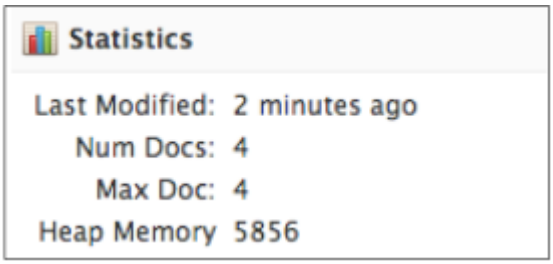
你会看到如下警告：

```
WARNING: This will irreparably remove EVERYTHING from your search index in connection 'default'. Your choices
after this are to restore from backups or rebuild via the ‘rebuild_index’ command.
Are you sure you wish to continue? [y/N]
```

输入 y。Haystack 将会清理搜索索引并且插入所有的发布状态的 blog 帖子。你会看到如下输出：

```
Removing all documents from your index because you said so. All documents removed. Indexing 4 posts
```

在浏览器中打开 http://127.0.0.1:8983/solr/#/blog 。在*Statistics* 下方，你会看到被编入索引（indexed）documents 的数量，如下所示：



django-3-13

现在，在浏览器中打开 http://127.0.0.1:8983/solr/#/blog/query 。这是一个 Solr 提供的查询接口。点击 *Execute query* 按钮。默认的查询会请求你的 core 中所有被编入索引（indexde）的 documents。你会看到一串带有这个查询结果的 *JSON* 输出。输出的 documents 如下所示：

```
{
    "id": "blog.post.1",
    "text": "Who was Django Reinhardt?\njazz, music\nThe Django web framework was named after the amazing jazz guitarist Django Reinhardt.",
    "django_id": "1",
    "publish": "2015-09-20T12:49:52Z",
    "django_ct": "blog.post"
},
```

这是每个帖子在搜索索引（index）中存储的数据。*text* 字段包含了标题，通过逗号分隔的标签（tags），还有帖子的内容，这个字段是在我们之前定义的模板（template）上构建的。
你已经使用过 `python manage.py rebuild_index` 来删除索引（index）中的所有信息然后再次对 documents 进行索引（index）。为了不删除所有对象而更新你的索引（index），你可以使用 `python manage.py update_index`。另外，你可以使用参数`--age=<num_hours>`来更新少量的对象。为了保证你的 Solr 索引更新，你可以为这个操作设置一个定时任务（Cron job）。

创建一个搜索视图（view）

现在，我们要开始创建一个自定义视图（view）来允许我们的用户搜索帖子。首先，我们需要一个搜索表单（form）。编辑 blog 应用下的 *forms.py* 文件，加入以下表单：

```
class SearchForm(forms.Form):
    query = forms.CharField()
```

我们会使用 *query* 字段来让用户引入搜索条件（terms）。编辑 blog 应用下的 *views.py* 文件，加入以下代码：

```
from .forms import EmailPostForm, CommentForm, SearchForm
from haystack.query import SearchQuerySet

def post_search(request):
    form = SearchForm()
    if 'query' in request.GET:
        form = SearchForm(request.GET)
        if form.is_valid():
```

```
            cd = form.cleaned_data
            results = SearchQuerySet().models(Post).filter(content=cd['query']).load_all()
            # count total results
            total_results = results.count()

    return render(request, 'blog/post/search.html',
            {'form': form,
            'cd': cd,
            'results': results,
            'total_results': total_results})
```

在这个视图（view）中，首先我们实例化了我们刚才创建的 *SearchForm*.我们准备使用 *GET* 方法来提交这个表单（form），这样可以使 URL 结果中包含查询的参数。假设这个表单（form）已经被提交，我们将在 *request.GET* 字典中查找 *query* 参数。当表单（form）被提交后，我们通过提交的 *GET* 数据来实例化它，然后我们要检查传入的数据是否有效（valid）。如果这个表单是有效（valid）的，我们使用 *SearchQuerySet* 为所有被编入索引的并且主要内容中包含给予的查询内容的 *Post* 对象来执行一次搜索。*load_all()*方法会立刻加载所有在数据库中有关联的 *Post* 对象。通过这个方法，我们使用数据库对象填充搜索结果，避免当遍历结果访问对象数据时，每个对象访问数据库 （译者注：这话不太好翻译，看不懂的话可以看下原文）。最后，我们存储 *total_results* 变量中结果的总数并传递本地的变量作为上下文(context)来渲染一个模板（template）。
搜索视图（view）已经准备好了。我们还需要创建一个模板（template）来展示表单（form）和用户执行搜索后返回的结果。在 *templates/blog/post/*目录下创建一个新的文件命名为 *search.html*，添加如下代码：

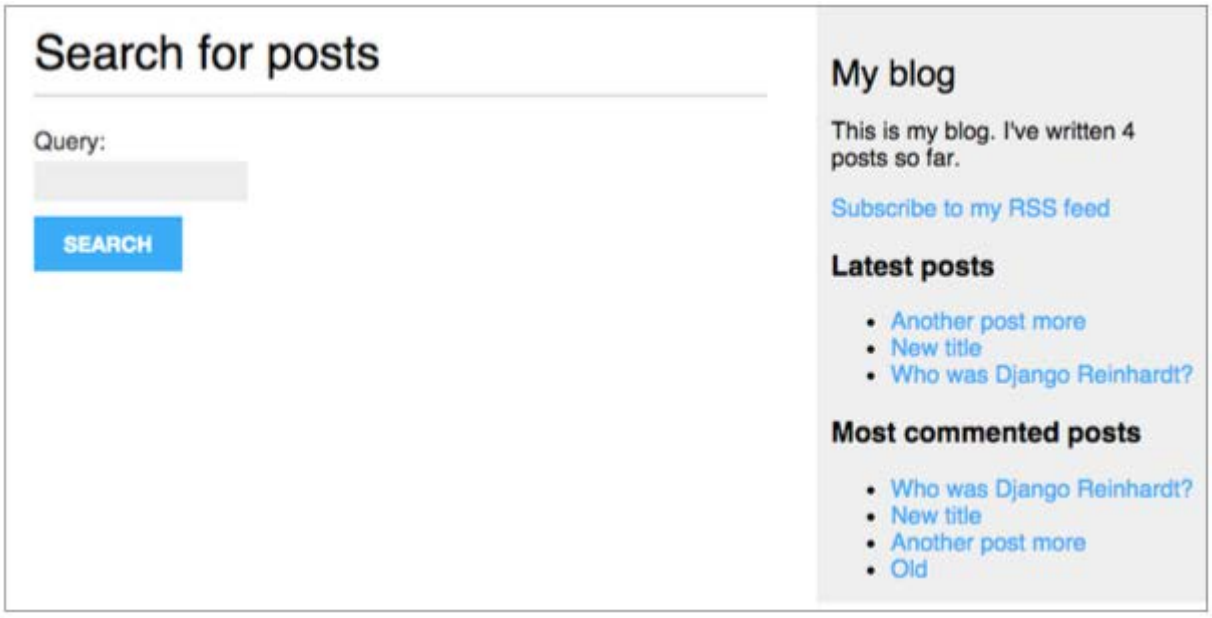
```
{% extends "blog/base.html" %}
{% block title %}Search{% endblock %}
{% block content %}
    {% if "query" in request.GET %}
        <h1>Posts containing "{{ cd.query }}"</h1>
        <h3>Found {{ total_results }} result{{ total_results|pluralize }}</h3>
        {% for result in results %}
            {% with post=result.object %}
                <h4><a href="{{ post.get_absolute_url }}">{{ post.title }}</a></h4>
                {{ post.body|truncatewords:5 }}
            {% endwith %}
            {% empty %}
                  <p>There are no results for your query.</p>
        {% endfor %}
        <p><a href="{% url "blog:post_search" %}">Search again</a></p>
    {% else %}
        <h1>Search for posts</h1>
        <form action="." method="get">
            {{ form.as_p }}
            <input type="submit" value="Search">
        </form>
    {% endif %}
{% endblock %}
```

就像在搜索视图（view）中，我们做了区分如果这个表单（form）是基于 *query* 参数存在的情况下提交。在这个 post 提交前，我们展示了这个表单和一个提交按钮。当这个 post 被提交，我们就展示查询

的操作结果，包含返回结果的总数和结果列表。每一个结果都是 Solr 返回和 Haystack 封装处理后的 document。我们需要使用 *result.object* 来获取真实的和这个结果相关联的 *Post* 对象。
最后，编辑 blog 应用下的 *urls.py* 文件，添加以下 URL 模式：

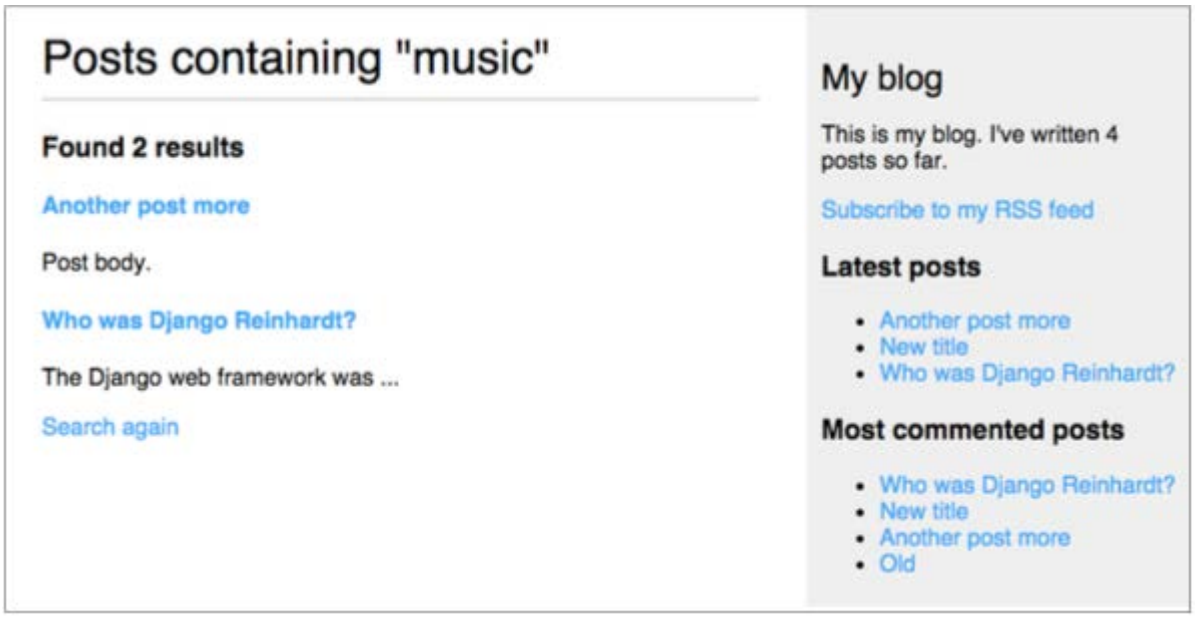
```
url(r'^search/$', views.post_search, name='post_search'),
```

现在，在浏览器中打开 http://127.0.0.1:8000/blog/search/。你会看到如下图所示的搜索表单（form）：



django-3-14

现在，输入一个查询条件然后点击 **Search** 按钮。你会看到查询搜索的结果，如下图所示：



django-3-15

如今，在你的项目中你已经构建了一个强大的搜索引擎，但这仅仅只是开始，还有更多丰富的功能可以通过 *Solr* 和 Haystack 做到。Haystack 包含视图（views），表单（forms）以及搜索引擎的高级功能。你可以在 http://django-haystack.readthedocs.org/en/latest/页面上阅读 Haystack 文档。
通过自定义架构（schema），Solr 搜索引擎可以适配各种需求。你可以结合分析仪，断词，和令牌过滤器，这些是在索引或搜索的时间执行，为你的网站内容提供更准确的搜索。 你可以在 https://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters 看到所有的可能性。

总结

在这一章中，你学习了如何创建自定义的 Django 模板标签（template tags）和过滤器（filters），提供给模板（template）实现一些自定义的功能。你还为搜索引擎创建了一个站点地图（sitemap），爬

取你的站点以及一个 RSS feed 给用户来订阅。你还通过在项目中集成 Slor 和 Haystack 为 blog 应用构建了一个搜索引擎。
在下一章中，你将会学习到通过使用 Django 认证框架，如何构建一个社交网站，创建自定义的用户画像，以及创建社交认证。

译者总结

终于将第三章也翻译完成了，隔了一个星期没翻译，相对于前两章，发现这章的翻译速度又加快了那么一点点，两天内完成翻译。本章中创建自己的模板标签和过滤器个人认为非常实用，我已经打算这段时间将手头上上线的几个项目都使用本章中提供的方法进行部分重构。本章最后部分的搜索引擎我倒是用不到，看官们可以也可以选择不看，毕竟书中提供的版本太老了。。。。。。前三章都是围绕博客应用展开（为什么大家都喜欢用博客应用做初始教程- -|||），下一章开始将开启新的项目应用，我们下周或下下周或下个月继续- -|||

第四章 创建一个社交网站

在上一章中，你学习了如何创建站点地图（sitemaps）和 feeds，你还为你的 blog 应用创建了一个搜索引擎。在本章中，你将开发一个社交应用。你会为用户创建一些功能，例如：登录，登出，编辑，以及重置他们的密码。你会学习如何为你的用户创建一个定制的 profile，你还会为你的站点添加社交认证。
本章将会覆盖一下几点：

- 使用认证（authentication）框架
- 创建用户注册视图（views）
- 通过一个定制的 profile 模型（model）扩展 *User* 模型（model）
- 使用 *python-social-auth* 添加社交认证

让我们开始创建我们的新项目吧。

创建一个社交网站项目

我们要创建一个社交应用允许用户分享他们在网上找到的图片。我们需要为这个项目构建以下元素：

- 一个用来给用户注册，登录，编辑他们的 profile，以及改变或重置密码的认证（authentication）系统
- 一个允许用户来关注其他人的关注系统（这里原文是 follow，‘跟随’，感觉用‘关注’更加适合点）
- 为用户从其他任何网站分享过来的图片进行展示和打上书签
- 每个用户都有一个活动流允许他们看到所关注的人上传的内容

本章主要讲述第一点。

开始你的社交网站项目

打开终端使用如下命令行为你的项目创建一个虚拟环境并且激活它：

mkdir evn
virtualenv evn/bookmarks
source env/bookmarks/bin/activate
shell 提示将会展示你激活的虚拟环境，如下所示：

```
(bookmarks)laptop:~ zenx$
```

通过以下命令在你的虚拟环境中安装 Django：

```
pip install Django==1.8.6
```

运行以下命令来创建一个新项目：

```
django-admin startproject bookmarks
```

在创建好一个初始的项目结构以后，使用以下命令进入你的项目目录并且创建一个新的应用命名为 *account*:

```
cd bookmarks/
django-admin startapp account
```

请记住在你的项目中激活一个新应用需要在 *settings.py* 文件中的 *INSTALLED_APPS* 设置中添加它。将新应用的名字添加在 *INSTALLED_APPS* 列中的所有已安装应用的最前面，如下所示：

```
INSTALLED_APPS = (
    'account',
    # ...
)
```

运行下一条命令为 *INSTALLED_APPS* 中默认包含的应用模型（models）同步到数据库中：

```
python manage.py migrate
```

我们将要使用认证（authentication）框架来构建一个认证系统到我们的项目中。

使用 Django 认证（authentication）框架

Django 拥有一个内置的认证(authentication)框架用来操作用户认证(authentication)，会话(sessions)，权限（permissions）以及用户组。这个认证（authentication）系统包含了一些普通用户的操作视图（views），例如：登录，登出，修改密码以及重置密码。

这个认证（authentication）框架位于 *django.contrib.auth*，被其他 Django 的 *contrib* 包调用。请记住你在*第一章 创建一个 Blog 应用*中使用过这个认证（authentication）框架并用来为你的 blog 应用创建了一个超级用户来使用管理站点。

当你使用 *startproject* 命令创建一个新的 Django 项目，认证（authentication）框架已经在你的项目设置中默认包含。它是由 *django.contrib.auth* 应用和你的项目设置中的 *MIDDLEWARE_CLASSES* 中的两个中间件类组成，如下：

- AuthenticationMiddleware:使用会话（sessions）将用户和请求（requests）进行关联
- SessionMiddleware:通过请求（requests）操作当前会话（sessions）

中间件就是一个在请求和响应阶段带有全局执行方法的类。你会在本书中的很多场景中使用到中间件。你将会在*第十三章 Going Live* 中学习如何创建一个定制的中间件（译者注：啥时候能翻译到啊）。

这个认证（authentication）系统还包含了以下模型（models）：

- User：一个包含了基础字段的用户模型（model）；这个模型（model）的主要字段有：username，password, email, first_name, last_name, is_active。
- Group：一个组模型（model）用来分类用户
- Permission：执行特定操作的标识

这个框架还包含默认的认证（authentication）视图（views）和表单（forms），我们之后会用到。

创建一个 log-in 视图（view）

我们将要开始使用 Django 认证(authentication)框架来允许用户登录我们的网站。我们的视图(view)需要执行以下操作来登录用户：

1. 通过提交的表单（form）获取 username 和 password
2. 通过存储在数据库中的数据对用户进行认证
3. 检查用户是否可用
4. 登录用户到网站中并且开始一个认证（authentication）会话（session）

首先，我们要创建一个登录表单（form）。在你的 account 应用目录下创建一个新的 *forms.py* 文件，添加如下代码：

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

这个表单(form)被用来通过数据库认证用户。请注意，我们使用 *PasswordInput* 控件来渲染 HTML`input`元素，包含 `type="password"`属性。编辑你的 *account* 应用中的 *views.py* 文件，添加如下代码：

```
from django.http import HttpResponse
from django.shortcuts import render
from django.contrib.auth import authenticate, login
from .forms import LoginForm


def user_login(request):
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            user = authenticate(username=cd['username'],
                                password=cd['password'])
            if user is not None:
                if user.is_active:
                    login(request, user)
                    return HttpResponse('Authenticated successfully')
                else:
                    return HttpResponse('Disabled account')
            else:
                return HttpResponse('Invalid login')
    else:
        form = LoginForm()
    return render(request, 'account/login.html', {'form': form})
```

以上就是我们在视图（view）中所作的基本登录操作：当 *user_login* 被一个 GET 请求（request）调用，我们实例化一个新的登录表单（form）并通过 `form = LoginForm()`在模板（template）中展示它。当用户通过 POST 方法提交表单（form）时，我们执行以下操作：

1、通过使用 `form = LoginForm(request.POST)`使用提交的数据实例化表单（form）

2、检查这个表单是否有效。如果无效，我们在模板（template）中展示表单错误信息（举个例子，比如用户没有填写其中一个字段就进行提交）

3、如果提交的数据是有效的，我们使用 `authenticate()`方法通过数据库对这个用户进行认证（authentication）。这个方法带入一个 *username* 和一个 *password*，如果这个用户成功的进行了认证则返回一个用户对象，否则是 *None*。如果用户没有被认证通过，我们返回一个 *HttpResponse* 展示一条消息。

4、如果这个用户认证（authentication）成功，我们使用 `is_active` 属性来检查用户是否可用。这是一个 Django 的 *User* 模型（model)属性。如果这个用户不可用，我们返回一个 *HttpResponse* 展示信息。
5、如果用户可用，我们登录这个用户到网站中。我们通过调用 `login()` 方法集合用户到会话（session）中然后返回一条成功消息。
请注意 *authentication* 和 *login* 中的不同点：`authenticate()` 检查用户认证信息，如果用户是正确的则返回一个用户对象；`login()` 将用户设置到当前的会话（session）中。
现在，你需要为这个视图（view）创建一个 URL 模式。在你的 *account* 应用目录下创建一个新的 *urls.py* 文件，添加如下代码：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    # post views
    url(r'^login/$', views.user_login, name='login'),
]
```

编辑位于你的 *bookmarks* 项目目录下的 *urls.py* 文件，将 *account* 应用下的 URL 模式包含进去：

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

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^account/',include('account.urls')),
]
```

这个登录视图（view）现在已经可以通过 URL 进行访问。现在是时候为这个视图（view）创建一个模板。因为之前你没有这个项目的任何模板，你可以开始创建一个主模板（template）能够被登录模板（template）继承使用。在 *account* 应用目录中创建以下文件和结构：

```
    templates/
        account/
            login.html
        base.html
```

编辑 *base.html* 文件添加如下代码：

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
    <head>
        <title>{% block title %}{% endblock %}</title>
        <link href="{% static "css/base.css" %}" rel="stylesheet">
    </head>
    <body>
        <div id="header">
            <span class="logo">Bookmarks</span>
        </div>
        <div id="content">
            {% block content %}
            {% endblock %}
```

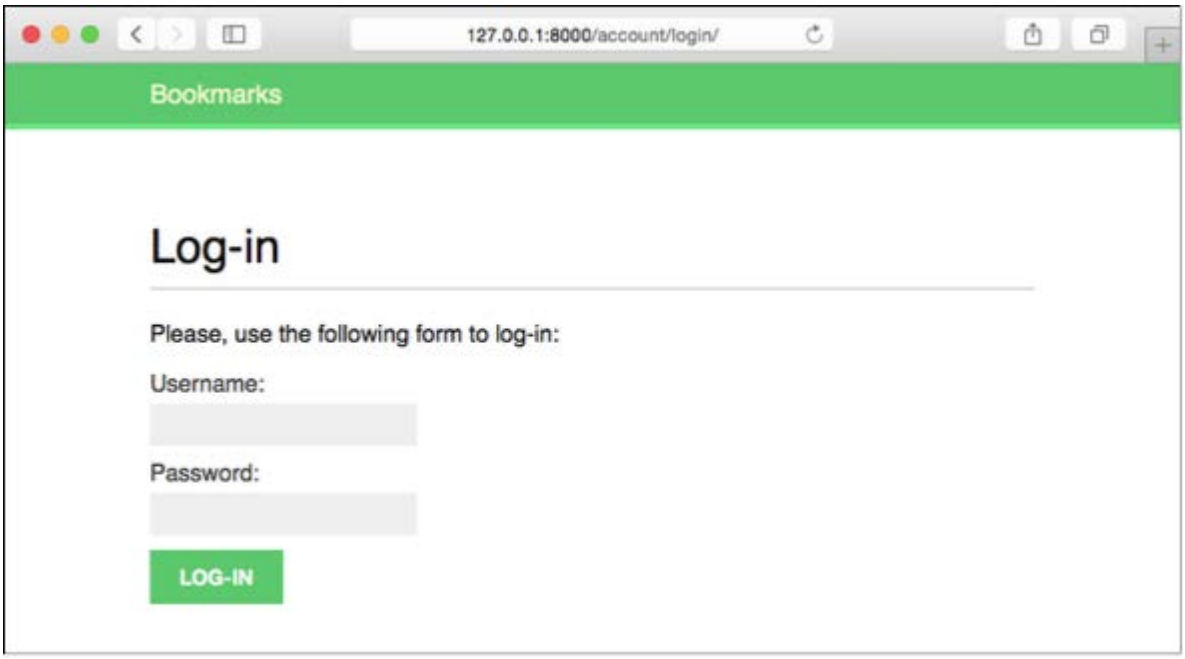
```
        </div>
    </body>
</html>
```

以上将是这个网站的基础模板（template）。就像我们在上一章项目中做过的一样，我们在这个主模板（template）中包含了 CSS 样式。你可以在本章的示例代码中找到这些静态文件。复制示例代码中的 *account* 应用下的 *static/*目录到你的项目中的相同位置，这样你就可以使用这些静态文件了。
基础模板（template）定义了一个 *title* 和一个 *content* 区块可以让被继承的子模板（template）填充内容。
让我们为我们的登录表单（form）创建模板（template）。打开 *account/login.html* 模板（template）添加如下代码：

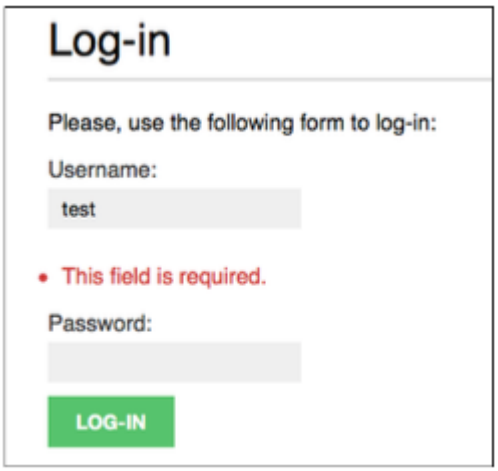
```
{% extends "base.html" %}
{% block title %}Log-in{% endblock %}
{% block content %}
    <h1>Log-in</h1>
    <p>Please, use the following form to log-in:</p>
    <form action="." method="post">
        {{ form.as_p }}
        {% csrf_token %}
        <p><input type="submit" value="Log-in"></p>
    </form>
{% endblock %}
```

这个模板（template）包含了视图（view）中实例化的表单（form）。因为我们的表单（form）将会通过 POST 方式进行提交，所以我们包含了`{% csrf_token %}`模板（template）标签(tag)用来通过 CSRF 保护。你已经在*第二章 使用高级特性扩展你的博客应用*中学习过 CSRF 保护。
目前还没有用户在你的数据库中。你首先需要创建一个超级用户用来登录管理站点来管理其他的用户。打开命令行执行 `python manage.py createsuperuser`。填写 username, e-mail 以及 password。之后通过命令 `python manage.py runserver` 运行开发环境，然后在你的浏览器中打开 http://127.0.0.1:8000/admin/ 。使用你刚才创建的 username 和 passowrd 来进入管理站点。你会看到 Django 管理站点包含 Django 认证（authentication）框架中的 *User* 和 *Group* 模型（models）。如下所示：

使用管理站点创建一个新的用户然后打开 http://127.0.0.1:8000/account/login/ 。你会看到被渲染过的模板（template），包含一个登录表单（form）：



现在，只填写一个字段保持另外一个字段为空进行表单（from）提交。在这例子中，你会看到这个表单（form）是无效的并且显示了一个错误信息：



如果你输入了一个不存在的用户或者一个错误的密码，你会得到一个 **Invalid login** 信息。
如果你输入了有效的认证信息，你会得到一个 **Authenticated successfully** 消息：

使用 Django 认证（authentication）视图（views）

Django 在认证（authentication）框架中包含了一些开箱即用的表单（forms）和视图（views）。你之前创建的登录视图（view）是一个非常好的练习用来理解 Django 中的用户认证（authentication）过程。无论如何，你可以在大部分的案例中使用默认的 Django 认证（authentication）视图（views）。
Django 提供以下视图（views）来处理认证（authentication）：

- login：操作表单（form）中的登录然后登录一个用户
- logout：登出一个用户
- logout_then_login：登出一个用户然后重定向这个用户到登录页面

Django 提供以下视图（views）来操作密码修改：

- password_change:操作一个表单（form）来修改用户密码
- password_change_done:当用户成功修改他的密码后提供一个成功提示页面

Django 还包含了以下视图（views）允许用户重置他们的密码：

- password_reset:允许用户重置他的密码。它会生成一条带有一个 token 的一次性使用链接然后发送到用户的邮箱中。
- password_reset_done:告知用户已经发送了一封可以用来重置密码的邮件到他的邮箱中。
- password_reset_complete:当用户重置完成他的密码后提供一个成功提示页面。

当你创建一个带有用户账号的网站时，以上的视图（views）可以帮你节省很多时间。你可以覆盖这些视图（views）使用的默认值，例如需要渲染的模板位置或者视图（view）需要使用到的表单（form）。你可以通过访问 https://docs.djangoproject.com/en/1.8/topics/auth/default/#module-django.contrib.auth.views 获取更多内置的认证（authentication）视图（views）信息。

登录和登出视图（views）

编辑你的 *account* 应用下的 *urls.py* 文件，如下所示：

```
from django.conf.urls import url
from . import views
urlpatterns = [
    # previous login view
    # url(r'^login/$', views.user_login, name='login'),
    # login / logout urls
    url(r'^login/$',
        'django.contrib.auth.views.login',
        name='login'),
    url(r'^logout/$',
        'django.contrib.auth.views.logout',
        name='logout'),
    url(r'^logout-then-login/$',
        'django.contrib.auth.views.logout_then_login',
        name='logout_then_login'),
]
```

我们将之前创建的 *user_login* 视图（view）URL 模式进行注释，然后使用 Django 认证（authentication）框架提供的 *login* 视图（view）。

【译者注】如果使用 Django 1.10 以上版本，urls.py 需要改写为以下方式(参见源书提供的源代码):

```
from django.conf.urls import url
from django.contrib.auth.views import login
# With django 1.10 I need to pass the callable instead of
# url(r'^login/$', 'django.contrib.auth.views.login', name='login')

from django.contrib.auth.views import logout
from django.contrib.auth.views import logout_then_login
from django.contrib.auth.views import password_change
from django.contrib.auth.views import password_change_done
from django.contrib.auth.views import password_reset
from django.contrib.auth.views import password_reset_done
from django.contrib.auth.views import password_reset_confirm
from django.contrib.auth.views import password_reset_complete
from . import views

urlpatterns = [
    # post views
```

```
    # url(r'^login/$', views.user_login, name='login'),

    # login logout
    url(r'^login/$', login, name='login'),
    url(r'^logout/$', logout, name='logout'),
    url(r'^logout-then-login/$', logout_then_login, name='logout_then_login'),
    # change password
    url(r'^password-change/$', password_change, name='password_change'),
    url(r'^password-change/done/$', password_change_done, name='password_change_done'),
    # reset password
    ## restore password urls
    url(r'^password-reset/$',
        password_reset,
        name='password_reset'),
    url(r'^password-reset/done/$',
        password_reset_done,
        name='password_reset_done'),
    url(r'^password-reset/confirm/(?P<uidb64>[-\w]+)/(?P<token>[-\w]+)/$',
        password_reset_confirm,
        name='password_reset_confirm'),
    url(r'^password-reset/complete/$',
        password_reset_complete,
        name='password_reset_complete'),
    ]
```

在你的 account 应用中的 template 目录下创建一个新的目录命名为 *registration*。这个路径是 Django 认证（authentication）视图（view）期望你的认证（authentication）模块（template）默认的存放路径。在这个新目录中创建一个新的文件，命名为 *login.html*，然后添加如下代码：

```
{% extends "base.html" %}
{% block title %}Log-in{% endblock %}
{% block content %}
  <h1>Log-in</h1>
  {% if form.errors %}
    <p>
      Your username and password didn't match.
      Please try again.
    </p>
  {% else %}
    <p>Please, use the following form to log-in:</p>
  {% endif %}
  <div class="login-form">
    <form action="{% url 'login' %}" method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <input type="hidden" name="next" value="{{ next }}" />
      <p><input type="submit" value="Log-in"></p>
    </form>
  </div>
```

```
{% endblock %}
```

这个登录模板(template)和我们之前创建的基本类似。Django 默认使用位于 *django.contrib.auth.forms* 中的 *AuthenticationForm*。这个表单（form）会尝试对用户进行认证，如果登录不成功就会抛出一个验证错误。在这个例子中，如果用户提供了错误的认证信息，我们可以在模板（template）中使用`{% if form.errors %}`来找到错误。请注意，我们添加了一个隐藏的 HTML`<input>`元素来提交叫做 *next* 的变量值。当你在请求（request）中传递一个 *next* 参数（举个例子：http://127.0.0.1:8000/account/login/?next=/account/），这个变量是登录视图(view)首个设置的参数。*next* 参数必须是一个 URL。当这个参数被给予的时候，Django 登录视图（view）将会在用户登录完成后重定向到给予的 URL。

现在，在 *registrtion* 模板（template）目录下创建一个 *logged_out.html* 模板（template）添加如下代码：

```
{% extends "base.html" %}
{% block title %}Logged out{% endblock %}
{% block content %}
  <h1>Logged out</h1>
  <p>You have been successfully logged out. You can <a href="{% url "login" %}">log-in again</a>.</p>
{% endblock %}
```

Django 会在用户登出的时候展示这个模板（template）。

在添加了 URL 模式以及登录和登出视图（view）的模板之后，你的网站已经准备好让用户使用 Django 认证（authentication）视图进行登录。

请注意，在我们的地址配置中所包含的 *logtou_then_login* 视图（view）不需要任何模板（template），因为它执行了一个重定向到登录视图（view）。

现在，我们要创建一个新的视图（view）给用户，在他或她登录他们的账号后来显示一个 dashboard。

打开你的 *account* 应用中的 *views.py* 文件，添加以下代码：

```
from django.contrib.auth.decorators import login_required
@login_required
def dashboard(request):
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard'})
```

我们使用认证（authentication）框架的 *login_required* 装饰器（decorator）装饰我们的视图（view）。*login_required* 装饰器（decorator）会检查当前用户是否通过认证，如果用户通过认证，它会执行装饰的视图（view），如果用户没有通过认证，它会把用户重定向到带有一个名为 *next* 的 GET 参数的登录 URL，该 GET 参数保存的变量为用户当前尝试访问的页面 URL。通过这些动作，登录视图(view)会将登录成功的用户重定向到用户登录之前尝试访问过的 URL。请记住我们在登录模板（template）中的登录表单（form）中添加的隐藏`<input>`就是为了这个目的。

我们还定义了一个 *section* 变量。我们会使用该变量来跟踪用户在站点中正在查看的页面。多个视图（views）可能会对应相同的 section。这是一个简单的方法用来定义每个视图(view)对应的 section。

现在，你需要创建一个给 dashboard 视图（view）使用的模板（template）。在 *templates/account/* 目录下创建一个新文件命名为 *dashboard.html*。添加如下代码：

```
{% extends "base.html" %}
{% block title %}Dashboard{% endblock %}
{% block content %}
  <h1>Dashboard</h1>
  <p>Welcome to your dashboard.</p>
```

```
{% endblock %}
```

之后，为这个视图（view）在 *account* 应用中的 *urls.py* 文件中添加如下 URL 模式：

```
urlpatterns = [
    # ...
    url(r'^$', views.dashboard, name='dashboard'),
]
```

编辑你的项目的 *settings.py* 文件，添加如下代码：

```
from django.core.urlresolvers import reverse_lazy
LOGIN_REDIRECT_URL = reverse_lazy('dashboard')
LOGIN_URL = reverse_lazy('login')
LOGOUT_URL = reverse_lazy('logout')
```

这些设置的意义：

- LOGIN_REDIRECT_URL：告诉 Django 用户登录成功后如果 *contrib.auth.views.login* 视图（view）没有获取到 *next* 参数将会默认重定向到哪个 URL。
- LOGIN_URL：重定向用户登录的 URL（例如：使用 *login_required* 装饰器（decorator））。
- LOGOUT_URL：重定向用户登出的 URL。

我们使用 *reverse_lazy()*来通过它们的名字动态构建 URL。*reverse_lazy()*方法就像 *reverse()*所做的一样 reverses URLs，但是你可以通过使用这种方式在你项目的 URL 配置被读取之前进行 reverse URLs。

让我们总结下目前为止我们都做了哪些工作：

- 你在项目中添加了 Django 内置的认证（authentication）登录和登出视图（views）。
- 你为每一个视图（view)创建了定制的模板（templates），并且定义了一个简单的视图（view）让用户登录后进行重定向。
- 最后，你配置了你的 Django 设置使用默认的 URLs。

现在，我们要给我们的主模板（template）添加登录和登出链接将所有的一切都连接起来。

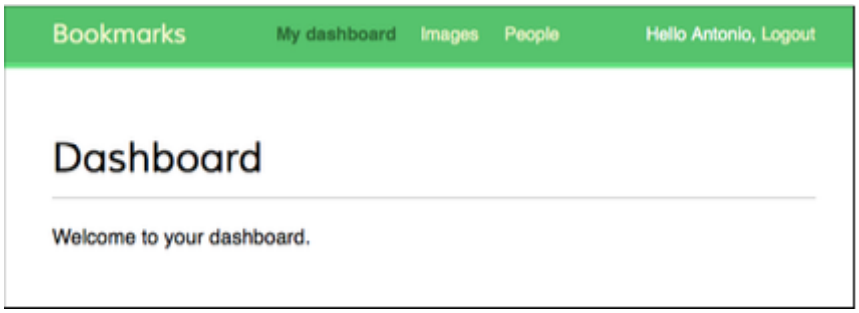
为了做到这点，我们必须确定无论用户是已登录状态还是没有登录的时候，都会显示适当的链接。通过认证（authentication）中间件当前的用户被设置在 HTTP 请求（request)对象中。你可以通过使用 `request.user` 访问用户信息。你会发现一个用户对象在请求（request)中，即便这个用户并没有认证通过。一个未认证的用户在请求（request）中被设置成一个 *AnonymousUser* 的实例。一个最好的方法来检查当前的用户是否通过认证是通过调用 `request.user.is_authenticated()`。

编辑你的 *base.html* 文件修改 ID 为 *header* 的`<div>`元素，如下所示：

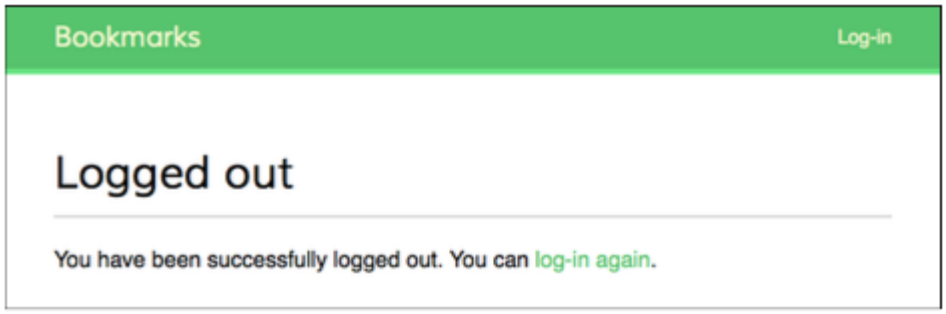
```
<div id="header">
<span class="logo">Bookmarks</span>
{% if request.user.is_authenticated %}
    <ul class="menu">
     <li {% if section == "dashboard" %}class="selected"{% endif %}>
       <a href="{% url "dashboard" %}">My dashboard</a>
     </li>
     <li {% if section == "images" %}class="selected"{% endif %}>
       <a href="#">Images</a>
     </li>
     <li {% if section == "people" %}class="selected"{% endif %}>
       <a href="#">People</a>
      </li>
     </ul>
    {% endif %}
```

```
    <span class="user">
      {% if request.user.is_authenticated %}
        Hello {{ request.user.first_name }},
        <a href="{% url "logout" %}">Logout</a>
      {% else %}
        <a href="{% url "login" %}">Log-in</a>
      {% endif %}
    </span>
</div>
```

就像你所看到的，我们只给通过认证（authentication）的用户显示站点菜单。我们还检查当前的 section 并通过使用 CSS 来给对应的`<li>`组件添加一个 *selected* 的 class 属性来使当前 section 在菜单中进行高亮显示。如果用户已经通过认证（authentication），我们还显示用户的名字（first_name)和一个登出的链接，否则，就是一个登出链接。
现在，在浏览器中打开 http://127.0.0.1:8000/account/login/ 。你会看到登录页面。输入可用的用户名和密码然后点击 **Log-in** 按钮。你会看到如下图所示：



你能看到通过 CSS 的作用 *My Dashboard* section 因为拥有一个 *selected* class 而高亮显示。因为当前用户已经通过了认证（authentication），所以用户的名字在右上角进行了显示。点击 **Logout** 链接。你会看到如下图所示：



在这个页面中，你能看到用户已经登出，然后，你无法看到当前网站的任何菜单。在右上角现在显示的是 **Log-in** 链接。
如果在你的登出页面中看到了 Django 管理站点的登出页面，检查项目 *settings.py* 中的 *INSTALLED_APPS*,确保 *django.contrib.admin* 在 *account* 应用的后面。每个模板（template）被定位在同样的相对路径时，Django 模板（template）读取器将会使用它找到的第一个应用中的模板（templates）。

修改密码视图（views）

我们还需要我们的用户在登录成功后可以进行修改密码。我们要集成 Django 认证（authentication）视图（views）来修改密码。打开 *account* 应用中的 *urls.py* 文件，添加如下 URL 模式：

```
# change password urls
url(r'^password-change/$',
   'django.contrib.auth.views.password_change',
   name='password_change'),
```

```
url(r'^password-change/done/$',
   'django.contrib.auth.views.password_change_done',
   name='password_change_done'),
```

password_change 视图（view）将会操作表单（form）进行修改密码，当用户成功的修改他的密码后 *password_change_done* 将会显示一条成功信息。让我们为每个视图（view）创建一个模板（template）。

在你的 *account* 应用 *templates/registration/*目录下添加一个新的文件命名为 *password_form.html*，在文件中添加如下代码：

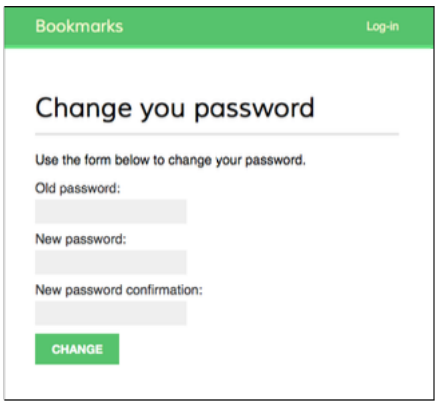
```
{% extends "base.html" %}
{% block title %}Change you password{% endblock %}
{% block content %}
  <h1>Change you password</h1>
  <p>Use the form below to change your password.</p>
  <form action="." method="post">
    {{ form.as_p }}
    <p><input type="submit" value="Change"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

这个模板（template）包含了修改密码的表单（form）。现在，在相同的目录下创建另一个文件，命名为 *password_change_done.html*，为它添加如下代码：

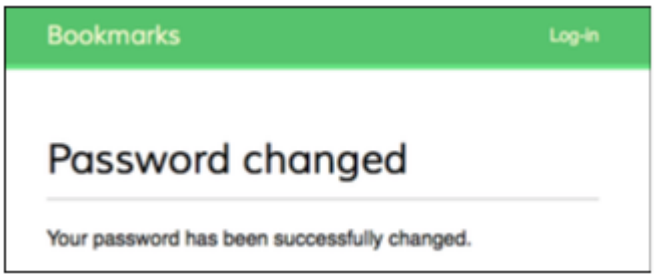
```
{% extends "base.html" %}
{% block title %}Password changed{% endblock %}
{% block content %}
  <h1>Password changed</h1>
  <p>Your password has been successfully changed.</p>
 {% endblock %}
```

这个模板（template）只包含显示一条成功的信息 当用户成功的修改他们的密码。

在浏览器中打开 http://127.0.0.1:8000/account/password-change/ 。如果你的用户没有登录，浏览器会重定向你到登录页面。在你成功认证（authentication）登录后，你会看到如下图所示的修改密码页面：



在表单（form）中填写你的旧密码和新密码，然后点击 **Change** 按钮。你会看到如下所示的成功页面：



登出再使用新的密码进行登录来验证每件事是否如预期一样工作。

重置密码视图（views）

在 *account* 应用 *urls.py* 文件中为密码重置添加如下 URL 模式：

```
# restore password urls
 url(r'^password-reset/$',
    'django.contrib.auth.views.password_reset',
    name='password_reset'),
 url(r'^password-reset/done/$',
     'django.contrib.auth.views.password_reset_done',
     name='password_reset_done'),
  url(r'^password-reset/confirm/(?P<uidb64>[-\w]+)/(?P<token>[-\w]+)/$',
       'django.contrib.auth.views.password_reset_confirm',
     name='password_reset_confirm'),
 url(r'^password-reset/complete/$',
     'django.contrib.auth.views.password_reset_complete',
      name='password_reset_complete'),
```

在你的 *account* 应用 *templates/registration/*目录下添加一个新的文件命名为 *password_reset_form.html*，为它添加如下代码：

```
{% extends "base.html" %}
{% block title %}Reset your password{% endblock %}
{% block content %}
  <h1>Forgotten your password?</h1>
  <p>Enter your e-mail address to obtain a new password.</p>
  <form action="." method="post">
    {{ form.as_p }}
    <p><input type="submit" value="Send e-mail"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

现在，在相同的目录下添加另一个文件命名为 *password_reset_email.html*，为它添加如下代码：

```
Someone asked for password reset for email {{ email }}. Follow the link below:
{{ protocol }}://{{ domain }}{% url "password_reset_confirm" uidb64=uid token=token %}
Your username, in case you've forgotten: {{ user.get_username }}
```

这个模板（template）会被用来渲染发送给用户的重置密码邮件。
在相同目录下添加另一个文件命名为 *password_reset_done.html*，为它添加如下代码：

```
{% extends "base.html" %}
{% block title %}Reset your password{% endblock %}
```

```
{% block content %}
  <h1>Reset your password</h1>
  <p>We've emailed you instructions for setting your password.</p>
  <p>If you don't receive an email, please make sure you've entered the address you registered with.</p>
{% endblock %}
```

再创建另一个模板（template）命名为 *password_reset_confirm.html*，为它添加如下代码：

```
{% extends "base.html" %}
{% block title %}Reset your password{% endblock %}
{% block content %}
  <h1>Reset your password</h1>
  {% if validlink %}
    <p>Please enter your new password twice:</p>
    <form action="." method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <p><input type="submit" value="Change my password" /></p>
    </form>
  {% else %}
    <p>The password reset link was invalid, possibly because it has already been used. Please request a new password reset.</p>
  {% endif %}
{% endblock %}
```

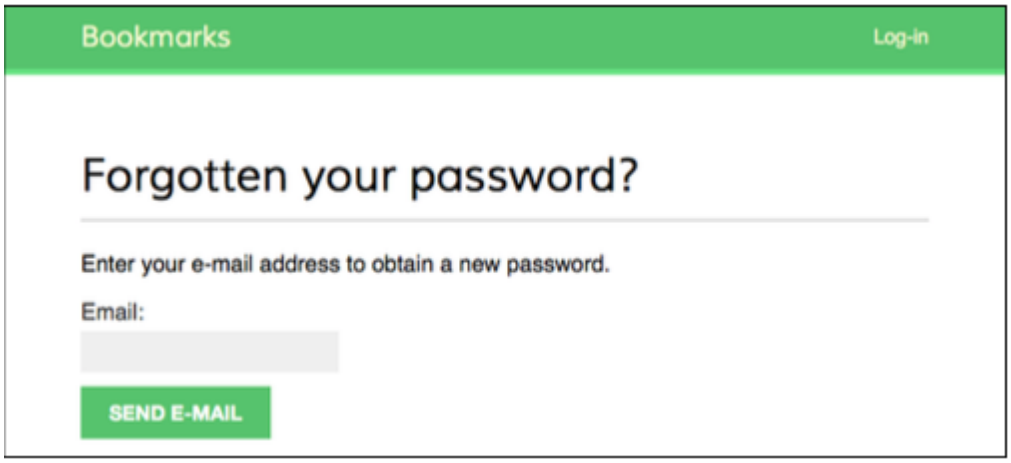
在以上模板中，我们将会检查重置链接是否有效。Django 重置密码视图（view）会设置这个变量然后将它带入这个模板(template)的上下文环境中。如果重置链接有效，我们展示用户密码重置表单(form)。创建另一个模板（template）命名为 *password_reset_complete.html*，为它添加如下代码：

```
{% extends "base.html" %}
{% block title %}Password reset{% endblock %}
{% block content %}
  <h1>Password set</h1>
  <p>Your password has been set. You can <a href="{% url "login" %}">log in now</a></p>
{% endblock %}
```

最后，编辑 *account* 应用中的*/registration/login.html* 模板（template），添加如下代码在`<form>`元素之后：

```
<p><a href="{% url "password_reset" %}">Forgotten your password?</a></p>
```

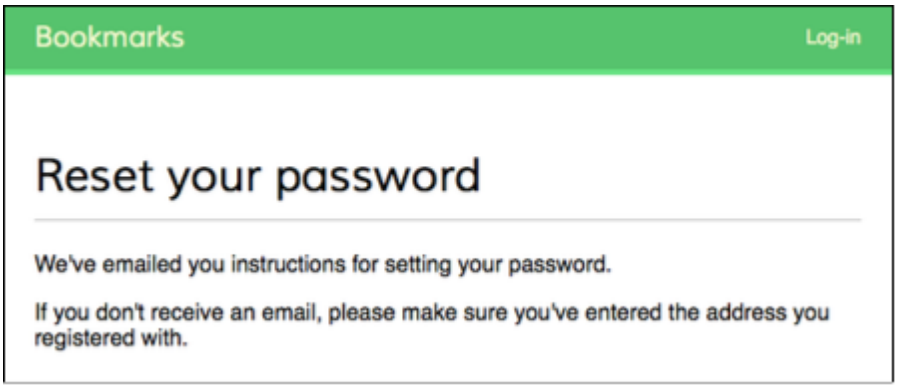
现在，在浏览器中打开 http://127.0.0.1:8000/account/login/ 然后点击 **Forgotten your password?**链接。你会看到如下图所示页面：



在这部分，你需要在你项目中的 *settings.py* 中添加一个 SMTP 配置，这样 Django 才能发送 e-mails。我们已经在*第二章 使用高级特性来优化你的 blog* 学习过如何添加 e-mail 配置。当然，在开发期间，我们可以配置 Django 在标准输出中输出 e-mail 内容来代替通过 SMTP 服务发送邮件。Django 提供一个 e-mail 后端来输出 e-mail 内容到控制器中。编辑项目中 *settings.py* 文件，添加如下代码：

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

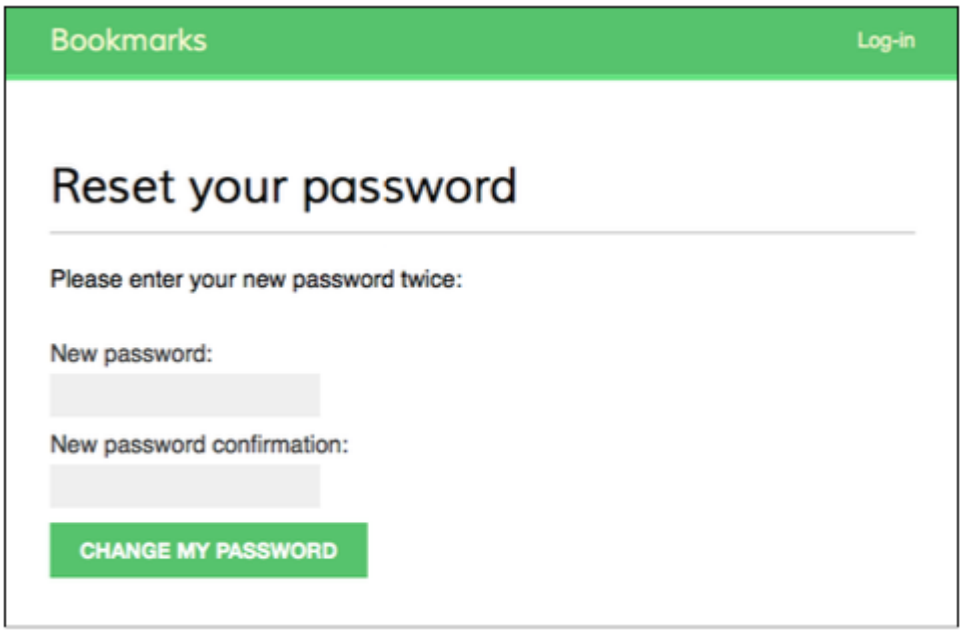
EMAIL_BACKEND 设置这个类用来发送 e-mails。
回到你的浏览器中，填写一个存在用户的 e-mail 地址，然后点击 **Send e-mail** 按钮。你会看到如下图所示页面：



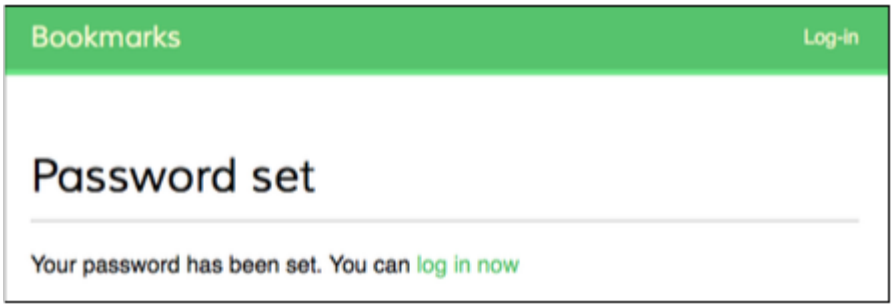
当你运行开发服务的时候看眼控制台输出。你会看到如下所示生成的 e-mail：

```
IME-Version: 1.0
 Content-Type: text/plain; charset="utf-8"
Content-Transfer-Encoding: 7bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: user@domain.com
Date: Thu, 24 Sep 2015 14:35:08 -0000
Message-ID: <20150924143508.62996.55653@zenx.local>
Someone asked for password reset for email user@domain.com. Follow the link below:
http://127.0.0.1:8000/account/password-reset/confirm/MQ/45f-9c3f30caafd523055fcc/
Your username, in case you've forgotten: zenx
```

这封 e-mail 被我们之前创建的 *password_reset_email.html* 模板给渲染。这个给你重置密码的 URL 带有一个 Django 动态生成的 token。在浏览器中打开这个链接，你会看到如下所示页面：



这个页面对应 *password_reset_confirm.html* 模板（template）用来设置新密码。填写新的密码然后点击 **Change my password button**。Django 会创建一个新的加密后密码保存进数据库，你会看到如下图所示页面：



现在你可以使用你的新密码登录你的账号。每个用来设置新密码的 token 只能使用一次。如果你再次打开你之前获取的链接，你会得到一条信息告诉你这个 token 已经无效了。
你在项目中集成了 Django 认证（authentication）框架的视图（views）。这些视图（views）对大部分的情况都适合。当然，如果你需要一种不同的行为你能创建你自己的视图。
（译者注：第四章上结束，之后内容将放到第四章

用户注册和用户 profiles

现有的用户已经可以登录，登出，修改他们的密码，以及当他们忘记密码的时候重置他们的密码。现在，我们需要构建一个视图（view）允许访问者创建他们的账号。

用户注册

让我们创建一个简单的视图（view）允许用户在我们的网站中进行注册。首先，我们需要创建一个表单（form）让用户填写用户名，他们的真实姓名以及密码。编辑 *account* 应用新目录下的 *forms.py* 文件添加如下代码：

```
from django.contrib.auth.models import User
class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password',
                               widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password',
                                widget=forms.PasswordInput)
    class Meta:
        model = User
        fields = ('username', 'first_name', 'email')
```

```
    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Passwords don\'t match.')
        return cd['password2']
```

我们为 *User* 模型（model）创建了一个 model 表单（form）。在我们的表单（form）中我们只包含了模型（model）中的 *username,first_name,email* 字段。这些字段会在它们对应的模型（model）字段上进行验证。例如：如果用户选择了一个已经存在的用户名，将会得到一个验证错误。我们还添加了两个额外的字段 *password* 和 *password2* 给用户用来填写他们的新密码和确定密码。我们定义了一个`clean_password2()`方法去检查第二次输入的密码是否和第一次输入的保持一致，如果不一致这个表单将会是无效的。当我们通过调用`is_valid()`方法验证这个表单（form）时这个检查会被执行。你可以提供一个`clean_<fieldname>()`方法给任何一个你的表单（form）字段用来清理值或者抛出表单（from）指定的字段的验证错误。表单（forms）还包含了一个`clean()`方法用来验证表单（form）的所有内容，这对验证需要依赖其他字段的字段是非常有用的。
Django 还提供一个 *UserCreationForm* 表单（form）给你使用，它位于 *django.contrib.auth.forms* 非常类似与我们刚才创建的表单（form）。
编辑 *account* 应用中的 *views.py* 文件，添加如下代码：

```
from .forms import LoginForm, UserRegistrationForm
def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Create a new user object but avoid saving it yet
            new_user = user_form.save(commit=False)
            # Set the chosen password
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Save the User object
            new_user.save()
            return render(request,
                          'account/register_done.html',
                          {'new_user': new_user})
    else:
        user_form = UserRegistrationForm()
    return render(request,
                  'account/register.html',
                  {'user_form': user_form})
```

这个创建用户账号的视图（view）非常简单。为了保护用户的隐私，我们使用 *User* 模型（model）的`set_password()`方法将用户的原密码进行加密后再进行保存操作。
现在，编辑 *account* 应用中的 *urls.py* 文件，添加如下 URL 模式：

```
url(r'^register/$', views.register, name='register'),
```

最后，创建一个新的模板（template）在 *account/*模板（template）目录下，命名为 *register.html*,为它添加如下代码：

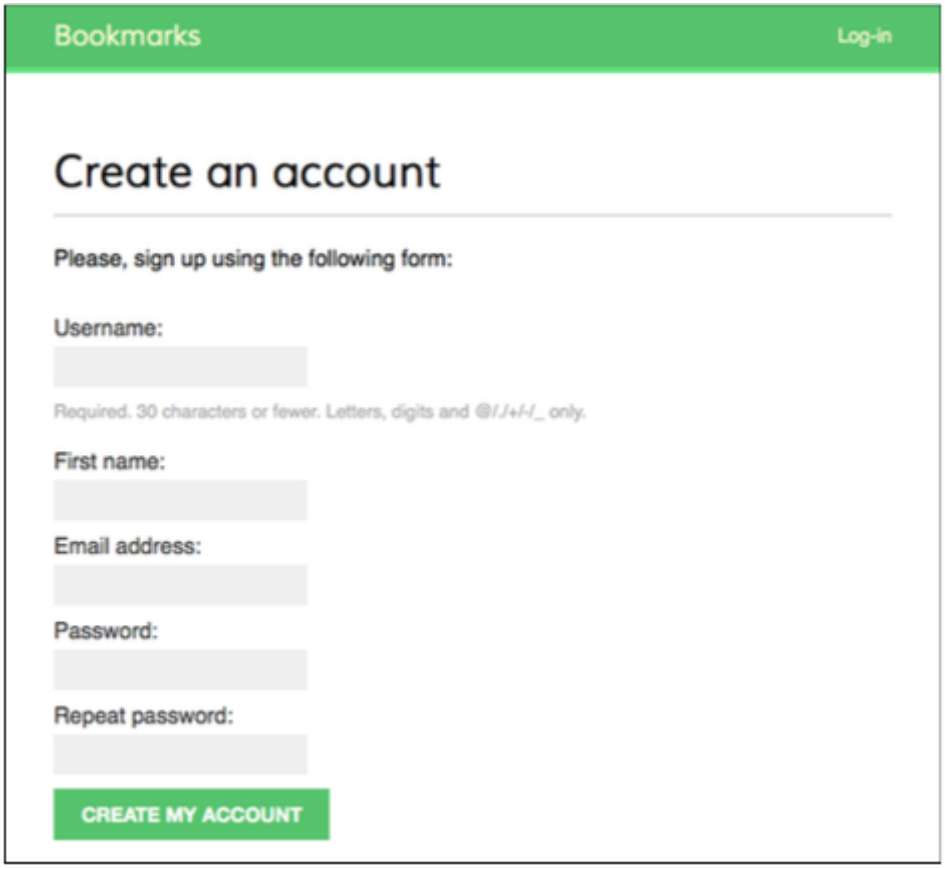
```
{% extends "base.html" %}
{% block title %}Create an account{% endblock %}
```

```
{% block content %}
  <h1>Create an account</h1>
  <p>Please, sign up using the following form:</p>
  <form action="." method="post">
    {{ user_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Create my account"></p>
  </form>
{% endblock %}
```

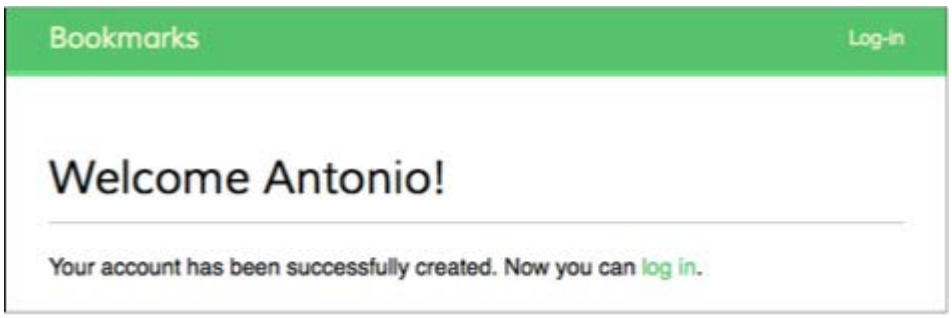
在相同的目录中添加一个模板（template）文件命名为 *register_done.html*，为它添加如下代码：

```
{% extends "base.html" %}
{% block title %}Welcome{% endblock %}
{% block content %}
  <h1>Welcome {{ new_user.first_name }}!</h1>
  <p>Your account has been successfully created. Now you can <a href="{% url "login" %}">log in</a>.</p>
{% endblock %}
```

现在，在浏览器中打开 http://127.0.0.1:8000/account/register/ 。你会看到你创建的注册页面：



填写用户信息然后点击 **Create my account** 按钮。如果所有的字段都验证都过，这个用户将会被创建然后会得到一条成功信息，如下所示：



点击 **log-in** 链接输入你的用户名和密码来验证账号是否成功创建。
现在，你还可以添加一个注册链接在你的登录模板（template）中。编辑 *registration/login.html* 模板（template）然后替换以下内容：

```
<p>Please, use the following form to log-in:</p>
```

为：

```
<p>Please, use the following form to log-in. If you don't have an account <a href="{% url "register" %}">register here</a></p>
```

如此，我们就可以从登录页面进入注册页面。

扩展 User 模型（model）

当你需要处理用户账号，你会发现 Django 认证（authentication）框架的 *User* 模型（model）只适应一般的案例。无论如何，*User* 模型（model）只有一些最基本的字段。你可能希望扩展 *User* 模型包含额外的数据。最好的办法就是创建一个 *profile* 模型（model）包含所有额外的字段并且和 Django 的 *User* 模型（model）做一对一的关联。
编辑 *account* 应用中的 *model.py* 文件，添加如下代码：

```
from django.db import models
from django.conf import settings


class Profile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL)
    date_of_birth = models.DateField(blank=True, null=True)
    photo = models.ImageField(upload_to='users/%Y/%m/%d', blank=True)

    def __str__(self):
        return 'Profile for user {}'.format(self.user.username)
```

为了保持你的代码通用化，当需要定义模型（model）和用户模型的关系时，使用 `get_user_model()`方法来取回用户模型(model)并使用 *AUTH_USER_MODEL* 设置来引用这个用户模型，替代直接引用 auth 的 *User* 模型（model)。
user 一对一字段允许我们关联用户和 profiles。*photo* 字段是一个 *ImageField* 字段。你需要安装一个 Python 包来管理图片，使用 **PIL(Python Imaging Library)**或者 **Pillow**（PIL 的分叉）,在 shell 中运行一下命令来安装 *Pillow*：

```
pip install Pillow==2.9.0
```

为了 Django 能在开发服务中管理用户上传的多媒体文件，在项目 *setting.py* 文件中添加如下设置：

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

MEDIA_URL 是管理用户上传的多媒体文件的主 URL，MEDIA_ROOT 是这些文件在本地保存的路径。我们动态的构建这些路径相对我们的项目路径来确保我们的代码更通用化。
现在，编辑 *bookmarks* 项目中的主 *urls.py* 文件，修改代码如下所示：

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

```
from django.conf.urls.static import static


urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^account/', include('account.urls')),
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

在这种方法中，Django 开发服务器将会在开发时改变对多媒体文件的服务。
*static()*帮助函数最适合在开发环境中使用而不是在生产环境使用。绝对不要在生产环境中使用 Django 来服务你的静态文件。
打开终端运行以下命令来为新的模型（model）创建数据库迁移：

```
python manage.py makemigrations
```

你会获得以下输出：

```
Migrations for 'account':
    0001_initial.py:
        - Create model Profile
```

接着，同步数据库通过以下命令：

```
python manage.py migrate
```

你会看到包含以下内容的输出：

```
Applying account.0001_initial... OK
```

编辑 *account* 应用中的 *admin.py* 文件，在管理站点注册 *Profiel* 模型（model），如下所示：

```
from django.contrib import admin
from .models import Profile
class ProfileAdmin(admin.ModelAdmin):
    list_display = ['user', 'date_of_birth', 'photo']
admin.site.register(Profile, ProfileAdmin)
```

使用 `python manage.py runnserver` 命令重新运行开发服务。现在，你可以看到 *Profile* 模型已经存在你项目中的管理站点中，如下所示：

现在，我们要让用户可以在网站编辑它们的 *profile*。添加如下的模型（model）表单（forms）到 *account* 应用中的 *forms.py* 文件：

```
from .models import Profile

class UserEditForm(forms.ModelForm):
```

```
    class Meta:
        model = User
        fields = ('first_name', 'last_name', 'email')

class ProfileEditForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ('date_of_birth', 'photo')
```

这两个表单（forms）的功能：

- UserEditForm：允许用户编辑它们的 *first name,last name, e-mail* 这些储存在 *User* 模型（model）中的内置字段。
- ProfileEditForm：允许用户编辑我们存储在定制的 *Profile* 模型（model）中的额外数据。用户可以编辑他们的生日数据以及为他们的 *profile* 上传一张照片。

编辑 *account* 应用中的 *view.py* 文件，导入 *Profile* 模型（model），如下所示：

```
from .models import Profile
```

然后添加如下内容到 *register* 视图（view）中的 `new_user.save()` 下方：

```
# Create the user profile
profile = Profile.objects.create(user=new_user)
```

当用户在我们的站点中注册，我们会创建一个对应的空的 *profile* 给他们。你需要在管理站点中为之前创建的用户们一个个手动创建对应的 *Profile* 对象。

现在，我们要让用户能够编辑他们的 *pfofile*。添加如下代码到相同文件中：

```
from .forms import LoginForm, UserRegistrationForm, \
UserEditForm, ProfileEditForm
@login_required
def edit(request):
    if request.method == 'POST':
        user_form = UserEditForm(instance=request.user,
                                 data=request.POST)
        profile_form = ProfileEditForm(instance=request.user.profile,
                                       data=request.POST,
                                       files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
    else:
        user_form = UserEditForm(instance=request.user)
        profile_form = ProfileEditForm(instance=request.user.profile)
    return render(request,
                  'account/edit.html',
                  {'user_form': user_form,
                  'profile_form': profile_form})
```

我们使用 *login_required* 装饰器 *decorator* 是因为用户编辑他们的 *profile* 必须是认证通过的状态。在这个例子中，我们使用两个模型（model）表单（forms）：*UserEditForm* 用来存储数据到内置的 *User* 模型（model）中，*ProfileEditForm* 用来存储额外的 *profile* 数据。为了验证提交的数据，通过 `is_valid()`

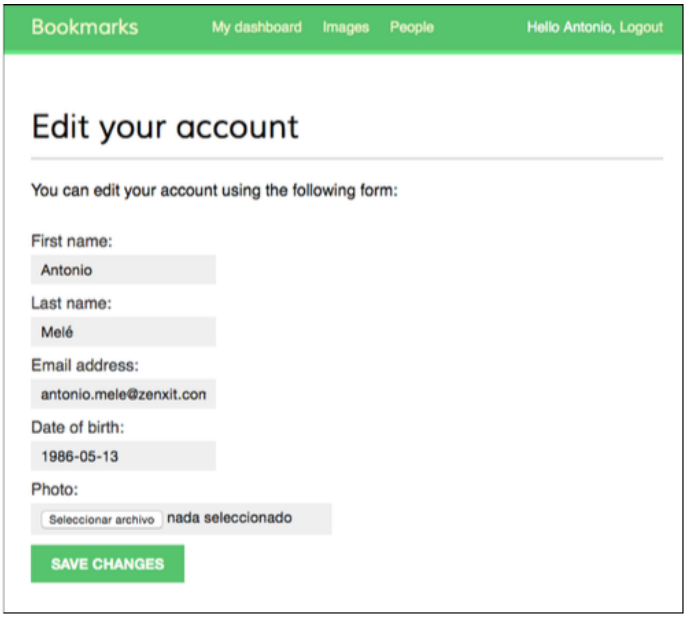
方法是否都返回 *True* 我们来检查每个表单（forms）。在这个例子中，我们保存两个表单（form）来更新数据库中对应的对象。
在 *account* 应用中的 *urls.py* 文件中添加如下 URL 模式：

```
url(r'^edit/$', views.edit, name='edit'),
```

最后，在 *templates/account/*中创建一个新的模板（template）命名为 *edit.html*，为它添加如下内容：

```
{% extends "base.html" %}
{% block title %}Edit your account{% endblock %}
{% block content %}
    <h1>Edit your account</h1>
    <p>You can edit your account using the following form:</p>
    <form action="." method="post" enctype="multipart/form-data">
        {{ user_form.as_p }}
        {{ profile_form.as_p }}
        {% csrf_token %}
    <p><input type="submit" value="Save changes"></p>
    </form>
{% endblock %}
```

我们在表单（form）中包含 `enctype="multipart/form-data"`用来支持文件上传。我们使用一个 HTML 表单来提交两个表单（forms）: *user_form* 和 *profile_form*。
注册一个新用户然后打开 http://127.0.0.1:8000/account/edit/。你会看到如下所示页面：



现在，你可以编辑 dashboard 页面包含编辑 *profile* 的页面链接和修改密码的页面链接。打开 *account/dashboard.html* 模板（model）替换如下代码：

```
<p>Welcome to your dashboard.</p>
```

为：

```
<p>Welcome to your dashboard.
You can <a href="{% url "edit" %}">edit your profile</a>
or <a href="{% url "password_change" %}">change your password</a>.</p>
```

用户现在可以从他们的 dashboard 访问编辑他们的 *profile* 的表单。

使用一个定制 *User* 模型（model）

Django 还提供一个方法可以使用你自己定制的模型（model）来替代整个 *User* 模型（model）。你自己的用户类需要继承 Django 的 *AbstractUser* 类，这个类提供了一个抽象的模型（model）用来完整执行默认用户。你可访问 https://docs.djangoproject.com/en/1.8/topics/auth/customizing/#substituting-a-custom-user-model 来获得这个方法的更多信息。

使用一个定制的用户模型（model）将会带给你很多的灵活性，但是它也可能给一些需要与 *User* 模型（model）交互的即插即用的应用集成带来一定的困难。

使用 messages 框架

当处理用户的操作时，你可能想要通知你的用户关于他们操作的结果。Django 有一个内置的 messages 框架允许你给你的用户显示一次性的提示。messages 框架位于 *django.contrib.messages*，当你使用 `python manage.py startproject` 命令创建一个新项目的时候，messages 框架就被默认包含在 *settings.py* 文件中的 *INSTALLED_APPS* 中。你会注意到你的设置文件中在 *MIDDLEWARE_CLASSES* 设置里包含了一个名为 *django.contrib.messages.middleware.MessageMiddleware* 的中间件。messages 框架提供了一个简单的方法添加消息给用户。消息被存储在数据库中并且会在用户的下一次请求中展示。你可以在你的视图（views）中导入 *messages* 模块来使用消息 messages 框架，用简单的快捷方式添加新的 messages，如下所示：

```
from django.contrib import messages
messages.error(request, 'Something went wrong')
```

你可以使用 `add_message()`方法创建新的 messages 或用以下任意一个快捷方法：

- success()：当操作成功后显示成功的 messages
- info()：展示 messages
- warning()：某些还没有达到失败的程度但已经包含有失败的风险，警报用
- error()：操作没有成功或者某些事情失败
- debug()：在生产环境中这种 messages 会移除或者忽略

让我们显示 messages 给用户。因为 messages 框架是被项目全局应用，我们可以在主模板（template）给用户展示 messages。打开 *base.html* 模板（template）在 id 为 *header* 的`<div>`和 id 为 *content* 的`<div>`之间添加如下内容：

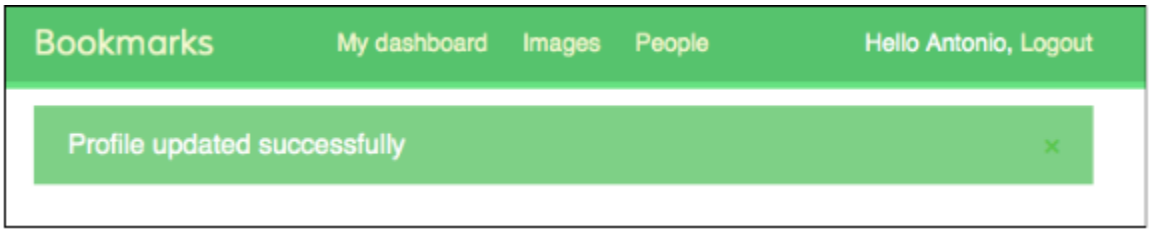
```
{% if messages %}
 <ul class="messages">
   {% for message in messages %}
     <li class="{{ message.tags }}">
    {{ message|safe }}
        <a href="#" class="close"> </a>
     </li>
   {% endfor %}
 </ul>
{% endif %}
```

messages 框架带有一个上下文环境（context）处理器用来添加一个 *messages* 变量给请求的上下文环境（context）。所以你可以在模板（template）中使用这个变量用来给用户显示当前的 messages。现在，让我们修改 *edit* 视图（view）来使用 messages 框架。编辑应用中的 *views.py* 文件，使 *edit* 视图（view）如下所示：

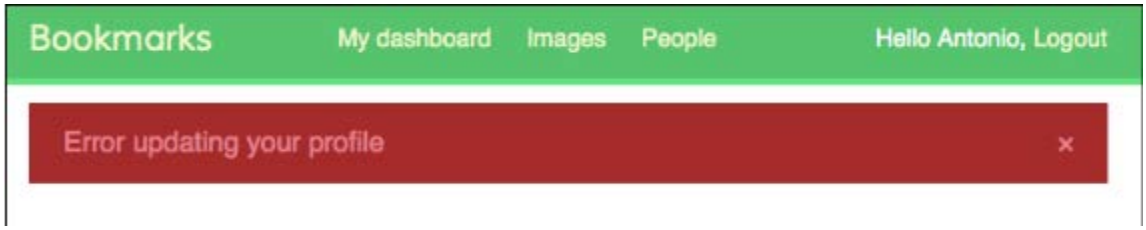
```
from django.contrib import messages
@login_required
def edit(request):
```

```
    if request.method == 'POST':
    # ...
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
            messages.success(request, 'Profile updated '\
                                      'successfully')
        else:
            messages.error(request, 'Error updating your profile')
    else:
        user_form = UserEditForm(instance=request.user)
        # ...
```

当用户成功的更新他们的 profile 时我们就添加了一条成功的 message，但如果某个表单（form）无效，我们就添加一个错误 message。
在浏览器中打开 http://127.0.0.1:8000/account/edit/ 编辑你的 profile。当 profile 更新成功，你会看到如下 message：



当表单（form）是无效的，你会看到如下 message：



创建一个定制的认证（authentication）后台

Django 允许你通过不同的来源进行认证（authentication）。*AUTHENTICATION_BACKENDS* 设置包含了所有的给你的项目的认证（authentication）后台。默认的，这个设置如下所示：

```
('django.contrib.auth.backends.ModelBackend',)
```

默认的 *ModelBackend* 通过数据库使用 *django.contrib.auth* 中的 *User* 模型（model）来认证（authentication）用户。这适用于你的大部分项目。当然，你还可以创建定制的后台通过其他的来源例如一个 *LDAP* 目录或者其他任何系统来认证你的用户。
你可以通过访问 https://docs.djangoproject.com/en/1.8/topics/auth/customizing/#other-authentication-sources 获得更多的信息关于自定义的认证（authentication）。
当你使用 *django.contrib.auth* 的 *authenticate()*函数，Django 会通过每一个定义在 *AUTHENTICATION_BACKENDS* 中的后台一个接一个地尝试认证（authentication）用户，直到其中有一个后台成功的认证该用户才会停止进行认证。只有所有的后台都无法进行用户认证（authentication），他或她才不会在你的站点中通过认证（authentication）。
Django 提供了一个简单的方法来定义你自己的认证（authentication）后台。一个认证（authentication）后台就是提供了如下两种方法的一个类：

- authenticate()：将用户信息当成参数，如果用户成功的认证（authentication）就需要返回 *True*，反之，需要返回 *False*。
- get_user()：将用户的 ID 当成参数然后需要返回一个用户对象。

创建一个定制认证（authentication）后台非常容易，就是编写一个 Python 类实现上面两个方法。我们要创建一个认证（authentication）后台让用户在我们的站点中使用他们 e-mail 替代他们的用户名来进行认证（authentication）。

在你的 *account* 应用中创建一个新的文件命名为 *authentication.py*，为它添加如下代码：

```
from django.contrib.auth.models import User


class EmailAuthBackend(object):
    """
    Authenticate using e-mail account.
    """
    def authenticate(self, username=None, password=None):
        try:
            user = User.objects.get(email=username)
            if user.check_password(password):
                return user
            return None
        except User.DoesNotExist:
            return None
    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
return None
```

这是一个简单的认证（authentication）后台。`authenticate()`方法接收了 *username* 和 *password* 两个可选参数。我们可以使用不同的参数，但是我们需要使用 *username* 和 *password* 来确保我们的后台可以立马在认证（authentication）框架视图（views）中工作。以上代码完成了以下工作内容：

- authenticate()：我们尝试通过给予的 e-mail 地址获取一个用户和使用 *User* 模型（model）中内置的 `check_password()` 方法来检查密码。这个方法会对给予密码进行哈希化来和数据库中存储的加密密码进行匹配。
- get_user()：我们通过 *user_id* 参数获取一个用户。Django 使用这个后台来认证用户之后取回 *User* 对象放置到持续的用户会话中。

编辑项目中的 *settings.py* 文件添加如下设置：

```
AUTHENTICATION_BACKENDS = (
   'django.contrib.auth.backends.ModelBackend',
   'account.authentication.EmailAuthBackend',
)
```

我们保留默认的 *ModelBacked* 用来保证用户仍然可以通过用户名和密码进行认证，接着我们包含进了我们自己的 email-based 认证（authentication）后台。现在，在浏览器中打开 http://127.0.0.1:8000/account/login/ 。请记住，Django 会对每个后台都尝试进行用户认证（authentication），所以你可以使用用户名或者使用 email 来进行无缝登录。

AUTHENTICATION_BACKENDS 设置中的后台排列顺序。如果相同的认证信息在多个后台都是有效的，Django 会停止在第一个成功认证（authentication）通过的后台，不再继续进行认证（authentication）。

为你的站点添加社交认证（authentication）

你可以希望给你的站点添加一些社交认证（authentication）服务，例如 *Facebook*，*Twitter* 或者 *Google*(国内就算了- -|||)。*Python-social-auth* 是一个 Python 模块提供了简化的处理为你的网站添加社交认证（authentication）。通过使用这个模块，你可以让你的用户使用他们的其他服务的账号来登录你的网站。你可以访问 https://github.com/omab/python-social-auth 得到这个模块的代码。
这个模块自带很多认证（authentication）后台给不同的 Python 框架，其中就包含 Django。
使用 pip 来安装这个包，打开终端运行如下命令：

```
pip install python-social-auth==0.2.12
```

安装成功后，我们需要在项目 *settings.py* 文件中的 *INSTALLED_APPS* 设置中添加 *social.apps.django_app.default*：

```
INSTALLED_APPS = (
    #...
    'social.apps.django_app.default',
)
```

这个 *default* 应用会在 Django 项目中添加 *python-social-auth*。现在，运行以下命令来同步 *python-social-auth* 模型（model）到你的数据库中：

```
python manage.py migrate
```

你会看到如下 *default* 应用的数据迁移输出：

```
Applying default.0001_initial... OK
Applying default.0002_add_related_name... OK
Applying default.0003_alter_email_max_length... OK
```

python-social-auth 包含了很多服务的后台。你可以访问 https://python-social-auth.readthedocs.org/en/latest/backends/index.html#supported-backends 看到所有的后台支持。
我们要包含的认证（authentication）后台包括 *Facebook*，*Twitter*，*Google*。
你需要在你的项目中添加社交登录 URL 模型。打开 *bookmarks* 项目中的主 *urls.py* 文件，添加如下 URL 模型：

```
url('social-auth/',
    include('social.apps.django_app.urls', namespace='social')),
```

为了确保社交认证（authentication）可以工作，你还需要配置一个 *hostname*，因为有些服务不允许重定向到 *127.0.0.1* 或 *localhost*。为了解决这个问题，在 *Linux* 或者 *Mac OSX* 下，编辑你的*/etc/hosts* 文件添加如下内容：

```
127.0.0.1 mysite.com
```

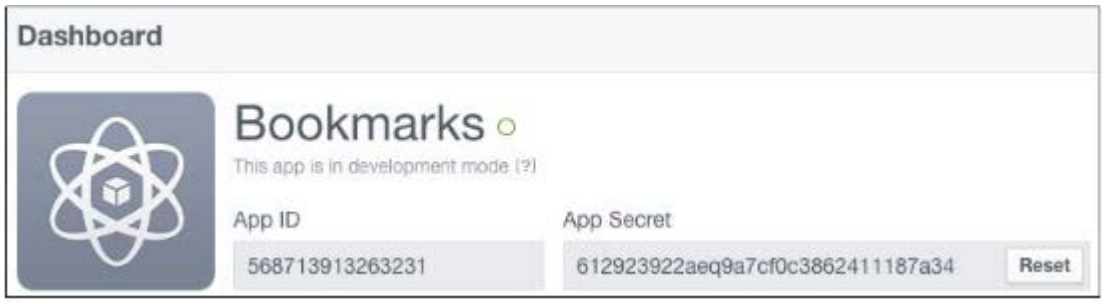
这是用来告诉你的计算机指定 *mysite.com hostname* 指向你的本地机器。如果你使用 *Windows*,你的 hosts 文件在 *C:\Winwows\ System32\Drivers\etc\hosts*。
为了验证你的 *host* 重定向是否可用，在浏览器中打开 http://mysite.com:8000/account/login/ 。如果你看到你的应用的登录页面，*host* 重定向已经可用。

使用 Facebook 认证（authentication）

（译者注：以下的几种社交认证操作步骤可能已经过时，请根据实际情况操作）
为了让你的用户能够使用他们的 Facebook 账号来登录你的网站，在项目 *settings.py* 文件中的 *AUTHENTICATION_BACKENDS* 设置中添加如下内容：

```
'social.backends.facebook.Facebook2OAuth2',
```

为了添加 Facebook 的社交认证（authentication），你需要一个 Facebook 开发者账号，然后你必须创建一个新的 Facebook 应用。在浏览器中打开 https://developers.facebook.com/apps/?action=create 点击 **Add new app** 按钮。点击 **Website** 平台然后为你的应用取名为 *Bookmarks*，输入 http://mysite.com:8000/ 作为你的网站 URL。跟随快速开始步骤然后点击 **Create App ID**。
回到网站的 Dashboard。你会看到类似下图所示的：



拷贝 **App ID** 和 **App Secret** 关键值，将它们添加在项目中的 *settings.py**文件中，如下所示：

```
SOCIAL_AUTH_FACEBOOK_KEY = 'XXX' # Facebook App ID
SOCIAL_AUTH_FACEBOOK_SECRET = 'XXX' # Facebook App Secret
```

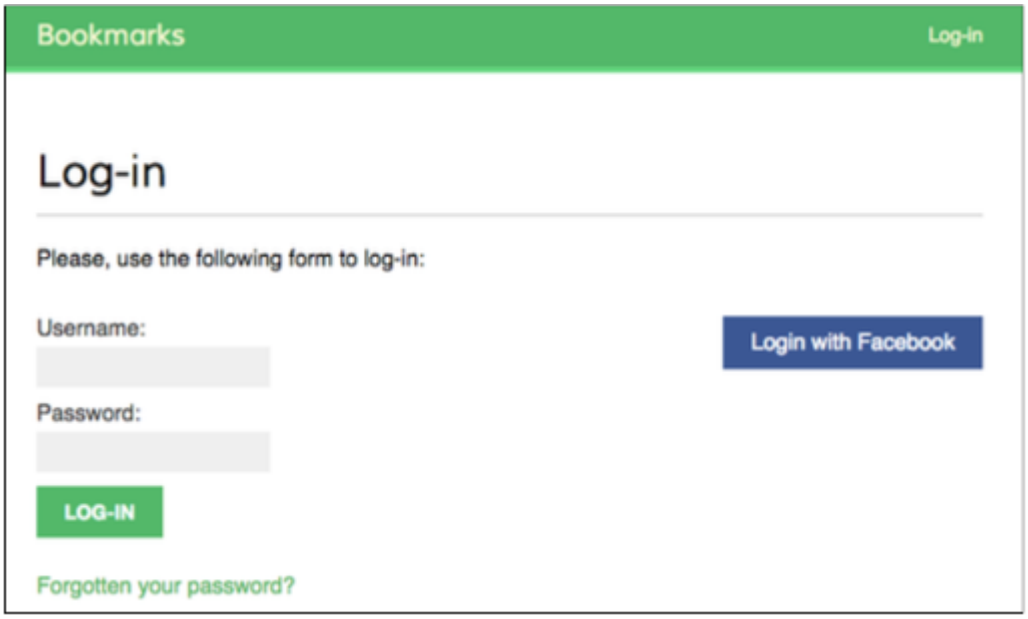
此外，你还可以定义一个 *SOCIAL_AUTH_FACEBOOK_SCOPE* 设置如果你想要访问 Facebook 用户的额外权限，例如：

```
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email']
```

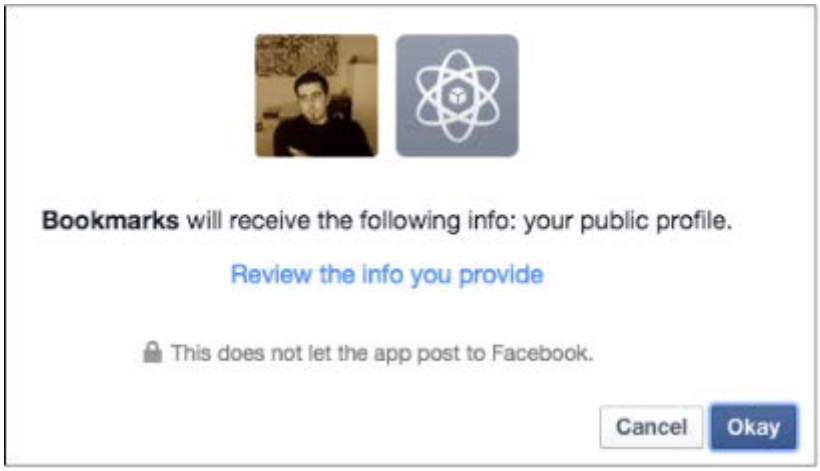
最后，打开 *registration/login.html* 模板（template）然后添加如下代码到 *content* block 中：

```
<div class="social">
  <ul>
    <li class="facebook"><a href="{% url "social:begin" "facebook" %}">Sign in with Facebook</a></li>
  </ul>
</div>
```

在浏览器中打开 http://mysite.com:8000/account/login/ 。现在你的登录页面会如下图所示：



点击*Login with Facebook* 按钮。你会被重定向到 Facebook，然后你会看到一个对话询问你的权限是否让 *Bookmarks* 应用访问你的公共 Facebook profile：



点击 **Okay** 按钮。Python-social-auth 会对认证（authentication）进行操作。如果每一步都没有出错，你会登录成功然后被重定向到你的网站的 dashboard 页面。请记住，我们已经使用过这个 URL 在 *LOGIN_REDIRECT_URL* 设置中。就像你所看到的，在你的网站中添加社交认证（authentication）是非常简单的。

使用 Twitter 认证（authentication）

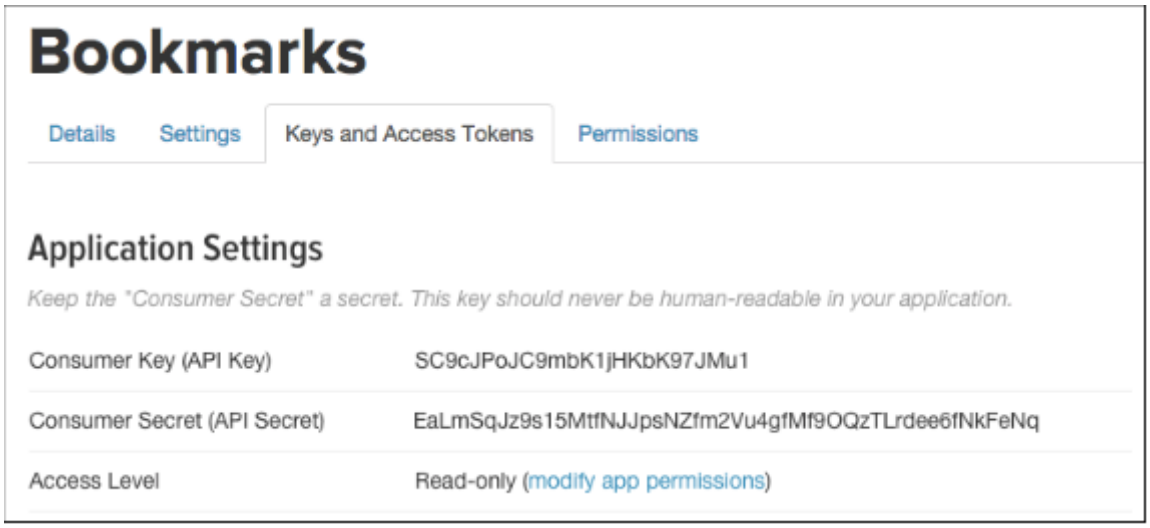
为了使用 Twitter 进行认证(authentication)，在项目 *settings.py* 中的 *AUTHENTICATION_BACKENDS* 设置中添加如下内容：

```
'social.backends.twitter.TwitterOAuth',
```

你需要在你的 Twitter 账户中创建一个新的应用。在浏览器中打开 https://apps.twitter.com/app/new 然后输入你应用信息，包含以下设置：

- Website：http://mysite.com:8000/
- Callback URL：http://mysite.com:8000/social-auth/complete/twitter/

确保你勾选了复选款 **Allow this application to be used to Sign in with Twitter**。之后点击 **Keys and Access Tokens**。你会看到如下所示信息：



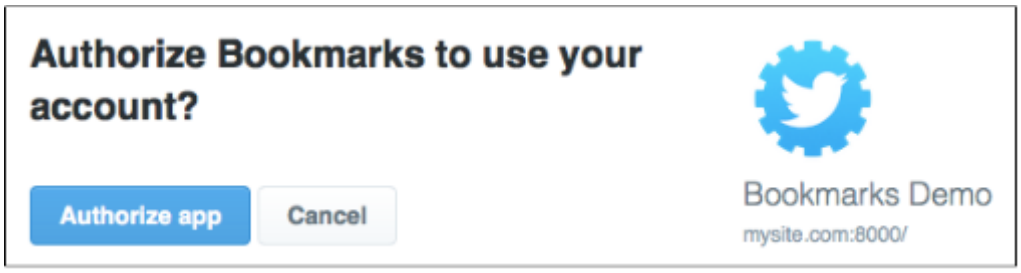
拷贝 **Consumer Key** 和 **Consumer Secret** 关键值，将它们添加到项目 *settings.py* 的设置中，如下所示：

```
SOCIAL_AUTH_TWITTER_KEY = 'XXX' # Twitter Consumer Key
SOCIAL_AUTH_TWITTER_SECRET = 'XXX' # Twitter Consumer Secret
```

现在，编辑 *login.html* 模板（template），在`<ul>`元素中添加如下代码：

```
<li class="twitter"><a href="{% url "social:begin" "twitter" %}">Login with Twitter</a></li>
```

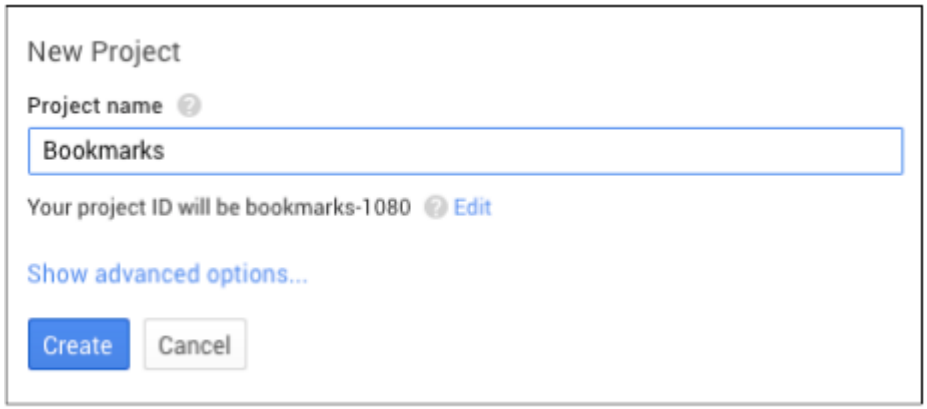
在浏览器中打开 http://mysite.com:8000/account/login/ 然后点击 **Login with Twitter** 链接。你会被重定向到 Twitter 然后它会询问你授权给应用，如下所示：



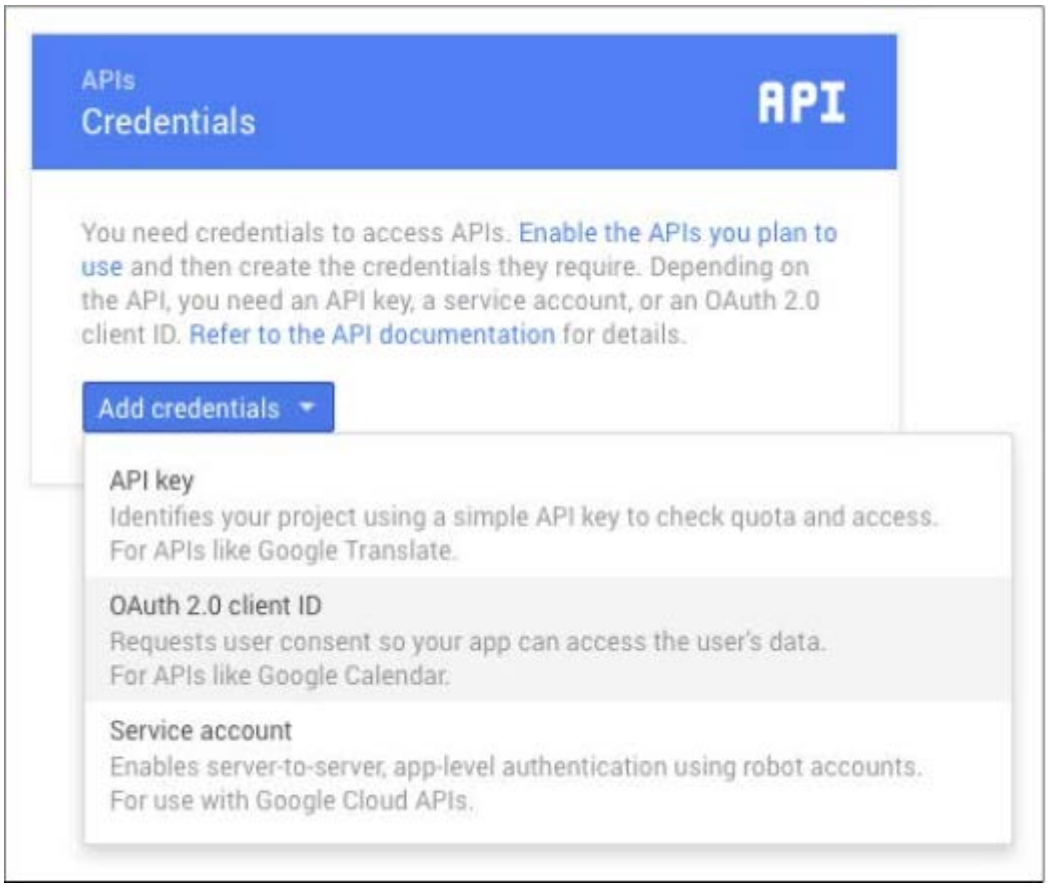
点击 **Authorize app** 按钮。你会登录成功并且重定向到你的网站 dashboard 页面。

使用 Google 认证（authentication）

Google 提供 *OAuth2* 认证（authentication）。你可以访问 https://developers.google.com/accounts/docs/OAuth2 获得关于 Google OAuth2 的信息。
首先，你徐闯创建一个 *API key* 在你的 Google 开发者控制台。在浏览器中打开 https://console.developers.google.com/project 然后点击 **Create project** 按钮。输入一个名字然后点击 **Create** 按钮，如下所示：



在项目创建之后，点击在左侧菜单的 **APIs & auth** 链接，然后点击 **Credentials** 部分。点击 **Add credentials** 按钮，然后选择 **OAuth2.0 client ID**，如下所示：

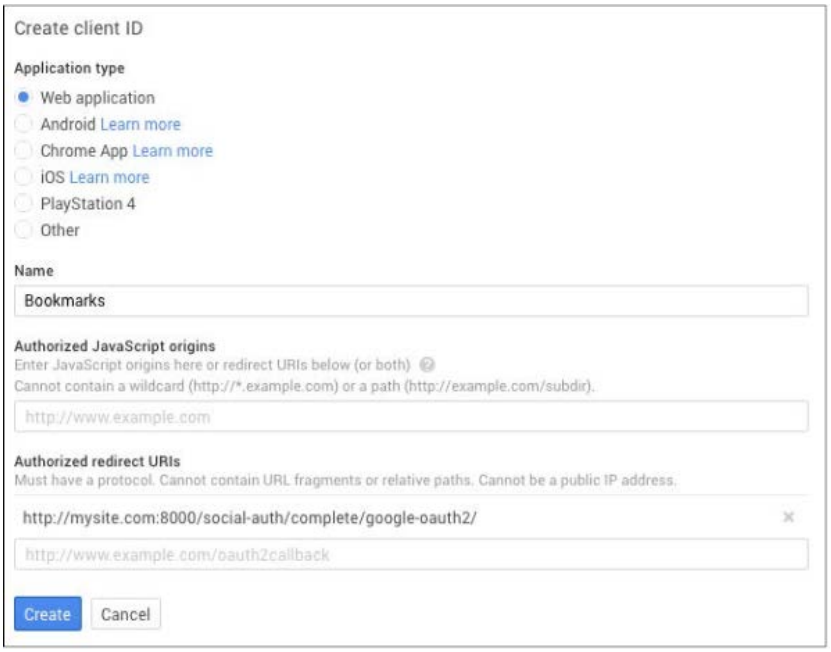


Google 首先会询问你配置同意信息页面。这个页面将会展示给用户告知他们是否同意使用他们的 Google 账号来登录访问你的网站。点击 *Configure consent screen* 按钮。选择你的 e-mail 地址，填写 *Bookmarks* 为 **Product name**，然后点击 **Save** 按钮。这个给你的项目使用的同意信息页面将会配置完成然后你会被重定向去完成创建你的 Client ID。

在表单（form）中填写以下内容：

- Application type： 选择 **Web application**
- Name： 输入 *Bookmarks*
- Authorized redirect URLs：输入 http://mysite.com:8000/social-auth/complete/google-oauth2/

这表单（form）将会如下所示：



点击 **Create** 按钮。你将会获得 **Client ID** 和 **Client Secret** 关键值。在你的 *settings.py* 中添加它们，如下所示：

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = '' # Google Consumer Key
```

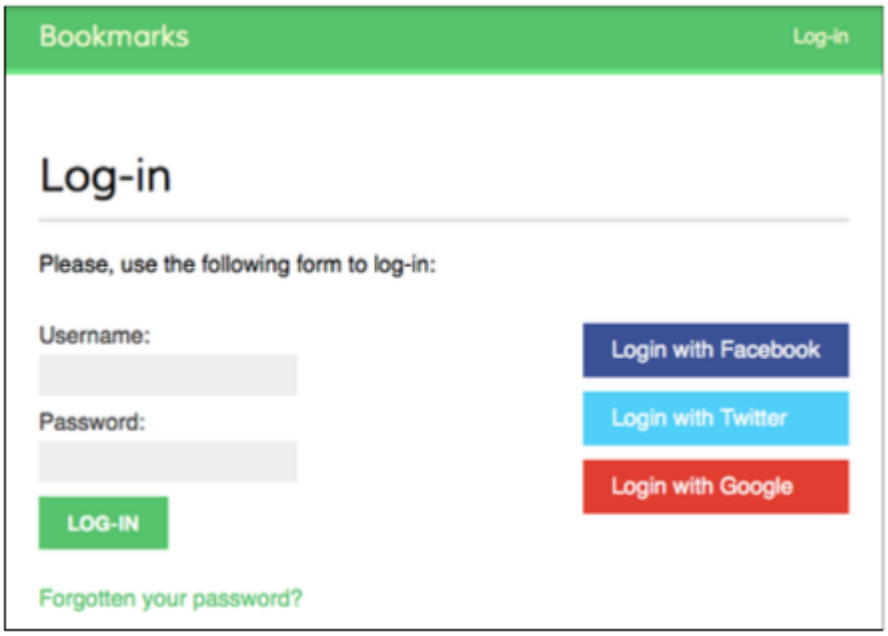
```
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = '' # Google Consumer Secret
```

在 Google 开发者控制台的左方菜单，**APIs & auth** 部分的下方，点击 **APIs** 链接。你会看到包含所有 Google Apis 的列表。点击 **Google+ API** 然后点击 **Enable API** 按钮在以下页面中：

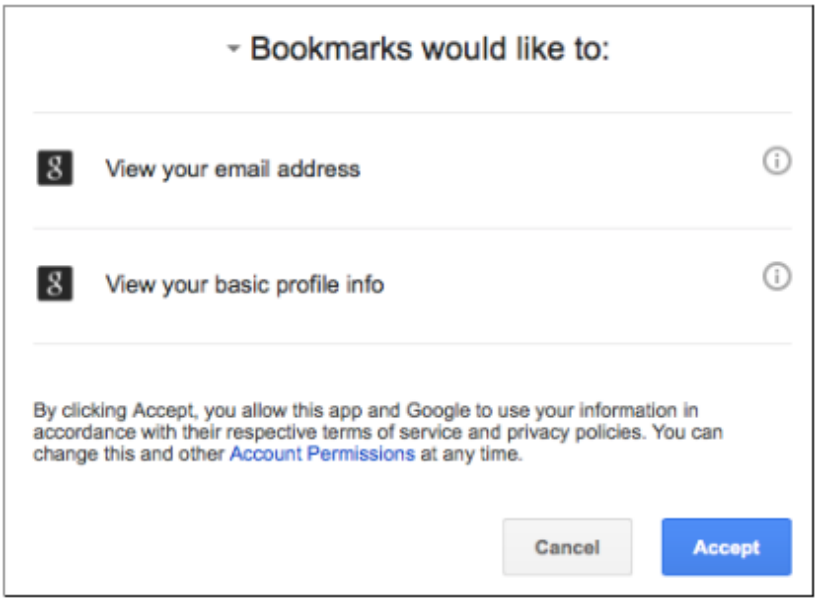
编辑 *login.html* 模板（template）在`<ul>`元素中添加如下代码：

```
<li class="google"><a href="{% url "social:begin" "google" %}">Login with Google</a></li>
```

在浏览器中打开 http://mysite.com:8000/account/login/ 。登录页面将会看上去如下图所示：



点击 **Login with Google** 按钮。你将会被重定向到 Google 并且被询问权限通过我们之前配置的同意信息页面：



点击 **Accept** 按钮。你会登录成功并重定向到你的网站的 dashboard 页面。

我们已经添加了社交认证（authentication）到我们的项目中。python-social-auth 模块还包含更多其他非常热门的在线服务。

总结

在本章中，你学习了如何创建一个认证（authentication）系统到你的网站并且创建了定制的用户 profile。你还为你的网站添加了社交认证（authentication）。
在下一章中，你会学习如何创建一个图片收藏系统（image bookmarking system），生成图片缩微图，创建 AJAX 视图（views）。

译者总结

终于写到了这里，呼出一口气，第四章的页数是前几章的两倍，在翻译之前还有点担心会不会坚持不下去，不过看样子我还是坚持了下来，而且发现一旦翻译起来就不想停止（- -|||莫非心中真的有翻译之魂！？）。这一章还是比较基础，主要介绍了集成用户的认证系统到网站中，比较有用的是通过第三方的平台账户登录，可惜 3 个平台 Facbook，Twitter，Google 国内都不好访问，大家练习的时候还是用国内的 QQ，微信，新浪等平台来练习吧。第五章的翻译不清楚什么时候能完成，也许过年前也可能过年后，反正不管如何，这本书我一定要翻译到最后！

第五章 在你的网站中分享内容

在上一章中，你为你的网站建立了用户注册和认证系统。你学习了如何为用户创建定制化的个人资料模型以及如何将主流的社交网络的认证添加进你的网站。
在这一章中，你将学习如何通过创建一个 JavaScript 书签来从其他的站点分享内容到你的网站，你也将通过使用 jQuery 在你的项目中实现一些 AJAX 特性。
这一章涵盖了以下几点：

- 创建一个 `many-to-many`（多对多）关系
- 定制表单（form）的行为
- 在 Django 中使用 jQuery
- 创建一个 jQuery 书签
- 通过使用 sorl.thumbnail 来生成缩略图
- 实现 AJAX 视图（views）并且使这些视图（views）和 jQuery 融合
- 为视图（views）创建定制化的装饰器 （decorators）
- 创建 AJAX 分页

建立一个能为图片打标签的网站

我们将允许用户可以在我们网站中分享他们在其他网站发现的图片，并且他们还可以为这些图片打上标签。为了达到这个目的，我们将要做以下几个任务：

- 定义一个模型来储存图片以及图片的信息
- 新建一个表单（form）和视图（view）来控制图片的上传
- 为用户创建一个可以上传他们在其他网站发现的图片的系统

首先，通过以下命令在你的 bookmarks 项目中新建一个应用：

```
django-admin startapp images
```

像如下所示一样在你的 `settings.py` 文件中 `INSTALED_APPS` 设置项下添加 'images' :

```
INSTALLED_APPS = [
    # ...
    'images',
]
```

现在 Django 知道我们的新应用已经被激活了。

创建图像模型

编辑 images 应用中的 `models.py` 文件，将以下代码添加进去：

```
from django.db import models
from django.conf import settings
class Image(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL,
    related_name='images_created')
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200,blank=True)
    url = models.URLField()
    image = models.ImageField(upload_to='images/%Y/%m/%d')
    description = models.TextField(blank=True)
    created = models.DateField(auto_now_add=True,
                               db_index=True)
    def __str__(self):
        return self.title
```

我们将要使用这个模型来储存来自各个不同网站中被标记的图片。让我们来看看在这个模型中的字段：

- `user`： 标记了这张图片 `User` 对象。这是一个 `ForeignKey` 字段 (译者注：外键，即一对多字段)，因为它指定了一个一对多关系： 一个用户可以 post 多张图片， 但是每张图片只能由一个用户上传
- `title`： 图片的标题
- `slug`： 一个只包含字母、数字、下划线、和连字符的标签， 用于创建优美的 搜索引擎友好（SEO-friendly）的 URL（译者注：slug 这个词在中文没有很好的对应翻译，所以就请大家记住“slug 表示的是只有字母、数字、下划线和连字符的标签”。如果有仔细看过 Django 官方文档的读者就会知道： slug 是一个新闻术语， 而 Django 的开发目的也是为了更好的编辑新闻， 所以这里就不难理解为什么 Django 中会出现 slug 字段了）
- `url`： 这张图片的源 URL
- `image`： 图片文件
- `description`： 一个可选的图片描述字段
- `created`： 用于表明一个对象在数据库中创建时的时间和日期。由于我们使用了 `auto_now_add`，当对象被创建时候时间和日期将会被自动设置，我们使用了 `db_index=True` ，所以 Django 将会在数据库中为这个字段创建索引

数据库索引改善了查询的执行。考虑为这个字段设置 `db_index=True` 是因为你将要很频繁地使用 `filter()`, `exclude()`,`order_by()` 来执行查询。`ForeignKey` 字段或者带有 `unique=True` 的字段表明了一个索引的创建。你也可以使用 `Meta.index_together` 来为多个字段创建索引。

我们将要重写 Image 模型的 `save()`方法来自动的生成 `slug` 字段。这个 `slug` 字段基于 `title` 字段的值。像下面这样导入 `slugify()`函数， 然后在 Image 模型中添加一个 `save()`方法：

```
from django.utils.text import slugify
class Image(models.Model):
    # ...
    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.title)
            super(Image, self).save(*args, **kwargs)
```

在这段代码中，我们使用了 Django 提供的 `slugify()`函数在没有提供 `slug` 字段时根据给定的图片标题自动生 slug,然后，我们保存了这个对象。我们自动生成 slug，这样的话用户就不用自己输入 `slug` 字段了。

建立多对多关系

我们将要在 Image 模型中再添加一个字段来保存喜欢这张图片的用户。因此，我们需要一个多对多关系。因为一个用户可能喜欢很多张图片，一张图片也可能被很多用户喜欢。
在 Image 模型中添加以下字段：

```
user_like = models.ManyToManyField(settings.AUTH_USER_MODEL,
                                   related_name='images_liked',
                                   blank=True)
```

当你定义一个 `ManyToMany` 字段时，Django 会用两张表主键（primary key）创建一个中介联接表（译者注：就是新建一张普通的表，只是这张表的内容是由多对多关系双方的主键构成的）。`ManyToMany` 字段可以在任意两个相关联的表中创建。
同 `ForeignKey` 字段一样， `ManyToMany` 字段的 `related_name` 属性使我们可以命名另模型回溯（或者是反查）到本模型对象的关系。`ManyToMany` 字段提供了一个多对多管理器（manager），这个管理器使我们可以回溯相关联的对象比如：`image.users_like.all()`或者从一个 `user` 中回溯，比如：`user.images_liked.all()`。
打开命令行，执行下面的命令以创建首次迁移：

```
python manage.py makemigrations images
```

你能看见以下输出：

```
Migrations for 'images':
    0001_initial.py:
        - Create model Image
```

现在执行这条命令来应用你的迁移：

```
python manage.py migrate images
```

你将会看到包含这一行输出：

```
Applying images.0001_initial... OK
```

现在 Image 模型已经在数据库中同步了。

注册 Image 模型到管理站点中

编辑 `images` 应用的 `admin.py` 文件，然后像下面这样将 `Image` 模型注册到管理站点中：

```
from django.contrib import admin
from .models import Image
class ImageAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug', 'image', 'created']
    list_filter = ['created']

admin.site.register(Image, ImageAdmin)
```

使用命令 `python manage.py runserver` 打开开发服务器，在浏览器中打开 `http://127.0.0.1:8000/admin/`,可以看到 `Image` 模型已经注册到了管理站点中：

Django-5-1

从其他网站上传内容

我们将使用户可以给从他们在其他网站发现的图片打上标签。用户将要提供图片的 URL ，标题，和一个可选的描述。我们的应用将要下载这幅图片，并且在数据库中创建一个新的 `Image` 对象。
我们从新建一个用于提交图片的表单开始。在 images 应用的路径下创建一个 `forms.py` 文件，在这个文件中添加如下代码：

```
from django import forms
from .models import Image
class ImageCreateForm(forms.ModelForm):
    class Meta:
        model = Image
        fields = ('title', 'url', 'description')
        widgets = {
            'url': forms.HiddenInput,
        }
```

如你所见，这是一个通过 `Image` 模型创建的 `ModelForm`（模型表单），但是这个表单只包含了 `title`,`url`,`description` 字段。我们的用户不会在表单中直接为图片添加 URL。相反的，他们将会使用一个 JavaScropt 工具来从其他网站中选择一张图片然后我们的表单将会以参数的形式接收这张图片的 URL。我们覆写 `url` 字段的默认控件（widget）为一个 `HiddenInput` 控件，这个控件将会被渲染为属性是 `type="hidden"` 的 HTML 元素。使用这个控件是因为我们不想让用户看见这个字段。

清洁表单字段

（译者注：原文标题是：cleaning form fields,在数据处理中有个术语是“清洗数据”，但是这里的清洁还有“使其整洁”的含义，感觉更加符合 `clean_url` 这个方法的定位。）
为了验证提供的图片 URL 是否合法，我们将检查以 `.jpg` 或 `.jpeg` 结尾的文件名，来只允许 JPG 文件的上传。Django 允许你自定义表单方法来清洁特定的字段，通过使用以 `clean_<fieldname>` 形式命名的方法来实现。这个方法会在你为一个表单实例执行 `is_valid()` 时执行。在清洁方法中，你可以改变字段的值或者为某个特定的字段抛出错误当需要的时候，将下面这个方法添加进 `ImageCreateForm`:

```
def clean_url(self):
    url = self.cleaned_data['url']
    valid_extensions = ['jpg', 'jpeg']
    extension = url.rsplit('.', 1)[1].lower()
    if extension not in valid_extensions:
        raise forms.ValidationError('The given URL does not ' \
                                    'match valid image extensions.')
    return url
```

在这段代码中，我们定义了一个 `clean_url` 方法来清洁 `url` 字段，这段代码的工作流程是：

- 1. 我们从表单实例的 `cleaned_data` 字典中获取了 `url` 字段的值
- 1. 我们分离了 URL 来获取文件扩展名，然后检查它是否为合法扩展名之一。如果它不是一个合法的扩展名，我们就会抛出 `ValidationError`,并且表单也不会被认证。我们执行的是一个非常简单的

认证。你可以使用更好的方法来验证所给的 URL 是否是一个合法的图片。
除了验证所给的 URL， 我们还需要下载并保存图片文件。比如，我们可以使用操作表单的视图来下载图片。不过，我们将采用一个更加通用的方法 ———— 通过覆写我们模型表单中 `save()` 方法来完成这个任务。

覆写模型表单中的 save()方法

如你所知，`ModelForm` 提供了一个 `save()` 方法来保存目前的模型实例到数据库中，并且返回一个对象。这个方法接受一个布尔参数 `commit`，这个参数允许你指定这个对象是否要被储存到数据库中。如果 `commit` 是 `False`，`save()` 方法将会返回一个模型实例但是并不会把这个对象保存到数据库中。我们将覆写表单中的 `save()` 方法，来下载图片然后保存它。
将以下的包在 `foroms.py` 中的顶部导入：

```
from urllib import request
from django.core.files.base import ContentFile
from django.utils.text import slugify
```

把 `save()` 方法加入 `ImageCreateForm` 中：

```
def save(self, force_insert=False,
         force_update=False,
         commit=True):
    image = super(ImageCreateForm, self).save(commit=False)
    image_url = self.cleaned_data['url']
    image_name = '{}.{}'.format(slugify(image.title),
    image_url.rsplit('.', 1)[1].lower())
# 从给定的 URL 中下载图片
    response = request.urlopen(image_url)
    image.image.save(image_name,
                     ContentFile(response.read()),
                     save=False)
    if commit:
        image.save()
    return image
```

我们覆写的 `save()` 方法保持了 `ModelForm` 中需要的参数、
这段代码：

1. 我们通过调用 `save()` 方法从表单中新建了一个 `image` 对象，并且 `commit=False`
2. 我们从表单的 `cleaned_data` 字典中获取了 URL
3. 我们通过结合 `image` 的标题 slug 和源文件的扩展名生成了图片的名字
4. 我们使用 Python 的 `urllib` 模块来下载图片，然后我们调用 `save()` 方法把图片传递给一个 `ContentFiel` 对象，这个对象被下载的文件所实例化。这样，我们就可以将我们的文件保存到项目中的 media 路径下。我们传递了参数 `comiit=False` 来避免对象被保存到数据库中。
5. 为了保持和我们覆写的 `save()` 方法一样的行为，我们将在 `commit` 参数为 `Ture` 时保存表单到数据库中。

现在我们需要一个新的视图来控制我们的表单。编辑 iamges 应用的 `views.py` 文件，然后将以下代码添加进去:

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
from django.contrib import messages
from .forms import ImageCreateForm

@login_required
```

```
def image_create(request):
    """
    View for creating an Image using the JavaScript Bookmarklet.
    """
    if request.method == 'POST':
        # form is sent
        form = ImageCreateForm(data=request.POST)
        if form.is_valid():
            # form data is valid
            cd = form.cleaned_data
            new_item = form.save(commit=False)
            # assign current user to the item
            new_item.user = request.user
            new_item.save()
            messages.success(request, 'Image added successfully')
            # redirect to new created item detail view
            return redirect(new_item.get_absolute_url())
    else:
        # build form with data provided by the bookmarklet via GET
        form = ImageCreateForm(data=request.GET)

    return render(request, 'images/image/create.html', {'section': 'images',
                                                        'form': form})
```

我们给 `image_create` 视图添加了一个 `login_required` 装饰器，来阻止未认证的用户的连接。这段代码完成下面的工作：

1. 我们先从 GET 中获取初始数据来创建一个表单实例。这个数据由来自外部网站图片的 `url` 和 `title` 属性构成，并且将由我们等会儿要创建的 JavaScript 工具提供。现在我们只是假设这里有初始数据。
2. 如果表单被提交我们将检查它是否合法。如果这个表单是合法的，我们将新建一个 `Image` 实例，但是我们通过传递 `commit=False` 来保证这个对象将不会保存到数据库中。
3. 我们将绑定当前用户（user）到一个新的 `iamge` 对象。这样我们就可以知道是谁上传了每一张图片。
4. 我们把 iamge 对象保存到了数据库中
5. 最后，我们使用 Django 的信息框架创建了一条上传成功的消息然后重定向用户到新图像的规范 URL 。我们没有在 Image 模型中实现 `get_absolute_url()`方法，我们等会儿将编写它。

在你的 images 应用中创建一个叫做 `urls.py` 的新文件，然后添加如下代码：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^create/$', views.image_create, name='create'),
]
```

像下面这样编辑在你项目文件夹中的主 `urls.py` 文件，将我们刚才为 images 应用创建的 url 模式添加进去：

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^account/', include('account.urls')),
    url(r'^images/', include('images.urls', namespace='images')),
```

```
]
```

最后，你需要创建一个模板来渲染你的表单。在你的 images 应用路径下创建如下路径结构：

```
templates/
   images/
      image/
         create.html
```

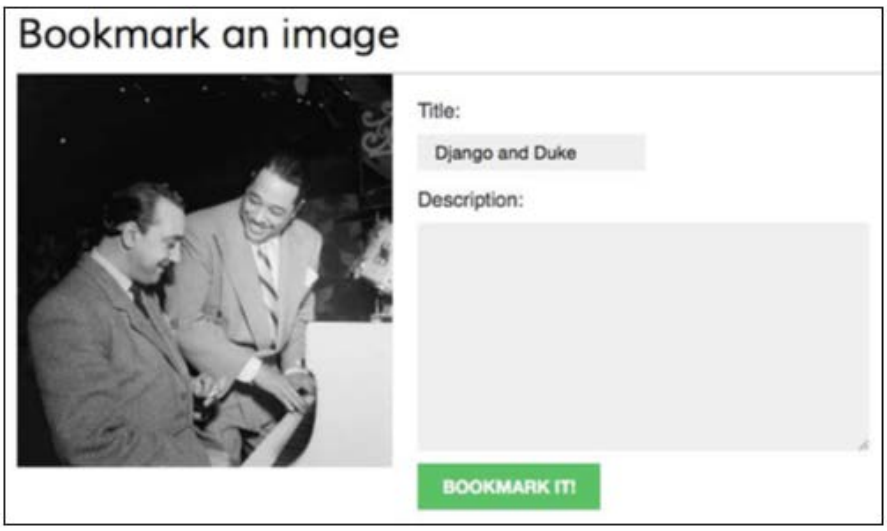
编辑新的 `create.html` 模板然后添加以下代码进去：

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

{% block title %}Bookmark an image{% endblock %}

{% block content %}
    <h1>Bookmark an image</h1>
    ![]({{ request.GET.url }})
    <form action="." method="post">
        {{ form.as_p }}
        {% csrf_token %}
        <input type="submit" value="Bookmark it!">
    </form>
{% endblock %}
```

现在在你的浏览器中打开 `http://127.0.0.1:8000/images/create/?title=...&url=...`，记得在 后面传递 GET 参数 `title` 和 `url` 来提供一个已存在的 JPG 图像的 URL 。
举个例子，你可以使用像下面这样的 URL：

```
http://127.0.0.1:8000/images/create/?title=%20Django%20and%20Duke&url=http://upload.wikimedia.org/wikipedia/commons/8/85/Django_Reinhardt_and_Duke_Ellington_%28Gottlieb%29.jpg
```

你可以看到一个带有图片预览的表单，就像下面这样：



Django-5-2

添加描述然后点击 **Bookmark it!** 按钮。一个新的 `Image` 对象将会被保存在你的数据库中。你将会得到一个错误，这个错误指示说 `Image` 模型没有 `get_absolute_url()` 方法。现在先不要担心这个，我们待会儿

将添加这个方法、在你的浏览器中打开 `http://127.0.0.1:8000/admin/images/image/`，确定新的图像对象已经被保存了。

用 jQuery 创建一个书签

书签是一个保存在浏览器中包含 JavaScript 代码的标签，用来拓展浏览器功能。当你点击书签的时候，JavaScript 代码会在浏览器显示的网站中被执行。这是一个在和其它网站交互时非常有用的工具。
一些在线服务，比如 Pinterest 实现了他们自己的书签来让用户可以在他们的平台中分享来自其他网站的内容，我们将以同样的方式创建一个书签，让用户可以在我们的网站中分享来自其他网站的图片。
我们将使用 jQuery 来创建我们的书签。 jQuery 是一个流行的 JavaScript 框架， 这个框架允许你快速开发客户端的功能。你可以在官网中更多的了解 jQuery: http://jquery.com/
你的用户将会像下面这样在他们的浏览器中添加书签然后使用它：

1. 用户从你的网站中拖拽一个链接到他的浏览器。这个链接在它的 `href` 属性中包含了 JavaScript 代码。这段代码将会被储存到书签当中。
2. 用户访问任意一个网站，然后点击这个书签， 这个书签的 JavaScript 代码就被执行了。

由于 JavaScript 代码将会以书签的形式被储存，之后你将不能更新它。这是个很显著的缺点，但是你可以通过实现一个简单的激活脚本来解决这个问题，这个脚本从一个 URL 中加载 JavaScript。你的用户将会以书签的形式来保存这个激活脚本，这样你就能在任何时候更新书签代码的内容了。我们将会采用这个方法来创建我们的书签。我们开始吧！
（译者注：上面这一段似乎有一点难以理解，其实很简单，就是把 JavaScript 保存在后端，只让用户保存一个能获取这段 JavaScript 的 url，url 是由书签来获取的。用户保存的就是这个含有获取 url 的 JavaScript 书签。）
在 image/templates/ 下创建一个新的模板，把它命名为 `bookmarklet_launcher.js`。这个就是我们的激活脚本了。将以下 JavaScript 代码添加进这个文件

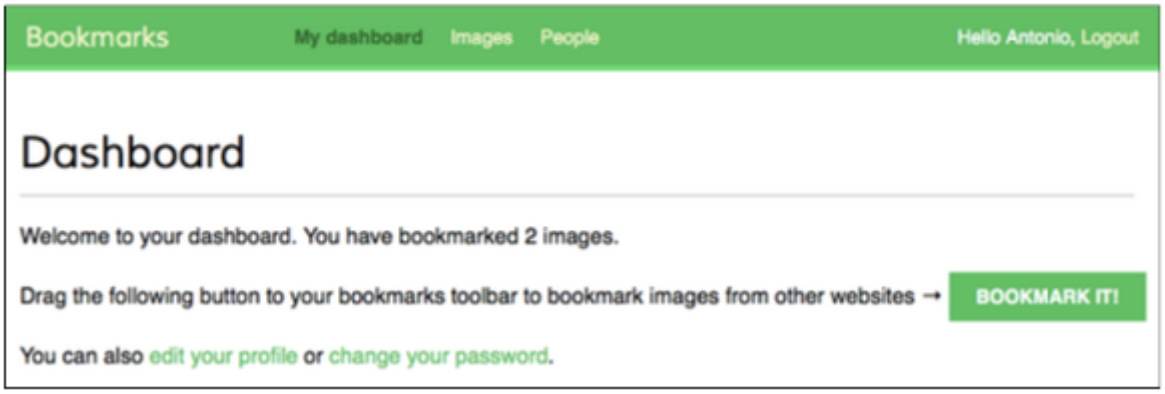
```
(function(){
    if(window.myBookmarklet!==undefined){
        myBookmarklet();
    }
    else{

document.body.appendChild(document.createElement('script')).src='http://127.0.0.1:8000/static/js/bookmarklet.js?r='+Math.floor(Math.random()*99999999999999999999);
    }
})();
```

这段脚本通过检查 `myBookmarklet` 变量是否被定义来检测书签是否被加载。这样，我们就可以避免在用户重复点击书签时重复加载。如果 `myBookmarklet` 没有被定义，我们就再加载一个 JavaScript 文件来在文档中添加一个`<script>`元素。 这个 `script` 标签加载 `bookmarklet_launcher.js` 脚本，将一个随机数作为参数来防止加载浏览器缓存中的文件。
我们当前的 bookmarklet 代码位于 `bookmarklet.js` 静态文件中。这使我们在不要求用户更新书签的情况下更新我们代码。让我们把书签添加进 dashboard 页，我们的用户就可以将它拷贝到他们的书签中。
编辑 account/dashboard.html 模板，像如下一样更改它：

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

{% block title %}Dashboard{% endblock %}

{% block content %}
    <h1>Dashboard</h1>

    {% with total_images_created=request.user.images_created.count %}
```

```
        <p>Welcome to your dashboard. You have bookmarked {{ total_images_created }}
image{{ total_images_created|pluralize }}.</p>
    {% endwith %}

    <p>Drag the following button to your bookmarks toolbar to bookmark images from other websites → <a
href="javascript:{% include "bookmarklet_launcher.js" %}" class="button">Bookmark it!</a><p>

    <p>You can also <a href="{% url "edit" %}">edit your profile</a> or <a href="{% url
"password_change" %}">change your password</a>.<p>
{% endblock %}
```

这个 danshboard 展示了用户所标记的图片总数。我们使用`{% with %}`模板标签来设置一个带有用户标记图片总数的参数。我们也引入了一个带有 `href` 属性的链接，这个链接含有我们的书签激活脚本。我们从 `bookmarklet_launcher.js` 模板中引入 JavaScript 脚本。
在你的浏览器中打开 `http://127.0.0.1:8000/account/`,你可以看到如下页面：



此处输入图片的描述

拖拽 `Bookmark it!`链接到你的浏览器的书签工具栏中。
现在创建下面几个路径和文件在 images 应用路径中：

- static/
- js/
- bookmarklet.js

你会在本章示例代码文件夹中的 images 应用路径下找到 `static/css/` 路径。复制 `css/`路径到你的代码文件夹下的 `static/`中。`css/bookmarklet.css` 文件为我们的 JavaScript 书签提供了样式。
编辑 `bookmarklet.js` 静态文件，然后添加以下 JavaScript 代码：

```
(function(){
  var jquery_version = '2.1.4';
  var site_url = 'http://127.0.0.1:8000/';
  var static_url = site_url + 'static/';
  var min_width = 100;
  var min_height = 100;

  function bookmarklet(msg) {
      // Here goes our bookmarklet code
);
 // Check if jQuery is loaded
  if(typeof window.jQuery != 'undefined') {
    bookmarklet();
  } else {
    // Check for conflicts
```

```
    var conflict = typeof window.$ != 'undefined';
    // Create the script and point to Google API
    var script = document.createElement('script');
    script.setAttribute('src','http://ajax.googleapis.com/ajax/libs/jquery/' + jquery_version +
'/jquery.min.js');
    // Add the script to the 'head' for processing
    document.getElementsByTagName('head')[0].appendChild(script);
    // Create a way to wait until script loading
    var attempts = 15;
    (function(){
      // Check again if jQuery is undefined
      if(typeof window.jQuery == 'undefined') {
        if(--attempts > 0) {
          // Calls himself in a few milliseconds
          window.setTimeout(arguments.callee, 250)
        } else {
          // Too much attempts to load, send error
          alert('An error ocurred while loading jQuery')
        }
      } else {
          bookmarklet();
      }
    })();
  }

})()
```

这是主要的 jQuery 加载脚本，当脚本已经加载到当前网站中时，它负责调用 JQuery 或者是从 Google 的 CDN 中加载 jQuery。当 jQuery 被加载，它会执行 `bookmarklet()`函数，该函数包含我们的 bookmarklet 代码。我们还在这个文件顶部设置几个变量：

- `jquery_version`: 加载的 jQuery 版本
- `site_url` 和 `static_url`：我们网站的主 URL 和各自静态文件的主 URL
- `min_width` 和 `min_height`：我们的书签在网站中将要寻找的图像支持的最小宽度和最小高度，

现在让我们来实现 `bookmarklet` 函数，编辑 `bookmarklet()`，让它看起来像这样：

```
  function bookmarklet(msg) {
    // load CSS
    var css = jQuery('<link>');
    css.attr({
      rel: 'stylesheet',
      type: 'text/css',
      href: static_url + 'css/bookmarklet.css?r=' + Math.floor(Math.random()*9999999999999999999)
    });
    jQuery('head').append(css);

    // load HTML
    box_html = '<div id="bookmarklet"><a href="#" id="close">×</a><h1>Select an image to bookmark:</h1><div
class="images"></div></div>';
    jQuery('body').append(box_html);
```

```
    // close event
    jQuery('#bookmarklet #close').click(function(){
    jQuery('#bookmarklet').remove();
    });
    };
```

这段代码运行如下：

1. 我们加载了 `bookmarklet.css` 样式表，使用一个随机的数字作为参数来避免浏览器的缓存
2. 我们添加了定制的 HTML 到当前网站的`<body>`元素中。这个 HTML 由包含在当前网站寻找到的图片的`<div>`元素构成的。
3. 我们添加了一个事件，当用户点击我们的 HTML 块中的关闭链接时，我们将移除我们添加进去的 HTML。我们使用 `#bookmarklet``#close` 选择器来找到带有一个 ID 为 `close` 的 HTML 元素，这个 HTML 元素的父 ID 是 `bookmarklet`。jQuery 选择器允许你寻找 HTML 元素。jQuery 选择器返回所有给定的 CSS 选择器找到的元素，你可以在这个链接中找到一组 jQuery 选择器：http://api.jquery.com/category/selectors/

在加载了 CSS 样式表和 HTML 后，我们需要在网站中找到图片。在 `bookmarklet()`函数的底部添加如下代码：

```
    // find images and display them
    jQuery.each(jQuery('img[src$="jpg"]'), function(index, image) {
      if (jQuery(image).width() >= min_width && jQuery(image).height() >= min_height)
      {
        image_url = jQuery(image).attr('src');
        jQuery('#bookmarklet .images').append('<a href="#">![]('+ image_url +')</a>');
      }
    });
```

这段代码使用了 `img[src$="jpg"]`选择器来找到所有的`<img>` HTML 元素，并且这些元素的 `src` 属性以 `jpg` 结尾。这意味着我们会找到当前网页中所有的 JPG 图片。我们通过 `each()`方法来遍历所有的结果。我们添加了`<div class="images">` HTML 容器用以放置图片，容器的的尺寸刚好比 `min_width` 和 `min_width` 大一点。

这个 HTML 容器现在包含了可以被打上标签的图片，我们想要用户点击他们需要的图片然后给他们打上标签。在 `bookmarklet()`函数中添加以下代码：

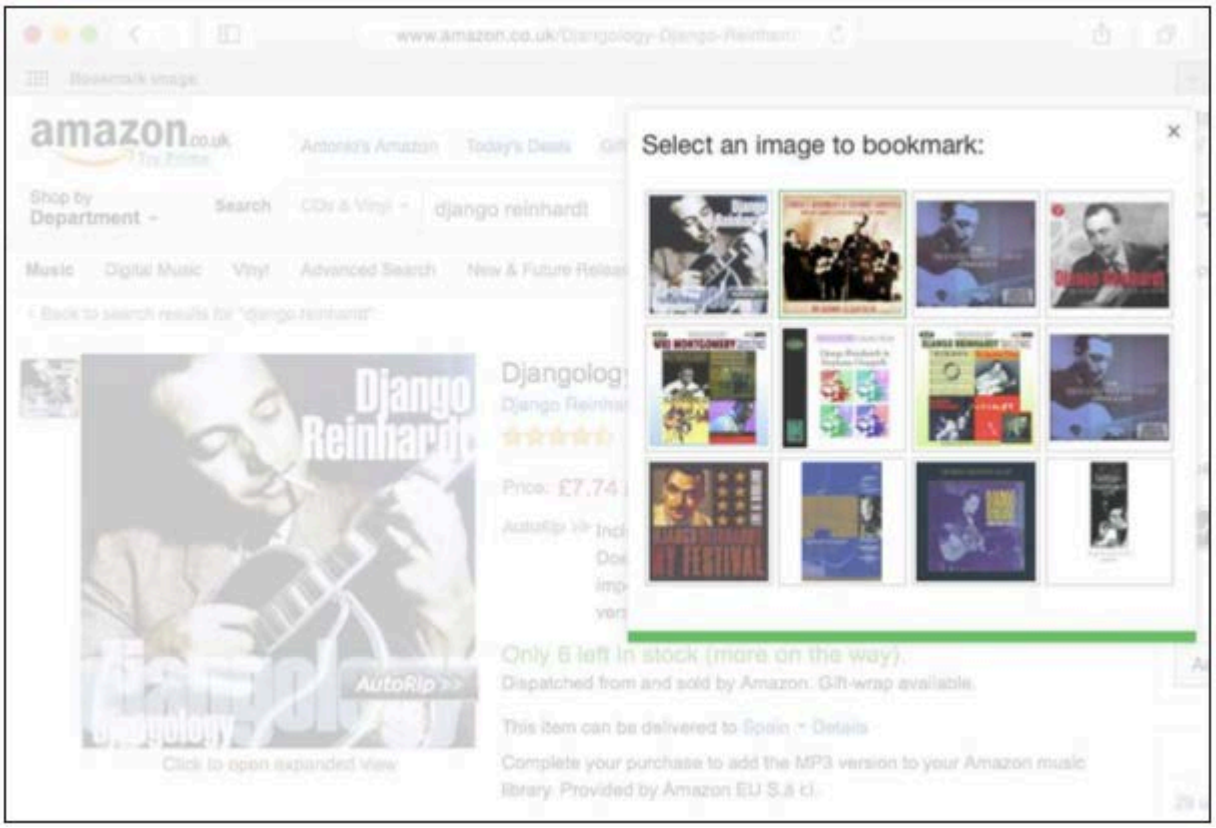
```
    // when an image is selected open URL with it
    jQuery('#bookmarklet .images a').click(function(e){
      selected_image = jQuery(this).children('img').attr('src');
      // hide bookmarklet
      jQuery('#bookmarklet').hide();
      // open new window to submit the image
      window.open(site_url +'images/create/?url='
                + encodeURIComponent(selected_image)
                + '&title=' + encodeURIComponent(jQuery('title').text()),
                '_blank');
    });
```

这段代码按照如下流程运行：

1. 我们把一个 `clck()`事件绑定到了图片的链接元素上
2. 当一个用户点击一个图片时我们新建了一个变量 `selected_image`，这个变量包含了被选择的图片的 URL。

3. 我们隐藏了书签然后在浏览器中打开一个新的窗口，这个窗口访问了我们的网站中为一个新的图片打标签的 URL 。我们传递了网站的 `title` 元素和被选中图片的 URL 作为 GET 参数。

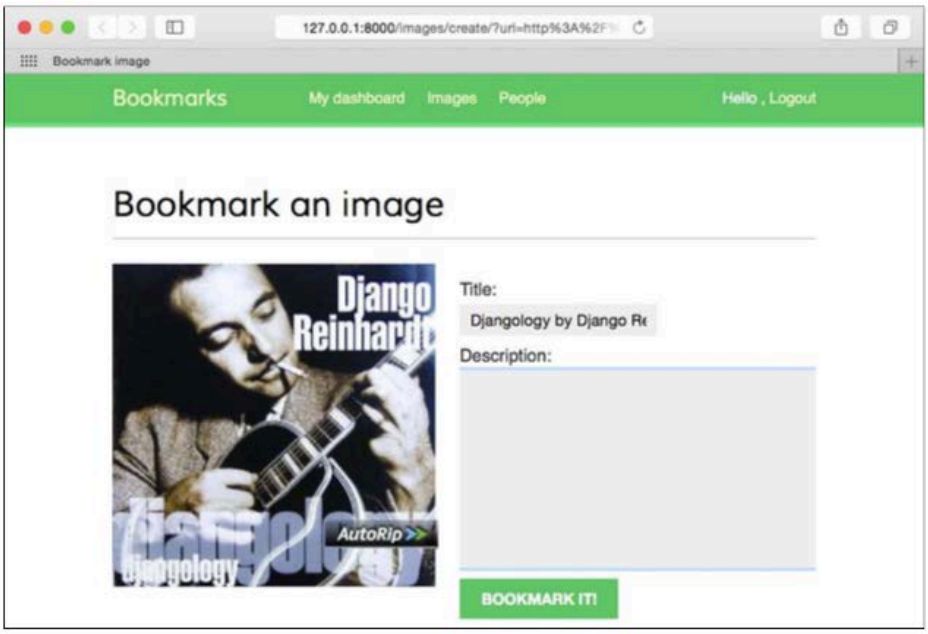
在你的浏览器中随便选择一个网址打开，然后点击你的书签。你将会看到一个白色的新窗口出现在当前网页上，它展示了所有尺寸大于 100*100px 的 JPG 图片，它看起来就像下面的例子一样：



django-5-4

因为我们已经开启了 Django 的开发服务器，使用 HTTP 来提供页面， 由于安全限制，书签将不能在 HTTPS 上工作。

如果你点击一幅图片，你将会被重定向到创建图片的页面，请求地址传递了网站的标题和被选中图片的 URL 作为 GET 参数。



Django-5-5

恭喜！这是你的第一个 JavaScript 书签！现在它已经和你的 Django 项目成为一体！

为你的图片创建一个详情视图

我们将创建一个简单的详情视图，用于展示一张已经保存在我们的网站中的图片。打开 images 应用的 `views.py`，将以下代码添加进去：

```
from django.shortcuts import get_object_or_404
from .models import Image
def image_detail(request, id, slug):
    image = get_object_or_404(Image, id=id, slug=slug)
    return render(request, 'images/image/detail.html', {'section':
'images','image': image})
```

这是一个用于展示图片的简单视图。编辑 iamges 应用的 `urls.py`，添加以下 URL 模式：

```
url(r'^detail/(?P<id>\d+)/(?P<slug>[-\w]+)/$',
            views.image_detail, name='detail'),
```

编辑 images 应用的 `models.py`,并且将 `get_absolute_url()`方法添加进 `Image` 模型:

```
from django.core.urlresolvers import reverse
class Image(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('images:detail',args=(self.id,self.slug))
```

记住，为对象提供精确 URL 的通用模式是在模型中定义 `get_absolute_url()`方法。
最后，在 images 应用的 模版路径`/images/image/`中新建一个模板，命名为 `detail.html`，添加以下代码：

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

{% block title %}{{ image.title }}{% endblock %}

{% block content %}
    <h1>{{ image.title }}</h1>
    ![]({{ image.image.url }})
    {% with total_likes=image.users_like.count %}
        <div class="image-info">
                <div>
                    <span class="count">
                        {{ total_likes }}like{{ total_likes|pluralize }}
                    </span>
                 </div>
                 {{ image.description|linebreaks }}
        <div class="image-likes">
            {% for user in image.users_like.all %}
                <div>
                    ![]({{ user.profile.photo.url }})
                    <p>{{ user.first_name }}</p>
                </div>
            {% empty %}
                Nobody likes this image yet.
            {% endfor %}
        </div>
    {% endwith %}
```

```
{% endblock %}
```

这个模版用来展示一张被打标签图片。我们使用`{% with %}`标签来保存所有统计 `user likes` 查询集（QuerySet）的结果，并将这个结果保存在一个新的变量 `total_likes` 中。这样我们就可以避免计算两次查询集（QuerySet）的结果。我们也引入了图片的描述，迭代了 `image.users_like.all` 来展示所有喜欢这张图片的用户。
使用`{% with %}`模版标签来防止 Django 做多次查询是很有用的
现在使用书签来为一张图片打上标签。在你提交图片之后你将会被重定向图片详情页面。这张图片将会包含一条提交成功的消息，效果如下：

Django-5-6

使用 sorl-thumbnail 创建缩略图

我们在详情页展示原图片，但是不同的图片的尺寸是不同的。一些图片源文件或许会非常大，加载他们会耗费很长时间。展示规范图片的最好方法是生成缩略图。我们将使用一个 Django 应用，叫做 `sorl-thumbnail`。
打开终端，用下面的命令来安装 `sorl-thumbnail`：

```
pip install sorl-thumbnail==12.3
```

编辑 bookmarklet 项目文件的 `settings.py`,将 `sorl-thumbnail` 添加进 `INSTALLED_APPS`.
运行下面的命令来同步你的数据库：

```
python manage.py migrate
```

你看到的输出中应该包含下面这一行：

```
Creating table thumbnail_kvstore
```

`sorl-thumbnail` 应用提供了不同的方法来定义一张图片的缩略图。它提供了`{% thumbnail %}`模版标签来在模版中生成缩略图，同时还有一个定制的 `ImageField` 字段，如果你想要在你的模型中定制缩略图的话。我们将要使用这个模版标签。编辑 `images/image/detail.html` 模版，删除这一行：

```
![]({{ image.image.url }})
```

替换成：

```
{% load thumbnail %}
{% thumbnail image.image "300" as im %}
<a href="{{ image.image.url }}">
![]({{ im.url }})
</a>
{% endthumbnail %}
```

这里，我们定义了一个固定宽度为 300px 的缩略图。当用户第一次加载这页面时，缩略图将会被创建。生成的缩略图将会在接下来的请求中被使用。运行 `python manage.py runserver` 开启开发服务器，连接到一张已有图片的详情页。缩略图将会生成并展示在网站中。
`sorl-thumbnail` 应用提供了几个选择来定制你的缩略图，包括裁减算法和能被应用的不同效果。如果你有任何生成缩略图的疑难点，你可以在你的设置中添加 `THUMBNAIL_DEBUG = TRUE` 来获得 debug 信息。你可以阅读 `sorl-thumbnail` 的完整文档：http://sorl-thumbnail.readthedocs.org/

用 jQuery 添加 AJAX 动作

现在我们将在你的应用中添加 AJAX 动作。AJAX 源于 Asynchronous JavaScript and XML（异步 JavaScript 和 XML）。这个术语包含一组可以制造异步 HTTP 请求的技术，它包含从服务器异步发送和接收数据，不需要重载整个页面，虽然它的名字里有 XML， 但是 XML 不是必需的。你可以以其他的格式发送或者接收数据，如 JSON， HTML，或者是纯文本。
我们将在图片详情页添加一个供用户点击的链接，表示他们喜欢这张图片。我们将会用 AJAX 来避免重载整个页面。首先，在 `views.py` 中创建一个可供用户点击“喜欢”或“不喜欢”的视图。编辑 images 应用的 `views.py`，将以下代码添加进去：

```
@login_required
@require_POST
def image_like(request):
    image_id = request.POST.get('id')
    action = request.POST.get('action')
    if image_id and action:
        try:
            image = Image.objects.get(id=image_id)
            if action == 'like':
                image.users_like.add(request.user)
            else:
                image.users_like.remove(request.user)
            return JsonResponse({'status':'ok'})
        except:
            pass
    return JsonResponse({'status':'ko'})
```

我们在这个视图中使用了两个装饰器。 `login_required` 装饰器阻止未登录的用户连接到这个视图。`require_GET` 装饰器返回一个 `HttpResponseNotAlloed` 对象（状态吗：405）如果 HTTP 请求不是 GET 。这样就可以只允许 GET 请求来访问这个视图。 Django 同样也提供了 `require_POST` 装饰器来只允许 POST 请求，以及一个可让你传递一组请求方法作为参数的 `require_http_methods` 装饰器。
在这个视图中我们使用了两个 GET 参数：
1. `image_id`:用户操作的 image 对象的 ID
2. `action`: 用户想要执行的动作。我们把它的值设定为 `like` 或者`dislike`
我们在 `Image` 模型的多对多字段 `users_like` 上使用 Django 提供的管理器来添加或者删除对象关系通过调用 `add()`或者 `remove()`方法来执行这些动作。调用 `add()`时传递一个存在于关联模型中的对象集不会重

复添加这个对象，同样，调用 `remove()`时传递一个不存在于关联模型中的对象集什操作也不会执行。另一个有用的多对多管理器是 `clear()`，它将删除所有的关联对象集。
最后，我们使用 Django 提供的 `JsonResponse` 类来将给你定的对象转换为一个 JSON 输出，这个类返回一个带有 `application/json` 内容类型的 HTTP 响应。
编辑 `images` 应用中的 `urls.py`,添加以下 URL 模式：

```
url(r'^like/$', views.image_like, name='like'),
```

加载 jQuery

我们需要在我们的图片详情页中添加 AJAX 功能。我们首先将在 `base.html` 模版中引入 AJAX。编辑 account 应用的 `base.html` 模版，然后将以下代码在`</body>`标签前添加以下代码：

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
<script>
  $(document).ready(function(){
    {% block domready %}
    {% endblock %}
    });
</script>
```

我们从 Google 加载 jQuery 框架，Google 提供了一个在高速内容分发网络中的流行 JavaScript 框架。你也可以自己下载 jQuery，地址：http://jquery.com/ 。然后将下载的文件添加进应用的 `static` 路径下。
我们添加`<script>`标签来引入 JavaScript 代码。`$(dovument).ready()`是一个 jQuery 函数，这个函数会在 DOM 层加载完毕后执行。 DON 源于 Document Object Model。当一个页面被载入时，DOM 会由浏览器创建， DOM 被创建为一个树对象。通过在这个函数中包含我们的代码来确保我们可以与 DOM 中加载的所有 HTML 元素都能进行交互操作。我们的代码仅仅会在 DOM 对象被加载完毕之后执行。
在文档预处理函数中，我们会在模板中引入一个 Django 模板块叫做 **domready**，在扩展了基础模版之后将会引入特定的 JavaScript 。
不要将 JavaScript 代码和 Django 模板标签搞混了。 Django 模板语言是在服务端被渲染并输出最终的 HTML 文档，JavaScript 是在客户端被执行的。在某些情况下，使用 Django 动态生成 JavaScript 很有用。
在这一章中，我们在 Django 模板中引入（include）了 JavaSript 代码。更好的引入方法是加载（load）JavaSript. `js` 文件是作为静态文件被提供的，特别在有大量脚本时尤其如此。

AJAX 请求中的跨站请求攻击（CSRF）

你已经在第二章了解到了跨站请求攻击，在 CSRF 保护激活的情况下， Django 会检查所有 POST 请求中的 CSRF token。当你提交表但时，你可以使用`{% csrf_token %}`模板标签来发送带有 token 的表单。无论如何，像 POST 请求一样对 AJAX 请求传递 CDRF token 有一点点不方便。因此，Django 允许你在你的 AJAX 请求中设置一个定制的 `X-CSRFToken` token 头（header）。这允许你安装一个 jQuery 或者任意 JavaScript 库来自动设置 `X-CSRFToken` 头在每一次请求中。
为了在所有的请求中加入 token ，你需要：
1. 从 `csrftoken` cookie 中检索 CSRF token，它在 CSRF 保护激活的情况下会被设置
2. 使用 `X-CSRFToken` 头发送 token 到 AJAX 中

你可以找到更多关于 CSRF 保护 和 AJAX 的信息：
http://docs.djangoproject.com/en/1.8/ref/csrf/#ajax
在你的 `base.html` 模板中添加最后一段代码：

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
```

```
<script src=" http://cdn.jsdelivr.net/jquery.cookie/1.4.1/jquery.
cookie.min.js "></script>
<script>
  var csrftoken = $.cookie('csrftoken');
  function csrfSafeMethod(method) {
    // these HTTP methods do not require CSRF protection
    return (/^(GET|HEAD|OPTIONS|TRACE)$/.test(method));
}
$.ajaxSetup({
  beforeSend: function(xhr, settings) {
    if (!csrfSafeMethod(settings.type) && !this.crossDomain) {
      xhr.setRequestHeader("X-CSRFToken", csrftoken);
  }
}
});
$(document).ready(function(){
    {% block domready %}
    {% endblock %}
  });
</script>
```

上面这段代码解释如下：

1. 我们从一个公共的 CDN 中载入了一个 jQuery Cookie 插件，这样我们就可以和 cookies 交互。
2. 读取 `csrftoken` cookie
3. 我们将定义 `csrfSafeMethod` 函数来检查一个 HTTP 方法是否安全。安全方法不要求 CSRF 保护，他们分别是 GET, HEAD, OPTIONS, TRACE。
4. 我们用`$.ajaxSetup()`设置了 jQuery AJAX 请求,在每个 AJAX 请求执行前，我们会检查请求方法是否安全和当前请求是否跨域名。如果请求是不安全的，我们将用从 cookie 中获得的值来设置 `X-CSRFToken` 头。这个设置将会应用到所有由 jQuery 执行的 AJAX 请求中

CSRF token 将会在所有的不安全 HTTP 方法的 AJAX 请求中引入，比如 POST, PUT

用 JQuery 执行 AJAX 请求

编辑 images 应用中的 `images/image/detailmhtml` 模板，删除这一行：

```
{% with total_likes=image.users_like.count %}
```

替换为：

```
{% with total_likes=image.users_like.count users_like=image.users_like.all %}
```

用 `image-info` 类属性修改`<div` 元素：

```
       <div class="image-info">
              <div>
                  <span class="count">
                      <span class="total">{{ total_likes }}</span>
                      like{{ total_likes|pluralize }}
                  </span>
                  <a href="#" data-id="{{ image.id }}" data-action="{% if request.user in users_like %}un{%
endif %}like" class="like button">
                      {% if request.user not in users_like %}
```

```
                    Like
                {% else %}
                    Unlike
                {% endif %}
            </a>
        </div>
        {{ image.description|linebreaks }}
    </div>
```

首先，我们添加了一个变量到`{% with %}`模板标签中来保存`image.uers_like.all`查询接的结果来避免执行两次查询。展示喜欢这张图片用户的总数，包含一个“like/unlike”链接。我们检查用户是否在关联对象`user_likes`中，基于当前的用户和图片的关系展示 like 或者 unlike。。我们将以下属性添加进了`<a>` HTML 元素中：

- `data-id`:被展示图片的 ID
- `data-action`: 当用户点击这个链接时执行这个动作。这个动作可以是 like 或者是 unlike

我们将会在向`iamge_like`视图的 AJAX 请求中添加这两个属性值。当一个用户点击`like/unlike`链接时，我们需要在客户端执行以下几个动作：

- 调用 AJAX 视图，并把图片的 ID 和动作参数传递进去
- 如果 AJAX 请求成功， 更新`<a>` HTML 元素的`data-action`属性（like / unlike），根据此来修改它展示的文本
- 更新展示的 likes 的总数

在`images/image/detail.html`模板中添加`domready`块，使用如下 JavaScript 代码：

```
{% block domready %}
    $('a.like').click(function(e){
        e.preventDefault();
        $.post('{% url "images:like" %}',
            {
                id: $(this).data('id'),
                action: $(this).data('action')
            },
            function(data){
                if (data['status'] == 'ok')
                {
                    var previous_action = $('a.like').data('action');

                    // toggle data-action
                    $('a.like').data('action', previous_action == 'like' ? 'unlike' : 'like');
                    // toggle link text
                    $('a.like').text(previous_action == 'like' ? 'Unlike' : 'Like');

                    // update total likes
                    var previous_likes = parseInt($('span.count .total').text());
                    $('span.count .total').text(previous_action == 'like' ? previous_likes + 1 : previous_likes
- 1);
                }
        });

    });
```

```
{% endblock %}
```

这段代码工作流程如下：

1. 我们使用`$.('a.like')` jQuery 选择器来找到所有的 class 属性是 like 的`<a>`标签
2. 我们为点击事件定义了一个控制器函数。这个函数会在用户每次点击 `like/unlike` 时执行
   3.在控制器函数中，我们使用 `e.preventDefault()`来避免`<a>`标签的默认行为。这会阻止链接把我们带到其他地方。
3. 我们使用`$.post()`向服务器执行一个异步 POST 请求。 jQuery 也会提供一个`$.get()`方法来执行 GET 请求和一个低级别的 `$.ajax()`方法。
4. 我们使用 Django 的`{% url %}`模板标签来构建为 AJAX 请求需要的 URL
5. 我们在请求中建立要发送的 POST 参数字典。他们是 Django 视图中期望的 `ID` 和 `action` 参数。我们从`<a>`元素的 `data-id` 和 `data-action` 中获取两个参数的值。
6. 我们定义了一个当 HTTP 应答被接收时的回调函数。它接收一个含有应答内容的数据属性。
7. 我们获取接收数据的 `status` 属性然后检查它的值是否是 `ok`。如果返回的 `data` 是期望中的那样，我们将切换 `data-action` 属性的链接和它的文本内容。这可以让用户取消这个动作。
8. 我们基于执行的动作来增加或者减少 likes 的总数

在你的浏览器中打开一张你上传的图片的详情页，你可以看到初始的 like 统计和一个 `LIKE` 按钮：

Django-5-7

点击 `LIKE` 按钮，你将会看见 likes 的总数上升了，按钮的文本也变成了 `UNLIKE`：

Django-5-8

当你点击 `UNLIKE` 按钮时动作被执行，按钮的文本也会变成 `LIKE`，统计的总数也会据此下降。
在编写 JavaScript 时，特别是在写 AJAX 请求时， 我们建议应该使用一个类似于 Firebug 的工具来调试你的 JavaScript 脚本以及监视 CSS 和 HTML 的变化，你可以下载 Firebug ： http://getfirebug.com/。一些浏览器比如*Chrome*或者*Safari*也包含一些调试 JavaScript 的开发者工具。在那些浏览器中，你可以在网页的任何地方右键然后点击 **Inspect element** 来使用网页开发者工具。

为你的视图创建定制化的装饰器

我们将会限制我们的 AJAX 视图只接收由 AJAX 发起的请求。Django Request 对象提供了一个 `is_ajax()`方法， 这个方法会检查请求是否带有 `XMLHttpRequest`,也就是说，会检查这个请求是否是一个 AJAX 请求。这个值被设置在 `HTTP_X_REQUESTED_WITH HTTP` 头中， 这个头被大多数的由 JavaScript 库发起的 AJAX 请求包含。
我们将在我们的视图中创建一个装饰器，来检测 `HTTP_X_RQUESTED_WITH` 头。装饰器是一个可以接收一个函数为参数的函数，并且它可以在不改变作为参数的函数的情况下，拓展此函数的功能。如果装饰器的概念对你来说还很陌生，在继续阅读之前你或许可以看看这个：
https:www.python.org/dev/peps/pep-0318/。
由于我们的装饰器将会是通用的，它将被应用到任何视图中，所以我们在我们的项目中将创建一个 `common`Python 包，在 bookmarklet 项目中创建如下路径：

- common/
- *_init_*.py
- decorators.py

编辑 `decorators.py`,添加如下代码：

```
from django.http import HttpResponseBadRequest

def ajax_required(f):
    def wrap(request, *args, **kwargs):
            if not request.is_ajax():
                return HttpResponseBadRequest()
            return f(request, *args, **kwargs)
    wrap.__doc__=f.__doc__
    wrap.__name__=f.__name__
    return wrap
```

这是我们定制的 `ajax_required` 装饰器。它定义一个当请求不是 AJAX 时返回 `HttpResponseBadRequest`（HTTP 400）对象的 wrap 函数，否则它将返回一个被装饰了的对象。
现在你可以编辑 images 应用的 `views.py`，为你的 `image_like` AJAX 视图添加这个装饰器：

```
from common.decorators import ajax_required

@ajax_required
@login_required
@require_POST
def image_like(request):
    # ...
```

如果你直接在你的浏览器中访问 `http://127.0.0.1:8000/images/like/`,你将会得到一个 HTTP 400 的错误。
如果你发现你正在视图中执行重复的检查，请为你的视图创建装饰器

在你的列表视图中添加 AJAX 分页

我们需要在你的网站中列出所有的被标签的图片。我们将要使用 AJAX 分页来建立一个不受限的滚屏功能。不受限的滚屏是在用户滚动到底部时，自动加载下一页的结果来实现的。
我们将实现一个图片列表视图，这个视图既可以支持标准的浏览器请求，也支持包含分页的 AJAX 请求。当用户首次加载列表页时，我们展示第一页的图片。当用户滚动到底部时，我们用 AJAX 加载下一页的内容，然后将内容加入到页面的底部。
编辑 images 应用的 `views.py`，添加以下代码：

```
from django.http import HttpResponse
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger

@login_required
def image_list(request):
    images = Image.objects.all()
    paginator = Paginator(images, 8)
    page = request.GET.get('page')
    try:
        images = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer deliver the first page
        images = paginator.page(1)
    except EmptyPage:
        if request.is_ajax():
            # If the request is AJAX and the page is out of range return an empty page
```

```
            return HttpResponse('')
        # If page is out of range deliver last page of results
        images = paginator.page(paginator.num_pages)
    if request.is_ajax():
        return render(request,
                      'images/image/list_ajax.html',
                      {'section': 'images', 'images': images})
    return render(request,
                  'images/image/list.html',
                  {'section': 'images', 'images': images})
```

在这个视图中，我们创建一个查询集（QuerySet）来从数据库中获得所有的图片。然后我们创建了一个 `Paginator` 对象来分页查询结果，每页有八张图片。如果请求的页面超出范围了，我们将会得到一个 `EmptyPage` 异常，在这种情况下并且请求又是由 AJAX 发起的话，我们将会返回一个空的 `HttpResponse`，这将帮助我们在客户端停止 AJAX 分页，我们将会把结果渲染给两个不同的模板：

1. 对于 AJAX 请求，我们渲染 `list_ajax.html` 模板。这个模板将只会包含我们请求页面的图片
2. 对于标准请求： 我们渲染 `list.html` 模板。这个模板将会继承 `base.html` 来展示整个页面，并且 `list_ajax.html` 页面也会被引入在其中。

编辑 images 应用的 `urls.py`,添加以下 URL 模式：

```
url(r'^$', views.image_list, name='list'),
```

最后，我们需要创建我们在上面提到的模板。在 images/image/ 下创建一个名为'list_ajax.html'的模板，添加以下代码：

```
{% load thumbnail %}

{% for image in images %}
    <div class="image">
        <a href="{{ image.get_absolute_url }}">
            {% thumbnail image.image "300x300" crop="100%" as im %}
                <a href="{{ image.get_absolute_url }}">
                    ![]({{ im.url }})
                </a>
            {% endthumbnail %}
        </a>
        <div class="info">
            <a href="{{ image.get_absolute_url }}" class="title">{{ image.title }}</a>
        </div>
    </div>
{% endfor %}
```

这个模板将用于展示一组图片，它会作为结果返回给 AJAX 请求。之后，在相同路径下创建'list.html'模板，添加以下代码：

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

{% block title %}Images bookmarked{% endblock %}

{% block content %}
    <h1>Images bookmarked</h1>
```

```
    <div id="image-list">
        {% include "images/image/list_ajax.html" %}
    </div>
{% endblock %}
```

这个列表模板继承了'base.html模板。为了避免重复编码，我们引入了 list_ajax.html 模板。这个 listmhtml 模板将会有一段 JavaScript 代码来加载当滚动到底部时的额外页面。
将以下代码添加进 list.html 模板中：

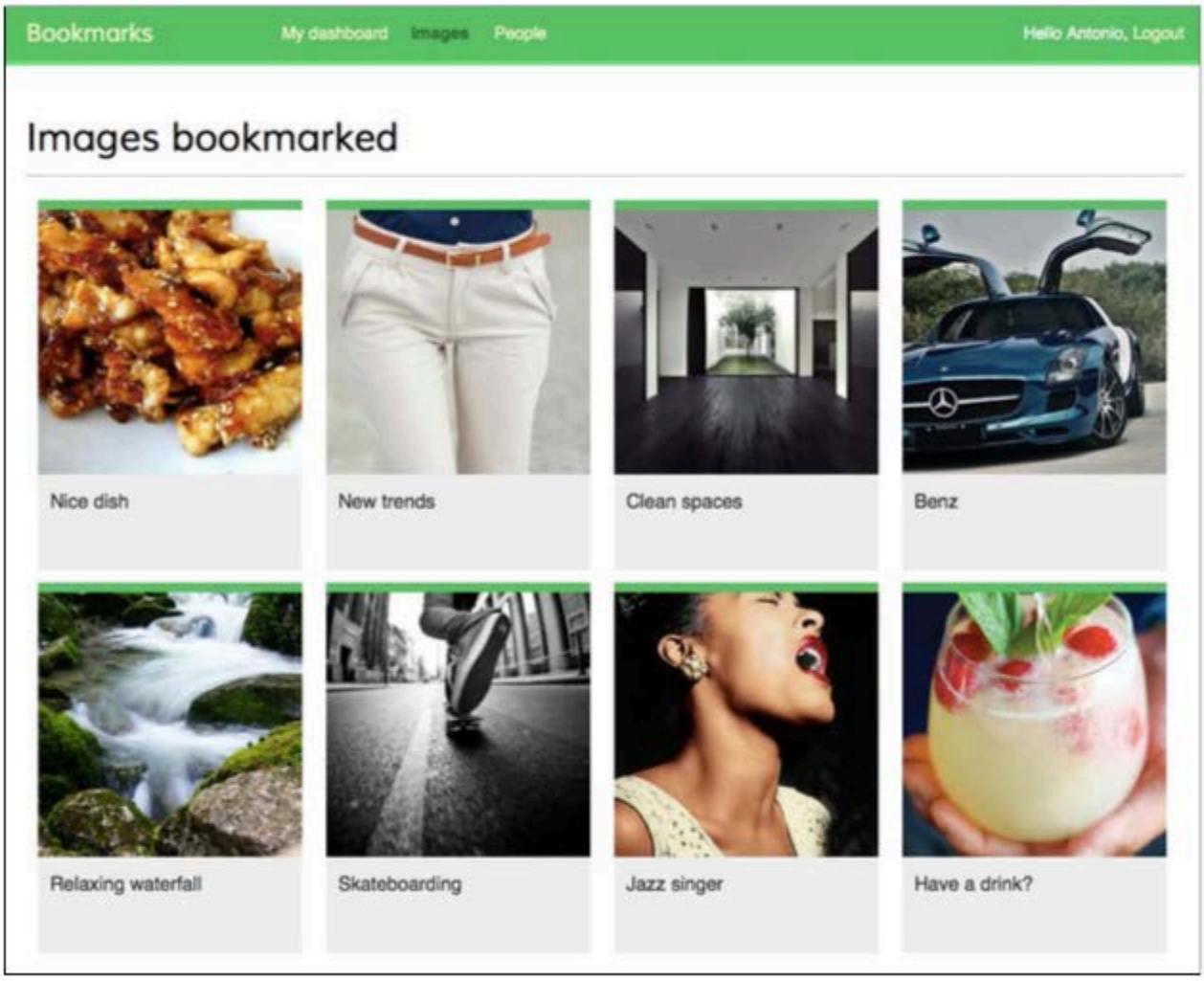
```
{% block domready %}
    var page = 1;
    var empty_page = false;
    var block_request = false;

    $(window).scroll(function() {
        var margin = $(document).height() - $(window).height() - 200;
        if  ($(window).scrollTop() > margin && empty_page == false && block_request == false) {
            block_request = true;
            page += 1;
            $.get('?page=' + page, function(data) {
                if(data == '')
                {
                    empty_page = true;
                }
                else {
                    block_request = false;
                    $('#image-list').append(data);
                }
            });
        }
    });
{% endblock %}
```

这段代码实现了不受限的滚屏功能。我们在 base.html 中定义的 domready 块中引入了 JavaScript 代码，这段代码的工作流程如下：

1. 我们定义了如下几个变量：
   - page：保存当前的页码
   - empt_page:让我们知道用户是否到了最后一页，然后接收一个空页面。只要接收到了一个空页面，我们就会停止发送额外的 AJAX 请求，因为我们确定此时已经没有结果了。
   - block_requests：当有进程中有 AJAX 请求时，阻止额外的请求。
2. 我们使用$(window).scroll()来捕获滚动事件，然后我们为此定义了一个控制器函数。
3. 我们计算边框变量来得到文档高度和窗口高度的差值，因为这个差值是用户将要滚动的内容的高度。我们从结果当中减去 200，这样我们就可以在用户接近底部 200pixels 时加载下一页的内容。
4. 我们只在以下两种条件满足时发送 AJAX 请求：没有其他 AJAX 请求被正在被执行时（译者注：就是同时只有一个 AJAX 请求）（block_request 必须是 false）,用户也没有到达页面底部（empty_page 也必须是 false）。
5. 我们将 block_request 设为 True 来避免滚动时间触发额外的 AJAX 请求，然后我们会在请求下一页时增加一次 page 计数。
6. 我们使用$.get()来执行一次 AJAX GET 请求，然后我们在一个叫做 data 的变量中接收 HTML 响应。这里有两种情况。

- 响应没有内容：我们已经到了结果的末尾，所以这里没有更多的页面来供我们加载。我们把 `empty_page` 设为 `True` 来阻止加载更多的 AJAX 请求。
- 响应含有数据：我们将数据添加到 id 为 `image-list` 的 HTML 元素中，当用户滚动到底部时页面将直接扩展添加的结果。

在浏览器中访问 `http://127.0.0.1:8000/images/`,你会看到你之前添加的一组图片，看起来像这样：



Django-5-9

滚动到底部将会加载下一页。确定你已经使用书签添加了多于 8 张图片，因为我们每一页展示的是 8 张图片。记得使用 Firebug 或者类似的工具来跟踪 AJAX 请求和调试你的 JavaScript 代码。
最后，编辑 `account` 应用中的 `base.html` 模板，为主菜单添加图片项：

```
<li {% if section == "images" %}class="selected"{% endif %}><a href="{% url "images:list" %}">Images</a></li>
```

现在你可以从主菜单连接到图片列表了。

总结

在这一章中，我们创建了一个 JavaScript 书签来从其他网站分享图片到我们的网站。你已经用 jQuery 实现了 AJAX 视图，还添加了 AJAX 分页。
在下一章中，将会教你如何创建一个粉丝系统和一个活动流。你将和通用关系、信号、与反规范化打交道。你也将学习如何在 Django 中使用 Redis。

第六章 跟踪用户动作

在上一章中，你在你的项目中实现了 AJAX 视图（views），通过使用 jQuery 并创建了一个 JavaScript 书签在你的平台中分享别的网站的内容。

在本章中，你会学习如何创建一个粉丝系统以及创建一个用户活动流（activity stream）。你会学习到 Django 信号（signals）的工作方式以及在你的项目中集成 Redis 快速 I/O 仓库用来存储视图（views）项。
本章将会覆盖以下几点：

- 通过一个中介模型（intermediate model）创建多对对的关系
- 创建 AJAX 视图（views）
- 创建一个活动流（activity stream）应用
- 给模型（modes）添加通用关系
- 取回对象的最优查询集（QuerySets）
- 使用信号（signals）给非规范化的计数
- 存储视图（views）项到 Redis 中

创建一个粉丝系统

我们将要在我们的项目中创建一个粉丝系统。我们的用户在平台中能够彼此关注并且跟踪其他用户的分享。这个关系在用户中的是多对多的关系，一个用户能够关注多个用户并且能被多个用户关注。

通过一个中介模型（intermediate model）（intermediary model）创建多对对的关系

在上一章中，你创建了多对对关系通过在其中一个有关联的模型（model）上添加了一个 *ManyToManyField* 然后让 Django 为这个关系创建了数据库表。这种方式支持大部分的场景，但是有时候你需要为这种关系创建一个中介模型（intermediate model）。创建一个中介模型（intermediate model）是非常有必要的当你想要为当前关系存储额外的信息，例如当前关系创建的时间点或者一个描述当前关系类型的字段。
我们会创建一个中介模型（intermediate model）用来在用户之间构建关系。有两个原因可以解释为什么我们要用一个中介模型（intermediate model）：

- 我们使用 Django 提供的 *user* 模型（model）并且我们想要避免修改它。
- 我们想要存储关系建立的时间

编辑你的 *account* 应用中的 *models.py* 文件添加如下代码：

```
from django.contrib.auth.models import User
   class Contact(models.Model):
       user_from = models.ForeignKey(User,
                                     related_name='rel_from_set')
       user_to = models.ForeignKey(User,
                                   related_name='rel_to_set')
       created = models.DateTimeField(auto_now_add=True,
                                      db_index=True)
       class Meta:
           ordering = ('-created',)
       def __str__(self):
           return '{} follows {}'.format(self.user_from,
self.user_to)
```

这个 *Contact* 模型我们将会给用户关系使用。它包含以下字段：

- user_form：一个 *ForeignKey* 指向创建关系的用户
- user_to：一个 *ForeignKey* 指向被关注的用户
- created：一个 `auto_now_add=True` 的 *DateTimeField* 字段用来存储关系创建时的时间

在 *ForeignKey* 字段上会自动生成一个数据库索引。我们使用 `db_index=True` 来创建一个数据库索引给 *created* 字段。这会提升查询执行的效率当通过这个字段对查询集（QuerySets）进行排序的时候。
使用 ORM ，我们可以创建一个关系给一个用户 *user1* 关注另一个用户 *user2*，如下所示：

```
user1 = User.objects.get(id=1)
user2 = User.objects.get(id=2)
Contact.objects.create(user_from=user1, user_to=user2)
```

关系管理器 *rel_form_set* 和 *rel_to_set* 会返回一个查询集（QuerySets）给 *Contace* 模型（model）。为了
从 *User* 模型（model）中存取最终的关系侧，*Contace* 模型（model）会期望 *User* 包含一个 *ManyToManyField*，如下所示（译者注：以下代码是作者假设的，实际上 *User* 不会包含以下代码）：

```
following = models.ManyToManyField('self',
                                   through=Contact,
                                   related_name='followers',
                                   symmetrical=False)
```

在这个例子中，我们告诉 Django 去使用我们定制的中介模型（intermediate model）来创建关系通过给 *ManyToManyField* 添加 `through=Contact`。这是一个从 *User* 模型到本身的多对对关系：我们在 *ManyToMnyfIELD* 字段中引用 `'self'`来创建一个关系给相同的模型（model）。
当你在多对多关系中需要额外的字段，创建一个定制的模型(model)，一个关系侧就是一个 *ForeignKey*。添加一个 *ManyToManyField* 在其中一个有关联的模型（models）中然后通过在 *through* 参数中包含该中介模型（intermediate model）指示 Django 去使用你的定制中介模型（intermediate model）。
如果 *User* 模型（model）是我们应用的一部分，我们可以添加以上的字段给模型（model）（译者注：所以说，上面的代码是作者假设存在）。但实际上，我们无法直接修改 *User* 类，因为它是属于 *django.contrib.auth* 应用的。我们将要做些轻微的改动，给 *User* 模型动态的添加这个字段。编辑 *account* 应用中的 *model.py* 文件，添加如下代码：

```
# Add following field to User dynamically
User.add_to_class('following',
                  models.ManyToManyField('self',
                                         through=Contact,
                                         related_name='followers',
                                         symmetrical=False))
```

在以上代码中，我们使用 Django 模型（models）的 `add_to_class()`方法给 *User* 模型（model）添加 *monkey-patch*(译者注：猴子补丁 *Monkey patch* 就是在运行时对已有的代码进行修改，而不需要修改原始代码）。你需要意识到，我们不推荐使用 `add_to_class()`为模型（models）添加字段。我们在这个场景中利用这种方法是因为以下的原因：

- 我们可以非常简单的取回关系对象使用 Django ORM 的 `user.followers.all()`以及 `user.following.all()`。我们使用中介（intermediary） *Contact* 模型（model）可以避免复杂的查询例如使用到额外的数据库操作 *joins*，如果在我们的定制 *Profile* 模型（model）中定义过了关系。
- 这个多对多关系的表将会被创建通过使用 *Contact* 模型（model）。因此，动态的添加 *ManyToManyField* 将不会对 Django *User* 模型（model）的数据库进行任意改变。
- 我们避免了创建一个定义的用户模型（model），保持了所有 Django 内置 *User* 的特性。

请记住，在大部分的场景中，在我们之前创建的 *Profile* 模型（model）添加字段是更好的方法，可以替代在 *User* 模型（model）上打上 *monkey-patch*。Django 还允许你使用定制的用户模型（models）。如果你想要使用你的定制用户模型（model），可以访问 https://docs.djangoproject.com/en/1.8/topics/auth/customizing/#specifying-a-custom-user-model 获得更多信息。
你能看到上述代码中的关系包含了 `symmetrical=Flase`来定义一个非对称（non-symmetric）关系。这表示如果我关注了你，你不会自动的关注我。
当你使用了一个中介模型（intermediate model）给多对多关系，一些关系管理器的方法将不可用，例如：`add()`，`create()`以及 `remove()`。你需要创建或删除中介模型（intermediate model）的实例来代替。

运行如下命令来生成 *account* 应用的初始迁移：

```
python manage.py makemigrations account
```

你会看到如下输出：

```
Migrations for 'account':
    0002_contact.py:
      - Create model Contact
```

现在继续运行以下命令来同步应用到数据库中：

```
python manage.py migrate account
```

你会看到如下内容包含在输出中：

```
Applying account.0002_contact... OK
```

Contact 模型（model）现在已经被同步进了数据库，我们可以在用户之间创建关系。但是，我们的网站还没有提供一个方法来浏览用户或查看详细的用户 profile。让我们为 *User* 模型构建列表和详情视图（views）。

为用户 profiles 创建列表和详情视图（views）

打开 *account* 应用中的 *views.py* 文件添加如下代码：

```
from django.shortcuts import get_object_or_404
from django.contrib.auth.models import User
  @login_required
  def user_list(request):
      users = User.objects.filter(is_active=True)
      return render(request,
                    'account/user/list.html',
                    {'section': 'people',
                     'users': users})
  @login_required
  def user_detail(request, username):
      user = get_object_or_404(User,
                               username=username,
                               is_active=True)
      return render(request,
                    'account/user/detail.html',
                    {'section': 'people',
                     'user': user})
```

以上是 *User* 对象的简单列表和详情视图（views）。`user_list` 视图（view）获得了所有的可用用户。Django *User* 模型（model）包含了一个标志（flag）`is_active` 来指示用户账户是否可用。我们通过 `is_active=True` 来过滤查询只返回可用的用户。这个视图（vies）返回了所有结果，但是你可以改善它通过添加页码，这个方法我们在 *image_list* 视图（view）中使用过。
user_detail 视图（view）使用 `get_object_or_404()`快捷方法来返回所有可用的用户通过传入的用户名。当使用传入的用户名无法找到可用的用户这个视图（view）会返回一个 HTTP 404 响应。
编辑 *account* 应用的 *urls.py* 文件，为以上两个视图（views）添加 URL 模式，如下所示：

```
urlpatterns = [
```

```
    # ...
    url(r'^users/$', views.user_list, name='user_list'),
    url(r'^users/(?P<username>[-\w]+)/$',
        views.user_detail,
        name='user_detail'),
]
```

我们会使用 `user_detail` URL 模式来给用户生成规范的 URL。你之前就在模型（model）中定义了一个 `get_absolute_url()`方法来为每个对象返回规范的 URL。另外一种方式为一个模型（model）指定一个 URL 是为你的项目添加 *ABSOLUTE_URL_OVERRIDES* 设置。
编辑项目中的 *setting.py* 文件，添加如下代码：

```
ABSOLUTE_URL_OVERRIDES = {
    'auth.user': lambda u: reverse_lazy('user_detail',
                                        args=[u.username])
}
```

Django 会为所有出现在 *ABSOLUTE_URL_OVERRIDES* 设置中的模型（models）动态添加一个 `get_absolute_url()`方法。这个方法会给设置中指定的模型返回规范的 URL。我们给传入的用户返回 *user_detail* URL。现在你可以在一个 User 实例上使用 `get_absolute_url()`来取回他自身的规范 URL。打开 Python shell 输入命令 `python manage.py shell` 运行以下代码来进行测试：

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.latest('id')
>>> str(user.get_absolute_url())
'/account/users/ellington/'
```

返回的 URL 如同期望的一样。我们需要为我们刚才创建的视图（views）创建模板（templates）。在 *account* 应用下的*templates/account/目录下添加以下目录和文件：

```
/user/
    detail.html
    list.html
```

编辑 *account/user/list.html* 模板（template）给它添加如下代码：

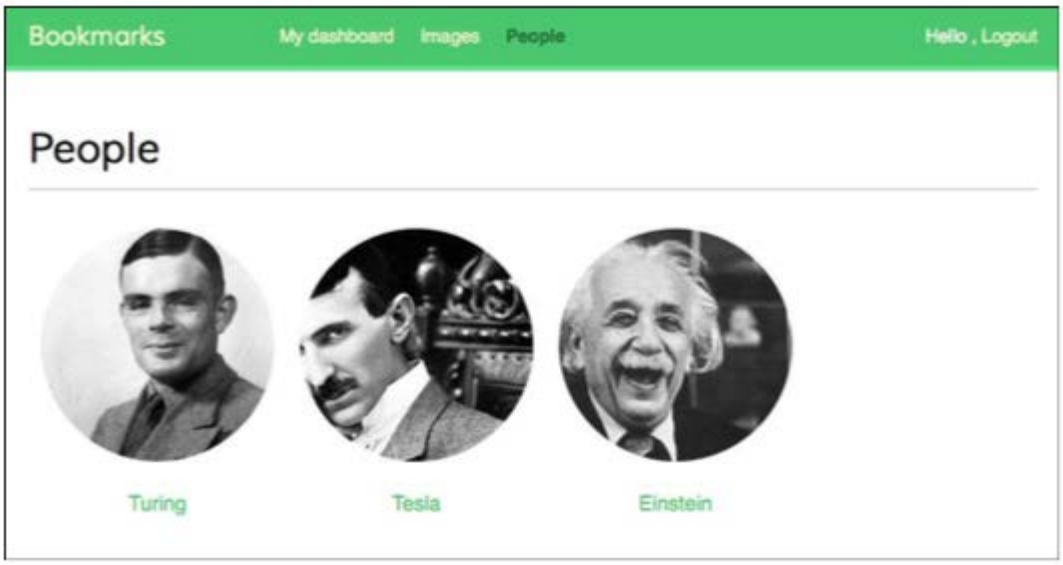
```
{% extends "base.html" %}
{% load thumbnail %}
{% block title %}People{% endblock %}
{% block content %}
    <h1>People</h1>
    <div id="people-list">
       {% for user in users %}
         <div class="user">
            <a href="{{ user.get_absolute_url }}">
             {% thumbnail user.profile.photo "180x180" crop="100%" as im %}
               ![]({{ im.url }})
             {% endthumbnail %}
           </a>
           <div class="info">
             <a href="{{ user.get_absolute_url }}" class="title">
               {{ user.get_full_name }}
```

```
            </a>
          </div>
        </div>
      {% endfor %}
    </div>
{% endblock %}
```

这个模板（template）允许我们在网站中排列所有可用的用户。我们对给予的用户进行迭代并且使用`{% thumbnail %}模板（template）标签（tag）来生成 profile 图片缩微图。
打开项目中的 *base.html* 模板（template），在以下菜单项的 *href* 属性中包含 *user_list*URL：

```
<li {% if section == "people" %}class="selected"{% endif %}>
    <a href="{% url "user_list" %}">People</a>
</li>
```

通过命令 `python manage.py runserver` 启动开发服务器然后在浏览器打开 http://127.0.0.1:8000/account/users/ 。你会看到如下所示的用户列：



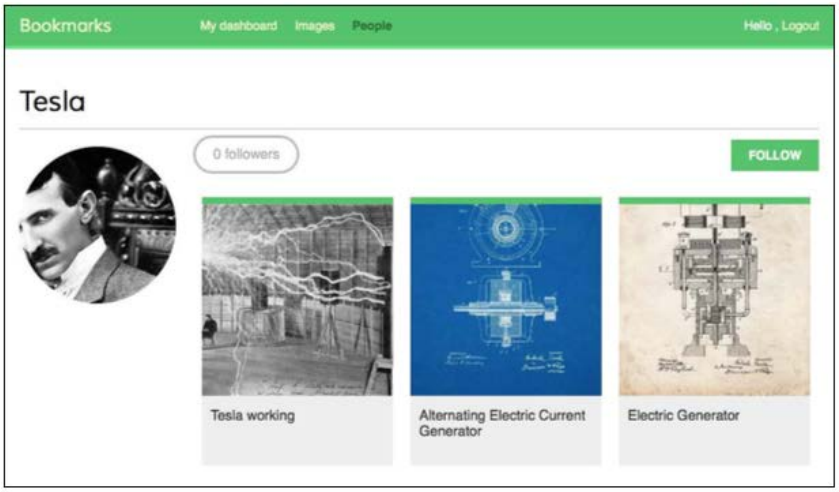
django-6-1

（译者注：图灵，特斯拉，爱因斯坦，都是大牛啊）
编辑 *account* 应用下的 *account/user/detail.html* 模板，添加如下代码：

```
{% extends "base.html" %}
{% load thumbnail %}
{% block title %}{{ user.get_full_name }}{% endblock %}
{% block content %}
    <h1>{{ user.get_full_name }}</h1>
    <div class="profile-info">
    {% thumbnail user.profile.photo "180x180" crop="100%" as im %}
        ![]({{ im.url }})
    {% endthumbnail %}
    </div>
    {% with total_followers=user.followers.count %}
    <span class="count">
        <span class="total">{{ total_followers }}</span>
        follower{{ total_followers|pluralize }}
    </span>
```

```
    <a href="#" data-id="{{ user.id }}" data-action="{% if request.user in user.followers.all %}un{%
endif %}follow" class="followbutton">
        {% if request.user not in user.followers.all %}
            Follow
        {% else %}
            Unfollow
        {% endif %}
    </a>
    <div id="image-list" class="image-container">
        {% include "images/image/list_ajax.html" with images = user.images_create.all %}
    </div>
    {% endwith %}
{% endblock %}
```

在详情模板（template）中我们展示用户 profile 并且我们使用`{% thumbnail %}`模板（template）标签（tag）来显示 profile 图片。我们显示粉丝的总数以及一个链接可以 *follow/unfollow* 该用户。我们会隐藏关注链接当用户在查看他们自己的 profile，防止用户自己关注自己。我们会执行一个 AJAX 请求来 *follow/unfollow* 一个指定用户。我们给 `<a>`HTML 元素添加 *data-id* 和 *data-action* 属性包含用户 ID 以及当该链接被点击的时候会执行的初始操作，*follow/unfollow* ，这个操作依赖当前页面的展示的用户是否已经被正在浏览的用户所关注。我们展示当前页面用户的图片书签通过 *list_ajax.html* 模板。
再次打开你的浏览器，点击一个拥有图片书签的用户链接，你会看到一个 profile 详情如下所示：



django-6-2

创建一个 AJAX 视图（view）来关注用户

我们将会创建一个简单的视图（view）使用 AJAX 来 *follow/unfollow* 用户。编辑 *account* 应用中的 *views.py* 文件添加如下代码：

```
from django.http import JsonResponse
from django.views.decorators.http import require_POST
from common.decorators import ajax_required
from .models import Contact
@ajax_required
@require_POST
@login_required
def user_follow(request):
    user_id = request.POST.get('id')
    action = request.POST.get('action')
    if user_id and action:
```

```
        try:
            user = User.objects.get(id=user_id)
            if action == 'follow':
                Contact.objects.get_or_create(
                    user_from=request.user,
                    user_to=user)
            else:
                Contact.objects.filter(user_from=request.user,
                                       user_to=user).delete()
            return JsonResponse({'status':'ok'})
        except User.DoesNotExist:
            return JsonResponse({'status':'ko'})
    return JsonResponse({'status':'ko'})
```

user_follow 视图（view）有点类似与我们之前创建的 *image_like* 视图（view）。因为我们使用了一个定制中介模型（intermediate model）给用户的多对多关系，所以 *ManyToManyField* 管理器默认的 `add()` 和 `remove()`方法将不可用。我们使用中介 *Contact* 模型（model）来创建或删除用户关系。
在 *account* 应用中的 *urls.py* 文件中导入你刚才创建的视图（view）然后为它添加 URL 模式：

```
    url(r'^users/follow/$', views.user_follow, name='user_follow'),
```

请确保你放置的这个 URL 模式的位置在 *user_detail*URL 模式之前。否则，任何对*/users/follow/* 的请求都会被 *user_detail* 模式给正则匹配然后执行。请记住，每一次的 HTTP 请求 Django 都会对每一条存在的 URL 模式进行匹配直到第一条匹配成功才会停止继续匹配。
编辑 *account* 应用下的 *user/detail.html* 模板添加如下代码：

```
{% block domready %}
    $('a.follow').click(function(e){
      e.preventDefault();
      $.post('{% url "user_follow" %}',
        {
          id: $(this).data('id'),
          action: $(this).data('action')
        },
        function(data){
          if (data['status'] == 'ok') {
            var previous_action = $('a.follow').data('action');

            // toggle data-action
            $('a.follow').data('action',
              previous_action == 'follow' ? 'unfollow' : 'follow');
            // toggle link text
            $('a.follow').text(
              previous_action == 'follow' ? 'Unfollow' : 'Follow');

            // update total followers
            var previous_followers = parseInt(
              $('span.count .total').text());
            $('span.count .total').text(previous_action == 'follow' ? previous_followers + 1 :
previous_followers - 1);
          }
```

```
        }
      });
    });
{% endblock %}
```

这段 JavaScript 代码执行 AJAX 请求来关注或不关注一个指定用户并且触发 *follow/unfollow* 链接。我们使用 jQuery 去执行 AJAX 请求的同时会设置 *follow/unfollow* 两种链接的 *data-aciton* 属性以及 HTML`<a>`元素的文本基于它上一次的值。当 AJAX 操作执行完成，我们还会对显示在页面中的粉丝总数进行更新。打开一个存在的用户的详情页面，然后点击 **Follow** 链接尝试下我们刚才构建的功能是否正常。

创建一个通用的活动流（activity stream）应用

许多社交网站会给他们的用户显示一个活动流（activity stream），这样他们可以跟踪其他用户在平台中的操作。一个活动流(activity stream)是一个用户或一个用户组最近活动的列表。举个例子，*Facebook* 的 *News Feed* 就是一个活动流（activity stream）。用户 X 给 Y 图片打上了书签或者用户 X 关注了用户 Y 也是例子操作。我们将会构建一个活动流（activity stream）应用这样每个用户都能看到他关注的用户最近进行的交互。为了做到上述功能，我们需要一个模型（modes）来保存用户在网站上的操作执行，还需要一个简单的方法来添加操作给 feed。
运行以下命令在你的项目中创建一个新的应用命名为 *actions*：

```
django-admin startapp actions
```

在你的项目中的 *settings.py* 文件中的 *INSTALLED_APPS* 设置中添加*'actions'*，这样可以让 Django 知道这个新的应用是可用状态：

```
INSTALLED_APPS = (
    # ...
    'actions',
)
```

编辑 *actions* 应用下的 *models.py* 文件添加如下代码：

```
from django.db import models
from django.contrib.auth.models import User

class Action(models.Model):
    user = models.ForeignKey(User,
                             related_name='actions',
                             db_index=True)
    verb = models.CharField(max_length=255)
    created = models.DateTimeField(auto_now_add=True,
                                   db_index=True)

    class Meta:
        ordering = ('-created',)
```

这个 *Action* 模型（model）将会用来记录用户的活动。模型（model）中的字段解释如下：

- user：执行该操作的用户。这个一个指向 Django *User* 模型（model）的 *ForeignKey*。
- verb：这是用户执行操作的动作描述。
- created：这个时间日期会在动作执行的时候创建。我们使用 `auto_now_add=True` 来动态设置它为当前的时间当这个对象第一次被保存在数据库中。

通过这个基础模型（model），我们只能够存储操作例如用户 X 做了哪些事情。我们需要一个额外的 *ForeignKey* 字段为了保存操作会涉及到的一个 *target*（目标）对象，例如用户 X 给图片 Y 打上了暑期那或者用户 X 现在关注了用户 Y。就像你之前知道的，一个普通的 *ForeignKey* 只能指向一个其他的模型（model）。但是，我们需要一个方法，可以让操作的 *target*（目标）对象是任何一个已经存在的模型（model）的实例。这个场景就由 Django 内容类型框架来上演。

使用内容类型框架

Django 包含了一个内容类型框架位于 *django.contrib.contenttypes*。这个应用可以跟踪你的项目中所有的模型（models）以及提供一个通用接口来与你的模型（models）进行交互。
当你使用 *startproject* 命令创建一个新的项目的时候这个 *contenttypes* 应用就被默认包含在 *INSTALLED_APPS* 设置中。它被其他的 *contrib* 包使用，例如认证(authentication）框架以及 admin 应用。
contenttypes 应用包含一个 *ContentType* 模型（model）。这个模型（model）的实例代表了你的应用中真实存在的模型（models），并且新的 *ContentTYpe* 实例会动态的创建当新的模型（models）安装在你的项目中。*ContentType* 模型（model）有以下字段：

- app_label：模型（model）属于的应用名，它会自动从模型（model）*Meta* 选项中的 *app_label* 属性获取到。举个例子：我们的 *Image* 模型（model）属于 *images* 应用
- model：模型（model）类的名字
- name：模型的可读名，它会自动从模型（model）*Meta* 选项中的 *verbose_name* 获取到。

让我们看一下我们如何实例化 *ContentType* 对象。打开 Python 终端使用 `python manage.py shell` 命令。你可以获取一个指定模型（model）对应的 *ContentType* 对象通过执行一个带有 *app_label* 和 *model* 属性的查询，例如：

```
>>> from django.contrib.contenttypes.models import ContentType
>>> image_type = ContentType.objects.get(app_label='images',model='image')
>>> image_type
<ContentType: image>
```

你还能反过来获取到模型（model）类从一个 *ContentType* 对象中通过调用它的 `model_class()` 方法：

```
>>> from images.models import Image
>>> ContentType.objects.get_for_model(Image)
<ContentType: image>
```

以上就是内容类型的一些例子。Django 提供了更多的方法来使用他们进行工作。你可以访问 https://docs.djangoproject.com/en/1.8/ref/contrib/contenttypes/ 找到关于内容类型框架的官方文档。

添加通用的关系给你的模型（models）

在通用关系中 *ContentType* 对象扮演指向模型（model）的角色被关联所使用。你需要 3 个字段在模型（model）中组织一个通用关系：

- 一个 *ForeignKey* 字段 *ContentType*。这个字段会告诉我们给这个关联的模型（model）。
- 一个字段用来存储被关联对象的 primary key。这个字段通常是一个 *PositiveIntegerField* 用来匹配 Django 自动的 primary key 字段。
- 一个字段用来定义和管理通用关系通过使用前面的两个字段。内容类型框架提供一个 *GenericForeignKey* 字段来完成这个目标。

编辑 *actions* 应用的 *models.py* 文件，添加如下代码：

```
from django.db import models
from django.contrib.auth.models import User
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey
class Action(models.Model):
```

```
    user = models.ForeignKey(User,
                             related_name='actions',
                             db_index=True)
    verb = models.CharField(max_length=255)
    target_ct = models.ForeignKey(ContentType,
                                  blank=True,
                                  null=True,
                                  related_name='target_obj')
    target_id = models.PositiveIntegerField(null=True,
                                            blank=True,
                                            db_index=True)
    target = GenericForeignKey('target_ct', 'target_id')
    created = models.DateTimeField(auto_now_add=True,
                                   db_index=True)
    class Meta:
        ordering = ('-created',)
```

我们给 *Action* 模型添加了以下字段：

- target_ct：一个 *ForeignKey* 字段指向 *ContentType* 模型（model）。
- target_id：一个 *PositiveIntegerField* 用来存储被关联对象的 primary key。
- target：一个 *GenericForeignKey* 字段指向被关联的对象基于前面两个字段的组合之上。

Django 没有创建任何字段在数据库中给 *GenericForeignKey* 字段。只有 *target_ct* 和 *target_id* 两个字段被映射到数据库字段。两个字段都有 `blank=True` 和 `null=True` 属性所以一个 *target*（目标）对象不是必须的当保存 *Action* 对象的时候。

你可以让你的应用更加灵活通过使用通用关系替代外键当它对拥有一个通用关系有意义。

运行以下命令来创建初始迁移为这个应用：

```
python manage.py makemigrations actions
```

你会看到如下输出：

```
    Migrations for 'actions':
        0001_initial.py:
            - Create model Action
```

接着，运行下一条命令来同步应用到数据库中：

```
python manage.py migrate
```

这条命令的输出表明新的迁移已经被应用：

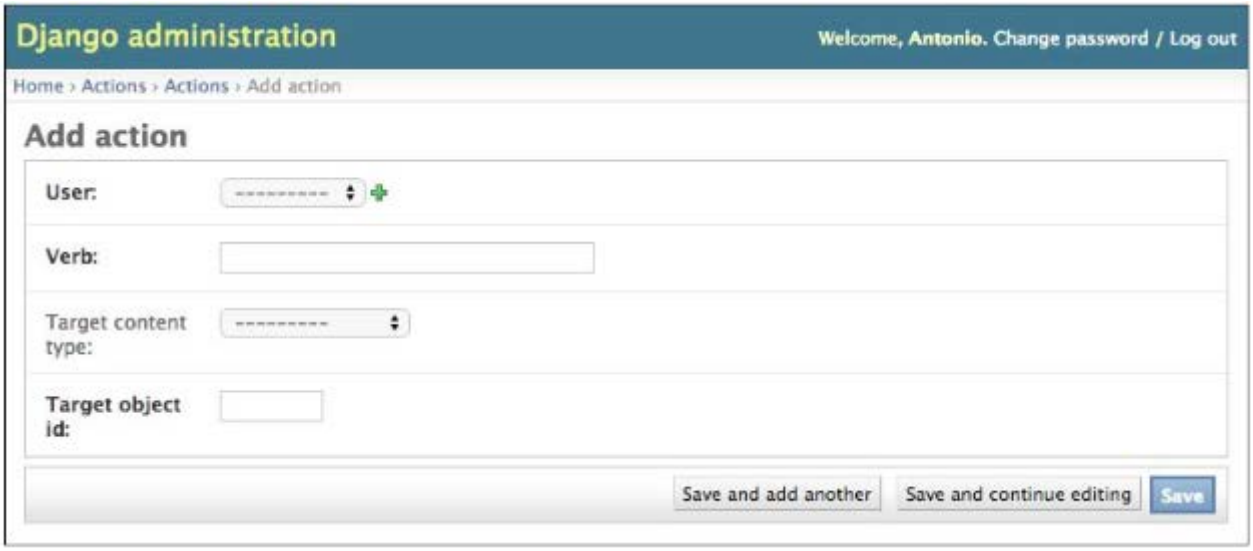
```
Applying actions.0001_initial... OK
```

让我们在管理站点中添加 *Action* 模型（model）。编辑 *actions* 应用的 *admin.py* 文件，添加如下代码：

```
from django.contrib import admin
from .models import Action

class ActionAdmin(admin.ModelAdmin):
    list_display = ('user', 'verb', 'target', 'created')
    list_filter = ('created',)
    search_fields = ('verb',)
```

```
admin.site.register(Action, ActionAdmin)
```

你已经将 *Action* 模型（model）注册到了管理站点中。运行命令 `python manage.py runserver` 来初始化开发服务器然后在浏览器中打开 http://127.0.0.1:8000/admin/actions/action/add/ 。你会看到如下页面可以创建一个新的 *Action* 对象：



django-6-3

如你所见，只有 *target_ct* 和 *target_id* 两个字段是映射为真实的数据库字段显示，并且 *GenericForeignKey* 字段不在这儿出现。*target_ct* 允许你选择任何一个在你的 Django 项目中注册的模型（models）。你可以限制内容类型从一个限制的模型（models）集合中选择通过在 *target-ct* 字段中使用 *limit_choices_to* 属性：*limit_choices_to* 属性允许你限制 *ForeignKey* 字段的内容通过给予一个特定值的集合。

在 *actions* 应用目录下创建一个新的文件命名为 *utils.py*。我们会定义一个快捷函数，该函数允许我们使用一种简单的方式创建新的 *Action* 对象。编辑这个新的文件添加如下代码给它：

```
from django.contrib.contenttypes.models import ContentType
from .models import Action
def create_action(user, verb, target=None):
    action = Action(user=user, verb=verb, target=target)
    action.save()
```

`create_action()`函数允许我们创建 *actions*，该 *actions* 可以包含一个 *target* 对象或不包含。我们可以使用这个函数在我们代码的任何地方添加新的 *actions* 给活动流（activity stream）。

在活动流（activity stream）中避免重复的操作

有时候你的用户可能多次执行同个动作。他们可能在短时间内多次点击 *like/unlike* 按钮或者多次执行同样的动作。这会导致你停止存储和显示重复的动作。为了避免这种情况我们需要改善 `create_action()` 函数来避免大部分的重复动作。

编辑 *actions* 应用中的 *utils.py* 文件使它看上去如下所示：

```
import datetime
from django.utils import timezone
from django.contrib.contenttypes.models import ContentType
from .models import Action

def create_action(user, verb, target=None):
    # check for any similar action made in the last minute
```

```
    now = timezone.now()
    last_minute = now - datetime.timedelta(seconds=60)
    similar_actions = Action.objects.filter(user_id=user.id,
                                            verb= verb,
                                        timestamp__gte=last_minute)
    if target:
        target_ct = ContentType.objects.get_for_model(target)
        similar_actions = similar_actions.filter(
                                            target_ct=target_ct,
                                            target_id=target.id)
    if not similar_actions:
        # no existing actions found
        action = Action(user=user, verb=verb, target=target)
        action.save()
        return True
    return False
```

我们通过修改 `create_action()`函数来避免保存重复的动作并且返回一个布尔值来告诉该动作是否保存。下面来解释我们是如何避免重复动作的：

- 首先，我们通过 Django 提供的 `timezone.now()`方法来获取当前时间。这个方法同 `datetime.datetime.now()` 相同，但是返回的是一个*timezone-aware*对象。Django 提供一个设置叫做*USE_TZ*用来启用或关闭时区的支持。通过使用 *startproject*命令创建的默认*settings.py*包含 USE_TZ=True`。
- 我们使用 *last_minute* 变量来保存一分钟前的时间，然后我们取回用户从那以后执行的任意一个相同操作。
- 我们会创建一个 *Action* 对象如果在最后的一分钟内没有存在同样的动作。我们会返回 *True* 如果一个 *Action* 对象被创建，否则返回 *False*。

添加用户动作给活动流（activity stream）

是时候添加一些动作给我们的视图(views)来给我的用户构建活动流（activity stream）了。我们将要存储一个动作为以下的每一个实例：

- 一个用户给某张图片打上书签
- 一个用户喜欢或不喜欢某张图片
- 一个用户创建一个账户
- 一个用户关注或不关注某个用户

编辑 *images* 应用下的 *views.py* 文件添加以下导入：

```
from actions.utils import create_action
```

在 *image_create* 视图（view）中，在保存图片之后添加 `create-action()`，如下所示：

```
new_item.save()
create_action(request.user, 'bookmarked image', new_item)
```

在 *image_like* 视图（view）中，在添加用户给 *users_like* 关系之后添加 `create_action()`，如下所示：

```
image.users_like.add(request.user)
create_action(request.user, 'likes', image)
```

现在编辑 *account* 应用中的 *view.py* 文件添加以下导入：

```
from actions.utils import create_action
```

在 *register* 视图（view）中，在创建 *Profile* 对象之后添加 `create-action()`，如下所示：

```
new_user.save()
profile = Profile.objects.create(user=new_user)
create_action(new_user, 'has created an account')
```

在 *user_follow* 视图（view）中添加 `create_action()`，如下所示：

```
Contact.objects.get_or_create(user_from=request.user,user_to=user)
create_action(request.user, 'is following', user)
```

就像你所看到的，感谢我们的 *Action* 模型（model）和我们的帮助函数，现在保存新的动作给活动流（activity stream）是非常简单的。

显示活动流（activity stream）

最后，我们需要一种方法来给每个用户显示活动流（activity stream）。我们将会在用户的 dashboard 中包含活动流（activity stream）。编辑 *account* 应用的 *views.py* 文件。导入 *Action* 模型然后修改 *dashboard* 视图（view）如下所示：

```
from actions.models import Action

@login_required
def dashboard(request):
    # Display all actions by default
    actions = Action.objects.exclude(user=request.user)
    following_ids = request.user.following.values_list('id',flat=True)
    if following_ids:
        # If user is following others, retrieve only their actions
        actions = actions.filter(user_id__in=following_ids)
    actions = actions[:10]

    return render(request,
                  'account/dashboard.html',
                 {'section': 'dashboard',
                   'actions': actions})
```

在这个视图（view），我们从数据库取回所有的动作（actions），不包含当前用户执行的动作。如果当前用户还没有关注过任何人，我们展示在平台中的其他用户的最新动作执行。这是一个默认的行为当当前用户还没有关注过任何其他的用户。如果当前用户已经关注了其他用户，我们就限制查询只显示当前用户关注的用户的动作执行。最后，我们限制结果只返回最前面的 10 个动作。我们在这儿并不使用 `order_by()`，因为我们依赖之前已经在 *Action* 模型（model）的 *Meta* 的排序选项。最新的动作会首先返回，因为我们在 *Action* 模型（model）中设置过 `ordering = ('-created',)`。

优化涉及被关联的对想的查询集（QuerySets）

每次你取回一个 *Aciton* 对象，你都可能存取它的有关联的 *User* 对象，
并且可能这个用户也关联它的 *Profile* 对象。Django ORM 提供了一个简单的方式一次性取回有关联的对象，避免对数据库进行额外的查询。

使用 *select_related*

Django 提供了一个叫做 `select_related()`的查询集（QuerySets）方法允许你取回关系为一对多的关联对象。该方法将会转化成一个单独的，更加复杂的查询集（QuerySets），但是你可以避免额外的查询

当存取这些关联对象。*select_relate* 方法是给 *ForeignKey* 和 *OneToOne* 字段使用的。它通过执行一个 *SQL JOIN* 并且包含关联对象的字段在 *SELECT* 声明中。
为了利用 `select_related()`,编辑之前代码中的以下行(译者注：请注意双下划线）：

```
actions = actions.filter(user_id__in=following_ids)
```

添加 *select_related* 在你将要使用的字段上：

```
actions = actions.filter(user_id__in=following_ids)\
                 .select_related('user', 'user__profile')
```

我们使用 `user__profile`（译者注：请注意是双下划线）来连接 profile 表在一个单独的 *SQL* 查询中。如果你调用 `select_related()`而不传入任何参数，它会取回所有 *ForeignKey* 关系的对象。给 `select_related()` 限制的关系将会在随后一直访问。
小心的使用 `select_related()`将会极大的提高执行时间

使用 *prefetch_related*

如你所见，`select_related()`将会帮助你提高取回一对多关系的关联对象的执行效率。但是，`select_related()` 无法给多对多或者多对一关系（ManyToMany 或者倒转 *ForeignKey* 字段）工作。Django 提供了一个不同的查询集（QuerySets）方法叫做 *prefetch_realted*，该方法在 `select_related()`方法支持的关系上增加了多对多和多对一的关系。`prefetch_related()`方法为每一种关系执行单独的查找然后对各个结果进行连接通过使用 Python。这个方法还支持 *GeneriRelation* 和 *GenericForeignKey* 的预先读取。
完成你的查询通过为它添加 `prefetch_related()`给目标 *GenericForeignKey* 字段，如下所示：

```
actions = actions.filter(user_id__in=following_ids)\
                 .select_related('user', 'user__profile')\
                 .prefetch_related('target')
```

这个查询现在已经被充分利用用来取回包含关联对象的用户动作（actions）。

为 *actions* 创建模板（templates）

我们要创建一个模板（template）用来显示一个独特的 *Action* 对象。在 *actions* 应用中创建一个新的目录命名为 *templates*。添加如下文件结构：

```
actions/
    action/
        detail.html
```

编辑 *actions/action/detail.html* 模板（template）文件添加如下代码：

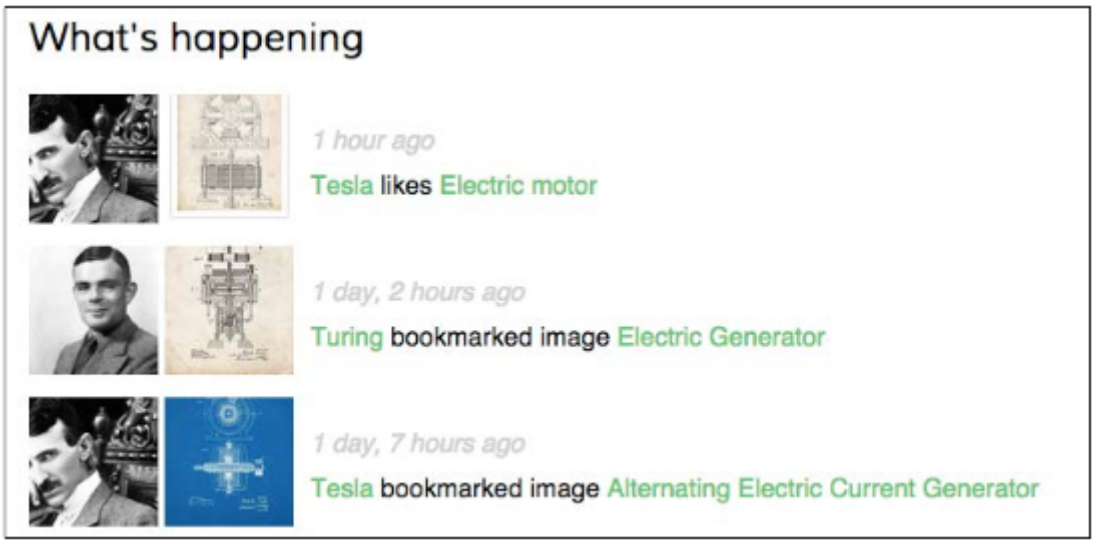
```
明天添加
```

这个模板用来显示一个 *Action* 对象。首先，我们使用`{% with %}`模板标签（template tag）来获取用户操作的动作（action）和他们的 profile。然后，我们显示目标对象的图片如果 *Action* 对象有一个关联的目标对象。最后，如果有执行过的动作（action），包括动作和目标对象，我们就显示链接给用户。
现在，编辑 *account/dashboard.html* 模板（template）添加如下代码到 *content* 区块下方：

```
<h2>What's happening</h2>
<div id="action-list">
    {% for action in actions %}
        {% include "actions/action/detail.html" %}
    {% endfor %}
```

```
</div>
```

在浏览器中打开 http://127.0.0.1:8000/account/ 。登录一个存在的用户并且该用户执行过一些操作已经被存储在数据库中。然后，登录其他用户，关注之前登录的用户，在 dashboard 页面可以看到生成的动作流。如下所示：



django-6-4

我们刚刚创建了一个完整的活动流（activity stream）给我们的用户并且我们还能非常容易的添加新的用户动作给它。你还可以添加无限的滚动功能给活动流（activity stream）通过集成 AJAX 分页处理，和我们之前在 *image_list* 视图（view）使用过的一样。

给非规范化（denormalizing）计数使用信号

有一些场景，你想要使你的数据非规范化。非规划化使指在一定的程度上制造一些数据冗余用来优化读取的性能。你必须十分小心的使用非规划化并且只有在你真的非常需要它的时候才能使用。你会发现非规划化的最大问题就是保持你的非规范化数据更新是非常困难的。

我们将会看到一个例子关于如何改善（improve）我们的查询通过使用非规范化计数。缺点就是我们不得不保持冗余数据的更新。我们将要从我们的 *Image* 模型（model）中使数据非规范化然后使用 Django 信号来保持数据的更新。

使用信号进行工作

Django 自带一个信号调度程序允许 *receiver* 函数在某个动作出现的时候去获取通知。信号非常有用，当你需要你的代码去执行某些事件的时候同时正在发生其他事件。你还能够创建你自己的信号这样一来其他人可以在某个事件发生的时候获得通知。

Django 模型（models）提供了几个信号，它们位于 *django.db.models.signales*。举几个例子：

- *pre_save* 和 *post_save*：前者会在调用模型（model）的 `save()`方法前发送信号，后者反之。
- *pre_delete* 和 *post_delete*：前者会在调用模型（model）或查询集（QuerySets）的 `delete()`方法之前发送信号，后者反之。
- *m2m_changed*：当在一个模型（model）上的 *ManayToManayField* 被改变的时候发送信号。

以上只是 Django 提供的一小部分信号。你可以通过访问 https://docs.djangoproject.com/en/1.8/ref/signals/ 获得更多信号资料。

打个比方，你想要获取热门图片。你可以使用 Django 的聚合函数来获取图片，通过图片获取的用户喜欢数量来进行排序。要记住你已经使用过 Django 聚合函数在*第三章 扩展你的 blog 应用*。以下代码将会获取图片并进行排序通过它们被用户喜欢的数量：

```
from django.db.models import Count
from images.models import Image
images_by_popularity = Image.objects.annotate(
```

```
    total_likes=Count('users_like')).order_by('-total_likes')
```

但是，通过统计图片的总喜欢数量进行排序比直接使用一个已经存储总统计数的字段进行排序要消耗更多的性能。你可以添加一个字段给 *Image* 模型（model）用来非规范化喜欢的数量用来提升涉及该字段的查询的性能。那么，问题来了，我们该如何保持这个字段是最新更新过的。
编辑 *images* 应用下的 *models.py* 文件，给 *Image* 模型（model）添加以下字段：

```
total_likes = models.PositiveIntegerField(db_index=True,
                                          default=0)
```

total_likes 字段允许我们给每张图片存储被用户喜欢的总数。非规范化数据非常有用当你想要使用他们来过滤或排序查询集（QuerySets）。
在你使用非规范化字段之前你必须考虑下其他几种提高性能的方法。考虑下数据库索引，最佳化查询以及缓存在开始规范化你的数据之前。
运行以下命令将新添加的字段迁移到数据库中：

```
python manage.py makemigrations images
```

你会看到如下输出：

```
Migrations for 'images':
    0002_image_total_likes.py:
        - Add field total_likes to image
```

接着继续运行以下命令来应用迁移：

```
python manage.py migrate images
```

输出中会包含以下内容：

```
Applying images.0002_image_total_likes... OK
```

我们要给 *m2m_changed* 信号附加一个 *receiver* 函数。在 *images* 应用目录下创建一个新的文件命名为 *signals.py*。给该文件添加如下代码：

```
from django.db.models.signals import m2m_changed
from django.dispatch import receiver
from .models import Image
@receiver(m2m_changed, sender=Image.users_like.through)
def users_like_changed(sender, instance, **kwargs):
    instance.total_likes = instance.users_like.count()
    instance.save()
```

首先，我们使用 `receiver()`装饰器将 `users_like_changed` 函数注册成一个 *receiver* 函数，然后我们将该函数附加给 *m2m_changed* 信号。我们将这个函数与 *Image.users_like.through* 连接，这样这个函数只有当 *m2m_changed* 信号被 *Image.users_like.through* 执行的时候才被调用。还有一个可以替代的方式来注册一个 *receiver* 函数，由使用 *Signal* 对象的 `connect()`方法组成。
Django 信号是同步阻塞的。不要使用异步任务导致信号混乱。但是，你可以联合两者来执行异步任务当你的代码只接受一个信号的通知。
你必须连接你的 *receiver* 函数给一个信号，只有这样它才会被调用当连接的信号发送的时候。有一个推荐的方法用来注册你的信号是在你的应用配置类中导入它们到 `ready()`方法中。Django 提供一个应用注册允许你对你的应用进行配置和内省。

典型的应用配置类

django 允许你指定配置类给你的应用们。为了提供一个自定义的配置给你的应用，创建一个继承 *django.apps* 的 *Appconfig* 类的自定义类。这个应用配置类允许你为应用存储元数据和配置并且提供内省。

你可以通过访问 https://docs. djangoproject.com/en/1.8/ref/applications/ 获取更多关于应用配置的信息。

为了注册你的信号 *receiver* 函数，当你使用 `receiver()`装饰器的时候，你只需要导入信号模块，这些信号模块被包含在你的应用的 *AppConfig* 类中的 `ready()`方法中。这个方法在应用注册被完整填充的时候就调用。其他给你应用的初始化都可以被包含在这个方法中。

在 *images* 应用目录下创建一个新的文件命名为 *apps.py*。为该文件添加如下代码：

```
from django.apps import AppConfig
class ImagesConfig(AppConfig):
    name = 'images'
    verbose_name = 'Image bookmarks'
    def ready(self):
        # import signal handlers
        import images.signals
```

name 属性定义该应用完整的 Python 路径。*verbose_name* 属性设置了这个应用可读的名字。它会在管理站点中显示。`ready()`方法就是我们为这个应用导入信号的地方。

现在我们需要告诉 Django 我们的应用配置位于哪里。编辑位于 *images* 应用目录下的 ***init.py*** 文件添加如下内容：

```
default_app_config = 'images.apps.ImagesConfig'
```

打开你的浏览器浏览一个图片的详细页面然后点击 **like** 按钮。再进入管理页面看下该图片的 *total_like* 属性。你会看到 *total_likes* 属性已经更新了最新的 like 数如下所示：

django-6-5

现在，你可以使用 *totla_likes* 属性来进行热门图片的排序或者在任何地方显示这个值，从而避免了复杂的查询操作。以下获取图片的查询通过图片的喜欢数量进行排序：

```
images_by_popularity = Image.objects.annotate(
    likes=Count('users_like')).order_by('-likes')
```

现在我们可以用新的查询来代替上面的查询：

```
images_by_popularity = Image.objects.order_by('-total_likes')
```

以上查询的返回结果只需要很少的 SQL 查询性能。以上就是一个例子关于如何使用 Django 信号。小心使用信号，因为它们会给理解控制流制造困难。在很多场景下你可以避免使用信号如果你知道哪个接收器需要被通知。

使用 Redis 来存储视图（views）项

Redis 是一个高级的 *key-value*（键值）数据库允许你保存不同类型的数据并且在 *I/O*(输入/输出）操作上非常非常的快速。Redis 可以在内存中存储任何东西，但是这些数据能够持续通过偶尔存储数据集到磁盘中或者添加每一条命令到日志中。Redis 是非常出彩的通过与其他的键值存储对比：它提供了一个强大的设置命令，并且支持多种数据结构，例如 string，hashes，lists，sets，ordered sets，甚至 bitmaps 和 HyperLogLogs。
SQL 最适合用于模式定义的持续数据存储，而 Redis 提供了许多优势当需要处理快速变化的数据，易失性存储，或者需要一个快速缓存的时候。让我们看下 Redis 是如何被使用的，当构建新的功能到我们的项目中。

安装 Redis

从 http://redis.io/download 下载最新的 Redis 版本。解压 *tar.gz* 文件，进入 *redis* 目录然后编译 Redis 通过使用以下 make 命令：

```
cd redis-3.0.4(译者注：版本根据自己下载的修改)
make （译者注：这里是假设你使用的是 linux 或者 mac 系统才有 make 命令，windows 如何操作请看下官方文档）
```

在 Redis 安装完成后允许以下 shell 命令来初始化 Redis 服务：

```
src/redis-server
```

你会看到输出的结尾如下所示：

```
# Server started, Redis version 3.0.4
* DB loaded from disk: 0.001 seconds
* The server is now ready to accept connections on port 6379
```

默认的，Redis 运行会占用 6379 端口，但是你也可以指定一个自定义的端口通过使用`--port`标志，例如：`redis-server --port 6655`。当你的服务启动完毕，你可以在其他的终端中打开 Redis 客户端通过使用如下命令：

```
src/redis-cli
```

你会看到 Redis 客户端 shell 如下所示：

```
127.0.0.1:6379>
```

Redis 客户端允许你在当前 shell 中立即执行 Rdis 命令。来我们来尝试一些命令。键入 *SET* 命令在 Redis 客户端中存储一个值到一个键中：

```
127.0.0.1:6379> SET name "Peter"
ok
```

以上的命令创建了一个带有字符串“Peter”值的 *name* 键到 Redis 数据库中。*OK* 输出表明该键已经被成功保存。然后，使用 *GET* 命令获取之前的值，如下所示：

```
127.0.0.1:6379> GET name
"Peter"
```

你还可以检查一个键是否存在通过使用 *EXISTS* 命令。如果检查的键存在会返回 1，反之返回 0：

```
127.0.0.1:6379> EXISTS name
```

```
(integer) 1
```

你可以给一个键设置到期时间通过使用 *EXPIRE* 命令，该命令允许你设置该键能在几秒内存在。另一个选项使用 *EXPIREAT* 命令来期望一个 Unix 时间戳。键的到期消失是非常有用的当将 Redis 当做缓存使用或者存储易失性的数据：

```
127.0.0.1:6379> GET name
"Peter"
127.0.0.1:6379> EXPIRE name 2
(integer) 1
Wait for 2 seconds and try to get the same key again:
127.0.0.1:6379> GET name
(nil)
```

（nil）响应是一个空的响应说明没有找到键。你还可以通过使用 *DEL* 命令删除任意键，如下所示：

```
127.0.0.1:6379> SET total 1
OK
127.0.0.1:6379> DEL total
(integer) 1
127.0.0.1:6379> GET total
(nil)
```

以上只是一些键选项的基本命令。Redis 包含了庞大的命令设置给一些数据类型，例如 strings，hashes，sets，ordered sets 等等。你可以通过访问 http://redis.io/commands 看到所有 Reids 命令以及通过访问 http://redis.io/topics/data-types 看到所有 Redis 支持的数据类型。

通过 Python 使用 Redis

我们需要绑定 Python 和 Redis。通过 pip 渠道安装 *redis-py* 命令如下：

```
pip install redis==2.10.3（译者注：版本可能有更新，如果需要最新版本，可以不带上'==2.10.3'后缀）
```

你可以访问 http://redis-py.readthedocs.org/ 得到 redis-py 文档。
redis-py 提供两个类用来与 Redis 交互：*StrictRedis* 和 *Redis*。两者提供了相同的功能。*StrictRedis* 类尝试遵守官方的 Redis 命令语法。*Redis* 类型继承 *Strictredis* 重写了部分方法来提供向后的兼容性。我们将会使用 *StrictRedis* 类，因为它遵守 Redis 命令语法。打开 Python shell 执行以下命令：

```
>>> import redis
>>> r = redis.StrictRedis(host='localhost', port=6379, db=0)
```

上面的代码创建了一个与 Redis 数据库的连接。在 Redis 中，数据库通过一个整形索引替代数据库名字来辨识。默认的，一个客户端被连接到数据库 0 。Reids 数据库可用的数字设置到 16，但是你可以在 *redis.conf* 文件中修改这个值。
现在使用 Python shell 设置一个键：

```
>>> r.set('foo', 'bar')
True
```

以上命令返回 Ture 表明这个键已经创建成功。现在你可以使用 `get()`命令取回该键：

```
>>> r.get('foo')
'bar'
```

如你所见，*StrictRedis* 方法遵守 Redis 命令语法。
让我们集成 Rdies 到我们的项目中。编辑 *bookmarks* 项目的 *settings.py* 文件添加如下设置：

```
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 0
```

以上设置了 Redis 服务器和我们将要在项目中使用到的数据库。

存储视图（vies）项到 Redis 中

让我们存储一张图片被查看的总次数。如果我们通过 Django ORM 来完成这个操作，它会在每次该图片显示的时候执行一次 SQL UPDATE 声明。使用 Redis，我们只需要对一个计数器进行增量存储在内存中，从而带来更好的性能。
编辑 *images* 应用下的 *views.py* 文件，添加如下代码：

```
import redis
from django.conf import settings
# connect to redis
r = redis.StrictRedis(host=settings.REDIS_HOST,
                      port=settings.REDIS_PORT,
                      db=settings.REDIS_DB)
```

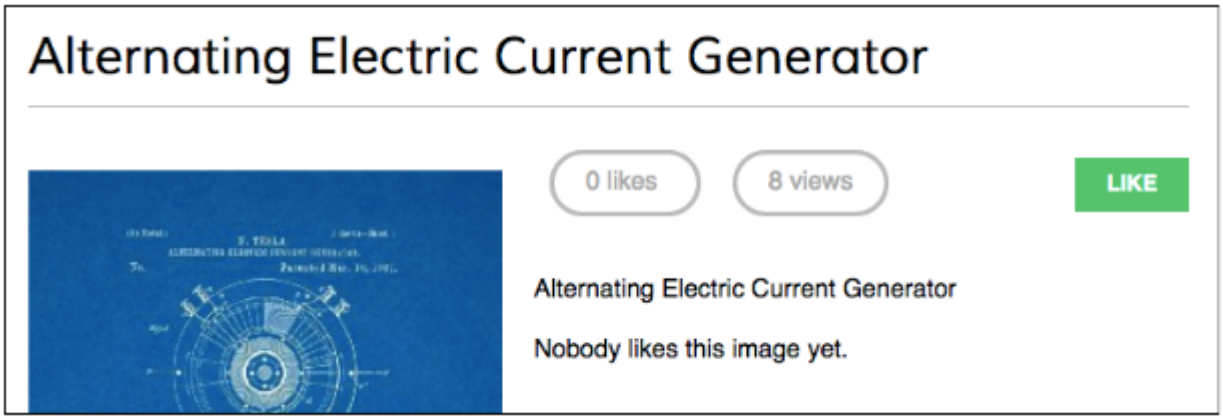
在这儿我们建立了 Redis 的连接为了能在我们的视图(views)中使用它。编辑 *images_detail* 视图(view)使它看上去如下所示：

```
def image_detail(request, id, slug):
image = get_object_or_404(Image, id=id, slug=slug)
# increment total image views by 1
total_views = r.incr('image:{}:views'.format(image.id))
return render(request,
              'images/image/detail.html',
              {'section': 'images',
               'image': image,
               'total_views': total_views})
```

在这个视图（view）中，我们使用 *INCR* 命令，它会从 1 开始增量一个键的值，在执行这个操作之前如果键不存在，它会将值设定为 0.`incr()`方法在执行操作后会返回键的值，然后我们可以存储该值到 *total_views* 变量中。我们构建 Rddis 键使用一个符号，比如 *object-type:id:field (for example image:33:id)* 。
对 Redis 的键进行命名有一个惯例是使用冒号进行分割来创建键的命名空间。做到这点，键的名字会特别冗长，有关联的键会分享部分相同的模式在它们的名字中。
编辑 *image/detail.html* 模板（template）在已有的`<span class="count">`元素之后添加如下代码：

```
<span class="count">
    <span class="total">{{ total_views }}</span>
    view{{ total_views|pluralize }}
</span>
```

现在在浏览器中打开一张图片的详细页面然后多次加载该页面。你会看到每次该视图（view）被执行的时候，总的观看次数会增加 1 。如下所示：



django-6-6

你已经成功的集成 Redis 到你的项目中来存储项统计。

存储一个排名到 Reids 中

让我们使用 Reids 构建更多的功能。我们要在我们的平台中创建一个最多浏览次数的图片排行。为了构建这个排行我们将要使用 Redis 分类集合。一个分类集合是一个非重复的字符串采集，其中每个成员和一个分数关联。其中的项根据它们的分数进行排序。
编辑 *images* 引用下的 *views.py* 文件，使 *image_detail* 视图（view）看上去如下所示：

```
def image_detail(request, id, slug):
image = get_object_or_404(Image, id=id, slug=slug)
# increment total image views by 1
total_views = r.incr('image:{}:views'.format(image.id)) # increment image ranking by 1
r.zincrby('image_ranking', image.id, 1)
return render(request,
             'images/image/detail.html',
             {'section': 'images',
              'image': image,
              'total_views': total_views})
```

我们使用 `zincrby()`命令存储图片视图（views）到一个分类集合中通过键 `image:ranking`。我们存储图片 *id*，和一个分数 1，它们将会被加到分类集合中这个元素的总分上。这将允许我们在全局上持续跟踪所有的图片视图（views），并且有一个分类集合，该分类集合通过图片的浏览次数进行排序。
现在创建一个新的视图（view）用来展示最多浏览次数图片的排行。在 *views.py* 文件中添加如下代码：

```
@login_required
def image_ranking(request):
    # get image ranking dictionary
    image_ranking = r.zrange('image_ranking', 0, -1,
                             desc=True)[:10]
    image_ranking_ids = [int(id) for id in image_ranking]
    # get most viewed images
    most_viewed = list(Image.objects.filter(
                     id__in=image_ranking_ids))
    most_viewed.sort(key=lambda x: image_ranking_ids.index(x.id))
    return render(request,
                 'images/image/ranking.html',
                 {'section': 'images',
                  'most_viewed': most_viewed})
```

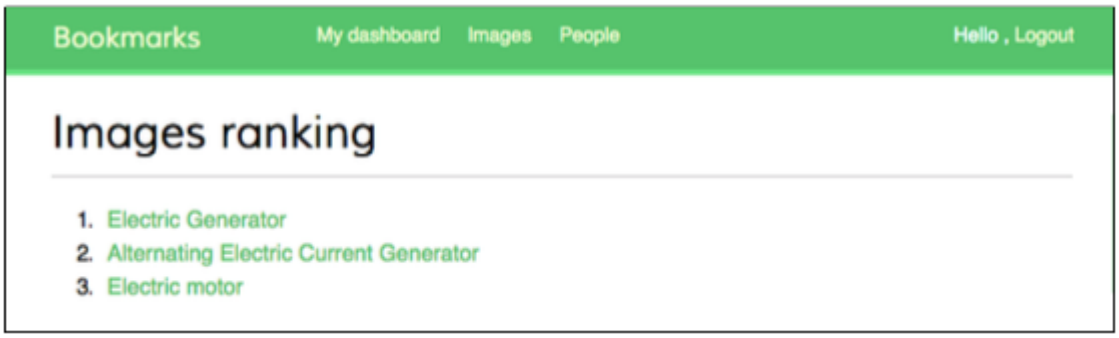
以上就是 *image_ranking* 视图。我们使用 `zrange()`命令获得分类集合中的元素。这个命令期望一个自定义的范围，最低分和最高分。通过将 0 定为最低分， -1 为最高分，我们告诉 Redis 返回分类集合中的所有元素。最终，我们使用`[:10]`对结果进行切片获取最前面十个最高分的元素。我们构建一个返回的图片 IDs 的列，然后我们将该列存储在 *image_ranking_ids* 变量中，这是一个整数列。我们通过这些 IDs 取回对应的 *Image* 对象，并将它们强制转化为列通过使用 `list()`函数。强制转化查询集（QuerySets）的执行是非常重要的，因为接下来我们要在该列上使用列的 `sort()`方法（就是因为这点所以我们需要的是一个对象列而不是一个查询集（QuerySets））。我们排序这些 *Image* 对象通过它们在图片排行中的索引。现在我们可以在我们的模板（template）中使用 *most_viewed* 列来显示 10 个最多浏览次数的图片。
创建一个新的 *image/ranking.html* 模板（template）文件，添加如下代码：

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

{% block title %}Images ranking{% endblock %}

{% block content %}
    <h1>Images ranking</h1>
     <ol>
       {% for image in most_viewed %}
         <li>
           <a href="{{ image.get_absolute_url }}">
             {{ image.title }}
           </a>
         </li>
       {% endfor %}
     </ol>
{% endblock %}
```

这个模板（template）非常简单明了，我们只是对包含在 *most_viewed* 中的 *Image* 对象进行迭代。
最后为新的视图（view）创建一个 URL 模式。编辑 *images* 应用下的 *urls.py* 文件，添加如下内容：

```
url(r'^ranking/$', views.image_ranking, name='create'),
```

在浏览器中打开 http://127.0.0.1:8000/images/ranking/ 。你会看到如下图片排行：



django-6-7

Redis 的下一步

Redis 并不能替代你的 SQL 数据库，但是它是一个内存中的快速存储，更适合某些特定任务。将它添加到你的栈中使用当你真的感觉它很需要。以下是一些适合 Redis 的场景：

- Counting：如你之前看到的，通过 Redis 管理计数器非常容易。你可以使用 `incr()`和`incrby()。

- Storing latest items：你可以添加项到一个列的开头和结尾通过使用 `lpush()`和 `rpush()`。移除和返回开头和结尾的元素通过使用 `lpop()`以及 `rpop()`。你可以削减列的长度通过使用 `ltrim()`来维持它的长度。
- Queues：除了 push 和 pop 命令，Redis 还提供堵塞的队列命令。
- Caching：使用 `expire()`和 `expireat()`允许你将 Redis 当成缓存使用。你还可以找到第三方的 Reids 缓存后台给 Django 使用。
- Pub/Sub：Redis 提供命令给订阅或不订阅，并且给渠道发送消息。
- Rankings and leaderboards：Redis 使用分数的分类集合使创建排行榜非常的简单。
- Real-time tracking：Redis 快速的 I/O(输入/输出)使它能完美支持实时场景。

总结

在本章中，你构建了一个粉丝系统和一个用户活动流（activity stream）。你学习了 Django 信号是如何进行工作并且在你的项目中集成了 Redis。
在下一章中，你会学习到如何构建一个在线商店。你会创建一个产品目录并且通过会话（sessions）创建一个购物车。你还会学习如何通过 Celery 执行异步任务。

第七章 建立一个在线商店

在上一章，你创建了一个用户跟踪系统和建立了一个用户活跃流。你也学习了 Django 信号是如何工作的，并且把 Redis 融合进了项目中来为图像视图计数。在这一章中，你将学会如何建立一个最基本的在线商店。你将会为你的产品创建目录和使用 Django sessions 实现一个购物车。你也将学习怎样定制上下文处理器（ context processors ）以及用 Celery 来激活动态任务。
在这一章中，你将学会：

- 创建一个产品目录
- 使用 Django sessions 建立购物车
- 管理顾客的订单
- 用 Celery 发送异步通知

创建一个在线商店项目（project）

我们将从新建一个在线商店项目开始。我们的用户可以浏览产品目录并且可以向购物车中添加商品。最后，他们将清点购物车然后下单。这一章涵盖了在线商店的以下几个功能：

- 创建产品目录模型（模型），将它们添加到管理站点，创建基本的视图（view）来展示目录
- 使用 Django sessions 建立一个购物车系统，使用户可以在浏览网站的过程中保存他们选中的商品
- 创建下单表单和功能
- 发送一封异步的确认邮件在用户下单的时候

首先，用以下命令来为你的新项目创建一个虚拟环境，然后激活它：

```
mkdir env
virtualenv env/myshop
source env/myshop/bin/activate
```

用以下命令在你的虚拟环境中安装 Django :

```
pip install django==1.8.6
```

创建一个叫做 `myshop` 的新项目，再创建一个叫做 `shop` 的应用，命令如下：

```
django-admin startproject myshop
cd myshop
django-admin startapp shop
```

编辑你项目中的 `settings.py` 文件，像下面这样将你的应用添加到 `INSTALLED_APPS` 中：

```
INSTALLED_APPS = [
    # ...
    'shop',
]
```

现在你的应用已经在项目中激活。接下来让我们为产品目录定义模型（models）。

创建产品目录模型（models）

我们商店中的目录将会由不同分类的产品组成。每一个产品会有一个名字，一段可选的描述，一张可选的图片，价格，以及库存。 编辑位于 `shop` 应用中的 `models.py` 文件，添加以下代码：

```
from django.db import models

class Category(models.Model):
     name = models.CharField(max_length=200,
                             db_index=True)
     slug = models.SlugField(max_length=200,
                         db_index=True,
                               unique=True)
     class Meta:
          ordering = ('name',)
          verbose_name = 'category'
          verbose_name_plural = 'categories'

    def __str__(self):
        return self.name

class Product(models.Model):
    category = models.ForeignKey(Category,
                              related_name='products')
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True)
    image = models.ImageField(upload_to='products/%Y/%m/%d',
                          blank=True)
    description = models.TextField(blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    stock = models.PositiveIntegerField()
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ('name',)
        index_together = (('id', 'slug'),)

    def __str__(self):
        return self.name
```

这是我们的 Category 和 Product 模型（models）。Category 模型（models）由一个 name 字段和一个唯一的 slug 字段构成。Product 模型（model）：

- category: 这是一个链接向 Category 的 ForeignKey 。这是个多对一（many-to-one）关系。一个产品可以属于一个分类，一个分类也可包含多个产品。
- name: 这是产品的名字
- slug: 用来为这个产品建立 URL 的 slug
- image: 可选的产品图片
- description: 可选的产品描述
- price: 这是个 DecimalField（**译者@ucag 注：十进制字段**）。这个字段使用 Python 的 decimal.Decimal 元类来保存一个固定精度的十进制数。max_digits 属性可用于设定数字的最大值， decimal_places 属性用于设置小数位数。
- stock: 这是个 PositiveIntegerField（**译者@ucag 注：正整数字段**） 来保存这个产品的库存。
- available: 这个布尔值用于展示产品是否可供购买。这使得我们可在目录中使产品废弃或生效。
- created: 当对象被创建时这个字段被保存。
- update: 当对象最后一次被更新时这个字段被保存。

对于 price 字段，我们使用 DecimalField 而不是 FloatField 来避免精度问题。
我们总是使用 DecimalField 来保存货币值。 FloatField 在内部使用 Python 的 float 类型。反之， DecimalField 使用的是 Python 中的 Decimal 类型，使用 Decimal 类型可以避免精度问题。
在 Product 模型（model）中的 Meta 类中，我们使用 index_together 元选项来指定 id 和 slug 字段的共同索引。我们定义这个索引，因为我们准备使用这两个字段来查询产品，两个字段被索引在一起来提高使用双字段查询的效率。
由于我们会在模型（models）中和图片打交道，打开 shell ，用下面的命令安装 Pillow ：

```
pip isntall Pillow==2.9.0
```

现在，运行下面的命令来为你的项目创建初始迁移：

```
python manage.py makemigrations
```

你将会看到以下输出：

```
Migrations for 'shop':
    0001_initial.py:
      - Create model Category
      - Create model Product
      - Alter index_together for product (1 constraint(s))
```

用下面的命令来同步你的数据库：

```
python mange.py migrate
```

你将会看到包含下面这一行的输出：

```
 Applying shop.0001_initial... OK
```

现在数据库已经和你的模型（models）同步了。

注册目录模型（models）到管理站点

让我们把模型（models）注册到管理站点，这样我们就可以轻松管理产品和产品分类了。编辑 shop 应用的 admin.py 文件，添加如下代码：

```
from django.contrib import admin
from .models import Category, Product
```

```
class CategoryAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug']
    prepopulated_fields = {'slug': ('name',)}
admin.site.register(Category, CategoryAdmin)

class ProductAdmin(admin.ModelAdmin):
    list_display = ['name', 'slug', 'price', 'stock',
                    'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated']
    list_editable = ['price', 'stock', 'available']
    prepopulated_fields = {'slug': ('name',)}
admin.site.register(Product, ProductAdmin)
```

记住，我们使用 `prepopulated_fields` 属性来指定那些要使用其他字段来自动赋值的字段。正如你以前看到的那样，这样做可以很方便的生成 slugs 。我们在 `ProductAdmin` 类中使用 `list_editable` 属性来设置可被编辑的字段，并且这些字段都在管理站点的列表页被列出。这样可以让你一次编辑多行。任何在 `list_editable` 的字段也必须在 `list_display` 中，因为只有这样被展示的字段才可以被编辑。
现在，使用如下命令为你的站点创建一个超级用户：

```
python manage.py createsuperuser
```

使用命令 `python manage.py runserver` 启动开发服务器。 访问 http://127.0.0.1:8000/admin/shop/product/add ,登录你刚才创建的超级用户。在管理站点的交互界面添加一个新的品种和产品。 product 的更改页面如下所示：

django-7-1

创建目录视图（views）

为了展示产品目录， 我们需要创建一个视图（view）来列出所有产品或者是给出的筛选后的产品。编辑 `shop` 应用中的 `views.py` 文件，添加如下代码：

```
from django.shortcuts import render, get_object_or_404
from .models import Category, Product

def product_list(request, category_slug=None):
    category = None
    categories = Category.objects.all()
    products = Product.objects.filter(available=True)
    if category_slug:
        category = get_object_or_404(Category, slug=category_slug)
        products = products.filter(category=category)
    return render(request,
```

```
                'shop/product/list.html',
                {'category': category,
                'categories': categories,
                'products': products})
```

我们只筛选 `available=True` 的查询集来检索可用的产品。我们使用一个可选参数 `category_slug` 通过所给产品类别来有选择性的筛选产品。
我们也需要一个视图来检索和展示单一的产品。把下面的代码添加进去：

```
def product_detail(request, id, slug):
    product = get_object_or_404(Product,
                                id=id,
                                slug=slug,
                                available=True)
    return render(request,
                'shop/product/detail.html',
                {'product': product})
```

`product_detail` 视图（view）接收 `id` 和 `slug` 参数来检索 `Product` 实例。我们可以只用 ID 就可以得到这个实例，因为它是一个独一无二的属性。尽管，我们在 URL 中引入了 slug 来建立搜索引擎友好（SEO-friendly）的 URL。
在创建了产品列表和明细视图（views）之后，我们该为它们定义 URL 模式了。在 `shop` 应用的路径下创建一个新的文件，命名为 `urls.py` ，然后添加如下代码：

```
from django.conf.urls import url
from . import views
urlpatterns = [
    url(r'^$', views.product_list, name='product_list'),
    url(r'^(?P<category_slug>[-\w]+)/$',
        views.product_list,
        name='product_list_by_category'),
    url(r'^(?P<id>\d+)/(?P<slug>[-\w]+)/$',
        views.product_detail,
        name='product_detail'),
```

这些是我们产品目录的 URL 模式。 我们为 `product_list` 视图（view）定义了两个不同的 URL 模式。命名为 `product_list` 的模式不带参数调用 `product_list` 视图（view）；命名为 `product_list_bu_category` 的模式向视图（view）函数传递一个 `category_slug` 参数，以便通过给定的产品种类来筛选产品。我们为 `product_detail` 视图（view）添加的模式传递了 `id` 和 `slug` 参数来检索特定的产品。
像这样编辑 `myshop` 项目中的 `urls.py` 文件：

```
from django.conf.urls import url, include
from django.contrib import admin
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^', include('shop.urls', namespace='shop')),
    ]
```

在项目的主要 URL 模式中，我们引入了 `shop` 应用的 URL 模式，并指定了一个命名空间，叫做 `shop`。现在，编辑 `shop` 应用中的 `models.py` 文件，导入 `reverse()` 函数，然后给 `Category` 模型和 `Product` 模型添加 `get_absolute_url()` 方法：

```
from django.core.urlresolvers import reverse
# ...
class Category(models.Model):
    # ...
    def get_absolute_url(self):
        return reverse('shop:product_list_by_category',
                       args=[self.slug])
class Product(models.Model):
# ...
    def get_absolute_url(self):
        return reverse('shop:product_detail',
            args=[self.id, self.slug])
```

正如你已经知道的那样， `get_absolute_url()` 是检索一个对象的 URL 约定俗成的方法。这里，我们将使用我们刚刚在 `urls.py` 文件中定义的 URL 模式。

创建目录模板（templates）

现在，我们需要为产品列表和明细视图创建模板（templates）。在 `shop` 应用的路径下创建如下路径和文件：

```
templates/
    shop/
        base.html
        product/
            list.html
            detail.html
```

我们需要定义一个基础模板（template），然后在产品列表和明细模板（templates）中继承它。 编辑 `shop/base.html` 模板（template），添加如下代码：

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>{% block title %}My shop{% endblock %}</title>
    <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
    <div id="header">
        <a href="/" class="logo">My shop</a>
    </div>
    <div id="subheader">
        <div class="cart">
            Your cart is empty.
        </div>
    </div>
    <div id="content">
        {% block content %}
        {% endblock %}
    </div>
```

```
</body>
</html>
```

这就是我们将为我们的商店应用使用的基础模板（template）。为了引入模板使用的 CSS 和图像，你需要复制这一章示例代码中的静态文件，位于 `shop` 应用中的 `static/`路径下。把它们复制到你的项目中相同的地方。

编辑 `shop/product/list.html` 模板（template），然后添加如下代码：

```
{% extends "shop/base.html" %}
{% load static %}

{% block title %}
    {% if category %}{{ category.name }}{% else %}Products{% endif %}
{% endblock %}

{% block content %}
    <div id="sidebar">
        <h3>Categories</h3>
        <ul>
            <li {% if not category %}class="selected"{% endif %}>
                <a href="{% url "shop:product_list" %}">All</a>
            </li>
        {% for c in categories %}
            <li {% if category.slug == c.slug %}class="selected"{% endif %}>
                <a href="{{ c.get_absolute_url }}">{{ c.name }}</a>
            </li>
        {% endfor %}
        </ul>
    </div>
    <div id="main" class="product-list">
        <h1>{% if category %}{{ category.name }}{% else %}Products{% endif %}</h1>
        {% for product in products %}
            <div class="item">
                <a href="{{ product.get_absolute_url }}">
                    ![]({% if product.image %}{{ product.image.url }}{% else %}{% static )
                </a>
                <a href="{{ product.get_absolute_url }}">{{ product.name }}</a><br>
                ${{ product.price }}
            </div>
        {% endfor %}
    </div>
{% endblock %}
```

这是产品列表模板（template）。它继承了 `shop/base.html` 并且使用了 `categories` 上下文变量来展示所有在侧边栏里的产品种类，以及 `products` 上下文变量来展示当前页面的产品。相同的模板用于展示所有的可用的产品以及经目录分类筛选后的产品。由于 `Product` 模型的 `image` 字段可以为空，我们需要为没有图片的产品提供一个默认图像。这个图片位于我们的静态文件路径下，相对路径为 `img/no_image.png`。因为我们在使用 `ImageField` 来保存产品图片，我们需要开发服务器来服务上传图片文件。编辑 `myshop` 项目的 `settings.py` 文件，添加以下设置：

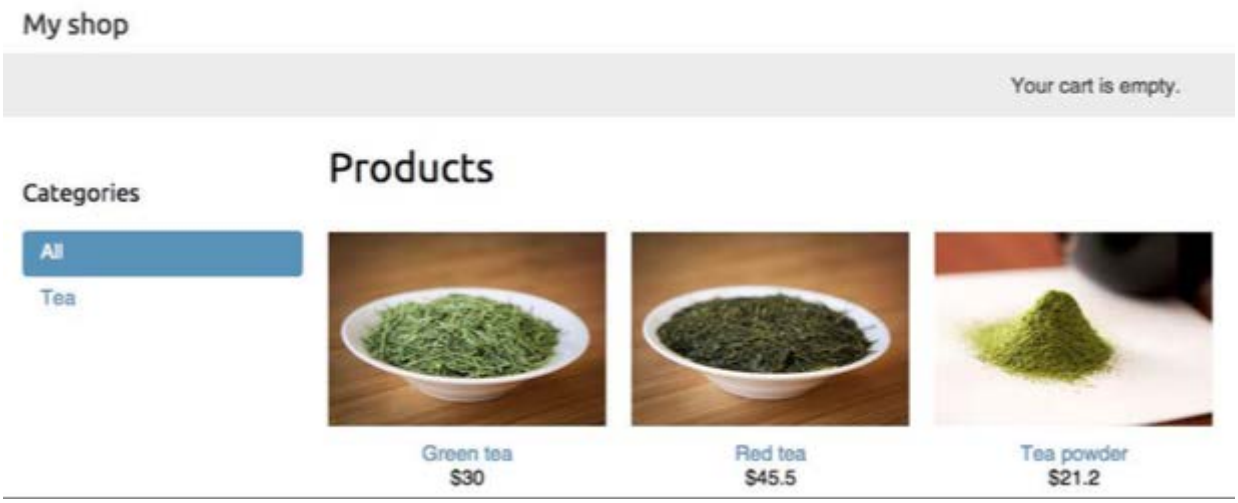
```
MEDIA_URL = '/media/'
```

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

`MEDIA_URL` 是基础 URL，它为用户上传的媒体文件提供服务。`MEDIA_ROOT` 是一个本地路径，媒体文件就在这个路径下，并且是由我们动态的将 `BASE_DIR` 添加到它的前面而得到的。
为了让 Django 给通过开发服务器上传的媒体文件提供服务，编辑 `myshop` 中的 `urls.py` 文件，添加如下代码：

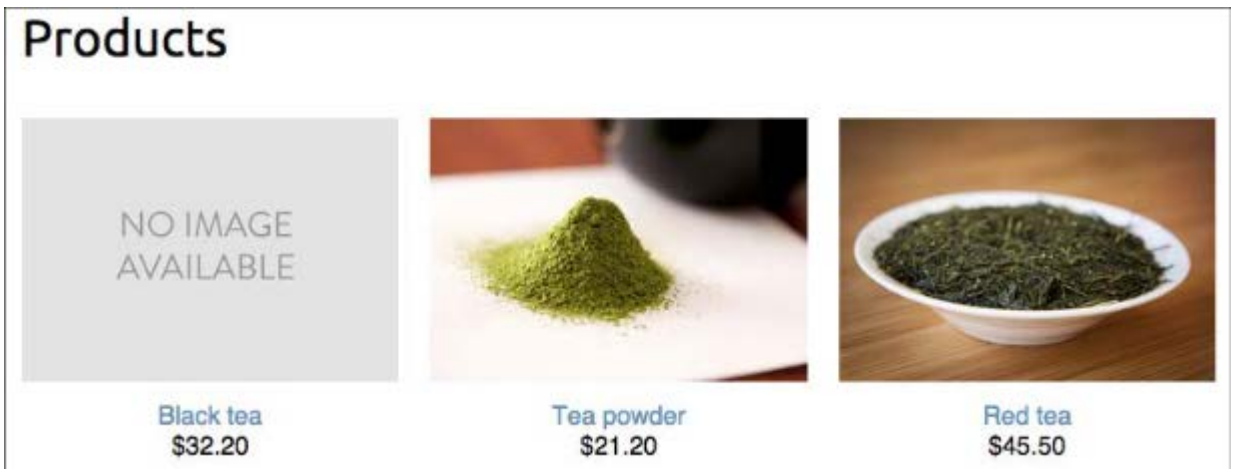
```
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    # ...
]
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

记住，我们仅仅在开发中像这样提供静态文件服务。在生产环境下，你不应该用 Django 来服务静态文件。
使用管理站点为你的商店添加几个产品，然后访问 http://127.0.0.1:8000/ 。你可以看到如下的产品列表页：



django-7-2

如果你用管理站点创建了几个产品，并且没有上传任何图片的话，就会显示默认的 `no_img.png` 。

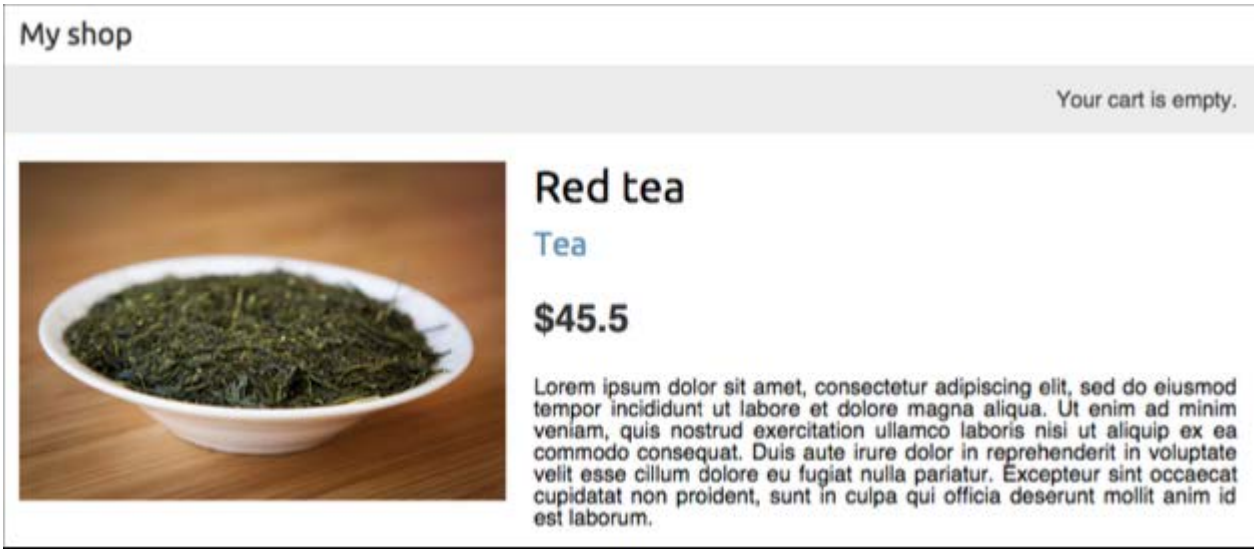


django-7-3

让我们编辑产品明细模板（template）。 编辑 `shop/product/detail.html` 模板（template），添加以下代码：

```
{% extends "shop/base.html" %}
{% load static %}

{% block title %}
    {% if category %}{{ category.title }}{% else %}Products{% endif %}
{% endblock %}

{% block content %}
    <div class="product-detail">
        ![]({% if product.image %}{{ product.image.url }}{% else %}{% static )
        <h1>{{ product.name }}</h1>
        <h2><a href="{{ product.category.get_absolute_url }}">{{ product.category }}</a></h2>
        <p class="price">${{ product.price }}</p>
            {{ product.description|linebreaks }}
    </div>
{% endblock %}
```

我们可以调用相关联的产品类别的 `get_absolute_url()` 方法来展示有效的属于同一目录的产品。现在，访问 http://127.0.0.1:8000 ，点击任意产品，查看产品明细页面。看起来像这样：



django-7-4

创建购物车

在创建了产品目录之后，下一步我们要创建一个购物车系统，这个购物车系统可以让用户选中他们想买的商品。购物车允许用户在最终下单之前选中他们想要的物品并且可以在用户浏览网站时暂时保存它们。购物车存在于会话中，所以购物车中的物品会在用户访问期间被保存。
我们将会使用 Django 的会话框架（seesion framework）来保存购物车。在购物车最终被完成或用户下单之前，购物车将会保存在会话中。我们需要为购物车和购物车里的商品创建额外的 Django 模型（models）。

使用 Django 会话

Django 提供了一个会话框架，这个框架支持匿名会话和用户会话。会话框架允许你为任意访问对象保存任何数据。会话数据保存在服务端，并且如果你使用基于 cookies 的会话引擎的话， cookies 会包含 session ID 。会话中间件控制发送和接收 cookies 。默认的会话引擎把会话保存在数据库中，但是正如你一会儿会看到的那样，你也可以选择不同的会话引擎。为了使用会话，你必须确认你项目

的 MIDDLEWARE_CLASSES 设置中包含了 django.contrib.sessions.middleware.SessionMiddleware 。这个中间件负责控制会话，并且是在你使用命令 startproject 创建项目时被默认添加的。
会话中间件使当前会话在 request 对象中可用。你可以用 request.seesion 连接当前会话，它的使用方式和 Python 的字典相似。会话字典接收任何默认的可被序列化为 JSON 的 Python 对象。你可以在会话中像这样设置变量：

```
request.session['foo'] = 'bar'
```

检索会话中的键:

```
request.session.get('foo')
```

删除会话中已有键：

```
del request.session['foo']
```

正如你所见，我们像使用 Python 字典一样使用 request.session 。
当用户登录时，他们的匿名会话将会丢失，然后新的会话将会为认证后的用户创建。如果你在匿名会话中储存了在登录后依然需要被持有的数据，你需要从旧的会话中复制数据到新的会话。

会话设置

你可以使用几种设置来为你的项目配置会话系统。最重要的部分是 SESSION_ENGINE .这个设置让你可以配置会话将会在哪里被储存。默认地， Django 用 django.contrib.sessions 的 Sessions 模型把会话保存在数据库中。
Django 提供了以下几个选择来保存会话数据：

- Database sessions（数据库会话）:会话数据将会被保存在数据库中。这是默认的会话引擎。
- File-based sessions（基于文件的会话）：会话数据保存在文件系统中。
- Cached sessions（缓存会话）:会话数据保存在缓存后端中。你可以使用 CACHES 设置来指定一个缓存后端。在缓存系统中保存会话拥有最好的性能。
- Cached sessions（缓存会话）：会话数据储存于缓存后端。你可以使用 CACHES 设置来制定一个缓存后端。在缓存系统中储存会话数据会有更好的性能表现。
- Cached database sessions（缓存于数据库中的会话）：会话数据保存于可高速写入的缓存和数据库中。只会在缓存中没有数据时才会从数据库中读取数据。
- Cookie-based sessions（基于 cookie 的会话）：会话数据储存于发送向浏览器的 cookie 中。

为了得到更好的性能，使用基于缓存的会话引擎（ cach-based session engine）吧。 Django 支持 Mercached ，以及 Redis 的第三方缓存后端和其他的缓存系统。
你可以用其他的设置来定制你的会话。这里有一些和会话有关的重要设置：

- SESSION_COOKIE_AGE：cookie 会话保持的时间。以秒为单位。默认值为 1209600 （2 周）。
- SESSION_COOKIE_DOMAIN：这是为会话 cookie 使用的域名。把它的值设置为 .mydomain.com 来使跨域名 cookie 生效。
- SESSION_COOKIE_SECURE：这是一个布尔值。它表示只有在连接为 HTTPS 时 cookie 才会被发送。
- SESSION_EXPIRE_AT_BROWSER_CLOSE：这是一个布尔值。它表示会话会在浏览器关闭时就过期。
- SESSION_SAVE_EVERY_REQUEST：这是一个布尔值。如果为 True ，每一次请求的 session 都将会被储存进数据库中。 session 的过期时间也会每次刷新。

在这个网站你可以看到所有的 session 设置：
https://docs.djangoproject.com/en/1.8/ref/settings/#sessions

会话过期

你可以通过 SESSION_EXPIRE_AT_BROWSER_CLOSE 选择使用 browser-length 会话或者持久会话。默认的设置是 False ，强制把会话的有效期设置为 SESSION_COOKIE_AGE 的值。如果你
把 SESSION_EXPIRE_AT_BROWSER_CLOSE 的值设为 True，会话将会在用户关闭浏览器时过期，
且 SESSION_COOKIE_AGE 将不会对此有任何影响。

你可以使用 `request.session` 的 `set_expiry()` 方法来覆写当前会话的有效期。

在会话中保存购物车

我们需要创建一个能序列化为 JSON 的简单结构，这样就可以把购物车中的东西储存在会话中。购物车必须包含以下数据，每个物品的数据都要包含在其中：

- `Product` 实例的 `id`
- 选择的产品数量
- 产品的总价格

因为产品的价格可能会变化，我们采取当产品被添加进购物车时同时保存产品价格和产品本身的办法。这样做，我们就可以保持用户在把商品添加进购物车时他们看到的商品价格不变了，即使产品的价格在之后有了变更。

现在，你需要把购物车和会话关联起来。购物车像下面这样工作：

- 当需要一个购物车时，我们检查顾客是否已经设置了一个会话键（ session key）。如果会话中没有购物车，我们就创建一个新的购物车，然后把它保存在购物车的会话键中。
- 对于连续的请求，我们在会话键中执行相同的检查和获取购物车内物品的操作。我们在会话中检索购物车的物品和他们在数据库中相关联的 `Product` 对象。

编辑你的项目中 `settings.py`，把以下设置添加进去：

```
CART_SESSION_ID = 'cart'
```

添加的这个键将会用于我们的会话中来储存购物车。因为 Django 的会话对于每个访问者是独立的**（译者@ucag 注：原文为 per-visitor ，没能想出一个和它对应的中文词，根据上下文，我就把这个词翻译为了一个短语）**，我们可以在所有的会话中使用相同的会话键。

让我们创建一个应用来管理我们的购物车。打开终端，然后创建一个新的应用，在项目路径下运行以下命令：

```
python manage.py startapp cart
```

然后编辑你添加的项目中的 `settings.py` ，在 `INSTALLED_APPS` 中添加 `cart`：

```
INSTALLED_APPS = (
    # ...
    'shop',
    'cart',
)
```

在 `cart` 应用路径内创建一个新的文件，命名为 `cart.py` ，把以下代码添加进去：

```
from decimal import Decimal
from django.conf import settings
from shop.models import Product

class Cart(object):
    def __init__(self, request):
        """
        Initialize the cart.
        """
        self.session = request.session
        cart = self.session.get(settings.CART_SESSION_ID)
        if not cart:
            # save an empty cart in the session
            cart = self.session[settings.CART_SESSION_ID] = {}
```

```
        self.cart = cart
```

这个 `Cart` 类可以让我们管理购物车。我们需要把购物车与一个 `request` 对象一同初始化。我们使用 `self.session = request.session` 保存当前会话以便使其对 `Cart` 类的其他方法可用。首先，我们使用 `self.session.get(settings.CART_SESSION_ID)` 尝试从当前会话中获取购物车。如果当前会话中没有购物车，我们就在会话中设置一个空字典，这样就可以在会话中设置一个空的购物车。我们希望我们的购物车字典使用产品 ID 作为键，以数量和价格为键值对的字典为值。这样做，我们就能保证一个产品在购物车当中不被重复添加；我们也能简化获取任意购物车物品数据的步骤。
让我们写一个方法来向购物车当中添加产品或者更新产品的数量。把 `save()` 和 `add()` 方法添加进 `Cart` 类当中：

```
def add(self, product, quantity=1, update_quantity=False):
    """
    Add a product to the cart or update its quantity.
    """
    product_id = str(product.id)
    if product_id not in self.cart:
        self.cart[product_id] = {'quantity': 0,
                                 'price': str(product.price)}
    if update_quantity:
        self.cart[product_id]['quantity'] = quantity
    else:
        self.cart[product_id]['quantity'] += quantity
    self.save()

def save(self):
    # update the session cart
    self.session[settings.CART_SESSION_ID] = self.cart
    # mark the session as "modified" to make sure it is saved
    self.session.modified = True
```

`add()` 函数接受以下参数：

- `product`：需要在购物车中更新或者向购物车添加的 `Product` 对象
- `quantity`：一个产品数量的可选参数。默认为 1
- `update_quantity`：这是一个布尔值，它表示数量是否需要按照给定的数量参数更新（`True`），不然新的数量必须要被加进已存在的数量中（`False`）

我们在购物车字典中把产品 `id` 作为键。我们把产品 `id` 转换为字符串，因为 Django 使用 JSON 来序列化会话数据，而 JSON 又只接受支字符串的键名。产品 `id` 为键，一个有 `quantity` 和 `price` 的字典作为值。产品的价格从十进制数转换为了字符串，这样才能将它序列化。最后，我们调用 `save()` 方法把购物车保存到会话中。
`save()` 方法会把购物车中所有的改动都保存到会话中，然后用 `session.modified = True` 标记改动了的会话。这是为了告诉 Django 会话已经被改动，需要将它保存起来。
我们也需要一个方法来从购物车当中删除购物车。把下面的方法添加进 `Cart` 类当中：

```
def remove(self, product):
    """
    Remove a product from the cart.
    """
    product_id = str(product.id)
    if product_id in self.cart:
        del self.cart[product_id]
```

```
        self.save()
```

`remove` 方法从购物车字典中删除给定的产品，然后调用 `save()` 方法来更新会话中的购物车。
我们将迭代购物车当中的物品，然后获取相应的 `Product` 实例。为恶劣达到我们的目的，你需要定义 `__iter__()` 方法。把下列代码添加进 `Cart` 类中：

```
def __iter__(self):
    """
    Iterate over the items in the cart and get the products
    from the database.
    """
    product_ids = self.cart.keys()
    # get the product objects and add them to the cart
    products = Product.objects.filter(id__in=product_ids)
    for product in products:
        self.cart[str(product.id)]['product'] = product

    for item in self.cart.values():
        item['price'] = Decimal(item['price'])
        item['total_price'] = item['price'] * item['quantity']
        yield item
```

在 `__iter__()` 方法中，我们检索购物车中的 `Product` 实例来把他们添加进购物车的物品中。之后，我们迭代所有的购物车物品，把他们的 `price` 转换回十进制数，然后为每个添加一个 `total_price` 属性。现在我们就可以很容易的在购物车当中迭代物品了。
我们还需要一个方法来返回购物车中物品的总数量。当 `len()` 方法在一个对象上执行时，Python 会调用对象的 `__len__()` 方法来检索它的长度。我们将会定义一个定制的 `__len__()` 方法来返回保存在购物车中保存的所有物品数量。把下面这个 `__len__()` 方法添加进 `Cart` 类中：

```
def __len__(self):
    """
    Count all items in the cart.
    """
    return sum(item['quantity'] for item in self.cart.values())
```

我们返回所有购物车物品的数量。
添加下列方法来计算购物车中物品的总价：

```
def get_total_price(self):
    return sum(Decimal(item['price']) * item['quantity'] for item in self.cart.values())
```

最后，添加一个方法来清空购物车会话：

```
def clear(self):
    # remove cart from session
    del self.session[settings.CART_SESSION_ID]
        self.session.modified = True
```

我们的 `Cart` 类现在已经准备好管理购物车了。

创建购物车视图

既然我们已经创建了 `Cart` 类来管理购物车，我们就需要创建添加，更新，或者删除物品的视图了。我们需要创建以下视图：

- 用于添加或者更新物品的视图，且能够控制当前的和更新的数量
- 从购物车中删除物品的视图
- 展示购物车物品和总数的视图

添加物品

为了把物品添加进购物车，我们需要一个允许用户选择数量的表单。在 `cart` 应用路径下创建一个 `forms.py` 文件，然后添加以下代码：

```
from django import forms

PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]

class CartAddProductForm(forms.Form):
    quantity = forms.TypedChoiceField(
                choices=PRODUCT_QUANTITY_CHOICES,
                coerce=int)
    update = forms.BooleanField(required=False,
                initial=False,
                widget=forms.HiddenInput)
```

我们将要使用这个表单来向购物车添加产品。我们的 `CartAddProductForm` 类包含以下两个字段：

- `quantity`：让用户可以在 1~20 之间选择产品的数量。我们使用了带有 `coerce=int` 的 `TypeChoiceField` 字段来把输入转换为整数
- `update`：让你展示数量是否要被加进已当前的产品数量上（`False`），否则如果当前数量必须被用给定的数量给更新（`True`）。我们为这个字段使用了 `HiddenInput` 控件，因为我们不想把它展示给用户。

让我们一个新的视图来想购物车中添加物品。编辑 `cart` 应用的 `views.py`，添加以下代码：

```
from django.shortcuts import render, redirect, get_object_or_404
from django.views.decorators.http import require_POST
from shop.models import Product
from .cart import Cart
from .forms import CartAddProductForm

@require_POST
def cart_add(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    form = CartAddProductForm(request.POST)
    if form.is_valid():
        cd = form.cleaned_data
        cart.add(product=product,
                 quantity=cd['quantity'],
                 update_quantity=cd['update'])
    return redirect('cart:cart_detail')
```

这个视图是为了想购物车添加新的产品或者更新当前产品的数量。我们使用 `require_POST` 装饰器来只响应 POST 请求，因为这个视图将会变更数据。这个视图接收产品 ID 作为参数。我们用给定的 ID 来检索 `Product` 实例，然后验证 `CartAddProductForm`。如果表单是合法的，我们将在购物车中添加或者更新产品。我们将创建 `cart_detail` 视图。

我们还需要一个视图来删除购物车中的物品。将以下代码添加进 `cart` 应用的 `views.py` 中：

```
def cart_remove(request, product_id):
    cart = Cart(request)
    product = get_object_or_404(Product, id=product_id)
    cart.remove(product)
    return redirect('cart:cart_detail')
```

`cart_detail` 视图接收产品 ID 作为参数。我们根据给定的产品 ID 检索相应的 `Product` 实例，然后将它从购物车中删除。然后，我们将用户重定向到 `cart_detail` URL。
最后，我们需要一个视图来展示购物车和其中的物品。讲一下代码添加进 `veiws.py` 中：

```
def cart_detail(request):
    cart = Cart(request)
    return render(request, 'cart/detail.html', {'cart': cart})
```

`cart_detail` 视图获取当前购物车并展示它。
我们已经创建了视图来向购物车中添加物品，或从购物车中更新数量，删除物品，还有展示他们。然我们为这些视图添加 URL 模式。在 `cart` 应用中创建一个新的文件，命名为 `urls.py`。把下面这些 URL 模式添加进去：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.cart_detail, name='cart_detail'),
    url(r'^add/(?P<product_id>\d+)/$',
            views.cart_add,
            name='cart_add'),
    url(r'^remove/(?P<product_id>\d+)/$',
            views.cart_remove,
            name='cart_remove'),
]
```

编辑 `myshop` 应用的主 `urls.py` 文件，添加以下 URL 模式来引用 `cart` URLs：

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^cart/', include('cart.urls', namespace='cart')),
    url(r'^', include('shop.urls', namespace='shop')),
]
```

确保你在 `shop.urls` 之前引用它，因为它比前者更加有限制性。

创建展示购物车的模板

`cart_add` 和 `cart_remove` 视图没有渲染任何模板，但是我们需要为 `cart_detail` 创建模板。
在 `cart` 应用路径下创建以下文件结构：

```
templates/
    cart/
        detail.html
```

编辑 `cart/detail.html` 模板，然后添加以下代码：

```
{% extends "shop/base.html" %}
{% load static %}

{% block title %}
    Your shopping cart
{% endblock %}

{% block content %}
    <h1>Your shopping cart</h1>
    <table class="cart">
        <thead>
            <tr>
                <th>Image</th>
                <th>Product</th>
                <th>Quantity</th>
                <th>Remove</th>
                <th>Unit price</th>
                <th>Price</th>
            </tr>
        </thead>
        <tbody>
        {% for item in cart %}
            {% with product=item.product %}
            <tr>
                <td>
                    <a href="{{ product.get_absolute_url }}">
                       ![]({% if product.image %}{{ product.image.url }}{% else %}{% static )
                    </a>
                </td>
                <td>{{ product.name }}</td>
                <td>{{ item.quantity }}</td>
                <td><a href="{% url "cart:cart_remove" product.id %}">Remove</a></td>
                <td class="num">${{ item.price }}</td>
                <td class="num">${{ item.total_price }}</td>
            </tr>
            {% endwith %}
        {% endfor %}
        <tr class="total">
            <td>Total</td>
            <td colspan="4"></td>
            <td class="num">${{ cart.get_total_price }}</td>
        </tr>
        </tbody>
    </table>
    <p class="text-right">
        <a href="{% url "shop:product_list" %}" class="button light">Continue shopping</a>
        <a href="#" class="button">Checkout</a>
    </p>
{% endblock %}
```

这个模板被用于展示购物车的内容。它包含了一个保存于当前购物车物品的表格。我们允许用用户使用发送到 `cart_add` 表单来改变选中的产品数量。我们通过提供一个 *Remove* 链接来允许用户从购物车中删除物品。

向购物车中添加物品

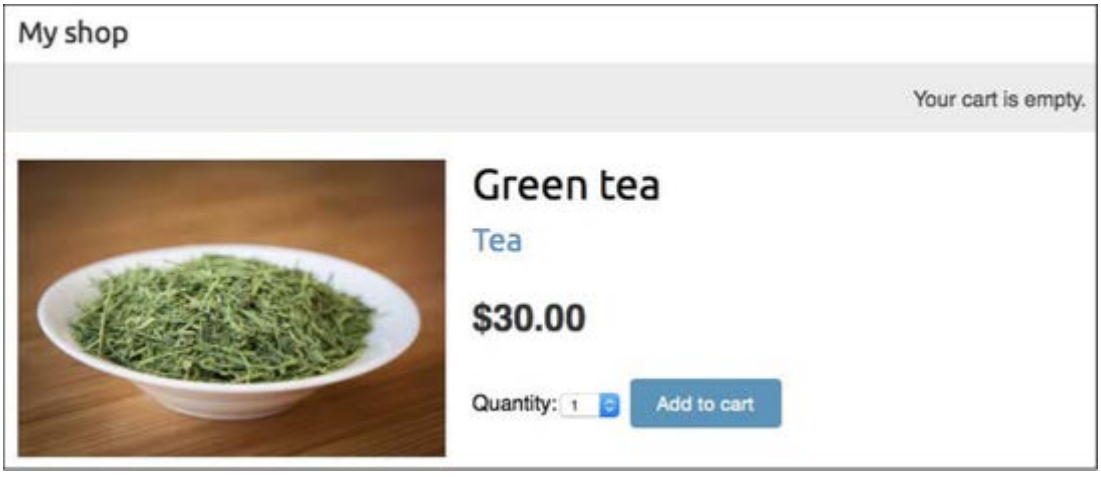
现在，我们需要在产品详情页添加一个 **Add to cart** 按钮。编辑 `shop` 应用中的 `views.py`，然后把 `CartAddProductForm` 添加进 `product_detail` 视图中：

```
from cart.forms import CartAddProductForm

def product_detail(request, id, slug):
    product = get_object_or_404(Product, id=id,
                                slug=slug,
                                available=True)
    cart_product_form = CartAddProductForm()
    return render(request,
           'shop/product/detail.html',
           {'product': product,
           'cart_product_form': cart_product_form})
```

编辑 `shop` 应用的 `shop/product/detail.html` 模板，然后将如下表格按照这样添加产品价格：

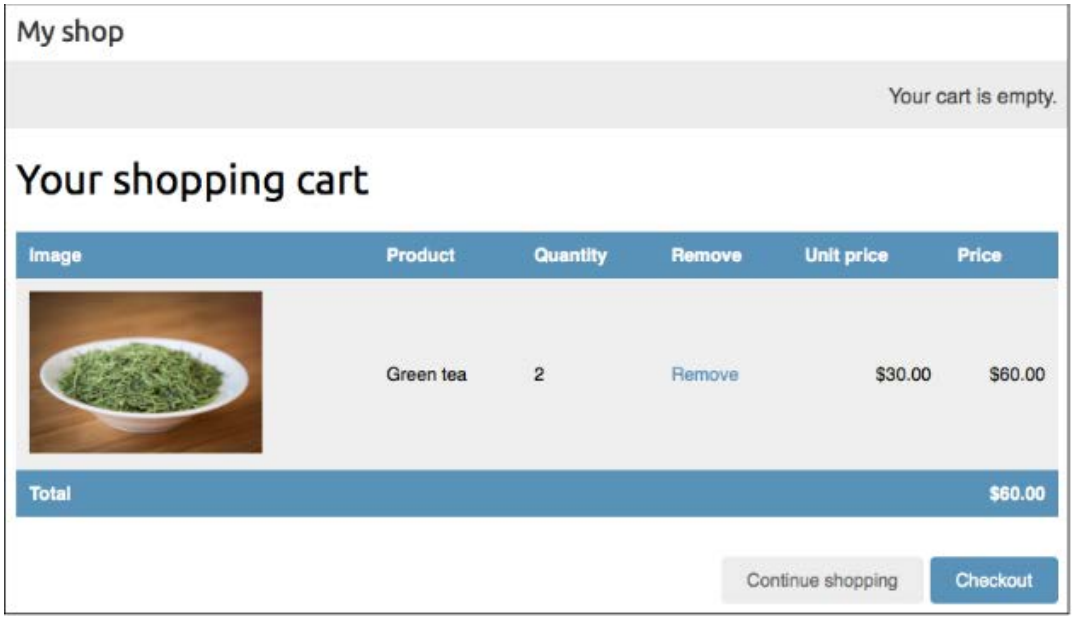
```
<p class="price">${{ product.price }}</p>
<form action="{% url "cart:cart_add" product.id %}" method="post">
{{ cart_product_form }}
{% csrf_token %}
<input type="submit" value="Add to cart">
</form>
```

确保用 `python manage.py runserver` 运行开发服务器。现在，打开 http://127.0.0.1:8000/，导航到产品详情页。现在它包含了一个表单来选择数量在将产品添加进购物车之前。这个页面看起来像这样：



django-7-5

选择一个数量，然后点击 **Add to cart** 按钮。表单将会通过 POST 方法提交到 `cart_add` 视图。视图会把产品添加进当前会话的购物车当中，包括当前产品的价格和选定的数量。然后，用户将会被重定向到购物车详情页，它长得像这个样子：



django-7-6

在购物车中更新产品数量

当用户看到购物车时，他们可能想要在下单之前改变产品数量。我们将会允许用户在详情页改变产品数量。

编辑 `cart` 应用的 `views.py`，然后把 `cart_detail` 改成这个样子：

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
                                initial={'quantity': item['quantity'],
                                'update': True})
    return render(request, 'cart/detail.html', {'cart': cart})
```

我们为每一个购物车中的物品创建了 `CartAddProductForm` 实例来允许用户改变产品的数量。我们把表单和当前物品数量一同初始化，然后把 `update` 字段设为 `True` ，这样当我们提交表单到 `cart_add` 视图时，当前的数量就被新的数量替换了。

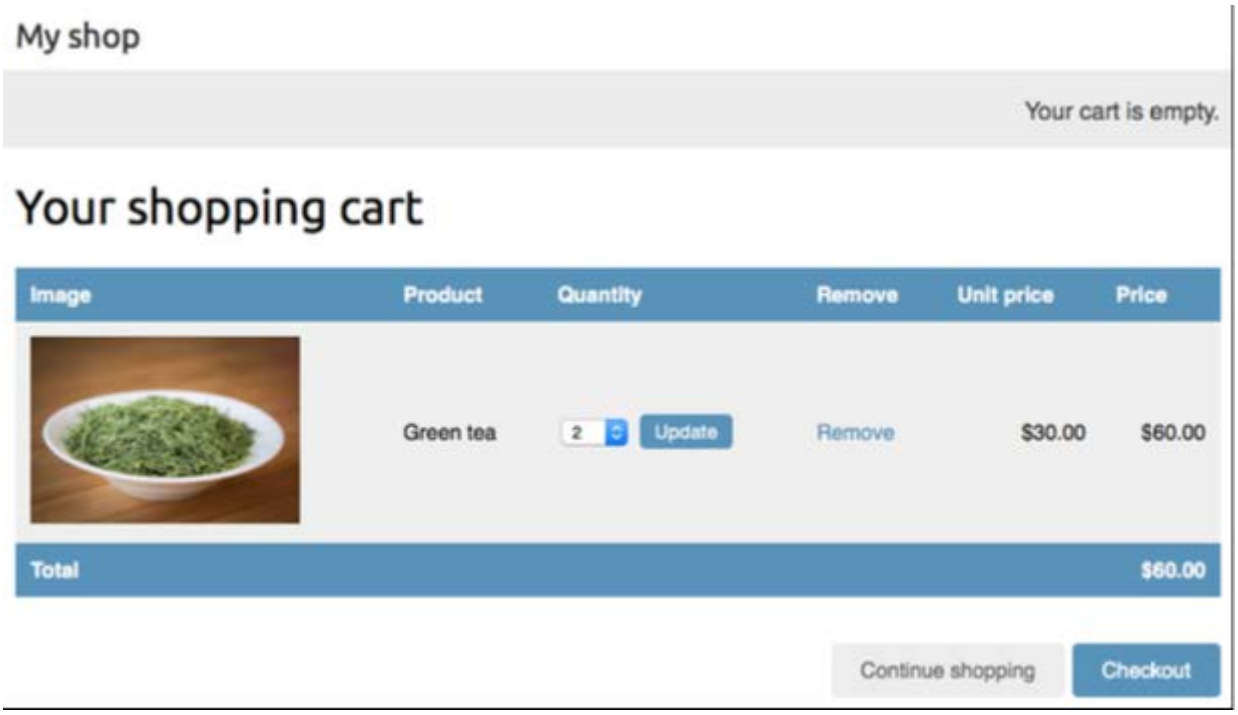
现在，编辑 `cart` 应用的 `cart/detail.html` 模板，然后找到这一行：

```
<td> {{ item.quantity }} </td>
```

把它替换为下面这样的代码：

```
<td>
<form action="{% url "cart:cart_add" product.id %}" method="post">
{{ item.update_quantity_form.quantity }}
{{ item.update_quantity_form.update }}
<input type="submit" value="Update">
{% csrf_token %}
</form>
</td>
```

在你的浏览器中打开 http://127.0.0.1:8000/cart/ 。你将会看到一个表单来编辑每个物品的数量，长得像下面这样：



django-7-7

改变物品的数量，然后点击 **Update** 按钮来测试新的功能。

为当前购物车创建上下文处理器

你可能已经注意到我们在网站的头部展示了 **Your cart is empty** 的信息。当我们开始向购物车添加物品时，我们将看到它已经替换为了购物车中物品的总数和总花费。由于这是个展示在整个页面的东西，我们将创建一个上下文处理器来引用当前请求中的购物车，尽管我们的视图函数已经处理了它。

上下文处理器

上下文处理器是一个接收 `request` 对象为参数并返回一个已经添加了请求上下文字典的 Python 函数。他们在你需要让什么东西在所有模板都可用时迟早会派上用场。

一般的，当你用 `startproject` 命令创建一个新的项目时，你的项目将会包含下面的模板上下文处理器，他们位于 `TEMPLATES` 设置中的 `context_processors` 内：

- `django.template.context_processors.debug`：在上下文中设置 `debug` 布尔值和 `sql_queries` 变量，来表示在 request 中执行的 SQL 查询语句表
- `django.template.context_processors.request`：在上下文中设置 request 变量
- `django.contrib.auth.context_processors.auth`：在请求中设置用户变量
- `django.contrib.messages.context_processors.messages`：在包含所有使用消息框架发送的信息的上下文中设置一个 `messages` 变量

Django 也使用 `django.template.context_processors.csrf` 来避免跨站请求攻击。这个上下文处理器不在设置中，但是它总是可用的并且由安全原因不可被关闭。

你可以在这个网站看到所有的内建上下文处理器：

https://docs.djangoproject.com/en/1.8/ref/templates/api/#built-in-template-context-processors

把购物车添加进请求上下文中

让我们创建一个上下文处理器来将当前购物车添加进模板请求上下文中。这样我们就可以在任意模板中获取任意购物车了。

在 `cart` 应用路径里添加一个新文件，并命名为 `context_processors.py` 。上下文处理器可以位于你代码中的任何地方，但是在这里创建他们将会使你的代码变得组织有序。将以下代码添加进去：

```
from .cart import Cart

def cart(request):
```

```
    return {'cart': Cart(request)}
```

如你所见，一个上下文处理器是一个函数，这个函数接收一个 `request` 对象作为参数，然后返回一个对象字典，这些对象可用于所有使用 `RequestContext` 渲染的模板。在我们的上下文处理器中，我们使用 `request` 对象实例化了购物车，然后让它作为一个名为 `cart` 的参数对模板可用。
编辑项目中的 `settings.py` ，然后把 `cart.context_processors.cart` 添加进 `TEMPLATE` 内的 `context_processors` 选项中。改变后的设置如下：

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'cart.context_processors.cart',
            ],
        },
    },
]
```

你的上下文处理器将会在使用 `RequestContext` 渲染 模板时执行。`cart` 变量将会被设置在模板上下文中。上下文处理器会在所有的使用 `RequestContext` 的请求中执行。你可能想要创建一个定制的模板标签来代替一个上下文处理器，如果你想要链接到数据库的话。
现在，编辑 `shop` 应用的 `shop/base.html` 模板，然后找到这一行：

```
<div class="cart">
Your cart is empty.
</div>
```

把它替换为下面的代码：

```
<div class="cart">
    {% with total_items=cart|length %}
        {% if cart|length > 0 %}
            Your cart:
            <a href="{% url "cart:cart_detail" %}">
                {{ total_items }} item{{ total_items|pluralize }},
                ${{ cart.get_total_price }}
            </a>
        {% else %}
            Your cart is empty.
        {% endif %}
    {% endwith %}
</div>
```

使用 `python manage.py runserver` 重载你的服务器。打开 http://127.0.0.1:8000/ ,添加一些产品到购物车里。在网站头里，你可以看到当前物品总数和总花费，就象这样：

django-7-8

保存用户订单

当购物车已经结账完毕时，你需要把订单保存进数据库中。订单将要保存客户信息和他们购买的产品信息。

使用下面的命令创建一个新的应用来管理用户订单：

```
python manage.py startapp orders
```

编辑项目中的 `settings.py` ，然后把 `orders` 添加进 `INSTALLED_APPS` 中：

```
INSTALLED_APPS = (
    # ...
    'orders',
)
```

现在你已经激活了你的新应用。

创建订单模型

你需要一个模型来保存订单的详细信息，第二个模型用来保存购买的物品，包括物品的价格和数量。

编辑 `orders` 应用的 `models.py` ，然后添加以下代码：

```
from django.db import models
from shop.models import Product

class Order(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField()
    address = models.CharField(max_length=250)
    postal_code = models.CharField(max_length=20)
    city = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    paid = models.BooleanField(default=False)

    class Meta:
        ordering = ('-created',)

    def __str__(self):
        return 'Order {}'.format(self.id)

    def get_total_cost(self):
        return sum(item.get_cost() for item in self.items.all())
```

```
class OrderItem(models.Model):
    order = models.ForeignKey(Order, related_name='items')
    product = models.ForeignKey(Product,
                  related_name='order_items')
    price = models.DecimalField(max_digits=10, decimal_places=2)
    quantity = models.PositiveIntegerField(default=1)

    def __str__(self):
        return '{}'.format(self.id)

    def get_cost(self):
        return self.price * self.quantity
```

`Order` 模型包含几个用户信息的字段和一个 `paid` 布尔值字段，这个字段默认值为 `False`。待会儿，我们将使用这个字段来区分支付和未支付订单。我们也定义了一个 `get_total_cost()` 方法来得到订单中购买物品的总花费。
`OrderItem` 模型让我们可以保存物品，数量和每个物品的支付价格。我们引用 `get_cost()`来返回物品的花费。
给 `orders` 应用下运行首次迁移：

```
python manage.py makemigrations
```

你将看到如下输出：

```
Migrations for 'orders':
    0001_initial.py:
        - Create model Order
        - Create model OrderItem
```

运行以下命令来应用新的迁移：

```
python manage.py migrate
```

你的订单模型已经同步到了数据库中

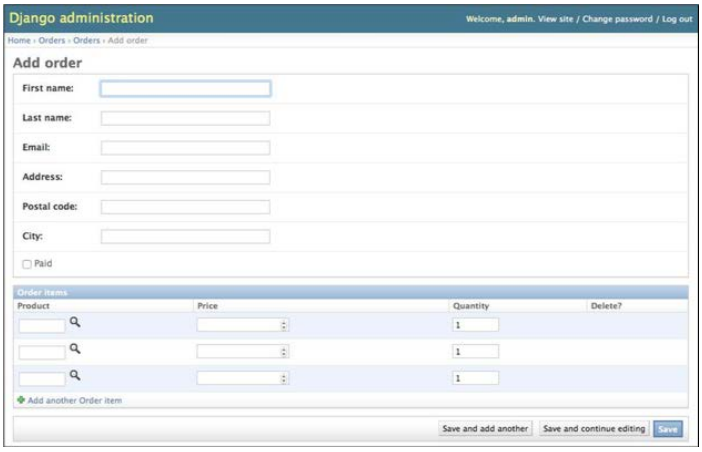
在管理站点引用订单模型

让我们把订单模型添加到管理站点。编辑 `orders` 应用的 `admin.py`：

```
from django.contrib import admin
from .models import Order, OrderItem

class OrderItemInline(admin.TabularInline):
    model = OrderItem
    raw_id_fields = ['product']

class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'first_name', 'last_name', 'email',
                    'address', 'postal_code', 'city', 'paid',
                    'created', 'updated']
    list_filter = ['paid', 'created', 'updated']
    inlines = [OrderItemInline]
```

```
admin.site.register(Order, OrderAdmin)
```

我们在 `OrderItem` 使用 `ModelInline` 来把它引用为 `OrderAdmin` 类的内联元素。一个内联元素允许你在同一编辑页引用模型，并且将这个模型作为父模型。
用 `python manage.py runserver` 命令打开开发服务器，访问 http://127.0.1:8000/admin/orders/order/add/ 。你将会看到如下页面：



django-7-9

创建顾客订单

我们需要使用订单模型来保存在用户最终下单时在购物车中的物品，创建新的订单的工作流程如下：

- 1. 向用户展示一个订单表来让他们填写数据
- 1. 我们用用户输入的数据创建一个新的 `Order` 实例，然后我们创建每个物品相关联的 `OrderItem` 实例。
- 1. 我们清空购物车，然后把用户重定向到成功页面

首先，我们需要一个表单来输入订单详情。在 `orders` 应用路径内创建一个新的文件，命名为 `forms.py` 。添加以下代码：

```
from django import forms
from .models import Order

class OrderCreateForm(forms.ModelForm):
    class Meta:
        model = Order
        fields = ['first_name', 'last_name', 'email', 'address',
                  'postal_code', 'city']
```

这是我们将要用于创建新的 `Order` 对象的表单。现在，我们需要一个视图来管理表格以及创建一个新的订单。编辑 `orders` 应用的 `views.py` ，添加以下代码：

```
from django.shortcuts import render
from .models import OrderItem
from .forms import OrderCreateForm
from cart.cart import Cart

def order_create(request):
    cart = Cart(request)
    if request.method == 'POST':
        form = OrderCreateForm(request.POST)
```

```
        if form.is_valid():
            order = form.save()
            for item in cart:
                OrderItem.objects.create(order=order,
                    product=item['product'],
                    price=item['price'],
                    quantity=item['quantity'])
            # clear the cart
            cart.clear()
            return render(request,
                'orders/order/created.html',
                {'order': order})
    else:
        form = OrderCreateForm()
    return render(request,
            'orders/order/create.html',
            {'cart': cart, 'form': form})
```

在 `order_create` 视图中，我们将用 `cart = Cart(request)` 获取到当前会话中的购物车。基于请求方法，我们将执行以下几个任务：

- **GET 请求**：实例化 `OrderCreateForm` 表单然后渲染模板 `orders/order/create.html`
- **POST 请求**：验证提交的数据。如果数据是合法的，我们将使用 `order = form.save()`来创建一个新的 `Order` 实例。然后我们将会把它保存进数据库中，之后再把它保存进 `order` 变量里。在创建 `order` 之后，我们将迭代无购车的物品然后为每个物品创建 `OrderItem`。最后，我们清空购物车。

现在，在 `orders` 应用路径下创建一个新的文件，把它命名为 `urls.py`。添加以下代码：

```
from django.conf.urls import url
from . import views

urlpatterns = [
        url(r'^create/$',
            views.order_create,
            name='order_create'),
]
```

这个是 `order_create` 视图的 URL 模式。编辑 `myshop` 的 `urls.py`，把下面的模式引用进去。记得要把它放在 `shop.urls` 模式之前：

```
url(r'^orders/', include('orders.urls', namespace='orders')),
```

编辑 `cart` 应用的 `cart/detail.html` 模板，找到下面这一行：

```
<a href="#" class="button">Checkout</a>
```

替换为：

```
<a href="{% url "orders:order_create" %}" class="button">
Checkout
</a>
```

用户现在可以从购物车详情页导航到订单表了。我们依然需要定义一个下单模板。在 `orders` 应用路径下创建如下文件结构：

```
templates/
    orders/
        order/
            create.html
            created.html
```

编辑 `ordrs/order/create.html` 模板，添加以下代码：

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

{% block title %}
Checkout
{% endblock %}

{% block content %}
    <h1>Checkout</h1>

    <div class="order-info">
        <h3>Your order</h3>
        <ul>
            {% for item in cart %}
            <li>
                {{ item.quantity }}x {{ item.product.name }}
                <span>${{ item.total_price }}</span>
            </li>
            {% endfor %}
        </ul>
    <p>Total: ${{ cart.get_total_price }}</p>
</div>

<form action="." method="post" class="order-form">
    {{ form.as_p }}
    <p><input type="submit" value="Place order"></p>
    {% csrf_token %}
</form>
{% endblock %}
```

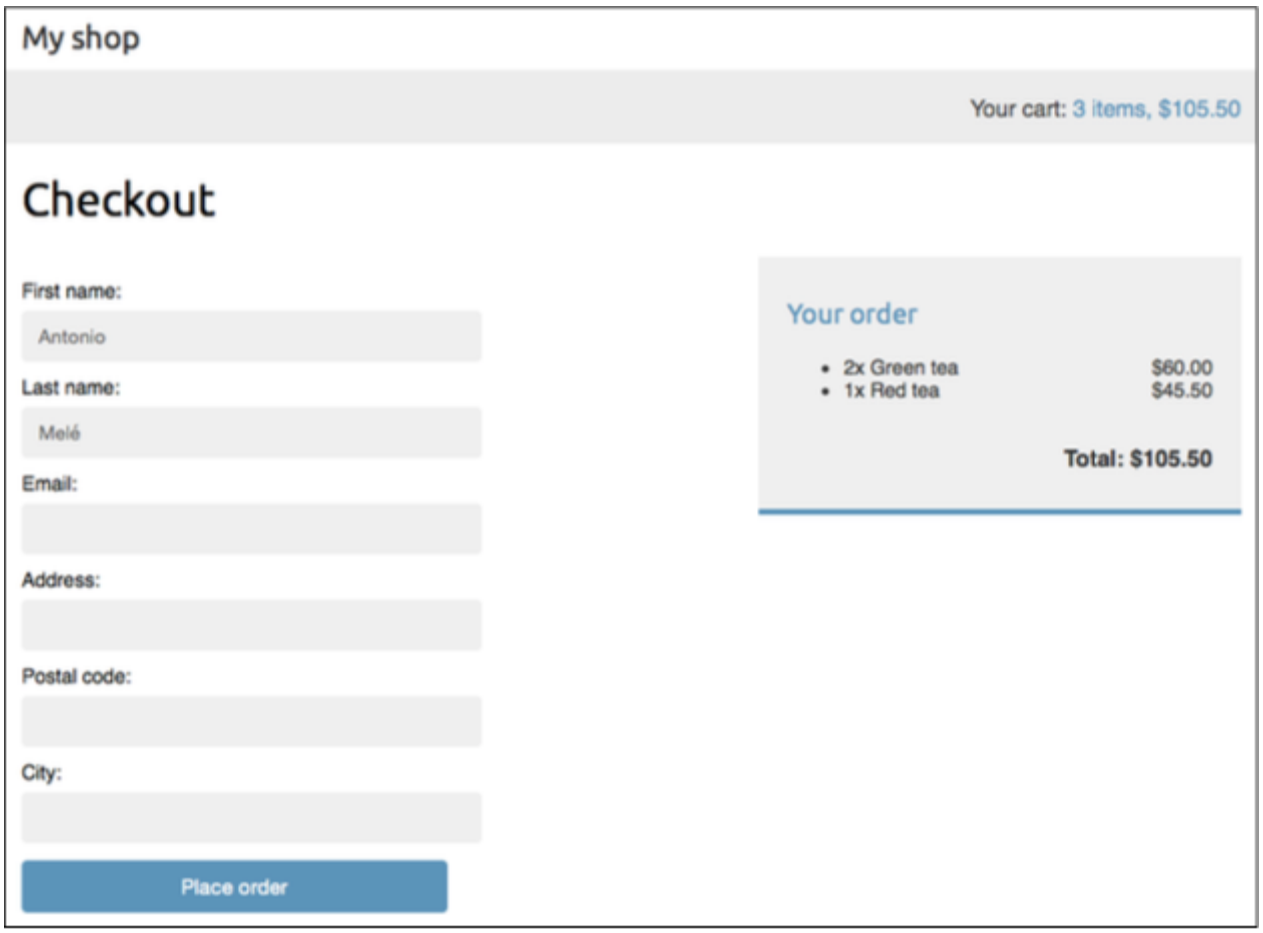
模板展示的购物车物品包括物品总量和下单表。

编辑 `orders/order/created.html` 模板，然后添加以下代码：

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

{% block title %}
Thank you
{% endblock %}

{% block content %}
    <h1>Thank you</h1>
    <p>Your order has been successfully completed. Your order number is
<strong>{{ order.id }}</strong>.</p>
```

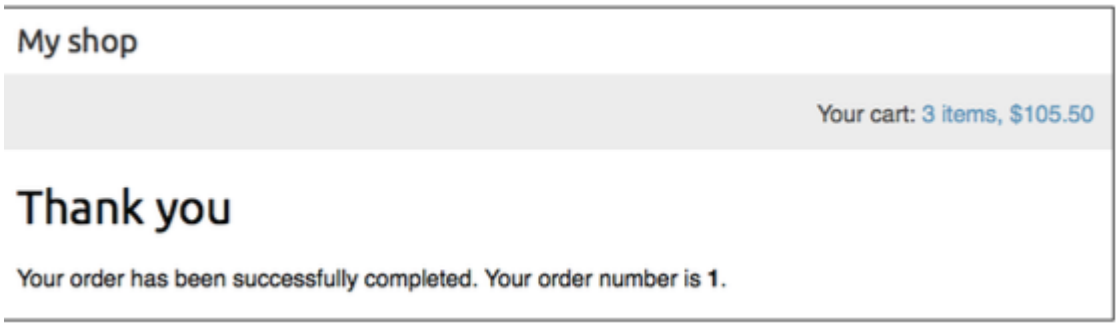
```
{% endblock %}
```

这是当订单成功创建时我们渲染的模板。打开开发服务器，访问 http://127.0.0.1:8000/，在购物车当中添加几个产品进去，然后结账。
你就会看到下面这个页面：



django-7-10

用合法的数据填写表单，然后点击 **Place order** 按钮。订单就会被创建，然后你将会看到成功页面：



django-7-11

使用 Celery 执行异步操作

你在视图执行的每个操作都会影响响应的时间。在很多场景下你可能想要尽快的给用户返回响应，并且让服务器异步地执行一些操作。这特别和耗时进程或从属于失败的进程时需要重新操作时有着密不可分的关系。比如，一个视频分享平台允许用户上传视频但是需要相当长的时间来转码上传的视频。这个网站可能会返回一个响应给用户，告诉他们转码即将开始，然后开始异步转码。另一个例子是给用户发送邮件。如果你的网站发送了通知邮件，SMTP 连接可能会失败或者减慢响应的速度。执行异步操作来避免阻塞执行就变得必要起来。
Celery 是一个分发队列，它可以处理大量的信息。它既可以执行实时操作也支持任务调度。使用 Celery 不仅可以让你很轻松的创建异步任务还可以让这些任务尽快执行，但是你需要在一个指定的时间调度他们执行。

你可以在这个网站找到 Celery 的官方文档：http://celery.readthedocs.org/en/latest/

安装 Celery

让我们安装 Celery 然后把它整合进你的项目中。用下面的命令安装 Celery：

```
pip install celery==3.1.18
```

Celery 需要一个消息代理（message broker）来管理请求。这个代理负责向 Celery 的 worker 发送消息，当接收到消息时 worker 就会执行任务。让我们安装一个消息代理。

安装 RabbitMQ

有几个 Celery 的消息代理可供选择，包括键值对储存，比如 Redis 或者是实时消息系统，比如 RabbitMQ。我们会用 RabbitMQ 配置 Celery ，因为它是 Celery 推荐的 message worker。
如果你用的是 Linux，你可以用下面这个命令安装 RabbitMQ ：

```
apt-get install rabbitmg
```

（译者@夜夜月注：这是 debian 系 linux 的安装方式）
如果你需要在 Mac OSX 或者 Windows 上安装 RabbitMQ，你可以在这个网站找到独立的支持版本：https://www.rabbitmq.com/download.html
在安装它之后，使用下面的命令执行 RabbitMQ：

```
rabbitmg-server
```

你将会在最后一行看到这样的输出：

```
Starting broker... completed with 10 plugins
```

RabbitMQ 正在运行了，准备接收消息。

把 Celery 添加进你的项目

你必须为 Celery 实例提供配置。在 `myshop` 的 `settings.py` 文件的旁边创建一个新的文件，命名为 `celery.py` 。这个文件会包含你项目的 Celery 配置。添加以下代码：

```
import os
from celery import Celery
from django.conf import settings

# set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'myshop.settings')

app = Celery('myshop')

app.config_from_object('django.conf:settings')
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

在这段代码中，我们为 Celery 命令行程序设置了 `DJANGO_SETTINGS_MODULE` 变量。然后我们用 `app=Celery('myshop')` 创建了一个实例。我们用 `config_from_object()` 方法来加载项目设置中任意的定制化配置。最后，我们告诉 Celery 自动查找我们列举在 `INSTALLED_APPS` 设置中的异步应用任务。Celery 将在每个应用路径下查找 `task.py` 来加载定义在其中的异步任务。
你需要在你项目中的 `__init__.py` 文件中导入 `celery` 来确保在 Django 开始的时候就会被加载。编辑 `myshop/__init__.py` 然后添加以下代码：

```
# import celery
from .celery import app as celery_app
```

现在，你可以为你的项目开始编写异步任务了。
`CELERY_ALWAYS_EAGER` 设置允许你在本地用异步的方式执行任务而不是把他们发送向队列中。这在不运行 Celery 的情况下， 运行单元测试或者是运行在本地环境中的项目是很有用的。

向你的应用中添加异步任务

我们将创建一个异步任务来发送消息邮件来让用户知道他们下单了。
约定俗成的一般用法是，在你的应用路径下的 `tasks` 模型里引入你应用的异步任务。在 `orders` 应用内创建一个新的文件，并命名为 `task.py` 。这是 Celery 寻找异步任务的地方。添加以下代码：

```
from celery import task
from django.core.mail import send_mail
from .models import Order

@task
def order_created(order_id):
    """
    Task to send an e-mail notification when an order is
    successfully created.
    """
    order = Order.objects.get(id=order_id)
    subject = 'Order nr. {}'.format(order.id)
    message = 'Dear {},\n\nYou have successfully placed an order.\
               Your order id is {}.'.format(order.first_name,
                                            order.id)
    mail_sent = send_mail(subject,
                          message,
                          'admin@myshop.com',
                          [order.email])
    return mail_sent
```

我们通过使用 `task` 装饰器来定义我们的 `order_created` 任务。如你所见，一个 Celery 任务 只是一个用 `task` 装饰的 Python 函数。我们的 `task` 函数接收一个 `order_id` 参数。通常推荐的做法是只传递 ID 给任务函数然后在任务被执行的时候需找相关的对象，我们使用 Django 提供的 `send_mail()` 函数来发送一封提示邮件给用户告诉他们下单了。如果你不想安装邮件设置，你可以通过一下 `settings`.py` 中的设置告诉 Django 把邮件传给控制台：

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

异步任务不仅仅适用于耗时进程，也适用于失败进程组中的进程，这些进程或许不会消耗太多时间，但是他们或许会链接失败或者需要再次尝试连接策略。
现在我们要把任务添加到 `order_create` 视图中。打开 `orders` 应用的 `views.py` 文件，按照如下导入任务：

```
from .tasks import order_created
```

然后在清除购物车之后调用 `order_created` 异步任务：

```
# clear the cart
cart.clear()
# launch asynchronous task
```

```
order_created.delay(order.id)
```

我们调用任务的 `delay()` 方法并异步地执行它。之后任务将会被添加进队列中，将会尽快被一个 worker 执行。

打开另外一个 shell ，使用以下命令开启 celery worker ：

```
celery -A myshop worker -l info
```

Celery worker 现在已经运行，准备好执行任务了。确保 Django 的开发服务器也在运行当中。访问 http://127.0.0.1:8000/ ,在购物车中添加一些商品，然后完成一个订单。在 shell 中，你已经打开过了 Celery worker 所以你可以看到以下的相似输出：

```
[2015-09-14 19:43:47,526: INFO/MainProcess] Received task: orders.
tasks.order_created[933e383c-095e-4cbd-b909-70c07e6a2ddf]
[2015-09-14 19:43:50,851: INFO/MainProcess] Task orders.tasks.
order_created[933e383c-095e-4cbd-b909-70c07e6a2ddf] succeeded in
3.318835098994896s: 1
```

任务已经被执行了，你会接收到一封订单通知邮件。

监控 Celery

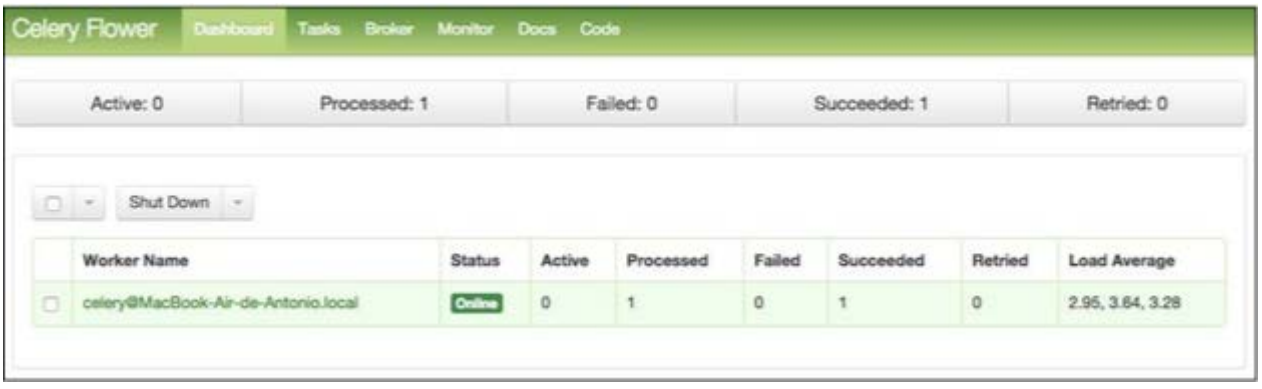
你或许想要监控执行了的异步任务。下面就是一个基于 web 的监控 Celery 的工具。你可以用下面的命令安装 Flower:

```
pip install flower
```

安装之后，你可以在你的项目路径下用以下命令启动 Flower ：

```
celery -A myshop flower
```

在你的浏览器中访问 http://localhost:5555/dashboard ，你可以看到激活了的 Celery worker 和正在执行的异步任务统计：



django-7-12

你可以在这个网站找到 Flower 的文档：http://flower.readthedocs.org/en/latest/

总结

在这一章中，你创建了一个最基本的商店应用。你创建了产品目录以及使用会话的购物车。你实现了定制化的上下文处理器来使购物车在你的模板中可用，实现了一个下单表格。你也学到了如何用 Celery 执行异步任务。

在下一章中，你将会学习在你的商店中整合一个支付网关，添加管理站点的定制化动作，以 CSV 的形式导出数据，以及动态的生成 PDF 文件。

第八章 管理付款和订单

在上一章，你创建了一个基础的在线商店包含一个产品列表以及订单系统。你还学习了如何执行异步的任务通过使用 Celery。在这一章中，你会学习到如何集成一个支付网关（**译者注：支付网关（Payment Gateway）是银行金融网络系统和 Internet 网络之间的接口，是由银行操作的将 Internet 上传输的数据转换为金融机构内部数据的一组服务器设备，或由指派的第三方处理商家支付信息和顾客的支付指令。以上是我百度的。**）到你的站点中。你还会扩展管理平台站点来管理订单和用不同的格式导出它们。

在这一章中，我们会覆盖以下几点：

- 集成一个支付网关到你的站点中
- 管理支付通知
- 导出订单为 CSV 格式
- 创建定制视图给管理页面
- 动态的生成 PDF 支票

集成一个支付网关

一个支付网关允许你在线处理支付。通过使用一个支付网关，你可以管理顾客的订单以及委托一个可靠的，安全的第三方处理支付。这意味着你无需担心存储信用卡信息到你的系统中。

PayPal 提供了多种方法来集成它的网管到你的站点中。标准的集成由一个 *Buy now* 按钮组成，这个按钮你可以已经在别的网站见到过（译者注：国内还是支付宝和微信比较多）。这个按钮会重定向购买者到 PayPal 去处理支付。我们将要集成 PayPal 支付标准包含一个定制的 *Buy now* 按钮到我们的站点中。PayPal 将会处理支付并且发送一个消息通知给我们的服务指明该笔支付的状态。

创建一个 PayPal 账户

你需要有一个 PayPal 商业账户来集成支付网关到你的站点中。如果你还没有一个 PayPal 账户，去 https://www.paypal.com/signup/account 注册。确保你选择了一个 *Bussiness Account* 并且注册成为 PayPal 支付标准解决方案，如下图所示：

django-8-0

填写你的详情在注册表单中并且完成注册流程。PayPal 会发送给你一封 e-mail 来核对你的账户。

安装 django-paypal

Django-paypal 是一个第三方 django 应用，它可以简化集成 PayPal 到 Django 项目中。我们将要使用它来集成 PayPal 支付标准解决方案到我们的商店中。你可以找到 django-paypal 的文档，访问 http://django-paypal.readthedocs.org/。

安装 django-paypal 在 shell 中通过以下命令：

```
pip install django-paypal==0.2.5
```

(译者注：现在应该有最新版本，书上使用的是 0.2.5 版本)
编辑你的项目中的 *settings.py* 文件，添加'paypal.standard.ipn'到 *INSTALLED_APPS* 设置中，如下所示：

```
INSTALLED_APPS = (
    # ...
    'paypal.standard.ipn',
)
```

这个应用提供自 django-paypal 来集成 PayPal 支付标准通过 **Instant Payment Notification(IPN)**。我们之后会操作支付通知。
添加以下设置到 *myshop* 的 *settings.py* 文件来配置 django-paypal：

```
# django-paypal settings
PAYPAL_RECEIVER_EMAIL = 'mypaypalemail@myshop.com'
PAYPAL_TEST = True
```

以上两个设置含义如下：

- PAYPAL_RECEIVER_EMAIL：你的 PayPal 账户 e-mail。使用你创建的 PayPal 账户 e-mail 替换 *mypaypalemail@myshop.com*。
- PAYPAL_TEST：一个布尔类型指示是否 PayPal 的沙箱环境，该环境可以用来处理支付。这个沙箱允许你测试你的 PayPal 集成在迁移到一个正式生产的环境之前。

打开 shell 运行如下命令来同步 django-paypal 的模型（models）到数据库中：

```
python manage.py migrate
```

你会看到如下类似的输出：

```
Running migrations:
    Rendering model states... DONE
    Applying ipn.0001_initial... OK
    Applying ipn.0002_paypalipn_mp_id... OK
    Applying ipn.0003_auto_20141117_1647... OK
```

django-paypal 的模型（models）如今已经同步到了数据库中。你还需要添加 django-paypal 的 URL 模式到你的项目中。编辑主的 *urls.py* 文件，该文件位于 *myshop* 目录，然后添加以下的 URL 模式。记住粘贴该 URL 模式要在 *shop.urls* 模式之前为了避免错误的模式匹配：

```
url(r'^paypal/', include('paypal.standard.ipn.urls')),
```

让我们添加支付网关到结账流程中。

添加支付网关

结账流程工作如下：

- 1.用户添加物品到他们的购物车中
- 2.用户结账他们的购物车
- 3.用户被重定向到 PayPal 进行支付
- 4.PayPal 发送一个支付通知给我们的站点
- 5.PayPal 重定向用户回到我们的网站

创建一个新的应用到你的项目中使用如下命令：

```
python manage.py startapp payment
```

我们将要使用这个应用去管理结账过程和用户支付。
编辑你的项目的 *settings.py* 文件，添加'payment'到 *INSTALLED_APPS* 设置中，如下所示：

```
INSTALLED_APPS = (
    # ...
    'paypal.standard.ipn',
    'payment',
)
```

payment 应用现在已经在项目中激活。编辑 *orders* 应用的 *views.py* 文件并且确保包含以下导入：

```
from django.shortcuts import render, redirect
from django.core.urlresolvers import reverse
```

替换以下 *order_create* 视图（view）的内容：

```
# launch asynchronous task
order_created.delay(order.id)
return render(request, 'orders/order/created.html', locals())
```

新的内容为：

```
# launch asynchronous task
order_created.delay(order.id) # set the order in the session
request.session['order_id'] = order.id # redirect to the payment
return redirect(reverse('payment:process'))
```

在成功的创建一个新的订单之后，我们设置这个订单 ID 到当前的会话中使用 *order_id* 会话键（session key）。之后，我们重定向用户到 `payment:process` URL，这个我们下一步就是创建。
编辑 *payment* 应用的 *views.py* 文件然后添加如下代码：

```
from decimal import Decimal
from django.conf import settings
from django.core.urlresolvers import reverse
from django.shortcuts import render, get_object_or_404
from paypal.standard.forms import PayPalPaymentsForm
from orders.models import Order

def payment_process(request):
    order_id = request.session.get('order_id')
    order = get_object_or_404(Order, id=order_id)
    host = request.get_host()
    paypal_dict = {
        'business': settings.PAYPAL_RECEIVER_EMAIL,
        'amount': '%.2f' % order.get_total_cost().quantize(
                                        Decimal('.01')),
        'item_name': 'Order {}'.format(order.id),
        'invoice': str(order.id),
        'currency_code': 'USD',
        'notify_url': 'http://{}{}'.format(host,
```

```
                                    reverse('paypal-ipn')),
        'return_url': 'http://{}{}'.format(host,
                                    reverse('payment:done')),
        'cancel_return': 'http://{}{}'.format(host,
                                  reverse('payment:canceled')),
      }
      form = PayPalPaymentsForm(initial=paypal_dict)
      return render(request,
                    'payment/process.html',
                    {'order': order, 'form':form})
```

在 `payment_process` 视图（view）中，我们生成了一个 PayPal 的 **Buy now** 按钮用来支付一个订单。首先，我们拿到当前的订单从 `order_id` 会话键中，这个键值被之前的 `order_create` 视图（view）设置。我们拿到这个 `order` 对象通过给予的 ID 并且构建一个新的 `PayPalPaymentsForm`，该表单表单包含以下字段：

- business：PayPal 商业账户用来处理支付。我们使用 e-mail 账户，该账户定义在 `PAYPAL_RECEIVER_EMAIL` 设置那里。
- amount：向顾客索要的总价。
- item_name：正在出售的商品名。我们使用订单 ID，因为订单可能包含很多产品。
- currency_code：本次支付的货币。我们设置这里为 USD 使用 U.S. Dollar**(译者注：传说中的美金)**。需要使用相同的货币，该货币被设置在你的 PayPal 账户中（例如：EUR 对应欧元）。
- notify_url：这个 URL PayPal 将会发送 IPN 请求过去。我们使用 django-paypal 提供的 `paypal-ipn` URL。这个视图（view）与这个 URL 关联来操作支付通知以及存储它们到数据库中。
- return_url：这个 URL 用来重定向用户当他的支付成功之后。我们使用 URL `payment:done`，这个我们接下来会创建。
- cancel_return：这个 URL 用来重定向用户如果这个支付被取消或者有其他问题。我们使用 URL `payment:canceled`，这个我们接下来会创建。

`PayPalpaymentsForm` 将会被渲染成一个标准表单带有隐藏的字段，并且用户将来只能看到 **Buy now** 按钮。当用户点击该按钮，这个表单将会提交到 PayPal 通过 POST 渠道。

让我们创建简单的视图（views）给 PayPal 用来重定向用户当支付成功，或者当支付被取消因为某些原因。添加以下代码到相同的 *views.py* 文件：

```
from django.views.decorators.csrf import csrf_exempt

@csrf_exempt
def payment_done(request):
    return render(request, 'payment/done.html')

@csrf_exempt
def payment_canceled(request):
    return render(request, 'payment/canceled.html')
```

我们使用 `csrf_exempt` 装饰器来避免 Django 期待一个 CSRF 标记，因为 PayPal 能重定向用户到以上两个视图（views）通过 POST 渠道。创建新的文件在 *payment* 应用目录下并且命名为 *urls.py*。添加以下代码：

```
from django.conf.urls import url
from . import views
urlpatterns = [
    url(r'^process/$', views.payment_process, name='process'),
    url(r'^done/$', views.payment_done, name='done'),
    url(r'^canceled/$', views.payment_canceled, name='canceled'),
```

```
]
```

这些 URL 是给支付工作流的。我们已经包含了以下 URL 模式：

- process：给这个视图（view）用来生成 PayPal 表单给 **Buy now** 按钮。
- done：给 PayPal 用来重定向用户当支付成功的时候。
- canceled：给 PayPal 用来重定向用户当支付取消的时候。

编辑主的 *myshop* 项目的 *urls.py* 文件，包含 URL 模式给 *payment* 应用：

```
url(r'^payment/', include('payment.urls',namespace='payment')),
```

记住粘贴以上内容在 `shop.urls` 模式之前用来避免错误的模式匹配。
创建以下文件建构在 *payment* 应用目录下：

```
templates/
    payment/
        process.html
        done.html
        canceled.html
```

编辑 *payment/process.html* 模板（template）并且添加以下代码：

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

{% block title %}Pay using PayPal{% endblock %}

{% block content %}
  <h1>Pay using PayPal</h1>
  {{ form.render }}
{% endblock %}
```

这个模板（template）会渲染 `PayPalPaymentsForm` 并且展示 **Buy now** 按钮。
编辑 *payment/done.html* 模板（template）并且添加如下代码：

```
{% extends "shop/base.html" %}
{% block content %}
    <h1>Your payment was successful</h1>
    <p>Your payment has been successfully received.</p>
{% endblock %}
```

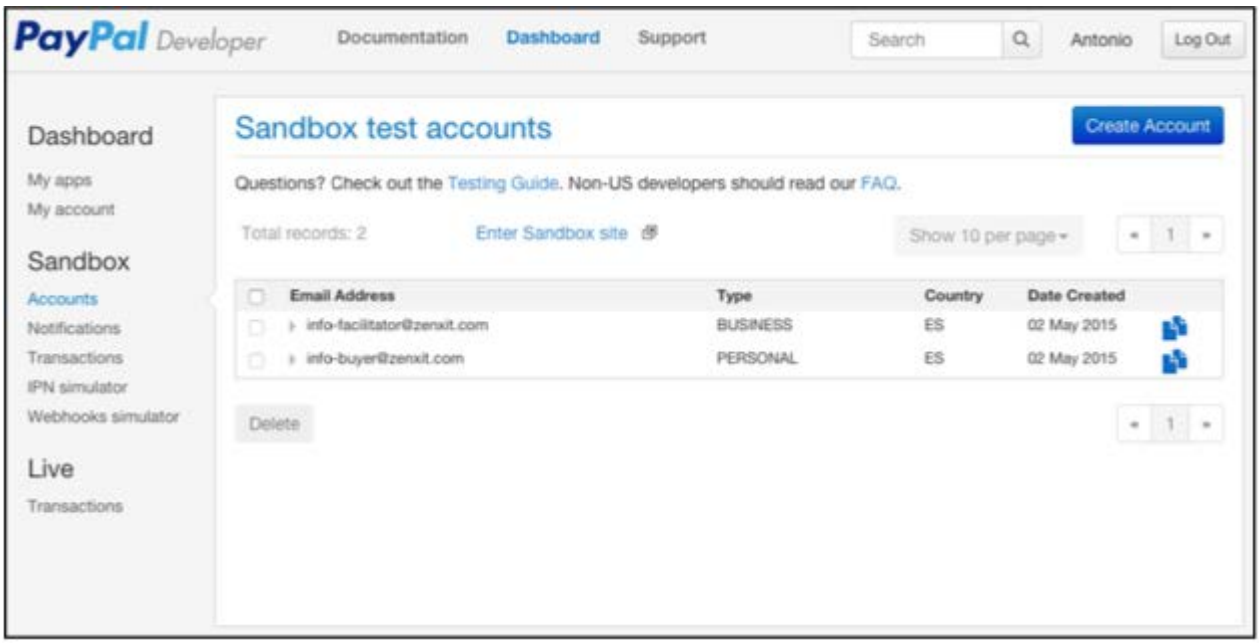
这个模板（template）的页面给用户重定向当成功支付之后。
编辑 *payment/canceled.html* 模板（template）并且添加以下代码：

```
{% extends "shop/base.html" %}
{% block content %}
    <h1>Your payment has not been processed</h1>
    <p>There was a problem processing your payment.</p>
{% endblock %}
```

这个模板（template）的页面给用户重定向当有这个支付过程出现问题或者用户取消了这次支付。
让我们尝试完成的支付过程。

使用 PayPal 的沙箱

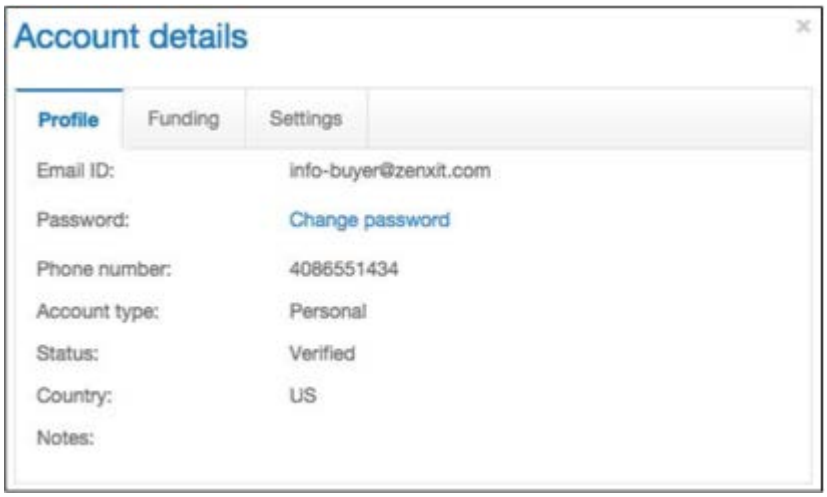
打开 http://developer.paypal.com 在你的浏览器中然后进行登录使用你的 PayPal 商业账户。点击 **Dashboard** 菜单项，在左方菜单点击 **Accounts** 选项在 **Sandbox** 下方。你会看到你的沙箱测试账户列，如下所示：



django-8-1

一开始，你将会看到一个商业以及一个个人测试账户由 PayPal 动态创建。你可以创建新的沙箱测试账户通过使用 **Create Account** 按钮。

点击 **Personal Account** 在列中扩大它，之后点击 **Profile** 链接。你会看到一些信息关于这个测试账户包含 e-mail 和 profile 信息，如下所示：



django-8-2

在 **Funding** tab 中，你会找到银行账户，信用卡日期，以及 PayPal 信用余额。

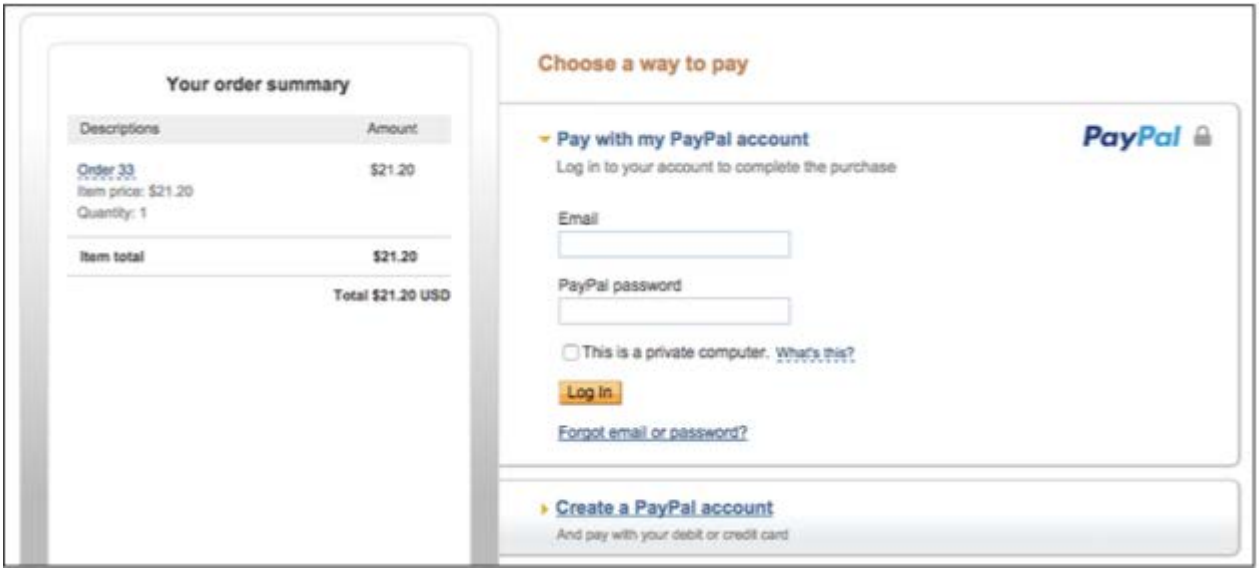
这些测试账户能够被用来做支付在你的网站中当使用沙箱环境。跳转到 **Profile** tab 然后点击 **Change password** 链接。创建一个定制密码给这个测试账户。

打开 shell 并且启动开发服务器使用命令 `python manage.py runserver`。打开 http://127.0.0.1:8000 在你的浏览器中，添加一些产品到购物车中，并且填写结账表单。当你点击 **Place order** 按钮，这个订单会被保存在数据库中，这个订单 ID 会被保存在当前的会话中，并且你会被重定向到支付处理页面。这个页面从会话中获取订单并且渲染 PayPal 表单显示一个 **Buy now** 按钮，如下所示：

django-8-3

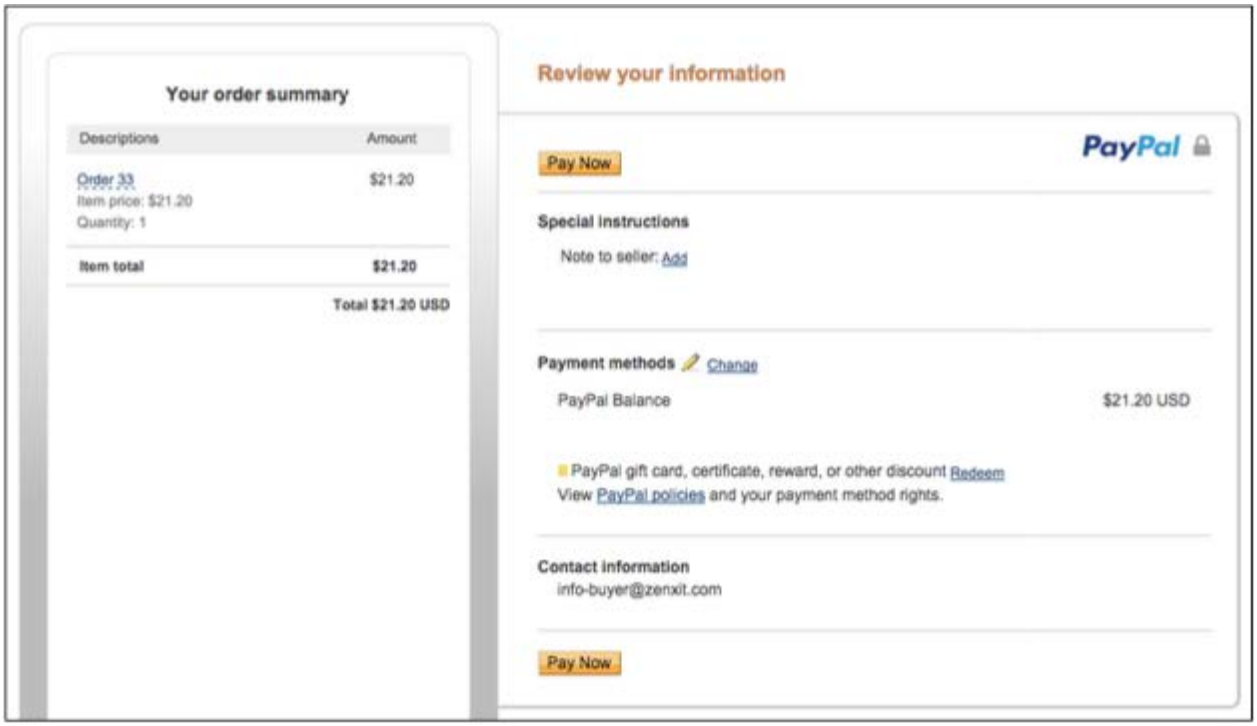
你可以看下 HTML 源码来看下生成的表单字段。

点击 **Buy now** 按钮。你会被重定向到 PayPal，并且你会看到如下页面：



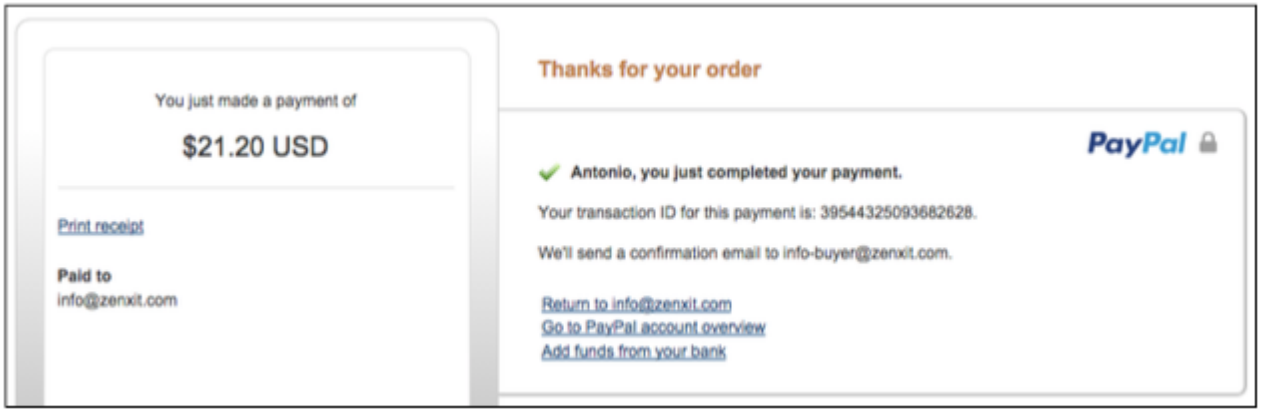
django-8-4

输入购买者测试账户 e-mail 和密码然后点击 **Log In** 按钮。你会被重定向到以下页面：



django-8-5

现在，点击 **Pay now** 按钮。最后，你会看到批准页面该页面包含你的交易 ID。这个页面看上去如下所示：



django-8-6

点击 **Return to e-mail@domain.com** 按钮。你会被重定向到的 URL 是你之前在 `PayPalPaymentsForm` 中的 `return_url` 字段中定义的。这个 URL 对应 `payment_done` 视图（view）。这个页面看上去如下所示：

django-8-7

这个支付已经成功了。然而，PayPal 并没有发送一个支付状态通知给我们的应用，因为我们运行我们的项目在我们本地主机，IP 是 127.0.0.1 这并不是一个公开地址。我们将要学习如何使我们的站点可以从 Internet 访问并且接收 IPN 通知。

获取支付通知

IPN 是一个方法提供自大部分的支付网关用来跟踪实时的购买。一个通知会立即发送到你的服务当这个网关处理了一个支付。这个通知包含所有支付详情，包括状态以及一个支付的签名，该签名可以用来确定这个消息的来源点。这个消息被发送通过一个单独的 HTTP 请求给你的服务。在出现连接问题的情况下，PayPal 将会多次企图通知你的站点。

django-paypal 应用内置两种不同的信号给 IPNs。如下：

- valid_ipn_received：会被触发当 IPN 信息获取自 PayPal 是正确的并且不是一个已存在数据库中的消息的复制。
- invalid_ipn_received：这个信号会触发当 IPN 获取自 PayPal 包含无效的数据或者不是一个良好的形式。

我们将要创建一个定制的接受函数并且连接它给 `valid_ipn_received` 信号用来确定支付。

创建新的文件在 *payment* 应用目录下，并且命名为 *signals.py*，添加如下代码：

```
from django.shortcuts import get_object_or_404
from paypal.standard.models import ST_PP_COMPLETED
from paypal.standard.ipn.signals import valid_ipn_received
from orders.models import Order

def payment_notification(sender, **kwargs):
    ipn_obj = sender
    if ipn_obj.payment_status == ST_PP_COMPLETED:
        # payment was successful
        order = get_object_or_404(Order, id=ipn_obj.invoice)
        # mark the order as paid
        order.paid = True
        order.save()

valid_ipn_received.connect(payment_notification)
```

我们连接 `payment_notification` 接收函数给 django-paypal 提供的 `valid_ipn_received` 信号。这个接收函数工作如下：

- 1.我们获取发送对象，该对象是一个 *PayPalIPN* 模型的实例，位于 `paypal.standard.ipn.models`。
- 2.我们检查 `payment_status` 属性来确保它和 django-payapl 的完整状态相同。这个状态指示这个支付已经成功处理。
- 3.之后我们使用 `get_object_or_404()`快捷函数来拿到订单，该订单的 ID 匹配 `invoice` 参数我们之前提供给 PayPal。

- 4.我们备注这个订单已经支付通过设置它的 `paid` 属性为 `True` 并且保存这个订单对象到数据库中。

你需要确保你的信号方法已经加载，这样这个接收函数会被调用当 `valid_ipn_received` 信号被触发的时候。The best practice is to load your signals when the application containing them is loaded. **（译者注：谁帮我翻一下，好拗口啊）**。这能够实现通过定义一个定制应用配置，这方面会在下一节进行解释。

配置我们的应用

你已经学习了关于应用的配置在*第六章 跟踪用户操作*。我们将要定义一个定制配置给我们的 *payment* 应用为了加载我们的信号接收函数。

创建一个新的文件在 *payment* 应用目录下命名为 *apps.py*。添加如下代码：

```
from django.apps import AppConfig

class PaymentConfig(AppConfig):
    name = 'payment'
    verbose_name = 'Payment'

    def ready(self):
        # import signal handlers
        import payment.signals
```

在上述代码中，我们定义了一个定制 `AppConfif` 类给 *payment* 应用。`name` 参数是这个应用的名字，*verbose_name* 包含可读的样式。我们导入信号方法在 `ready()` 方法中确保它们会被加载当这个应用初始化的时候。

编辑 *payment* 应用的 ***init.py*** 文件，添加以下行：

```
default_app_config = 'payment.apps.PaymentConfig'
```

以上操作可以使 Django 动态加载你的定制应用配置类。你可以找到更进一步的信息关于应用配置，通过访问 https://docs.djangoproject.com/en/1.8/ref/applications/ 。

测试支付通知

由于我们工作在本地环境中，我们需要确保我们的站点可以被 PayPal 获得。有不少应用允许你使你的开发环境在 Internet 中可获得。我们将要使用 Ngrok，它就是其中一个最著名的。

```
./ngrok http 8000
```

通过这条命名，你告诉 Ngrok 去创建一条隧道给你的本地主机在端口 8000 上并且分配一个 Internet 可访问主机名给它。你可以看到如下类似输出：

```
Tunnel Status     online
Version           2.0.17/2.0.17
Web Interface     http://127.0.0.1:4040
Forwarding        http://1a1b50f2.ngrok.io -> localhost:8000
Forwarding        https://1a1b50f2.ngrok.io -> localhost:8000

Connnections      ttl     opn     rt1     rt5     p50     p90
                  0       0       0.00    0.00    0.00    0.00
```

Ngrok 告诉我们关于我们的站点，运行在本地 8000 端口使用 Django 开发服务器，已经可以在 Internet 访问到通过 URLs http://1a1b50f2.ngrok.io 以及 https://1a1b50f2.ngrok.io ， 前者是 HTTP，后者是 HTTPS。Ngrok 还提供一个 URL 来访问一个 web 接口用来显示信息关于发送到这个服务的请求。

打开 Ngrok 提供的 URL 在浏览器中；例如，http://1a1b50f2.ngrok.io 。添加一些产品到购物车中，放置一个订单，然后使用你的 PayPal 测试账户进行支付。这个时候，PayPal 将能够拿到这个 URL，这个 URL 由 `PayPalPaymentsForm` 的 `notify_url` 字段生成，在 `payment_process` 视图（view）中。如果你看一下这个渲染过的表单，你会看到这个 HTML 表单字段看上去如下所示：

```
<input id="id_notify_url" name="notify_url" type="hidden"
value="http://1a1b50f2.ngrok.io/paypal/">
```

在结束支付过程之后，打开 http://127.0.0.1:8000/admin/ipn/paypalipn/ 在你的浏览器中。你会看到一个 IPN 对象对应最新的支付状态为 **Completed**。这个对象包含所有的支付信息，该对象由 PayPal 发送给你提供给 IPN 通知的 URL。IPN 管理列展示页面看上去如下所示：

django-8-8

你还可以启动 IPNs 通过使用 PayPal 的 IPN 模拟器位于 https://developer.paypal.com/developer/ipnSimulator/ 。这个模拟器允许你指定字段和发送的通知类型。

除了 PayPal 支付标准外，PayPal 提供 Website Payments Pro，它是一个订购服务允许你接受支付在你的站点中而不需要重定向用户到 PayPal。你可以找到更多信息关于如何集成 Website Payments Pro，通过访问 http://django-paypal.readthedocs.org/en/v0.2.5/pro/index.html。

导出订单为 CSV 文件

有时候，你可能想要导出包含在模型的信息到一个文件中，这样你可以导入它到其他的系统中。其中一个范围最广的格式用来导出/导入数据就是 **Comma-Separated Values(CSV)**。一个 CSV 文件就是一个纯文本文件包含若干记录。一个 csv 文件中，通常一行记录为一个订单，以及一些分割符（通常是逗号）。我们将要定制管理平台站点能够导出订单为 CSV 文件。

添加定制操作到管理平台站点中

Django 提供你多种不同的选项来定制管理平台站点。我们将要修改对象列视图（view）来包含一个定制的管理操作。一个管理操作工作如下：一个用户选择对象从管理对象列页面通过复选框，之后选择一个操作去执行在所有被选择的项上，然后执行该操作。以下展示操作会位于管理页面的哪个地方：

django-8-9

创建定制管理操作允许管理人员一次性应用操作多个元素。

你可以创建一个定制操作通过编写一个经常性的函数获取以下参数：

- 当前展示的 *ModelAdmin*
- 当前请求对象，一个 *HttpRequest* 实例
- 一个查询集（QuerySet）给用户所选择的对象

这个函数将会被执行当这个操作被触发在管理平台站点上。
我们将要创建一个定制管理操作来下载订单列表的 CSV 文件。编辑 *orders* 应用的 *admin.py* 文件，添加如下代码在 `OrderAdmin` 类之前：

```
import csv
import datetime
from django.http import HttpResponse
def export_to_csv(modeladmin, request, queryset):

    opts = modeladmin.model._meta
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment; \
          filename={}.csv'.format(opts.verbose_name)
    writer = csv.writer(response)
    fields = [field for field in opts.get_fields() if not field.many_to_many and not field.one_to_many]
    # Write a first row with header information
    writer.writerow([field.verbose_name for field in fields])
    # Write data rows
    for obj in queryset:
        data_row = []
        for field in fields:
            value = getattr(obj, field.name)
            if isinstance(value, datetime.datetime):
                value = value.strftime('%d/%m/%Y')
            data_row.append(value)
        writer.writerow(data_row)
    return response
export_to_csv.short_description = 'Export to CSV'
```

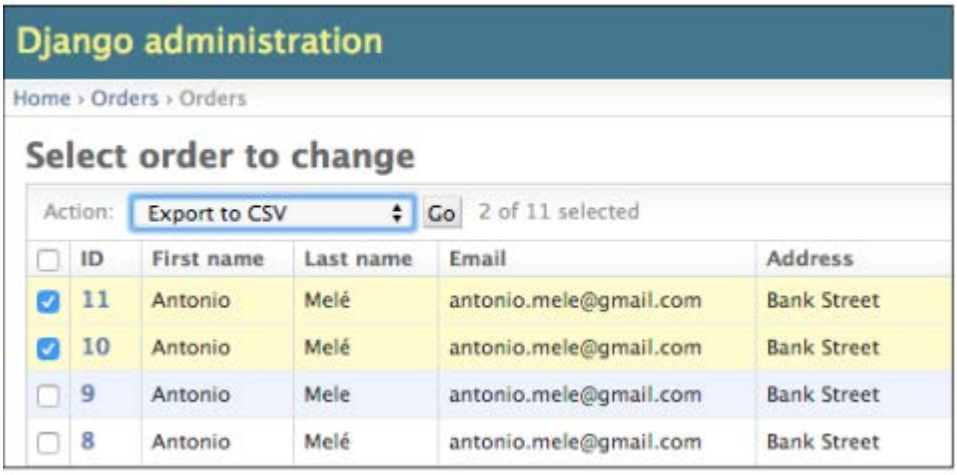
在这代码中，我们执行以下任务：

- 1.我们创建一个 `HttpResponse` 实例包含一个定制 `text/csv` 内容类型来告诉浏览器这个响应需要处理为一个 CSV 文件。我们还添加一个 `Content-Disposition` 头来指示这个 HTTP 响应包含一个附件。
- 2.我们创建一个 CSV `writer` 对象，该对象将会被写入 `response` 对象。
- 3.我们动态的获取 `model` 字段通过使用模型（moedl）`_meta` 选项的 `get_fields()`方法。我们排除多对多以及一对多的关系。
- 4.我们编写了一个头行包含字段名。
- 5.我们迭代给予的查询集（QuerySet）并且为每一个查询集中返回的对象写入行。我们注意格式化 `datetime` 对象因为这个输出值给 CSV 必须是一个字符串。
- 6.我们定制这个操作的显示名在模板(template)中通过设置一个 `short_description` 属性给这个函数。

我们已经创建了一个普通的管理操作可以添加到任意的 *ModelAdmin* 类。
最后，添加新的 `export_to_csv` 管理操作给 `OrderAdmin` 类如下所示：

```
class OrderAdmin(admin.ModelAdmin):
    # ...
    actions = [export_to_csv]
```

打开 http://127.0.0.1:8000/admin/orders/order/ 在你的浏览器中。管理操作看上去如下所示：



django-8-10

选择一些订单然后选择 **Export to CSV** 操作从下拉选框中，之后点击 **Go** 按钮。你的浏览器会下载生成的 CSV 文件名为 *order.csv*。打开下载的文件使用一个文本编辑器。你会看到的内容如以下的格式，包含一个头行以及你之前选择的每行订单对象：

```
ID,first name,last name,email,address,postal
code,city,created,updated,paid
3,Antonio,Melé,antonio.mele@gmail.com,Bank Street 33,WS J11,London,25/05/2015,25/05/2015,False
...
```

如你所见，创建管理操作是非常简单的。

扩展管理站点通过定制视图（view）

有时候你可能想要定制管理平台站点，比如处理 *ModelAdmin* 的配置，管理操作的创建，以及覆盖管理模板（templates）。在这样的场景中，你需要创建一个定制的管理视图（view）。通过一个定制的管理视图（view），你可以构建任何你需要的功能。你只需要确保只有管理用户能访问你的视图并且你维护这个管理的外观和感觉通过你的模板（template）扩展自一个管理模板（template）。
让我们创建一个定制视图（view）来展示关于一个订单的信息。编辑 *orders* 应用下的 *views.py* 文件，添加以下代码：

```
from django.contrib.admin.views.decorators import staff_member_required
from django.shortcuts import get_object_or_404
from .models import Order

@staff_member_required
def admin_order_detail(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    return render(request,
                  'admin/orders/order/detail.html',
                  {'order': order})
```

这个 `staff_member_required` 装饰器检查用户请求这个页面的 `is_active` 以及 `is_staff` 字段是被设置为 `True`。在这个视图（view）中，我们获取 `Order` 对象通过给予的 id 以及渲染一个模板来展示这个订单。
现在，编辑 *orders* 应用中的 *urls.py* 文件并且添加以下 URL 模式：

```
url(r'^admin/order/(?P<order_id>\d+)/$',
    views.admin_order_detail,
    name='admin_order_detail'),
```

创建以下文件结构在 *orders* 应用的 *templates/*目录下：

```
admin/
    orders/
        order/
            detail.html
```

编辑 *detail.html* 模板（template），添加以下内容：

```
{% extends "admin/base_site.html" %}
{% load static %}

{% block extrastyle %}
     <link rel="stylesheet" type="text/css" href="{% static "css/admin.css" %}" />
{% endblock %}

{% block title %}
     Order {{ order.id }} {{ block.super }}
{% endblock %}

{% block breadcrumbs %}
  <div class="breadcrumbs">
    <a href="{% url "admin:index" %}">Home</a> ›
    <a href="{% url "admin:orders_order_changelist" %}">Orders</a>
    ›
    <a href="{% url "admin:orders_order_change" order.id %}">Order {{ order.id }}</a>
    › Detail
  </div>
{% endblock %}

{% block content %}
  <h1>Order {{ order.id }}</h1>
  <ul class="object-tools">
    <li>
      <a href="#" onclick="window.print();">Print order</a>
    </li>
  </ul>
  <table>
    <tr>
      <th>Created</th>
      <td>{{ order.created }}</td>
    </tr>
    <tr>
      <th>Customer</th>
      <td>{{ order.first_name }} {{ order.last_name }}</td>
    </tr>
    <tr>
      <th>E-mail</th>
      <td><a href="mailto:{{ order.email }}">{{ order.email }}</a></td>
    </tr>
    <tr>
    <th>Address</th>
```

```
    <td>{{ order.address }}, {{ order.postal_code }} {{ order.city
}}</td>
  </tr>
    <tr>
      <th>Total amount</th>
      <td>${{ order.get_total_cost }}</td>
    </tr>
    <tr>
      <th>Status</th>
      <td>{% if order.paid %}Paid{% else %}Pending payment{% endif %}</td>
    </tr>
  </table>

  <div class="module">
    <div class="tabular inline-related last-related">
      <table>
        <h2>Items bought</h2>
        <thead>
          <tr>
            <th>Product</th>
            <th>Price</th>
            <th>Quantity</th>
            <th>Total</th>
          </tr>
        </thead>
        <tbody>
          {% for item in order.items.all %}
            <tr class="row{% cycle "1" "2" %}">
              <td>{{ item.product.name }}</td>
              <td class="num">${{ item.price }}</td>
              <td class="num">{{ item.quantity }}</td>
              <td class="num">${{ item.get_cost }}</td>
            </tr>
          {% endfor %}
          <tr class="total">
            <td colspan="3">Total</td>
            <td class="num">${{ order.get_total_cost }}</td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
{% endblock %}
```

这个模板（template）是用来显示一个订单详情在管理平台站点中。这个模板（template）扩展 Djnago 的管理平台站点的 *admin/base_site.html* 模板，它包含管理的主要 HTML 结构和 CSS 样式。我们加载定制的静态文件 *css/admin.css*。

为了使用静态文件，你需要拿到它们从这章教程的实例代码中。复制位于 *orders* 应用的 *static/*目录下的静态文件然后添加它们到你的项目的相同位置。

我们使用定义在父模板（template）的区块包含我们自己的内容。我们展示信息关于订单和购买的商品。
当你想要扩展一个管理模板（template），你需要知道它的结构以及确定存在的区块。你可以找到所有管理模板（template），通过访问 https://github.com/django/django/tree/1.8.6/django/contrib/admin/templates/admin 。
你也可以重写一个管理模板（template）如果你需要的话。为了重写一个管理模板（template），拷贝它到你的 *template* 目录保持相同的相对路径以及文件名。Django 管理平台站点将会使用你的定制模板（template）替代默认的模板。
最后，让我们添加一个链接给每个 *Order* 对象在管理平台站点的列展示页面。编辑 *orders* 应用的 *admin.py* 文件然后添加以下代码，在 `OrderAdmin` 类上面：

```
from django.core.urlresolvers import reverse
def order_detail(obj):
    return '<a href="{}">View</a>'.format(
        reverse('orders:admin_order_detail', args=[obj.id]))
order_detail.allow_tags = True
```

这个函数需要一个 *Order* 对象作为参数并且返回一个 HTML 链接给 `admind_order_detail`URL。Django 会避开默认的 HTML 输出。我们必须设置 `allow_tags` 属性为 `True` 来避开 auto-escaping。
设置 `allow_tags` 属性为 `True` 来避免 HTML-escaping 在一些 *Model* 方法，*ModelAdmin* 方法，以及任何其他的调用中。当你使用 `allow_tags` 的时候，能确保避开用户输入的跨域脚本。
之后，编辑 `OrderAdmin` 类来展示链接：

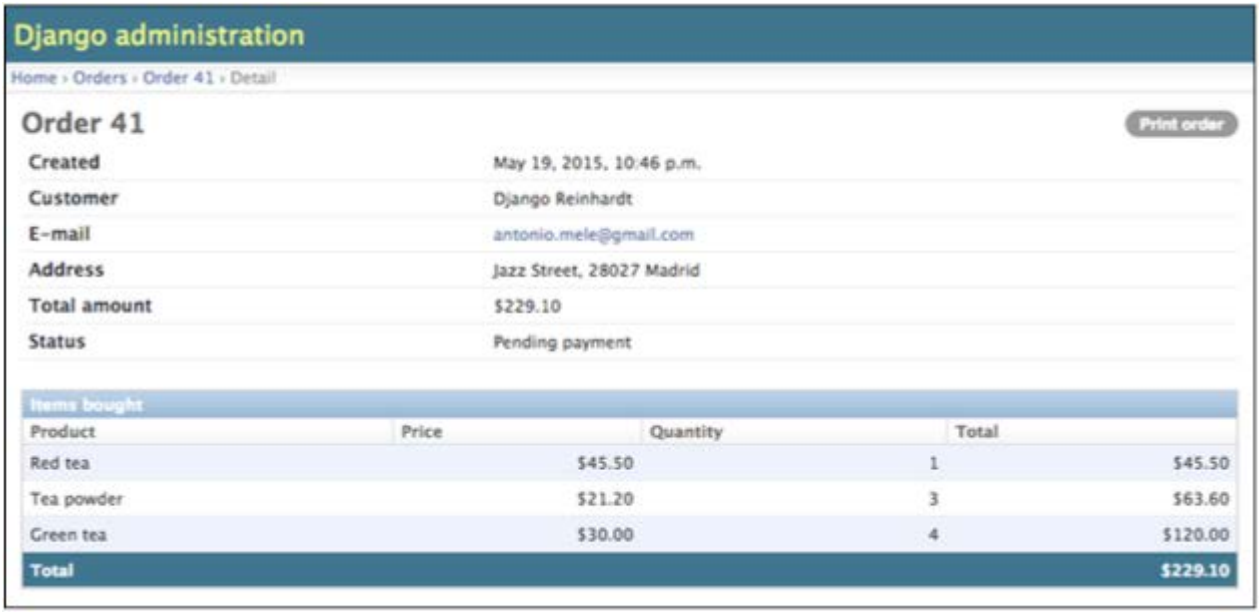
```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id',
                    'first_name',
                    # ...
                    'updated',
                    order_detail]
```

打开 http://127.0.0.1:8000/admin/orders/order/ 在你的浏览器中。每一行现在都会包含一个 **View** 链接如下所示：

City	Paid	Created	Updated	Order detail
Madrid	⛔	May 19, 2015, 3:16 p.m.	May 19, 2015, 3:16 p.m.	View

django-8-11

点击某个订单的 **View** 链接来加载定制订单详情页面。你会看到一个页面如下所示：



django-8-12

生成动态的 PDF 发票

如今我们已经有了一个完整的结账和支付系统，我们可以生成一张 PDF 发票给每个订单。有几个 Python 库可以生成 PDF 文件。一个最流行的生成 PDF 的 Python 库是 Reportlab。你可以找到关于如何使用 Reportlab 输出 PDF 文件的信息，通过访问 https://docs.djangoproject.com/en/1.8/howto/outputting-pdf/ 。
在大部分的场景中，你还需要添加定制样式和格式给你的 PDF 文件。你会发现渲染一个 HTML 模板（template）以及转化该模板（template）为一个 PDF 文件更加的方便，保持 Python 远离表现层。我们要遵循这个方法并且使用一个模块来生成 PDF 文件通过 Django。我们将要使用 WeasyPrint，它是一个 Python 库可以生成 PDF 文件从 HTML 模板中。

安装 WeasyPrint

首先，安装 WeasyPrint 的依赖给你的 OS，这些依赖你可以找到通过访问 http://weasyprint.org/docs/install/#platforms 。
之后，安装 WeasyPrint 通过 *pip* 渠道使用如下命令：

```
pip install WeasyPrint==0.24
```

创建一个 PDF 模板（template）

我们需要一个 HTML 文档给 WeasyPrint 输入。我们将要创建一个 HTML 模板（template），渲染它使用 Django，并且传递它给 WeasyPrint 来生成 PDF 文件。
创建一个新的模板（template）文件在 *orders* 应用的 *templates/orders/order/目录下命名为* pdf.html*。
添加如下内容：

```
<html>
<body>
    <h1>My Shop</h1>
    <p>
      Invoice no. {{ order.id }}</br>
      <span class="secondary">
        {{ order.created|date:"M d, Y" }}
      </span>
    </p>
```

```
    <h3>Bill to</h3>
    <p>
      {{ order.first_name }} {{ order.last_name }}<br>
      {{ order.email }}<br>
      {{ order.address }}<br>
      {{ order.postal_code }}, {{ order.city }}
    </p>
    <h3>Items bought</h3>
    <table>
      <thead>
        <tr>
          <th>Product</th>
          <th>Price</th>
          <th>Quantity</th>
          <th>Cost</th>
        </tr>
      </thead>
      <tbody>
        {% for item in order.items.all %}
          <tr class="row{% cycle "1" "2" %}">
            <td>{{ item.product.name }}</td>
            <td class="num">${{ item.price }}</td>
            <td class="num">{{ item.quantity }}</td>
            <td class="num">${{ item.get_cost }}</td>
          </tr>
        {% endfor %}
        <tr class="total">
          <td colspan="3">Total</td>
          <td class="num">${{ order.get_total_cost }}</td>
        </tr>
      </tbody>
    </table>

    <span class="{% if order.paid %}paid{% else %}pending{% endif %}">
      {% if order.paid %}Paid{% else %}Pending payment{% endif %}
    </span>
</body>
</html>
```

这个模板（template）就是 PDF 发票。在这个模板（template）中，我们展示所有订单详情以及一个 HTML `<table>` 元素包含所有商品。我们还包含了一条消息来展示如果该订单已经支付或者支付还在进行中。

渲染 PDF 文件

我们将要创建一个视图（view）来生成 PDF 发票给存在的订单通过使用管理平台站点。编辑 *order* 应用的 *views.py* 文件添加如下代码：

```
from django.conf import settings
from django.http import HttpResponse
from django.template.loader import render_to_string
```

```
import weasyprint

@staff_member_required
def admin_order_pdf(request, order_id):
    order = get_object_or_404(Order, id=order_id)
    html = render_to_string('orders/order/pdf.html',
                            {'order': order})
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = 'filename=\
          "order_{}.pdf"'.format(order.id)
    weasyprint.HTML(string=html).write_pdf(response,
        stylesheets=[weasyprint.CSS(
            settings.STATIC_ROOT + 'css/pdf.css')])
    return response
```

这个视图（view）用来生成一个 PDF 发票给一个订单。我们使用 `staff_member_required` 装饰器来确保只有管理人员能够访问这个视图（view）。我们获取 *Order* 对象通过给予的 ID 并且我们使用 `rander_to_string()`函数提供自 Django 来渲染 *orders/order/pdf.html*。这个渲染过的 HTML 会被保存到 `html` 变量中。之后，我们生成一个新的 `HttpResponse` 对象指定 `application/pdf` 的内容类型并且包含 `Content-Disposition` 头来指定这个文件名。我们使用 WeasyPrint 来生成一个 PDF 文件从渲染的 HTML 代码中并且将该文件写入 `HttpResponse` 对象中。我们加载它从本地路径通过使用 `STATIC_ROOT` 设置。最后，我们返回这个生成的响应。

由于我们需要使用 `STATIC_ROOT` 设置，我们需要添加它到我们的项目中。这个项目将会是静态文件的所在地。编辑 *myshop* 项目的 *settings.py* 文件，添加如下设置：

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static/')
```

之后，运行命令 `python manage.py collectstatic`。你会在输出末尾看到如下输出：

```
You have requested to collect static files at the destination
location as specified in your settings:

    code/myshop/static
This will overwrite existing files!
Are you sure you want to do this?
```

输入 *yes* 然后回车。你会得到一条消息，告知那个静态文件已经复制到 `STATIC_ROOT` 目录中。*collectstatic* 命令复制所有静态文件从你的应用到定义在 `STATIC_ROOT` 设置的目录中。这允许每个应用去提供它自己的静态文件通过使用一个 *static/*目录来包含它们。你还可以提供额外的静态文件来源在 `STATICFILES_DIRS` 设置。所有的目录被指定在 `STATICFILED_DIRS` 列中的都将会被复制到 `STATIC_ROOT` 目录中当 *collectstatic* 被执行的时候。

编辑 *orders* 应用目录下的 *urls.py* 文件并且添加如下 URL 模式：

```
url(r'^admin/order/(?P<order_id>\d+)/pdf/$',
    views.admin_order_pdf,
    name='admin_order_pdf'),
```

现在，我们可以编辑管理列展示页面给 *Order* 模型（model）来添加一个链接给 PDF 文件给每一个结果。编辑 *orders* 应用的 *admin.py* 文件并且添加以下代码在 `OrderAdmin` 类上面：

```
def order_pdf(obj):
    return '<a href="{}">PDF</a>'.format(
```

```
        reverse('orders:admin_order_pdf', args=[obj.id]))
order_pdf.allow_tags = True
order_pdf.short_description = 'PDF bill'
```

添加 `order_pdf` 给 `OrderAdmin` 类的 `list_display` 属性：

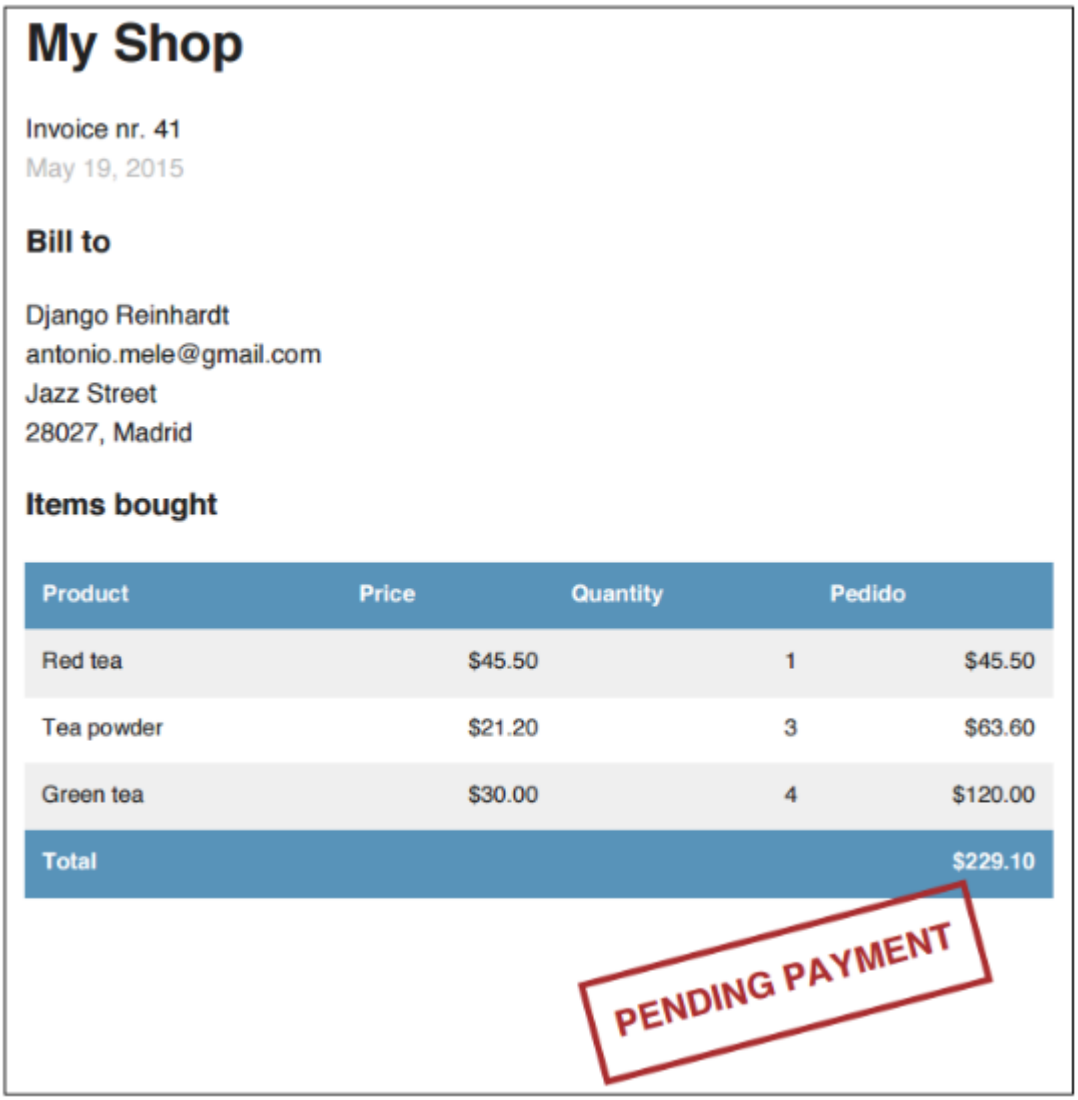
```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id',
                    # ...
                    order_detail,
                    order_pdf]
```

如果你指定一个 `short_description` 属性给你的调用，Django 将会使用它给这个列命名。
打开 http://127.0.0.1:8000/admin/orders/order/ 在你的浏览器中。每一行现在都包含一个 PDF 链接，如下所示：

City	Paid	Created	Updated	Order detail	PDF bill
Madrid	⛔	May 19, 2015, 10:46 p.m.	May 19, 2015, 10:46 p.m.	View	PDF

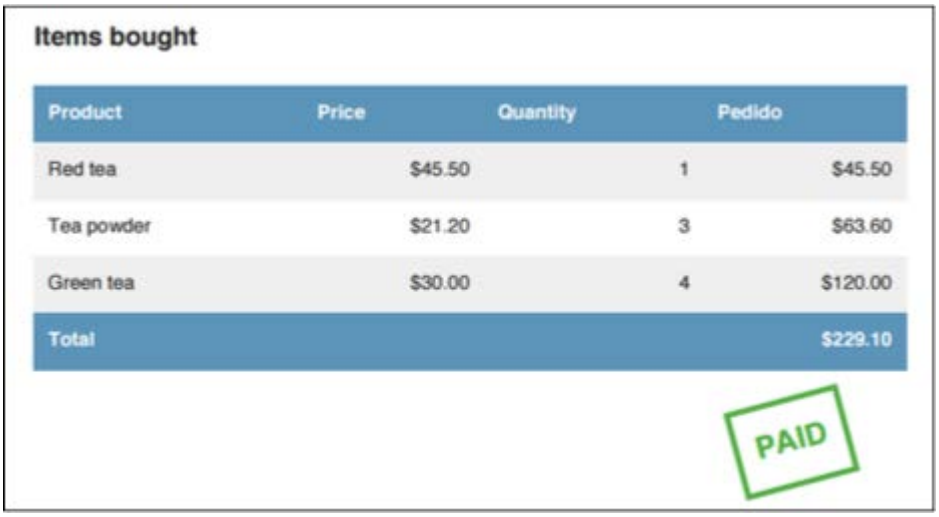
django-8-13

点击某一个订单的 **PDF**。你会看到一个生成的 PDF 文件，如下所示一个订单还没有支付完成：



django-8-14

对于支付完成的订单，你会看到如下所示的 PDF 文件：



django-8-15

通过 e-mail 发送 PDF 文件

让我们发送一封 e-mail 给我们的顾客包含生成的 PDF 发表但一个支付被接收的时候。编辑 *payment* 应用下的 *signals.py* 文件并且添加如下导入：

```
from django.template.loader import render_to_string
from django.core.mail import EmailMessage
from django.conf import settings
import weasyprint
from io import BytesIO
```

之后添加如下代码在 `order.save()`行之后，需要同样的缩进等级：

```
# create invoice e-mail
subject = 'My Shop - Invoice no. {}'.format(order.id)
message = 'Please, find attached the invoice for your recent
purchase.'
email = EmailMessage(subject,
                     message,
                     'admin@myshop.com',
                     [order.email])
# generate PDF
html = render_to_string('orders/order/pdf.html', {'order': order})
out = BytesIO()
stylesheets=[weasyprint.CSS(settings.STATIC_ROOT + 'css/pdf.css')]
weasyprint.HTML(string=html).write_pdf(out,
                                       stylesheets=stylesheets)
# attach PDF file
email.attach('order_{}.pdf'.format(order.id),
             out.getvalue(),
             'application/pdf')
# send e-mail
email.send()
```

在这个信号中，我们使用 Django 提供的 `EmailMessage` 类来创建一个 e-mail 对象。之后我们渲染这个模板（template）到 `html` 变量中。我们生成 PDF 文件从渲染的模板（template）中，并且我们输出它到

一个 `BytesIO` 实例中，该实例是一个内容字节缓存。之后我们附加这个生成的 PDF 文件到 `EmailMessage` 对象通过使用它的 `attach()`方法，包含这个 `out` 缓存的内容。
记住设置你的 SMTP 设置在项目的 *settings.py* 文件中来发送 e-mail。你可以到**第二章 通过高级特性扩展你的 blog** 去看下一个 SMTP 配置的例子。
现在你可以打开 Ngrok 提供给你的应用的 URL 然后完成一个新的支付处理为了收到 PDF 发票到你的 e-mail 中。

总结

在这一章中，你集成了一个支付网关到你的项目中。你定制了 Django 管理平台页面并且学习到了如何动态的生成 CSV 以及 PDF 文件。
在下一章中将会给你一个深刻理解关于国际化和本地化给 Django 项目。你还会学习到创建一个赠券系统已经构建一个产品推荐引擎。

译者总结

不知不觉，第八章也翻译完成了，还是渣翻，精校之前大家先凑合着看吧，有问题我会及时更新。目前全书翻译已完成三分之二，离不开各位的支持，我们下章再见！对了，本章完成日是三八妇女（女神？）节，各位女看客们节日快乐！

第九章 拓展你的商店

在上一章中，你学习了如何把支付网关整合进你的商店。你处理了支付通知，学会了如何生成 CSV 和 PDF 文件。在这一章中，你会把优惠券添加进你的商店中。你将学到国际化（internationalization）和本地化（localization）是如何工作的，你还会创建一个推荐引擎。
在这一章中将会包含一下知识点：

- 创建一个优惠券系统来应用折扣
- 把国际化添加进你的项目中
- 使用 Rosetta 来管理翻译
- 使用 django-parler 来翻译模型（model）
- 建立一个产品推荐引擎

创建一个优惠券系统

很多的在线商店会送出很多优惠券，这些优惠券可以在顾客的采购中兑换为相应的折扣。在线优惠券通常是由一串给顾客的代码构成，这串代码在一个特定的时间段内是有效的。这串代码可以被兑换一次或者多次。
我们将会为我们的商店创建一个优惠券系统。优惠券将会在顾客在某一个特定的时间段内输入时生效。优惠券没有任何兑换数的限制，他们也可用于购物车的总金额中。对于这个功能，我们将会创建一个模型（model）来储存优惠券代码，优惠券有效的时间段，以及折扣的力度。
在 `myshop` 项目内使用如下命令创建一个新的应用：

```
python manage.py startapp coupons
```

编辑 `myshop` 的 `settings.py` 文件，像下面这样把把应用添加到 `INSTALLED_APPS` 中：

```
INSTALLED_APPS = (
    # ...
    'coupons',
)
```

新的应用已经在我们的 Django 项目中激活了。

创建优惠券模型（model）

让我们开始创建 Coupon 模型（model）。编辑 coupons 应用中的 models.py 文件，添加以下代码：

```
from django.db import models
from django.core.validators import MinValueValidator,\
                                   MaxValueValidator
class Coupon(models.Model):
    code = models.CharField(max_length=50,
                            unique=True)
    valid_from = models.DateTimeField()
    valid_to = models.DateTimeField()
    discount = models.IntegerField(
               validators=[MinValueValidator(0),
                           MaxValueValidator(100)])
    active = models.BooleanField()

    def __str__(self):
        return self.code
```

我们将会用这个模型（model）来储存优惠券。 Coupon 模型（model）包含以下几个字段：

- code：用户必须要输入的代码来将优惠券应用到他们购买的商品中
- valid_from：表示优惠券会在何时生效的时间和日期值
- valid_to：表示优惠券会在何时过期
- discount：折扣率（这是一个百分比，所以它的值的范围是 0 到 1000）。我们使用验证器来限制接收的最小值和最大值
- active：表示优惠券是否激活的布尔值

执行下面的命令来为 coupons 生成首次迁移：

```
python manage.py makemigrations
```

输出应该包含以下这几行：

```
Migrations for 'coupons':
    0001_initial.py:
        - Create model Coupon
```

之后我们执行下面的命令来应用迁移：

```
python manage.py migrate
```

你可以看见包含下面这一行的输出：

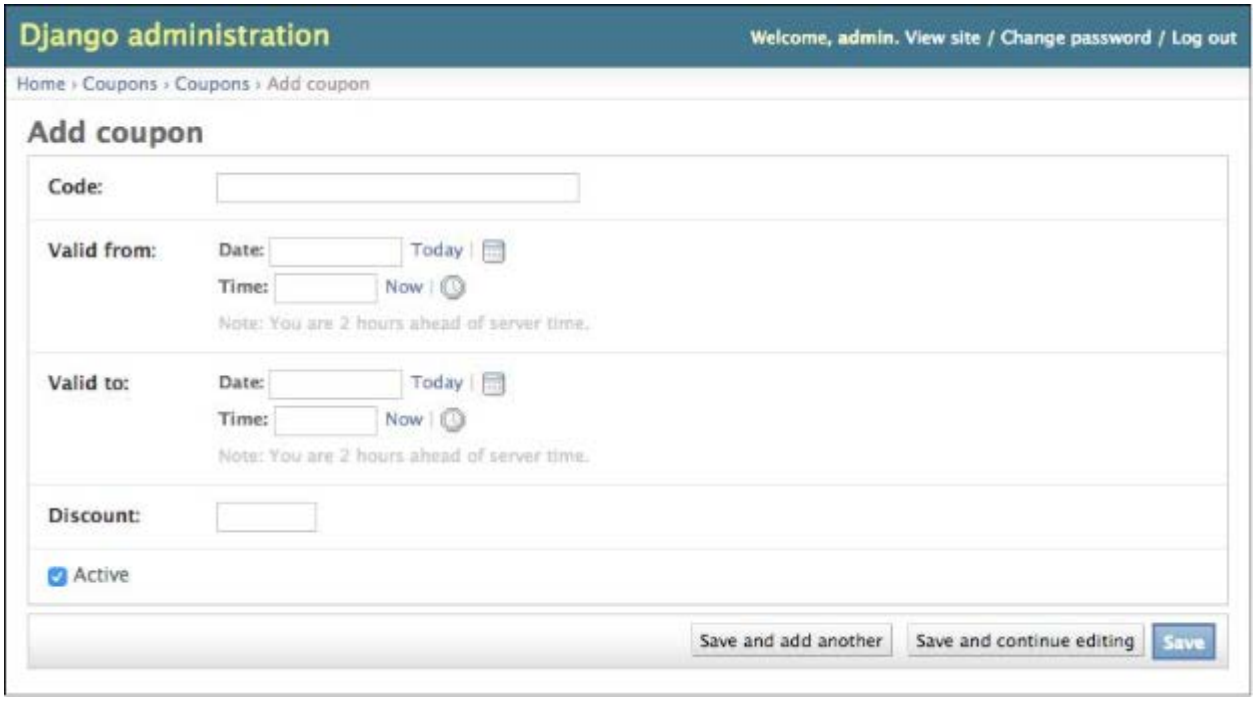
```
Applying coupons.0001_initial... OK
```

迁移现在已经被应用到了数据库中。让我们把 Coupon 模型（model）添加到管理站点。编辑 coupons 应用的 admin.py 文件，添加以下代码：

```
from django.contrib import admin
from .models improt Coupon

class CouponAdmin(admin.ModelAdmin):
    list_display = ['code', 'valid_from', 'valid_to',
                    'discount', 'active']
    list_filter = ['active', 'valid_from', 'valid_to']
```

```
    search_fields = ['code']
admin.site.register(Coupon, CouponAdmin)
```

`Coupon` 模型（model）现在已经注册进了管理站点中。确保你已经用命令 `python manage.py runserver` 打开了开发服务器。访问 http://127.0.0.1:8000/admin/coupons/add。你可以看见下面的表单：



django-9-1

填写表单创建一个在当前日期有效的新优惠券，确保你点击了 **Active** 复选框，然后点击 **Save** 按钮。

把应用优惠券到购物车中

我们可以保存新的优惠券以及检索目前的优惠券。现在我们需要一个方法来让顾客可以应用他们的优惠券到他们购买的产品中。花点时间来想想这个功能该如何实现。应用一张优惠券的流程如下：

```
1. 用户将产品添加进购物车
2. 用户在购物车详情页的表单中输入优惠代码
3. 当用户输入优惠代码然后提交表单时，我们查找一张和所给优惠代码相符的有效优惠券。我们必须检查用户输入的优惠券代码，
`active` 属性为 `True` ，当前时间位于 `valid_from 和 `valid_to` 之间。
4. 如果查找到了相应的优惠券，我们把它保存在用户会话中，然后展示包含折扣了的购物车以及更新总价。
5. 当用户下单时，我们把优惠券保存到所给的订单中。
```

在 `coupons` 应用路径下创建一个新的文件，命名为 `forms.py` 文件，添加以下代码：

```
from django import forms

class CouponApplyForm(forms.Form):
    code = forms.CharField()
```

我们将会用这个表格来让用户输入优惠券代码。编辑 `coupons` 应用中的 `views.py` 文件，添加以下代码：

```
from django.shortcuts import render, redirect
from django.utils import timezone
from django.views.decorators.http import require_POST
from .models import Coupon
from .forms import CouponApplyForm
```

```
@require_POST
def coupon_apply(request):
    now = timezone.now()
    form = CouponApplyForm(request.POST)
    if form.is_valid():
        code = form.cleaned_data['code']
        try:
            coupon = Coupon.objects.get(code__iexact=code,
                                        valid_from__lte=now,
                                        valid_to__gte=now,
                                        active=True)
            request.session['coupon_id'] = coupon.id
        except Coupon.DoesNotExist:
            request.session['coupon_id'] = None
    return redirect('cart:cart_detail')
```

`coupon_apply` 视图（view）验证优惠券然后把它保存在用户会话（session）中。我们使用 `require_POST` 装饰器来限制这个视图（view）仅接受 POST 请求。在视图（view）中，我们执行了以下几个任务：

1.我们用上传的数据实例化了 `CouponApplyForm` 然后检查表单是否合法。
2. 如果表单是合法的，我们就从表单的 `cleaned_data` 字典中获取 `code` 。我们尝试用所给的代码检索 `Coupon` 对象。我们使用 `iexact` 字段来对照查找大小写不敏感的精确匹配项。优惠券在当前必须是激活的（`active=True`）以及必须在当前日期内是有效的。我们使用 Django 的 `timezone.now()` 函数来获得当前的时区识别时间和日期（time-zone-aware） 然后我们把它和 `valid_from` 和 `valid_to` 字段做比较，对这两个日期分别执行 `lte` （小于等于）运算和 `gte` （大于等于）运算来进行字段查找。
3. 我们在用户的会话中保存优惠券的 `id``。
4. 我们把用户重定向到 `cart_detail` URL 来展示应用了优惠券的购物车。

我们需要一个 `coupon_apply` 视图（view）的 URL 模式。在 `coupon` 应用路径下创建一个新的文件，命名为 `urls.py` ，添加以下代码：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^apply/$', views.coupon_apply, name='apply'),
]
```

然后，编辑 `myshop` 项目的主 `urls.py` 文件，引入 `coupons` 的 URL 模式：

```
url(r'^coupons/', include('coupons.urls', namespace='coupons')),
```

记得把这个放在 `shop.urls` 模式之前。
现在编辑 `cart` 应用的 `cart.py`，包含以下导入：

```
from coupons.models import Coupon
```

把下面这行代码添加进 `Cart` 类的 `__init__()` 方法中来从会话中初始化优惠券：

```
# store current applied coupon
self.coupon_id = self.session.get('coupon_id')
```

在这行代码中，我们尝试从当前会话中得到 `coupon_id` 会话键，然后把它保存在 `Cart` 对象中。把以下方法添加进 `Cart` 对象中：

```
@property
def coupon(self):
    if self.coupon_id:
        return Coupon.objects.get(id=self.coupon_id)
    return None

def get_discount(self):
    if self.coupon:
        return (self.coupon.discount / Decimal('100')) \
               * self.get_total_price()
    return Decimal('0')

def get_total_price_after_discount(self):
    return self.get_total_price() - self.get_discount()
```

下面是这几个方法：

- `coupon()`：我们定义这个方法作为 `property` 。如果购物车包含 `coupon_id` 函数，就会返回一个带有给定 `id` 的 `Coupon` 对象
- `get_discount()`：如果购物车包含 `coupon` ，我们就检索它的折扣比率，然后返回购物车中被扣除折扣的总和。
- `get_total_price_after_discount()`：返回被减去折扣之后的总价。

`Cart` 类现在已经准备好处理应用于当前会话的优惠券了，然后将它应用于相应折扣中。
让我们在购物车详情视图（view）中引入优惠券系统。编辑 `cart` 应用的 `views.py` ，然后把下面这一行添加到顶部：

```
from coupons.forms import CouponApplyForm
```

之后，编辑 `cart_detail` 视图（view），然后添加新的表单：

```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
                    initial={'quantity': item['quantity'],
                    'update': True})
    coupon_apply_form = CouponApplyForm()
    return render(request,
            'cart/detail.html',
            {'cart': cart,
            'coupon_apply_form': coupon_apply_form})
```

编辑 `cart` 应用的 `acrt/detail.html` 文件，找到下面这几行：

```
<tr class="total">
    <td>Total</td>
    <td colspan="4"></td>
    <td class="num">${{ cart.get_total_price }}</td>
</tr>
```

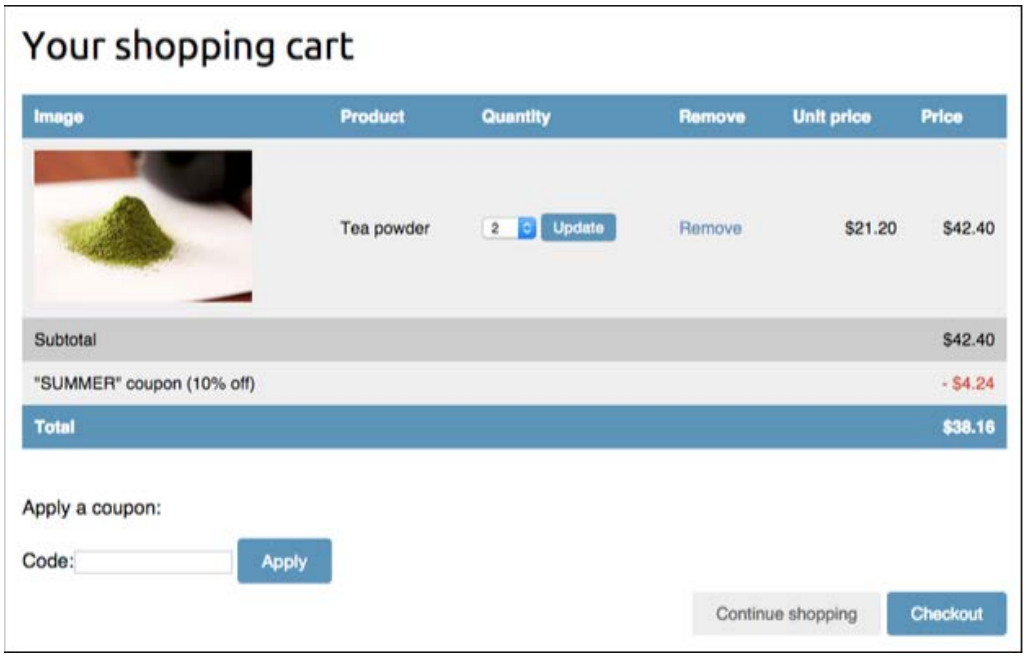
把它们换成以下几行：

```
{% if cart.coupon %}
<tr class="subtotal">
    <td>Subtotal</td>
    <td colspan="4"></td>
    <td class="num">${{ cart.get_total_price }}</td>
</tr>
<tr>
<td>
    "{{ cart.coupon.code }}" coupon
    ({{ cart.coupon.discount }}% off)
    </td>
    <td colspan="4"></td>
    <td class="num neg">
    - ${{ cart.get_discount|floatformat:"2" }}
    </td>
</tr>
{% endif %}
<tr class="total">
    <td>Total</td>
    <td colspan="4"></td>
    <td class="num">
    ${{ cart.get_total_price_after_discount|floatformat:"2" }}
    </td>
</tr>
```

这段代码用于展示可选优惠券以及折扣率。如果购物车中有优惠券，我们就在第一行展示购物车的总价作为 **小计**。然后我们在第二行展示当前可应用于购物车的优惠券。最后，我们调用 `cart` 对象的 `get_total_price_after_discount()` 方法来展示折扣了的总价格。
在同一个文件中，在 `</table>` 标签之后引入以下代码：

```
<p>Apply a coupon:</p>
<form action="{% url "coupons:apply" %}" method="post">
    {{ coupon_apply_form }}
    <input type="submit" value="Apply">
    {% csrf_token %}
</form>
```

我们将会展示一个表单来让用户输入优惠券代码，然后将它应用于当前的购物车当中。
访问 http://127.0.0.1:8000 ，向购物车当中添加一个商品，然后在表单中输入你创建的优惠代码来应用你的优惠券。你可以看到购物车像下面这样展示优惠券折扣：



django-9-2

让我们把优惠券添加到购物流程中的下一步。编辑 `orders` 应用的 `orders/order/create.html` 模板（template），找到下面这几行：

```
<ul>
{% for item in cart %}
<li>
    {{ item.quantity }}x {{ item.product.name }}
    <span>${{ item.total_price }}</span>
</li>
{% endfor %}
</ul>
```

把它替换为下面这几行：

```
<ul>
{% for item in cart %}
    <li>
    {{ item.quantity }}x {{ item.product.name }}
    <span>${{ item.total_price }}</span>
    </li>
{% endfor %}
{% if cart.coupon %}
<li>
"{{ cart.coupon.code }}" ({{ cart.coupon.discount }}% off)
<span>- ${{ cart.get_discount|floatformat:"2" }}</span>
</li>
{% endif %}
</ul>
```

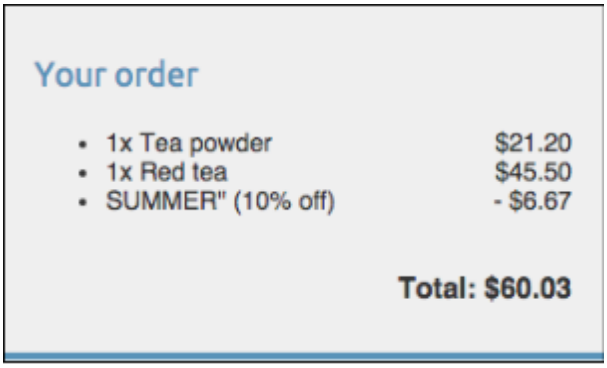
订单汇总现在已经包含使用了的优惠券，如果有优惠券的话。现在找到下面这一行：

```
<p>Total: ${{ cart.get_total_price }}</p>
```

把他们换成以下一行：

```
<p>Total: ${{ cart.get_total_price_after_discount|floatformat:"2" }}</p>
```

这样做之后，总价也将会在减去优惠券折扣被计算出来。
访问 http://127.0.0.1:8000/orders/create/ 。你会看到包含使用了优惠券的订单小计：



django-9-3

用户现在可以在购物车当中使用优惠券了。尽管，当用户结账时，我们依然需要在订单中储存优惠券信息。

在订单中使用优惠券

我们会储存每张订单中使用的优惠券。首先，我们需要修改 `Order` 模型（model）来储存相关联的 `Coupon` 对象，如果有这个对象的话。
编辑 `orders` 应用的 `models.py` 文件，添加以下代码：

```
from decimal import Decimal
from django.core.validators import MinValueValidator, \
                                   MaxValueValidator
from coupons.models import Coupon
```

然后，把下列字段添加进 `Order` 模型（model）中：

```
coupon = models.ForeignKey(Coupon,
                           related_name='orders',
                           null=True,
                           blank=True)
discount = models.IntegerField(default=0,
                       validators=[MinValueValidator(0),
                           MaxValueValidator(100)])
```

这些字段让用户可以在订单中储存可选的优惠券信息，以及优惠券的相应折扣。折扣被存在关联的 `Coupon` 对象中，但是我们在 `Order` 模型（model）中引入它以便我们在优惠券被更改或者删除时好保存它。
因为 `Order` 模型（model）已经被改变了，我们需要创建迁移。执行下面的命令：

```
python manage.py makemigrations
```

你可以看到如下输出：

```
Migrations for 'orders':
    0002_auto_20150606_1735.py:
        - Add field coupon to order
        - Add field discount to order
```

用下面的命令来执行迁移：

```
python manage.py migrate orders
```

你必须要确保新的迁移已经被应用了。 `Order` 模型（model）的字段变更现在已经同步到了数据库中。
回到 `models.py` 文件中，按照如下更改 `Order` 模型（model）的 `get_total_cost()` 方法：

```
def get_total_cost(self):
    total_cost = sum(item.get_cost() for item in self.items.all())
    return total_cost - total_cost * \
        (self.discount / Decimal('100'))
```

`Order` 模型（model）的 `get_total_cost()` 现在已经把使用了的折扣包含在内了，如果有折扣的话。
编辑 `orders` 应用的 `views.py` 文件，更改 `order_create` 视图（view）以便在创建新的订单时保存相关联的优惠券和折扣。找到下面这一行：

```
order = form.save()
```

把它替换为下面这几行：

```
order = form.save(commit=False)
if cart.coupon:
    order.coupon = cart.coupon
    order.discount = cart.coupon.discount
order.save()
```

在新的代码中，我们使用 `OrderCrateForm` 表单的 `save()` 方法创建了一个 `Order` 对象。我们使用 `commit=False` 来避免将它保存在数据库中。如果购物车当中有优惠券，我们就会保存相关联的优惠券和折扣。然后才把 `order` 对象保存到数据库中。
确保用 `python manage.py runserver` 运行了开发服务器。使用 `./ngrok http:8000` 命令来启动 Ngrok 。
在你的浏览器中打开 Ngrok 提供的 URL，然后用用你创建的优惠券完成一次购物。当你完成一次成功的支付后，有可以访问 http://127.0.0.1:8000/admin/orders/order/ ，检查 `order` 对象是否包含了优惠券和折扣，如下：

django-9-4

你也可以更改管理界面的订单详情模板（template）和订单的 PDF 账单，以便用展示购物车的方式来展示使用了的优惠券。
下面。我们将会为我们的项目添加国际化（internationalization）。

添加国际化（internationalization）和本地化（localization）

Django 提供了完整的国际化和本地化的支持。这使得你可以把你的项目翻译为多种语言以及处理地区特性的 日期，时间，数字，和时区 的格式。让我们一起来搞清楚国际化和本地化的区别。国际化（通常简写为：**i18n**）是为让软件适应潜在的不同语言和多种语言的使用做的处理，这样它就不是以某种

特定语言或某几种语言为硬编码的软件了。本地化（简写为：**l10n**）是对软件的翻译以及使软件适应某种特定语言的处理。Django 自身已经使用自带的国际化框架被翻译为了 50 多种语言。

使用 Django 国际化

国际化框架让你可以容易的在 Python 代码和模板（template）中标记需要翻译的字符串。它依赖于 GNU 文本获取集来生成和管理消息文件。**消息文件**是一个表示一种语言的纯文本文件。它包含某一语言的一部分或者所有的翻译字符串。消息文件有 `.po` 扩展名。
一旦翻译完成，信息文件就会被编译一遍快速的连接到被翻译的字符串。编译后的翻译文件的扩展名为 `.mo` 。

国际化和本地化设置

Django 提供了几种国际化的设置。下面是几种最有关联的设置：

- `USE_I18N`：布尔值。用于设定 Django 翻译系统是否启动。默认值为 `True`。
  -`USE_L10N`：布尔值。表示本地格式化是否启动。当被激活式，本地格式化被用于展示日期和数字。默认为 `False` 。
- `USE_TZ`：布尔值。用于指定日期和时间是否是时区别（timezone-aware）。
- `LANGUAGE_CODE`：项目的默认语言。在标准的语言 ID 格式中，比如，`en-us` 是美式英语，`en-gb` 是英式英语。这个设置要 `USE_I18N` 为 `True` 才能生效。你可以在这个网站找到一个合法的语言 ID 表：`http://www.i18nguy.com/unicode/language-identifiers.html`。
- `LANGUAGES`：包含项目可用语言的元组。它们由包含 **语言代码** 和 **语言名字** 的双元组构成的。你可以在 `django.conf.global_settions` 里看到可用语言的列表。当你选择你的网站将会使用哪一种语言时，你就把 `LANGUAGES` 设置为那个列表的子列表。
- `LOCAL_PATHS`：Django 寻找包含翻译的信息文件的路径列表。
- `TIME_ZONE`：代表项目时区的字符串。当你使用 `startproject` 命令创建新项目时它被设置为 `UTC` 。你也可以把它设置为其他的时区，比如 `Europe/Madrid` 。

这是一些可用的国际化和本地化的设置。你可以在这个网站找到全部的（设置）列表：
https://docs.djangoproject.com/en/1.8/ref/settings/#globalization-i18n-l10n 。

国际化管理命令

Django 使用 `manage.py` 或者 `django-admin.py` 工具包管理翻译，包含以下命令：

- `makemessages`：运行于源代码树中，寻找所有被用于翻译的字符串，然后在 `locale` 路径中创建或更新 `.po` 信息文件。每一种语言创建一个单一的 `.po` 文件。
- `compilemessages`： 把扩展名为 `.po` 的信息文件编译为用于检索翻译的 `.mo` 文件。

你需要文本获取工具集来创建，更新，以及编译信息文件。大部分的 Linux 发行版都包含文本获取工具集。如果你在使用 Mac OS X，用 Honebrew (`http://brew.sh`)是最简单的安装它的方法，使用以下命令 ： `brew install gettext`。你或许也需要将它和命令行强制连接 `brew link gettext --force` 。对于 Windows ， 安装步骤如下 ：
https://docs.djangoproject.com/en/1.8/topics/i18n/translation/#gettext-on-windows 。

怎么把翻译添加到 Django 项目中

让我们看看国际化项目的过程。我们需要像下面这样做：

```
1. 我们在 Python 代码和模板（template）中中标记需要翻译的字符串。
2. 我们运行 `makemessages` 命令来创建或者更新包含所有翻译字符串的信息文件。
3. 我们翻译包含在信息文件中的字符串，然后使用 `compilemessages` 编译他们。
```

Django 如何决定当前语言

Django 配备有一个基于请求数据的中间件，这个中间件用于决定当前语言是什么。这个中间件是位于 `django.middleware.locale.localMiddleware` 的 `LocaleMiddleware` ,它执行下面的任务：

1. 如果使用 `i18_patterns` — 这是你使用的被翻译的 URL 模式，它在被请求的 URL 中寻找一个语言前缀来决定当前语言。
2. 如果没有找到语言前缀，就会在当前用户会话中寻找当前的 `LANGUAGE_SESSION_KEY` 。
3. 如果在会话中没有设置语言，就会寻找带有当前语言的 cookie 。这个 cookie 的定制名可在 `LANGUAGE_COOKIE_NAME` 中设置。默认地，这个 cookie 的名字是 `django_language` 。
4. 如果没有找到 cookie，就会在请求 HTTP 头中寻找 `Accept-Language` 。
5. 如果 `Accept-Language` 头中没有指定语言，Django 就使用在 `LANGUAGE_CODE` 设置中定义的语言。

默认的，Django 会使用 LANGUAGE_OCDE 中设置的语言，除非你正在使用 LocaleMiddleware。上述操作进会在使用这个中间件时生效。

为我们的项目准备国际化

让我们的项目准备好使用不同的语言吧。我们将要为我们的商店创建英文版和西班牙语版。编辑项目中的 settings.py 文件，添加下列 LANGUAGES 设置。把它放在 LANGUAGE_OCDE 旁边：

```
LANGUAGES = (
       ('en', 'English'),
       ('es', 'Spanish'),
)
```

LANGUAGES 设置包含两个由语言代码和语言名的元组构成，比如 en-us 或 en-gb ，或者一般的设置为 en 。这样设置之后，我们指定我们的应用只会对英语和西班牙语可用。如果我们不指定 LANGUAGES 设置，网站将会对所有 Django 的翻译语言有效。
确保你的 LANGUAGE_OCDE 像如下设置：

```
LANGUAGE_OCDE = 'en'
```

把 django.middleware.locale.LocaleMiddleware 添加到 MIDDLEWARE_CLASSES 设置中。确保这个设置在 SessionsMiddleware 之后，因为 LocaleMiddleware 需要使用会话数据。它同样必须放在 CommonMiddleware 之前，因为后者需要一种激活了的语言来解析请求 URL 。MIDDLEWARE_CLASSES 设置看起来应该如下：

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.locale.LocaleMiddleware',
    'django.middleware.common.CommonMiddleware',
    # ...
)
```

中间件的顺序非常重要，因为每个中间件所依赖的数据可能是由其他中间件处理之后的才获得的。中间件按照 MIDDLEWARE_CLASSES 的顺序应用在每个请求上，以及反序应用于响应中。
在主项目路径下，在 manage.py 文件同级，创建以下文件结构：

```
locale/
    en/
    es/
```

locale 路径是放置应用信息文件的地方。再次编辑 settings.py ，然后添加以下设置：

```
LOCALE_PATHS = (
    os.path.join(BASE_DIR, 'locale/'),
)
```

LOCALE_PATHS 设置指定了 Django 寻找翻译文件的路径。第一个路径有最高优先权。

当你在你的项目路径下使用 `makemessages` 命令时，信息文件将会在 `locale` 路径下生成。尽管，对于包含 `locale` 路径的应用，信息文件就会保存在这个应用的 `locale` 路径中。

翻译 Python 代码

我们翻译在 Python 代码中的字母，你可以使用 `django.utils.translation` 中的 `gettext()`函数来标记需要翻译的字符串。这个函数翻译信息然后返回一个字符串。约定俗成的用法是导入这个函数后把它命名为 `_`（下划线）。
你可以在这个网站找到所有关于翻译的文档 ：
https://docs.djangoproject.com/en/1.8/topics/i18n/translation/ 。

标准翻译

下面的代码展示了如何标记一个翻译字符串：

```
from django.utils.translation import gettext as _
output = _('Text to be translated.')
```

惰性翻译（Lazy translation）

Django 对所有的翻译函数引入了惰性（lazy）版，这些惰性函数都带有后缀 `_lazy()` 。当使用惰性函数时，字符串在值被连接时就会被翻译，而不是在被调用时翻译（这也是它为什么被惰性翻译的原因）。惰性函数迟早会派上用场，特别是当标记字符串在模型（model）加载的执行路径中时。
使用 `gettext_lazy()` 而不是 `gettext()`，字符串就会在连接到值的时候被翻译而不会在函数调用的时候被翻译。Django 为所有的翻译都提供了惰性版本。

翻译引入的变量

标记的翻译字符串可以包含占位符来引入翻译中的变量。下面就是一个带有占位符的翻译字字符串的例子：

```
from django.utils.translation import gettext as _
month = _('April')
day = '14'
output = _('Today is %(month)s %(day)s') % {'month': month,
                                            'day': day}
```

通过使用占位符，你可以重新排序文字变量。举个例子，以前的英文版本是 'Today is April 14' ，但是西班牙的版本是这样的 'Hoy es 14 de Abril' 。当你的翻译字符串有多于一个参数时，我们总是使用字符串插值来代替位置插值

翻译中的复数形式

对于复数形式，你可以使用 `gettext()` 和 `gettext_lazy()` 。这些函数基于一个可以表示对象数量的参数来翻译单数和复数形式。下面这个例子展示了如何使用它们：

```
output = ngettext('there is %(count)d product',
                  'there are %(count)d products',
                  count) % {'count': count}
```

现在你已经基本知道了如何在你的 Python 代码中翻译字符了，现在是把翻译应用到项目中的时候了。

翻译你的代码

编辑项目中的 `settings.py` ， 导入 `gettext_lazy()` 函数，然后像下面这样修改 `LANGUAGES` 的设置：

```
from django.utils.translation import gettext_lazy as _
```

```
LANGUAGES = (
    ('en', _('English')),
    ('es', _('Spanish')),
)
```

我们使用 `gettext_lazy()` 而不是 `gettext()` 来避免循环导入，这样就可以在语言名被连接时就翻译它们。
在你的项目路径下执行下面的命令：

```
django-admin makemessages --all
```

你可以看到如下输出：

```
processing locale es
processing locale en
```

看下 `locale/` 路径。你可以看到如下文件结构：

```
en/
    LC_MESSAGES/
        django.po
es/
    LC_MESSAGES/
        django.po
```

`.po` 文件已经为每一个语言创建好了。用文本编辑器打开 `es/LC_MESSAGES/django.po` 。在文件的末尾，你可以看到如下几行：

```
#: settings.py:104
msgid "English"
msgstr ""

#: settings.py:105
msgid "Spanish"
msgstr ""
```

每一个翻译字符串前都有一个显示该文件详情的注释以及它在哪一行被找到。每个翻译都包含两个字符串：

- `msgid`：在源代码中的翻译字符串
- `msgstr`：对应语言的翻译，默认为空。这是你输入所给字符串翻译的地方。

按照如下，根据所给的 `msgid` 在 `msgtsr` 中填写翻译：

```
#: settings.py:104
msgid "English"
msgstr "Inglés"

#: settings.py:105
msgid "Spanish"
msgstr "Español"
```

保存修改后的信息文件，打开 shell ，运行下面的命令：

```
django-admin compilemessages
```

如果一切顺利，你可以看到像如下的输出：

```
processing file django.po in myshop/locale/en/LC_MESSAGES
processing file django.po in myshop/locale/es/LC_MESSAGES
```

输出给出了被编译的信息文件的相关信息。再看一眼 `locale` 路径的 `myshop` 。你可以看到下面的文件：

```
en/
    LC_MESSAGES/
        django.mo
        django.po
es/
    LC_MESSAGES/
        django.mo
        django.po
```

你可以看到每个语言的 `.mo` 的编译文件已经生成了。
我们已经翻译了语言本身的名字。现在让我们翻译展示在网站中模型（model）字段的名字。编辑 `orders` 应用的 `models.py` ，为 `Order` 模型（model）添加翻译的被标记名：

```
from django.utils.translation import gettext_lazy as _

class Order(models.Model):
    first_name = models.CharField(_('first name'),
                                  max_length=50)
    last_name = models.CharField(_('last name'),
                                  max_length=50)
    email = models.EmailField(_('e-mail'),)
    address = models.CharField(_('address'),
                                  max_length=250)
    postal_code = models.CharField(_('postal code'),
                                  max_length=20)
    city = models.CharField(_('city'),
                        max_length=100)
#...
```

我们添加为当用户下一个新订单时展示的字段添加了名字，它们分别是 `first_name`,`last_name`, `email`, `address`, `postal_code` ,`city`。记住，你也可以使用每个字段的 `verbose_name` 属性。
在 `orders` 应用路径内创建以下路径：

```
locale/
    en/
    es/
```

通过创建 `locale` 路径，这个应用的翻译字符串就会储存在这个路径下的信息文件里，而不是在主信息文件里。这样做之后，你就可以为每个应用生成独自的翻译文件。
在项目路径下打开 shell ，运行下面的命令：

```
django-admin makemessages --all
```

你可以看到如下输出：

```
processing locale es
processing locale en
```

用文本编辑器打开 `es/LC_MESSAGES/django.po` 文件。你将会看到每一个模型（model）的翻译字符串。为每一个所给的 `msgid` 字符串填写 `msgstr` 的翻译：

```
#: orders/models.py:10
msgid "first name"
msgstr "nombre"

#: orders/models.py:12
msgid "last name"
msgstr "apellidos"

#: orders/models.py:14
msgid "e-mail"
msgstr "e-mail"

#: orders/models.py:15
msgid "address"
msgstr "dirección"

#: orders/models.py:17
msgid "postal code"
msgstr "código postal"

#: orders/models.py:19
msgid "city"
msgstr "ciudad"
```

在你添加完翻译之后，保存文件。

在文本编辑器内，你可以使用 Poedit 来编辑翻译。 Poedit 是一个用来编辑翻译的软件，它使用 gettext 。它有 Linux ，Windows，Mac OS X 版，你可以在这个网站下载 Poedit : http://poedit.net/ 。

让我们来翻译项目中的表单吧。 `orders` 应用的 `OrderCrateForm` 还没有被翻译，因为它是一个 `ModelForm` ，使用 `Order` 模型（model）的 `verbose_name` 属性作为每个字段的标签。我们将会翻译 `cart` 和 `coupons` 应用的表单。

编辑 `cart` 应用路径下的 `forms.py` 文件，给 `CartAddProductForm` 的 `quantity` 字段添加一个 `label` 属性，然后按照如下标记这个需要翻译的字段：

```
from django import forms
from django.utils.translation import gettext_lazy as _

PRODUCT_QUANTITY_CHOICES = [(i, str(i)) for i in range(1, 21)]

class CartAddProductForm(forms.Form):
    quantity = forms.TypedChoiceField(
                        choices=PRODUCT_QUANTITY_CHOICES,
```

```
                    coerce=int,
                    label=_('Quantity'))
    update = forms.BooleanField(required=False,
                    initial=False,
                    widget=forms.HiddenInput)
```

编辑 `coupons` 应用的 `forms.py` ，按照如下翻译 `CouponApplyForm` ：

```
from django import forms
from django.utils.translation import gettext_lazy as _

class CouponApplyForm(forms.Form):
    code = forms.CharField(label=_('Coupon'))
```

翻译模板（templates）

Django 提供了 `{% trans %}` 和 `{% blocktrans %}` 模板（template）标签来翻译模板（template）中的字符串。为了使用翻译模板（template）标签，你需要在你的模板（template）顶部添加 `{% load i18n %}` 来载入她们。

{% trans %}模板（template）标签

`{% trans %}`模板（template）标签让你可以标记需要翻译的字符串，常量，或者是参数。在内部，Django对所给文本执行 `gettext()` 。这是如何标记模板（template）中的翻译字符串：

```
{% trans "Text to be translated" %}
```

你可使用 `as` 来储存你在模板（template）内使用的全局变量里的被翻译内容。下面这个例子保存了一个变量中叫做 `greeting` 的被翻译文本：

```
{% trans "Hello!" as greeting %}
<h1>{{ greeting }}</h1>
```

`{% trans %}` 标签对于简单的翻译字符串是很有用的，但是它不能包含变量的翻译内容。

{% blocktrans %}模板（template）标签

`{% blocktrans %}` 模板（template）标签让你可以标记含有占位符的变量和字符的内容。线面这个例子展示了如何使用 `{% blocktrans %}` 标签来标记包含一个 `name` 变量的翻译内容：

```
{% blocktrans %}Hello {{ name }}!{% endblocktrans %}
```

你可以用 `with` 来引入模板（template）描述，比如连接对象的属性或者应用模板（template）的变量过滤器。你必须为他们用占位符。你不能在 `blocktrans` 内连接任何描述或者对象属性。下面的例子展示了如何使用 `with` 来引入一个应用了 `capfirst` 过滤器的对象属性：

```
{% blocktrans with name=user.name|capfirst %}
Hello {{ name }}!
{% endblocktrans %}
```

当你的字符串中含有变量时使用 `{% blocktrans %}` 代替 `{% trans %}` 。

翻译商店模板（template）

编辑 `shop` 应用的 `shop/base.html` 。确保你已经在顶部载入了 `i18n` 标签，然后按照如下标记翻译字符串：

```
{% load i18n %}
{% load static %}
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>
{% block title %}{% trans "My shop" %}{% endblock %}
</title>
<link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
<div id="header">
<a href="/" class="logo">{% trans "My shop" %}</a>
</div>
<div id="subheader">
<div class="cart">
{% with total_items=cart|length %}
{% if cart|length > 0 %}
{% trans "Your cart" %}:
<a href="{% url "cart:cart_detail" %}">
{% blocktrans with total_items_plural=total_
items|pluralize
total_price=cart.get_total_price %}
{{ total_items }} item{{ total_items_plural }},
${{ total_price }}
{% endblocktrans %}
</a>
{% else %}
{% trans "Your cart is empty." %}
{% endif %}
{% endwith %}
</div>
</div>
<div id="content">
{% block content %}
{% endblock %}
</div>
</body>
</html>
```

注意展示在购物车小计的 `{% blocktrans %}` 标签。购物车小计在之前是这样的：

```
{{ total_items }} item{{ total_items|pluralize }},
${{ cart.get_total_price }}
```

我们使用 `{% blocktrans with ... %}` 来使用 `total_ items|pluralize` （模板（template）标签生效的地方）和 `cart_total_price` （连接对象方法的地方）的占位符：

```
{% blocktrans with total_items_plural=total_items|pluralize
total_price=cart.get_total_price %}
```

```
{{ total_items }} item{{ total_items_plural }},
${{ total_price }}
{% endblocktrans %}
```

下面，编辑 `shop` 应用的 `shop/product/detail.html` 模板（template）然后在顶部载入 `i18n` 标签，但是它必须位于 `{% extends %}` 标签的下面：

```
{% load i18n %}
```

找到下面这一行：

```
<input type="submit" value="Add to cart">
```

把它替换为下面这一行：

```
<input type="submit" value="{% trans "Add to cart" %}">
```

现在翻译 `orders` 应用模板（template）。编辑 `orders` 应用的 `orders/order/create.html` 模板（template），然后标记翻译文本：

```
{% extends "shop/base.html" %}
{% load i18n %}
{% block title %}
{% trans "Checkout" %}
{% endblock %}
{% block content %}
<h1>{% trans "Checkout" %}</h1>
<div class="order-info">
<h3>{% trans "Your order" %}</h3>
<ul>
{% for item in cart %}
<li>
{{ item.quantity }}x {{ item.product.name }}
<span>${{ item.total_price }}</span>
</li>
{% endfor %}
{% if cart.coupon %}
<li>
{% blocktrans with code=cart.coupon.code
discount=cart.coupon.discount %}
"{{ code }}" ({{ discount }}% off)
{% endblocktrans %}
<span>- ${{ cart.get_discount|floatformat:"2" }}</span>
</li>
{% endif %}
</ul>
<p>{% trans "Total" %}: ${{
cart.get_total_price_after_discount|floatformat:"2" }}</p>
</div>
<form action="." method="post" class="order-form">
{{ form.as_p }}
<p><input type="submit" value="{% trans "Place order" %}"></p>
```

```
{% csrf_token %}
</form>
{% endblock %}
```

看看本章中下列文件中的代码，看看字符串是如何被标记的：

- `shop` 应用：`shop/product/list.html`
- `orders` 应用：`orders/order/created.html`
- `cart` 应用：`cart/detail.html`

让位我们更新信息文件来引入新的翻译字符串。打开 shell ，运行下面的命令：

```
django-admin makemessages --all
```

`.po` 文件已经在 `myshop` 项目的 `locale` 路径下，你将看到 `orders` 应用现在已经包含我们标记的所有需要翻译的字符串。

编辑项目和 `orderes` 应用中的 `.po` 翻译文件，然后引入西班牙语翻译。你可以参考本章中翻译了的 `.po` 文件：

在项目路径下打开 shell ，然后运行下面的命令：

```
cd orders/
django-admin compilemessages
cd ../
```

我们已经编译了 `orders` 应用的翻译。

运行下面的命令，这样应用中不包含 `locale` 路径的翻译就被包含进了项目的信息文件中：

```
django-admin compilemessages
```

使用 Rosetta 翻译交互界面

Rosetta 是一个让你可以编辑翻译的第三方应用，它有类似 Django 管理站点的交互界面。Rosetta 让编辑 `.po` 文件和更新它变得很简单，让我们把它添加进项目中：

```
pip install django-rosetta==0.7.6
```

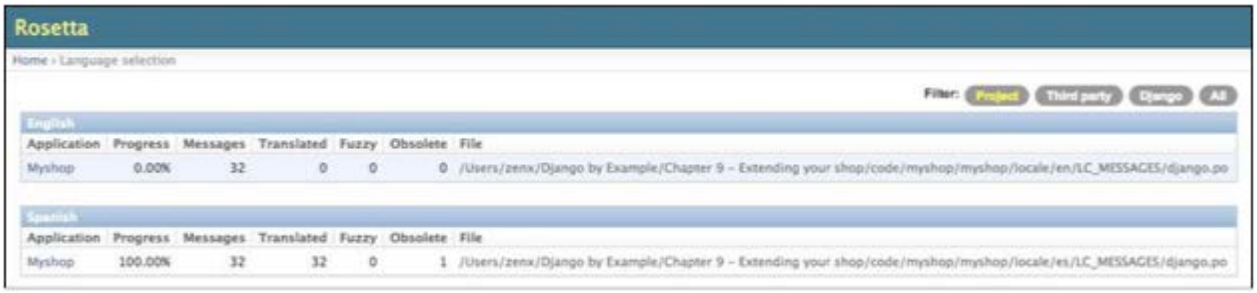
然后，把 `rosetts` 添加进项目中的 `setting.py` 文件中的 `INSTALLED_APPS` 设置中。

你需要把 Rosetta 的 URL 添加进主 URL 配置中。编辑项目中的主 `urls.py` 文件，把下列 URL 模式添加进去：

```
url(r'^rosetta/', include('rosetta.urls')),
```

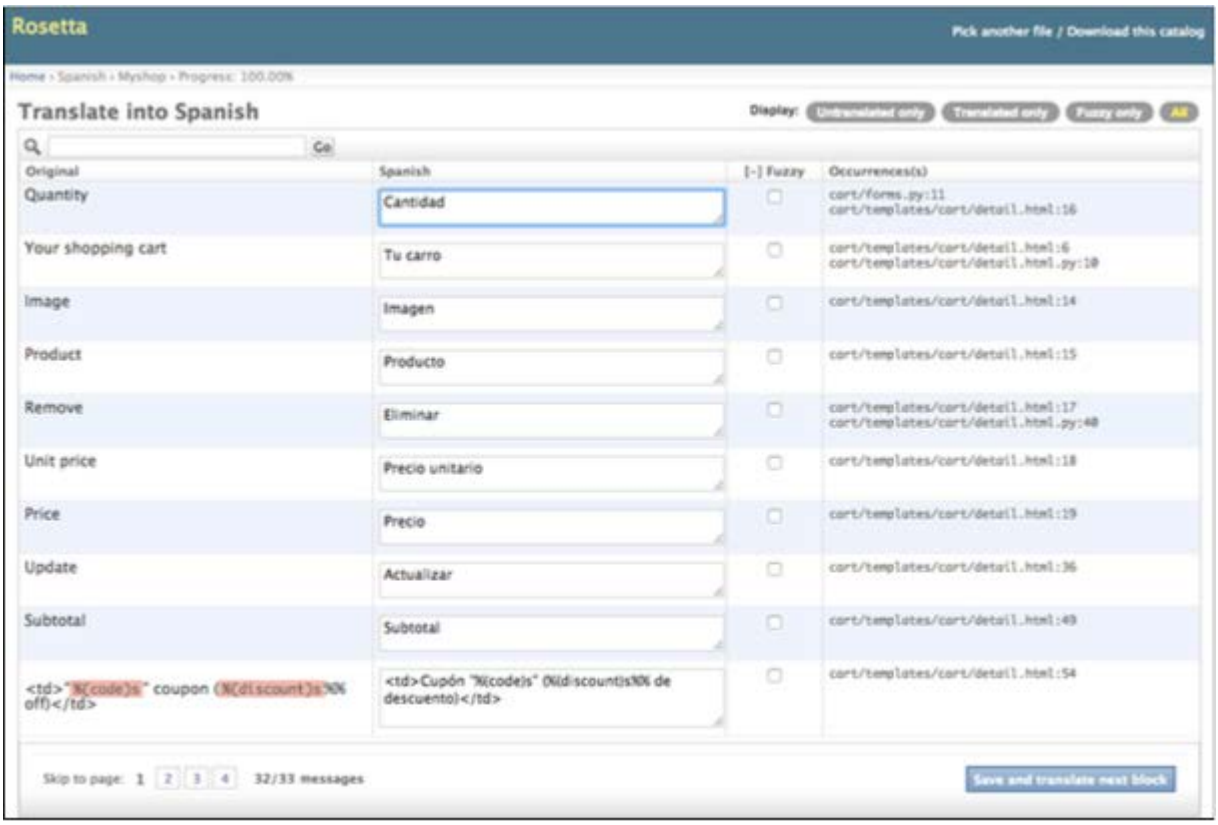
确保你把它放在了 `shop.urls` 模式之前以避免错误的匹配。

访问 http://127.0.0.1:8000/admin/ ,然后使用超级管理员账户登录。然后导航到 http://127.0.0.1:8000/rosetta/ 。你可以看到当前语言列表：



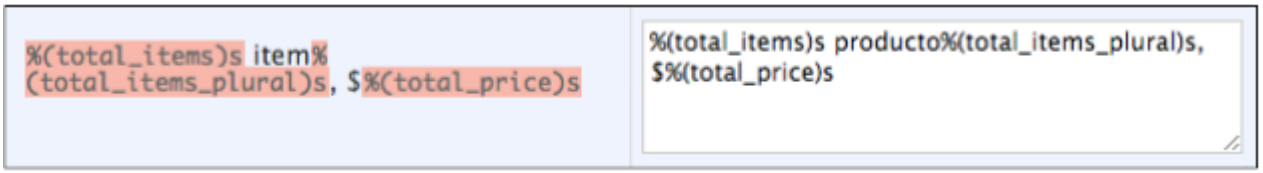
django-9-5

在 **Filter** 模块，点击 **All** 来展示所有可用的信息文件，包括属于 `orders` 应用的信息文件。点击 **Spanish** 模块下的 **Myshop** 链接来编辑西班牙语翻译。你可以看到翻译字符串的列表：



django-9-6

你可以在 **Spanish** 行下输入翻译。**Occurences** 行展示了代码中翻译字符串被找到的文件和行数、包含占位符的翻译看起来像这样：



django-9-7

Rosetta 使用了不同的背景色来展示占位符。当你翻译内容时，确保你没有翻译占位符。比如，用下面这个字符串举例：

```
%(total_items)s item%(total_items_plural)s, $%(total_price)s
```

它翻译为西班牙语之后长得像这样：

```
%(total_items)s producto%(total_items_plural)s, $%(total_price)s
```

你可以看看本章项目中使用相同西班牙语翻译的文件源代码。

当你完成编辑翻译后，点击 **Save and translate next block** 按钮来保存翻译到 `.po` 文件中。Rosetta 在你保存翻译时编辑信息文件，所以并不需要你运行 `compilemessages` 命令。尽管，Rosetta 需要 `locale` 路径的写入权限来写入信息文件。确保路径有有效权限。

如果你想让其他用户编辑翻译，访问：http://127.0.0.1:8000/admin/auth/group/add/,然后创建一个名为 `translations` 的新组。当编辑一个用户时，在 **Permissions** 模块下，把 `translations` 组添加进 **ChosenGroups** 中。Rosetta 只对超级用户和 `translations` 中的用户是可用的。

你可以在这个网站阅读 Rosetta 的文档：http://django-rosetta.readthedocs.org/en/latest/。

当你在生产环境中添加新的翻译时，如果你的 Django 运行在一个真实服务器上，你必须在运行 `compilemessages` 或保存翻译之后重启你的服务器来让 Rosetta 的更改生效。

惰性翻译

你或许已经注意到了 Rosetta 有 **Fuzzy** 这一行。这不是 Rosetta 的特性，它是由 gettext 提供的。如果翻译的 flag 是激活的，那么它就不会被包含进编译后的信息文件中。flag 用于需要翻译器修订的翻译字符串。当 `.po` 文件在更新新的翻译字符串时，一些翻译字符串可能被自动标记为 fuzzy. 当 gettext 找到一些变动不大的 `msgid` 时就会发生这样的情况，gettext 就会把它认为的旧的翻译和匹配在一起然后会在回把它标记为 fuzzy 以用于回查。翻译器之后会回查模糊翻译，会移除 fuzzy 标签然后再次编译信息文件。

国际化的 URL 模式

Django 提供了用于国际化的 URLs。它包含两种主要用于国际化的 URLs：

- **URL 模式中的语言前缀**：把语言的前缀添加到 URLs 当中，以便在不同的基本 URL 下提供每一种语言的版本。
- **翻译后的 URL 模式**：标记要翻译的 URL 模式，这样同样的 URL 就可以服务于不同的语言。

翻译 URLs 的原因是这样就可以优化你的站点，方便搜索引擎搜索。通过添加语言前缀，你就可以为每一种语言提供索引，而不是所有语言用一种索引。并且，把 URLs 为不同语言，你就可以提供给搜索引擎更好的搜索序列。

把语言前缀添加到 URLs 模式中

Django 允许你可以把语言前缀添加到 URLs 模式中。举个例子，网站的英文版可以服务于 `/en/` 起始路径之下，西班牙语版服务于 `/es/` 起始路径之下。
为了在你的 URLs 模式中使用不同语言，你必须确保 `settings.py` 中的 `MIDDLEWARE_CLASSES` 设置中有 `django.middleware.localMiddlewar` 。Django 将会使用它来辨别当前请求中的语言。
让我们把语言前缀添加到 URLs 模式中。编辑 `myshop` 项目的 `urls.py` ，添加以下库：

```
from django.conf.urls.i18n import i18n_patterns
```

然后添加 `i18n_patterns()`：

```
urlpatterns = i18n_patterns(
    url(r'^admin/', include(admin.site.urls)),
    url(r'^cart/', include('cart.urls', namespace='cart')),
    url(r'^orders/', include('orders.urls', namespace='orders')),
    url(r'^payment/', include('payment.urls',
                      namespace='payment')),
    url(r'^paypal/', include('paypal.standard.ipn.urls')),
    url(r'^coupons/', include('coupons.urls',
                  namespace='coupons')),
    url(r'^rosetta/', include('rosetta.urls')),
    url(r'^', include('shop.urls', namespace='shop')),
    )
```

你可以把 `i18n_patterns()` 和 `patterns()` URLs 模式结合起来，这样一些模式就会包含语言前缀另一些就不会包含。尽管，最好还是使用翻译后的 URLs 来避免 URL 匹配一个未翻译的 URL 模式的可能。
打开开发服务器，访问 http://127.0.0.1:8000/ ，因为你在 Django 中使用 `LocaleMiddleware` 来执行 `How Django determines the current language` 中描述的步骤来决定当前语言，然后它就会把你重定向到包含相同语言前缀的 URL。看看浏览器中 URL ，它看起来像这样：http://127.0.0.1:8000/en/.当前语言将会被设置在浏览器的 `Accept-Language` 头中，设为英语或者西班牙语或者是 `LANGUAGE_OCDE`（English） 中的默认设置。

翻译 URL 模式

Django 支持 URL 模式中有翻译了的字符串。你可以为每一种语言使用不同的 URL 模式。你可以使用 `ugettet_lazy()` 函数标记 URL 模式中需要翻译的字符串。

编辑 `myshop` 项目中的 `urls.py` 文件，把翻译字符串添加进 `cart`,`orders`,`payment`,`coupons` 的 URLs 模式的正则表达式中：

```
from django.utils.translation import gettext_lazy as _

urlpatterns = i18n_patterns(
    url(r'^admin/', include(admin.site.urls)),
    url(_(r'^cart/'), include('cart.urls', namespace='cart')),
    url(_(r'^orders/'), include('orders.urls',
                  namespace='orders')),
    url(_(r'^payment/'), include('payment.urls',
                  namespace='payment')),
    url(r'^paypal/', include('paypal.standard.ipn.urls')),
    url(_(r'^coupons/'), include('coupons.urls',
                  namespace='coupons')),
    url(r'^rosetta/', include('rosetta.urls')),
    url(r'^', include('shop.urls', namespace='shop')),
)
```

编辑 `orders` 应用的 `urls.py` 文件，标记需要翻译的 URLs 模式：

```
from django.conf.urls import url
from .import views
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    url(_(r'^create/$'), views.order_create, name='order_create'),
    # ...
]
```

编辑 `payment` 应用的 `urls.py` 文件，把代码改成这样：

```
from django.conf.urls import url
from . import views
from django.utils.translation import gettext_lazy as _

urlpatterns = [
    url(_(r'^process/$'), views.payment_process, name='process'),
    url(_(r'^done/$'), views.payment_done, name='done'),
    url(_(r'^canceled/$'),
        views.payment_canceled,
        name='canceled'),
]
```

我们不需要翻译 `shop` 应用的 URL 模式，因为它们是用变量创建的，而且也没有包含其他的任何字符。打开 shell ，运行下面的命令来把新的翻译更新到信息文件：

```
django-admin makemessages --all
```

确保开发服务器正在运行中，访问：http://127.0.0.1:8000/en/rosetta/ ,点击 **Spanish** 下的 **Myshop** 链接。你可以使用显示过滤器（display filter）来查看没有被翻译的字符串。确保你的 URL 翻译有正则表达式中的特殊字符。翻译 URLs 是一个精细活儿；如果你替换了正则表达式，你可能会破坏 URL。

允许用户切换语言

因为我们正在提供多语种服务，我们应当让用户可以选择站点的语言。我们将会为我们的站点添加语言选择器。语言选择器由可用的语言列表组成，我们使用链接来展示这些语言选项：
编辑 `shop/base.html` 模板（template），找到下面这一行：

```
<div id="header">
<a href="/" class="logo">{% trans "My shop" %}</a>
</div>
```

把他们换成以下几行：：

```
<div id="header">
<a href="/" class="logo">{% trans "My shop" %}</a>
{% get_current_language as LANGUAGE_CODE %}
{% get_available_languages as LANGUAGES %}
{% get_language_info_list for LANGUAGES as languages %}
<div class="languages">
<p>{% trans "Language" %}:</p>
<ul class="languages">
{% for language in languages %}
<li>
<a href="/{{ language.code }}/" {% if language.code ==
LANGUAGE_CODE %} class="selected"{% endif %}>
{{ language.name_local }}
</a>
</li>
{% endfor %}
</ul>
</div>
</div>
```

我们是这样创建我们的语言选择器的：

```
1. 首先使用 `{% load i18n %}` 加载国际化
2. 使用 `{% get_current_language %}` 标签来检索当前语言
3. 使用 `{% get_available_languages %}` 模板（template）标签来过去 `LANGUAGES` 中定义语言
4. 使用 `{% get_language_info_list %}` 来提供简易的语言属性连接入口
5. 我们创建了一个 HTML 列表来展示所有的可用语言列表然后我们为当前激活语言添加了 `selected` 属性
```

我们使用基于项目设置中语言变量的 i18n 提供的模板（template）标签。现在访问：http://127.0.0.1:8000/` ，你应该能在站点顶部的右边看到语言选择器：

django-9-8

用户现在可以轻易的切换他们的语言了。

使用 django-parler 翻译模型（models）

Django 没有提供开箱即用的模型（models）翻译的解决办法。你必须要自己实现你自己的解决办法来管理不同语言中的内容或者使用第三方模块来管理模型（model）翻译。有几种第三方应用允许你翻

译模型（model）字段。每一种手采用了不同的方法保存和连接翻译。其中一种应用是 django-parler 。这个模块提供了非常有效的办法来翻译模型（models）并且它和 Django 的管理站点整合的非常好。django-parler 为每一个模型（model）生成包含翻译的分离数据库表。这个表包含了所有的翻译的字段和源对象的外键翻译。它也包含了一个语言字段，一位内每一行都会保存一个语言的内容。

安装 django-parler

使用 pip 安装 django-parler :

```
pip install django-parler==1.5.1
```

编辑项目中 `settings.py` ，把 `parler` 添加到 `INSTALLED_APPS` 中，同样也把下面的配置添加到配置文件中：

```
PARLER_LANGUAGES = {
    None: (
        {'code': 'en',},
        {'code': 'es',},
    ),
    'default': {
        'fallback': 'en',
        'hide_untranslated': False,
    }
}
```

这个设置为 django-parler 定义了可用语言 `en` 和 `es` 。我们指定默认语言为 `en` ，然后我们指定 django-parler 应该隐藏未翻译的内容。

翻译模型（model）字段

让我们为我们的产品目录添加翻译。 django-parler 提供了 `TranslatedModel` 模型（model）类和 `TranslatedFields` 闭包（wrapper）来翻译模型（model）字段。编辑 `shop` 应用路径下的 `models.py` 文件，添加以下导入：

```
from parler.models import TranslatableModel, TranslatedFields
```

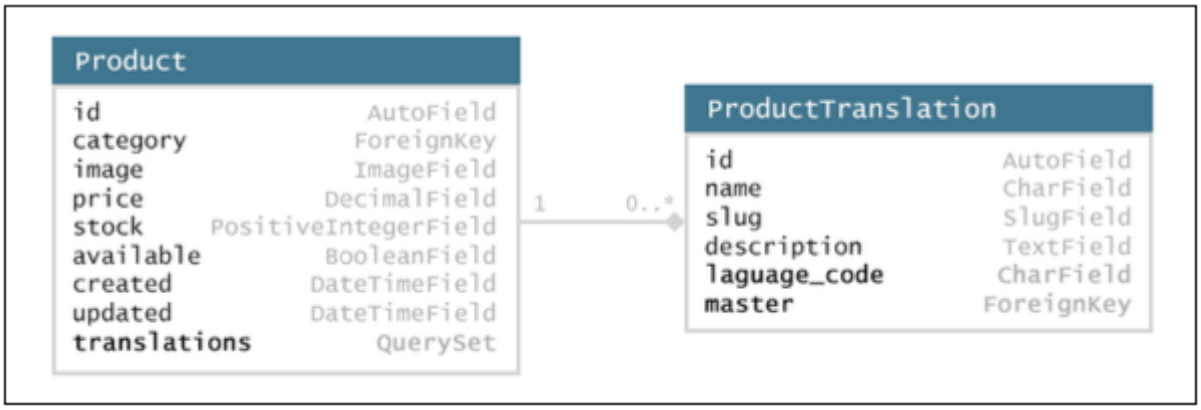
然后更改 `Category` 模型（model）来使 `name` 和 `slug` 字段可以被翻译。
我们依然保留了每行未翻译的字段：

```
class Category(TranslatableModel):
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200,
                            db_index=True,
                            unique=True)
    translations = TranslatedFields(
        name = models.CharField(max_length=200,
                                db_index=True),
        slug = models.SlugField(max_length=200,
                                db_index=True,
                                unique=True)
)
```

`Category` 模型（model）继承了 `TranslatableModel` 而不是 `models.Model`。并且 `name` 和 `slug` 字段都被引入到了 `TranslatedFields` 闭包（wrapper）中。
编辑 `Product` 模型（model），为 `name`,`slug` ,`description` 添加翻译，同样我们也保留了每行未翻译的字段：

```
class Product(TranslatableModel):
    name = models.CharField(max_length=200, db_index=True)
    slug = models.SlugField(max_length=200, db_index=True)
    description = models.TextField(blank=True)
    translations = TranslatedFields(
        name = models.CharField(max_length=200, db_index=True),
        slug = models.SlugField(max_length=200, db_index=True),
        description = models.TextField(blank=True)
    )
    category = models.ForeignKey(Category,
                      related_name='products')
    image = models.ImageField(upload_to='products/%Y/%m/%d',
                      blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    stock = models.PositiveIntegerField()
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
```

django-parler 为每个可翻译模型（model）生成了一个模型（model）。在下面的图片中，你可以看到 `Product` 字段和生成的 `ProductTranslation` 模型（model）：



django-9-9

django-parler 生成的 `ProductTranslation` 模型（model）有 `name`,`slug`,`description` 可翻译字段，一个 `language_code` 字段，和主要的 `Product` 对象的 `ForeignKey` 字段。`Product` 和 `ProductTranslation` 之间有一个一对多关系。一个 `ProductTranslation` 对象会为每个可用语言生成 `Product` 对象。

因为 Django 为翻译都使用了相互独立的表，所以有一些 Django 的特性我们是用不了。使用翻译后的字段来默认排序是不可能的。你可以在查询集（QuerySets）中使用翻译字段来过滤，但是你不可以在 `ordering Meta` 选项中引入可翻译字段。编辑 `shop` 应用的 `models.py`，然后注释掉 `Category Meta` 类的 `ordering` 属性：

```
class Meta:
    # ordering = ('name',)
    verbose_name = 'category'
    verbose_name_plural = 'categories'
```

我们也必须注释掉 `Product Meta` 类的 `index_together` 属性，因为当前的 django-parler 的版本不支持对它的验证。编辑 `Product Meta`：

```
class Meta:
    ordering = ('-created',)
```

```
    # index_together = (('id', 'slug'),)
```

你可以在这个网站阅读更多 django-parler 兼容的有关内容：
http://django-parler.readthedocs.org/en/latest/compatibility.html 。

创建一次定制的迁移

当你创建新的翻译模型（models）时，你需要执行 `makemessages` 来生成模型（models）的迁移，然后把变更同步到数据库中。尽管当使已存在字段可翻译化时，你或许有想要在数据库中保留的数据。我们将迁移当前数据到新的翻译模型（models）中。因此，我们添加了翻译字段但是有意保存了原始字段。
翻译添加到当前字段的步骤如下：

```
1. 在保留原始字段的情况下，创建新翻译模型（model）的迁移
2. 创建定制化迁移，从当前已有的字段中拷贝一份数据到翻译模型（models）中
3. 从源模型（models）中删除字段
```

运行下面的命令来为我们添加到 `Category` 和 `Product` 模型（model）中的翻译字段创建迁移：

```
python manage.py makemigrations shop --name "add_translation_model"
```

你可以看到如下输出：

```
Migrations for 'shop':
    0002_add_translation_model.py:
        - Create model CategoryTranslation
        - Create model ProductTranslation
        - Change Meta options on category
        - Alter index_together for product (0 constraint(s))
        - Add field master to producttranslation
        - Add field master to categorytranslation
        - Alter unique_together for producttranslation (1 constraint(s))
        - Alter unique_together for categorytranslation (1 constraint(s))
```

迁移已有数据

现在我们需要创建定制迁移来把已有数据拷贝到新的翻译模型（model）中。使用这个命令创建一个空的迁移：

```
python manage.py makemigrations --empty shop --name "migrate_translatable_fields"
```

你将会看到如下输出：

```
Migrations for 'shop':
    0003_migrate_translatable_fields.py
```

编辑 `shop/migrations/0003_migrate_translatable_fields.py` ，然后添加下面的代码：

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from django.db import models, migrations
from django.apps import apps
from django.conf import settings
from django.core.exceptions import ObjectDoesNotExist
```

```
translatable_models = {
    'Category': ['name', 'slug'],
    'Product': ['name', 'slug', 'description'],
}

def forwards_func(apps, schema_editor):
    for model, fields in translatable_models.items():
        Model = apps.get_model('shop', model)
        ModelTranslation = apps.get_model('shop',
                              '{}Translation'.format(model))

    for obj in Model.objects.all():
        translation_fields = {field: getattr(obj, field) for field in fields}
        translation = ModelTranslation.objects.create(
                       master_id=obj.pk,
                       language_code=settings.LANGUAGE_CODE,
                       **translation_fields)

def backwards_func(apps, schema_editor):
    for model, fields in translatable_models.items():
        Model = apps.get_model('shop', model)
        ModelTranslation = apps.get_model('shop',
                              '{}Translation'.format(model))
    for obj in Model.objects.all():
        translation = _get_translation(obj, ModelTranslation)
        for field in fields:
            setattr(obj, field, getattr(translation, field))
        obj.save()

def _get_translation(obj, MyModelTranslation):
    translations = MyModelTranslation.objects.filter(master_id=obj.pk)
    try:
        # Try default translation
        return translations.get(language_code=settings.LANGUAGE_CODE)
    except ObjectDoesNotExist:
        # Hope there is a single translation
        return translations.get()

class Migration(migrations.Migration):
    dependencies = [
    ('shop', '0002_add_translation_model'),
    ]
    operations = [
    migrations.RunPython(forwards_func, backwards_func),
    ]
```

这段迁移包括了用于执行应用和反转迁移的 `forwards_func()` 和 `backwards_func()` 。
迁移工作流程如下：

1. 我们在 `translatable_models` 字典中定义了模型（model）和它们的可翻译字段
2. 为了应用迁移，我们使用 `app.get_model()` 来迭代包含翻译的模型(model)来得到这个模型(model)和其可翻译的模型(model)
3. 我们在数据库中迭代所有的当前对象，然后为定义在项目设置中的 `LANGUAGE_CODE` 创建一个翻译对象。我们引入了 `ForeignKey` 到源对像和一份从源字段中可翻译字段的拷贝。

backwards 函数执行的是反转操作，它检索默认的翻译对象，然后把可翻译字段的值拷贝到新的模型（model）中。
最后，我们需要删除我们不再需要的源字段。编辑 `shop` 应用的 `models.py`，然后删除 `Category` 模型(model)的 `name` 和 `slug` 字段：

```
class Category(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=200, db_index=True),
        slug = models.SlugField(max_length=200,
                        db_index=True,
                        unique=True)
    )
```

删除 `Product` 模型（model）的 `name` 和 `slug` 字段：

```
class Product(TranslatableModel):
    translations = TranslatedFields(
        name = models.CharField(max_length=200, db_index=True),
        slug = models.SlugField(max_length=200, db_index=True),
        description = models.TextField(blank=True)
    )
    category = models.ForeignKey(Category,
                        related_name='products')
    image = models.ImageField(upload_to='products/%Y/%m/%d',
                        blank=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    stock = models.PositiveIntegerField()
    available = models.BooleanField(default=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
```

现在我们必须要创建最后一次迁移来应用其中的变更。如果我们尝试 `manage.py` 工具，我们将会得到一个错误，因为我们还没有让管理站点适应翻译模型（models）。让我们先来修整一下管理站点。

在管理站点中整合翻译

Django-parler 很好和 Django 的管理站点相融合。它用 `TranslatableAdmin` 重写了 Django 的 `ModelAdmin` 类 来管理翻译。
编辑 shop 应用的 `admin.py` 文件，添加下面的库：

```
from parler.admin import TranslatableAdmin
```

把 `CategoryAdmin` 和 `ProductAdmin` 的继承类改为 `TranslatableAdmin` 取代原来的 `ModelAdmin`。 Django-parler 现在还不支持 `prepopulated_fields` 属性，但是它支持 `get_prepopulated_fields()` 方法来提供相同的功能。让我们据此做一些改变。 `admin.py` 文件看起来像这样：

```
from django.contrib import admin
from .models import Category, Product
```

```
from parler.admin import TranslatableAdmin

class CategoryAdmin(TranslatableAdmin):
    list_display = ['name', 'slug']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}
admin.site.register(Category, CategoryAdmin)

class ProductAdmin(TranslatableAdmin):
    list_display = ['name', 'slug', 'category', 'price', 'stock',
                    'available', 'created', 'updated']
    list_filter = ['available', 'created', 'updated', 'category']
    list_editable = ['price', 'stock', 'available']

    def get_prepopulated_fields(self, request, obj=None):
        return {'slug': ('name',)}

admin.site.register(Product, ProductAdmin)
```

我们已经使管理站点能够和新的翻译模型（model）工作了。现在我们就能把变更同步到数据库中了。

应用翻译模型（model）迁移

我们在变更管理站点之前已经从模型(model)中删除了旧的字段。现在我们需要为这次变更创建迁移。
打开 shell ，运行下面的命令：

```
python manage.py makemigrations shop --name "remove_untranslated_fields"
```

你将会看到如下输出：

```
Migrations for 'shop':
    0004_remove_untranslated_fields.py:
        - Remove field name from category
        - Remove field slug from category
        - Remove field description from product
        - Remove field name from product
        - Remove field slug from product
```

迁移之后，我们就已经删除了源字段保留了可翻译字段。
总结一下，我们创建了以下迁移：

```
1. 将可翻译字段添加到模型（models）中
2. 将源字段中的数据迁移到可翻译字段中
3. 从源模型（models）中删除源字段
```

运行下面的命令来应用我们创建的迁移：

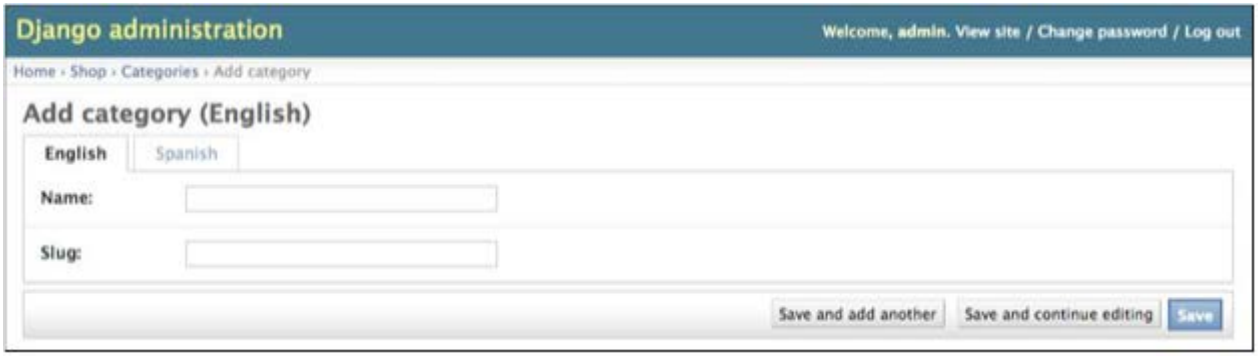
```
python manage.py migrate shop
```

你将会看到如下输出：

```
Applying shop.0002_add_translation_model... OK
```

```
Applying shop.0003_migrate_translatable_fields... OK
Applying shop.0004_remove_untranslated_fields... OK
```

我们的模型（models）现在已经同步到了数据库中。让我们来翻译一个对象。
用命令 `pythong manage.py runserver` 运行开发服务器，在浏览器中访问：
http://127.0.0.1:8000/en/admin/shop/category/add/ 。你就会看到包含两个标签的 **Add category** 页，一个标签是英文的，一个标签是西班牙语的。



django-9-10

你现在可以添加一个翻译然后点击 **Save** 按钮。确保你在切换标签之前保存了他们，不然让门就会丢失。

使视图（views）适应翻译

我们要使我们的 `shop` 视图（views）适应我们的翻译查询集（QuerySets）。在命令行运行 `python manage.py shell` ，看看你是如何检索和查询翻译字段的。为了从当前激活语言中得到一个字段的内容，你只需要用和连接一般字段相同的办法连接这个字段：

```
>>> from shop.models import Product
>>> product=Product.objects.first()
>>> product.name
'Black tea'
```

当你连接到被翻译了字段时，他们已经被用当前语言处理过了。你可以为一个对象设置不同的当前语言，这样你就可以获得指定的翻译了：

```
>>> product.set_current_language('es')
>>> product.name
'Té negro'
>>> product.get_current_language()
'es
```

当使用 `filter()` 执行一次查询集（QuerySets）时，你可以使用 `translations__` 语法筛选相关联对象:

```
>>> Product.objects.filter(translations__name='Black tea')
[<Product: Black tea>]
```

你也可以使用 `language()` 管理器来为每个检索的对象指定特定的语言：

```
>>> Product.objects.language('es').all()
[<Product: Té negro>, <Product: Té en polvo>, <Product: Té rojo>,
<Product: Té verde>]
```

如你所见，连接到翻译字段的方法还是很直接的。

让我们修改一下产品目录的视图（views）吧。编辑 `shop` 应用的 `views.py`，在 `product_list` 视图（view）中找到下面这一行：

```
category = get_object_or_404(Category, slug=category_slug)
```

把它替换为下面这几行：

```
language = request.LANGUAGE_CODE
category = get_object_or_404(Category,
                  translations__language_code=language,
                  translations__slug=category_slug)
```

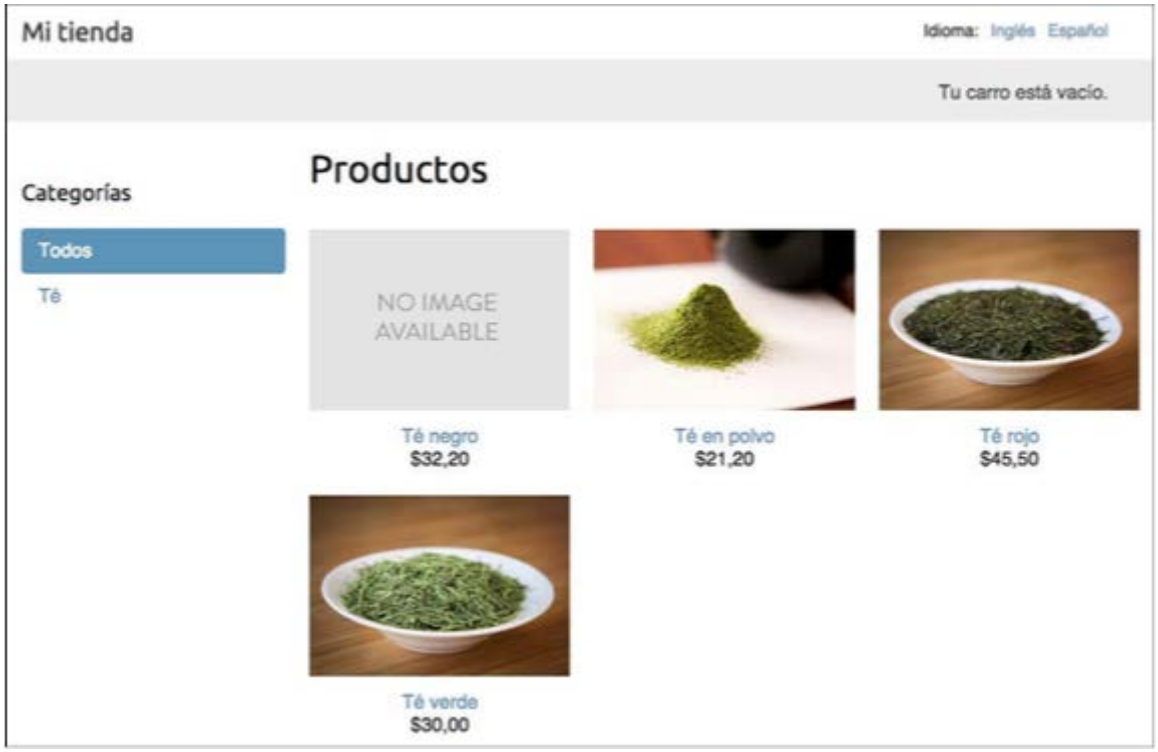
然后，编辑 `product_detail` 视图（view），找到下面这几行：

```
product = get_object_or_404(Product,
                            id=id,
                            slug=slug,
                            available=True)
```

把他们换成以下几行：：

```
language = request.LANGUAGE_CODE
get_object_or_404(Product,
                  id=id,
                  translations__language_code=language,
                  translations__slug=slug,
                  available=True)
```

现在 `product_list` 和 `product_detail` 已经可以使用翻译字段来检索了。运行开发服务器，访问：http://127.0.0.1:8000/es/ ，你可以看到产品列表页，包含了所有被翻译为西班牙语的产品：



django-9-11

现在每个产品的 URLs 已经使用 `slug` 字段被翻译为了的当前语言。比如，一个西班牙语产品的 URL 为：http://127.0.0.1:8000/es/2/te-rojo/ ,但是它的英文 URL 为：http://127.0.0.1:8000/en/2/red-tea/ 。如果你导航到产品详情页，你可以看到用被选语言翻译后的 URL 和内容，就像下面的例子：

django-9-12

如果你想要更多了解 django-parler ，你可以在这个网站找到所有的文档：http//django-parler.readthedocs.org/en/latest/ 。

你已经学习了如何翻译 Python 代码，模板（template），URL 模式，模型（model）字段。为了完成国际化和本地化的工作，我们需要展示本地化格式的时间和日期、以及数字。

本地格式化

基于用户的语言，你可能想要用不同的格式展示日期、时间和数字。本第格式化可通过修改 `settings.py` 中的 `USE_L10N` 设置为 `True` 来使本地格式化生效。

当 `USE_L10N` 可用时，Django 将会在模板（template）任何输出一个值时尝试使用某一语言的特定格式。你可以看到你的网站中用一个点来做分隔符的十进制英文数字，尽管在西班牙语版中他们使用逗号来做分隔符。这和 Django 里为 `es` 指定的语言格式。你可以在这里查看西班牙语的格式配置：https://github.com/django/django/blob/stable/1.8.x/django/conf/locale/es/formats.py。

通常，你把 `USE_L10N` 的值设为 `True` 然后 Django 就会为每一种语言应用本地化格式。虽然在某些情况下你有不想要被格式化的值。这和输出特别相关， JavaScript 或者 JSON 必须要提供机器可读的格式。

Django 提供了 `{% localize %}` 模板（template）表标签来让你可以开启或者关闭模板（template）的本地化。这使得你可以控制本地格式化。你需要载入 `l10n` 标签来使用这个模板（template）标签。下面的例子是如何在模板（template）中开启和关闭它：

```
{% load l10n %}

{% localize on %}
{{ value }}
{% endlocalize %}

{% localize off %}
{{ value }}
{% endlocalize %}
```

Django 同样也提供了 `localize` 和 `unlocalize` 模板（template）过滤器来强制或取消一个值的本地化。过滤器可以像下面这样使用：

```
{{ value|localize }}
{{ value|unlocalize }}
```

你也可以创建定制化的格式文件来指定特定语言的格式。你可以在这个网站找到所有关于本第格式化的信息：https://docs.djangoproject.com/en/1.8/topics/i18n/formatting/ 。

使用 django-localflavor 来验证表单字段

django-localflavor 是一个包含特殊工具的第三方模块，比如它的有些表单字段或者模型（model）字段对于每个国家是不同的。它对于验证本地地区，本地电话号码，身份证号，社保号等等都是非常有用的。这个包被集成进了一系列以 ISO 3166 国家代码命名的模块里。
使用下面的命令安装 django-localflavor :

```
pip install django-localflavor==1.1
```

编辑项目中的 `settings.py`，把 `localflavor` 添加进 `INSTALLED_APPS` 设置中。
我们将会添加美国（U.S.）邮政编码字段，这样之后创建一个新的订单就需要一个美国邮政编码了。
编辑 `orders` 应用的 `forms.py`，让它看起来像这样：

```
from django import forms
from .models import Order
from localflavor.us.forms import USZipCodeField

class OrderCreateForm(forms.ModelForm):
    postal_code = USZipCodeField()
    class Meta:
    model = Order
    fields = ['first_name', 'last_name', 'email', 'address',
                      'postal_code', 'city',]
```

我们从 `localflavor` 的 `us` 包中引入了 `USZipCodeField` 然后将它应用在 `OrderCreateForm` 的 `postal_code` 字段中。访问：http://127.0.0.1:8000/en/orders/create/ 然后尝试输入一个 3 位邮政编码。你将会得到 `USZipCodeField` 引发的验证错误：

```
Enter a zip code in the format XXXXX or XXXXX-XXXX.
```

这只是一个在你的项目中使用定制字段来达到验证目的的简单例子，localflavor 提供的本地组件对于让你的项目适应不同的国家是非常有用的。你可以阅读 django-localflavor 的文档，看看所有的可用地区组件：https://django-localflavor.readthedocs.org/en/latest/。
下面，我们将会为我们的店铺创建一个推荐引擎。

创建一个推荐引擎

推荐引擎是一个可以预测用户对于某一物品偏好和比率的系统。这个系统会基于用户的行为和它对于用户的了解选出相关的商品。如今，推荐系统用于许多的在线服务。它们帮助用户从大量的相关数据中挑选他们可能感兴趣的产品。提供良好的推荐可以鼓励用户更多的消费。电商网站受益于日益增长的相关产品推荐销售中。
我们将会创建一个简单但强大的推荐引擎来推荐经常一起购买的商品。我们将基于历史销售来推荐商品，这样就可以辨别出哪些商品通常是一起购买的了。我们将会在两种不同的场景下推荐互补的产品:

- **产品详情页**：我们将会展示和所给商品经常一起购买的产品列表。比如像这样展示：**Users who bought this also bought X, Y, Z**（购买了这个产品的用户也购买了 X, Y,Z）。我们需要一个数据结构来让我们能储被展示产品中和每个产品一起购买的次数。
- **购物车详情页**：基于用户添加到购物车当中的产品，我们将会推荐经常和他们一起购买的产品。在这样的情况下，我们计算的包含相关产品的评分一定要相加。

我们将会使用 Redis 来储存被一起购买的产品。记得你已经在**第六章 跟踪用户操作** 使用了 Redis 。如果你还没有安装 Redis ，你可以在那一章找到安装指导。

推荐基于历时购物的产品

现在，让我们推荐给用户一些基于他们添加到购物车的产品。我们将会在 Redis 中为每个产品储存一个键（key）。产品的键将会和它的评分一同储存在 Redis 的有序集中。在一次新的订单被完成时，我们每次都会为一同购买的产品的评分加一。
当一份订单付款成功后，我们保存每个购买产品的键，包括同意订单中的有序产品集。这个有序集让我们可以为一起购买的产品打分。
编辑项目中的额 `settings.py` ,添加以下设置：

```
REDIS_HOST = 'localhost'
REDIS_PORT = 6379
REDIS_DB = 1
```

这里的设置是为了和 Redis 服务器建立连接。在 `shop` 应用内创建一个新的文件，命名为 `recommender.py` 。添加以下代码：

```
import redis
from django.conf import settings
from .models import Product

# connect to redis
r = redis.StrictRedis(host=settings.REDIS_HOST,
                      port=settings.REDIS_PORT,
                      db=settings.REDIS_DB)

class Recommender(object):

    def get_product_key(self, id):
        return 'product:{}:purchased_with'.format(id)

    def products_bought(self, products):
        product_ids = [p.id for p in products]
        for product_id in product_ids:
            for with_id in product_ids:
                # get the other products bought with each product
                if product_id != with_id:
                    # increment score for product purchased together
                    r.zincrby(self.get_product_key(product_id),
                              with_id,
                              amount=1)
```

`Recommender` 类使我们可以储存购买的产品然后检索所给产品的产品推荐。`get_product_key()` 方法检索 `Product` 对象的 `id` ，然后为储存产品的有序集创建一个 Redis 键（key），看起来像这样：`product:[id]:purchased_with`
`products_bought()` 方法检索被一起购买的产品列表（它们都属于同一个订单）。在这个方法中，我们执行以下几个任务：

```
1. 得到所给 `Product` 对象的产品 ID
2. 迭代所有的产品 ID。对于每个 `id` ，我们迭代所有的产品 ID 并且跳过所有相同的产品，这样我们就可以得到和每个产品一起购买的产品。
3. 我们使用 `get_product_id()` 方法来获取 Redis 产品键。对于一个 ID 为 33 的产品，这个方法返回键 `product:33:purchased_with` 。这个键用于包含和这个产品被一同购买的产品 ID 的有序集。
4. 我们把有序集中的每个产品 `id` 的评分加一。评分表示另一个产品和所给产品一起购买的次数。
```

我们有一个方法来保存和对一同购买的产品评分。现在我们需要一个方法来检索被所给购物列表的一起购买的产品。把 `suggest_product_for()` 方法添加到 `Recommender` 类中：

```
def suggest_products_for(self, products, max_results=6):
    product_ids = [p.id for p in products]
    if len(products) == 1:
        # only 1 product
        suggestions = r.zrange(
                      self.get_product_key(product_ids[0]),
                      0, -1, desc=True)[:max_results]
    else:
        # generate a temporary key
        flat_ids = ''.join([str(id) for id in product_ids])
        tmp_key = 'tmp_{}'.format(flat_ids)
        # multiple products, combine scores of all products
        # store the resulting sorted set in a temporary key
        keys = [self.get_product_key(id) for id in product_ids]
        r.zunionstore(tmp_key, keys)
        # remove ids for the products the recommendation is for
        r.zrem(tmp_key, *product_ids)
        # get the product ids by their score, descendant sort
        suggestions = r.zrange(tmp_key, 0, -1,
                      desc=True)[:max_results]
        # remove the temporary key
        r.delete(tmp_key)
    suggested_products_ids = [int(id) for id in suggestions]

    # get suggested products and sort by order of appearance
    suggested_products = list(Product.objects.filter(id__in=suggested_
    products_ids))
    suggested_products.sort(key=lambda x: suggested_products_ids.
    index(x.id))
    return suggested_products
```

`suggest_product_for()` 方法接收下列参数：

- `products`：这是一个 `Product` 对象列表。它可以包含一个或者多个产品
- `max_results`：这是一个整数，用于展示返回的推荐的最大数量

在这个方法中，我们执行以下的动作：

```
1. 得到所给 `Product` 对象的 ID
2. 如果只有一个产品，我们就检索一同购买的产品的 ID，并按照他们的购买时间来排序。这样做，我们就可以使用 Redis 的 `ZRANGE` 命令。我们通过 `max_results` 属性来限制结果的数量。
3. 如果有多于一个的产品被给予，我们就生成一个临时的和产品 ID 一同创建的 Redis 键。
4. 我们把包含在每个所给产品的有序集中东西的评分组合并相加，我们使用 Redis 的 `ZUNIONSTORE` 命令来实现这个操作。`ZUNIONSTORE` 命令执行了对有序集的所给键的求和，然后在新的 Redis 键中保存每个元素的求和。你可以在这里阅读更多关于这个命令的信息：http://redisio/commands/ZUNIONSTORE 。我们在临时键中保存分数的求和。
5. 因为我们已经求和了评分，我们或许会获取我们推荐的重复商品。我们就使用 `ZREM` 命令来把他们从生成的有序集中删除。
6. 我们从临时键中检索产品 ID，使用 `ZREM` 命令来按照他们的评分排序。我们把结果的数量限制在 `max_results` 属性指定的值内。然后我们删除了临时键。
7. 最后，我们用所给的 `id` 获取 `Product` 对象，并且按照同样的顺序来排序。
```

为了更加方便使用，让我们添加一个方法来清除所有的推荐。
把下列方法添加进 `Recommender` 类中：

```
def clear_purchases(self):
    for id in Product.objects.values_list('id', flat=True):
        r.delete(self.get_product_key(id))
```

让我们试试我们的推荐引擎。确保你在数据库中引入了几个 `Product` 对象并且在 shell 中使用了下面的命令来初始化 Redis 服务器：

```
src/redis-server
```

打开另外一个 shell ，执行 `python manage.py shell` ，输入下面的代码来检索几个产品：

```
from shop.models import Product
black_tea = Product.objects.get(translations__name='Black tea')
red_tea = Product.objects.get(translations__name='Red tea')
green_tea = Product.objects.get(translations__name='Green tea')
tea_powder = Product.objects.get(translations__name='Tea powder')
```

然后，添加一些测试购物到推荐引擎中：

```
from shop.recommender import Recommender
r = Recommender()
r.products_bought([black_tea, red_tea])
r.products_bought([black_tea, green_tea])
r.products_bought([red_tea, black_tea, tea_powder])
r.products_bought([green_tea, tea_powder])
r.products_bought([black_tea, tea_powder])
r.products_bought([red_tea, green_tea])
```

我们已经保存了下面的评分：

```
black_tea: red_tea (2), tea_powder (2), green_tea (1)
red_tea: black_tea (2), tea_powder (1), green_tea (1)
green_tea: black_tea (1), tea_powder (1), red_tea(1)
tea_powder: black_tea (2), red_tea (1), green_tea (1)
```

让我们看看为单一产品的推荐吧：

```
>>> r.suggest_products_for([black_tea])
[<Product: Tea powder>, <Product: Red tea>, <Product: Green tea>]
>>> r.suggest_products_for([red_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Green tea>]
>>> r.suggest_products_for([green_tea])
[<Product: Black tea>, <Product: Tea powder>, <Product: Red tea>]
>>> r.suggest_products_for([tea_powder])
[<Product: Black tea>, <Product: Red tea>, <Product: Green tea>]
```

正如你所看到的那样，推荐产品的顺序正式基于他们的评分。让我们来看看为多个产品的推荐吧：

```
>>> r.suggest_products_for([black_tea, red_tea])
[<Product: Tea powder>, <Product: Green tea>]
>>> r.suggest_products_for([green_tea, red_tea])
```

```
[<Product: Black tea>, <Product: Tea powder>]
>>> r.suggest_products_for([tea_powder, black_tea])
[<Product: Red tea>, <Product: Green tea>]
```

你可以看到推荐产品的顺序和评分总和的顺序是一样的。比如 `black_tea`, `red_tea`, `tea_powder(2+1)` 和 `green_tea(1=1)` 的产品推荐就是这样。
我们必须保证我们的推荐算法按照预期那样工作。让我们在我们的站点上展示我们的推荐吧。
编辑 `shop` 应用的 `views.py`，添加以下库：

```
from .recommender import Recommender
```

把下面的代码添加进 `product_detail` 视图（view）中的 `render()` 之前：

```
r = Recommender()
recommended_products = r.suggest_products_for([product], 4)
```

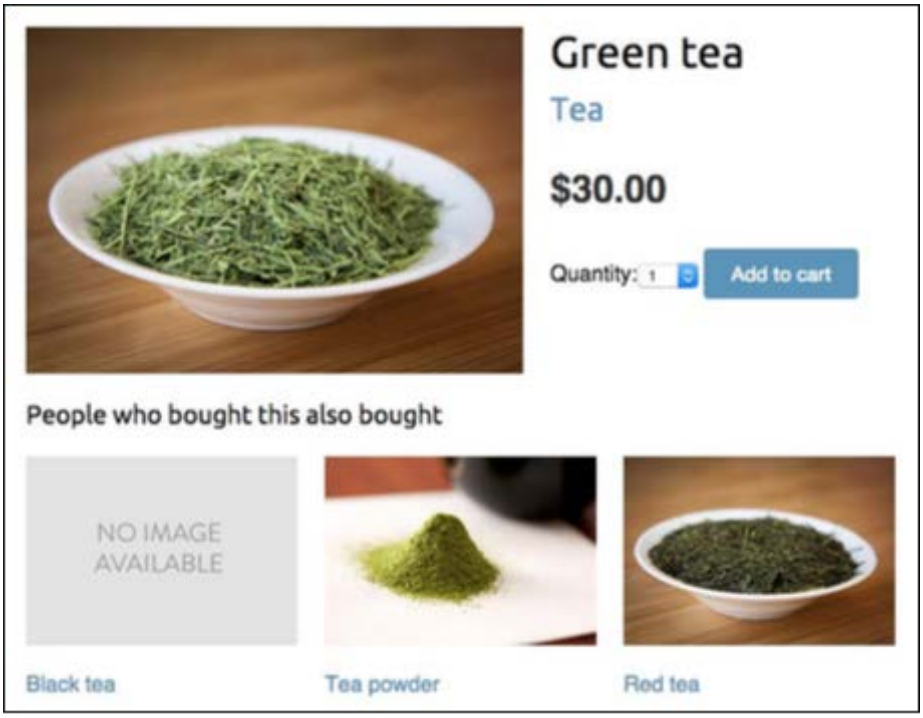
我们得到了四个最多产品推荐。 `product_detail` 视图（view）现在看起来像这样：

```
from .recommender import Recommender

def product_detail(request, id, slug):
    product = get_object_or_404(Product,
                                id=id,
                                slug=slug,
                                available=True)
    cart_product_form = CartAddProductForm()
    r = Recommender()
    recommended_products = r.suggest_products_for([product], 4)
    return render(request,
                  'shop/product/detail.html',
                  {'product': product,
                  'cart_product_form': cart_product_form,
                  'recommended_products': recommended_products})
```

现在编辑 `shop` 应用的 `shop/product/detail.html` 模板（template），把以下代码添加在 `{{ product.description|linebreaks }}` 的后面：

```
{% if recommended_products %}
<div class="recommendations">
<h3>{% trans "People who bought this also bought" %}</h3>
{% for p in recommended_products %}
<div class="item">
<a href="{{ p.get_absolute_url }}">
<img src="{% if p.image %}{{ p.image.url }}{% else %}{%
static "img/no_image.png" %}{% endif %}">
</a>
<p><a href="{{ p.get_absolute_url }}">{{ p.name }}</a></p>
</div>
{% endfor %}
</div>
{% endif %}
```

运行开发服务器，访问：http://127.0.0.1:8000/en/ 。点击一个产品来查看它的详情页。你应该看到展示在下方的推荐产品，就象这样：



django-9-13

我们也将在购物车当中引入产品推荐。产品推荐将会基于用户购物车中添加的产品。编辑 `cart` 应用的 `views.py` ，添加以下库：

```
from shop.recommender import Recommender
```

编辑 `cart_detail` 视图（view），让它看起来像这样：

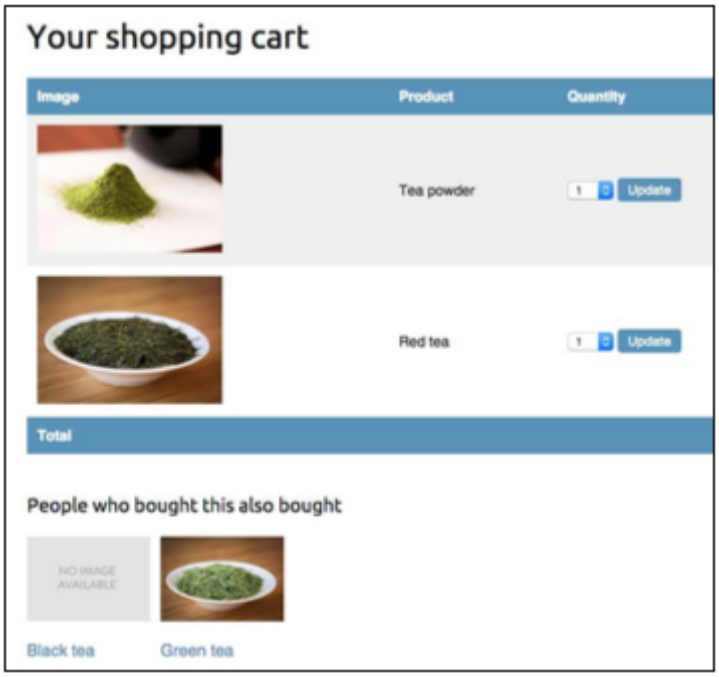
```
def cart_detail(request):
    cart = Cart(request)
    for item in cart:
        item['update_quantity_form'] = CartAddProductForm(
                    initial={'quantity': item['quantity'],
                             'update': True})
    coupon_apply_form = CouponApplyForm()

    r = Recommender()
    cart_products = [item['product'] for item in cart]
    recommended_products = r.suggest_products_for(cart_products,
                             max_results=4)
    return render(request,
                'cart/detail.html',
                {'cart': cart,
                'coupon_apply_form': coupon_apply_form,
                'recommended_products': recommendeproducts})
```

编辑 `cart` 应用的 `cart/detail.html` ，把下列代码添加在 `<table>`HTML 标签后面：

```
{% if recommended_products %}
<div class="recommendations cart">
<h3>{% trans "People who bought this also bought" %}</h3>
{% for p in recommended_products %}
```

```
<div class="item">
<a href="{{ p.get_absolute_url }}">
<img src="{% if p.image %}{{ p.image.url }}{% else %}{%
static "img/no_image.png" %}{% endif %}">
</a>
<p><a href="{{ p.get_absolute_url }}">{{ p.name }}</a></p>
</div>
{% endfor %}
</div>
{% endif %}
```

访问 http://127.0.0.1:8000/en/ ，在购物车当中添加一些商品。东航拟导航向 http://127.0.0.1:8000/en/cart/ 时，你可以在购物车下看到锐减的产品，就像下面这样：



django-9-14

恭喜你！你已经使用 Django 和 Redis 构建了一个完整的推荐引擎。

总结

在这一章中，你使用会话创建了一个优惠券系统。你学习了国际化和本地化是如何工作的。你也使用 Redis 构建了一个推荐引擎。

在下一章中，你将开始一个新的项目。你将会用 Django 创建一个在线学习平台，并且使用基于类的视图（view）。你将学会创建一个定制的内容管理系统（CMS）。

第十章 创建一个在线学习平台（e-Learning Platform）

在上一章，你添加国际化到你的在线商店项目中。你还构建了一个优惠券系统和一个商品推荐引擎。

在这章中，你会创建一个新项目。你将构建一个在线学习平台创建一个定制内容管理系统。

在这章中，你会学习以下操作：

- 创建 fixtures 给你的模型
- 使用模型继承
- 创建定制模型字段
- 使用基于类的视图和 mixins
- 构建 formsets
- 管理组合权限
- 创建一个内容管理系统

创建一个在线学习平台

我们最实际的项目将会是一个在线学习平台。在本章中，我们将要构建一个灵活的**内容管理系统（CMS）**用来允许教师来创建课程和管理它们的内容。
首先，创建一个虚拟环境给你的新项目并且激活它通过以下命令：

```
mkdir env
virtualenv env/educa
source env/educa/bin/activate
```

安装 Django 到你的虚拟环境中通过以下命令：

```
pip install Django==1.8.6
```

我们将要管理图片上传在我们的项目中，所以我们还需要安装 Pillow 通过以下命令：

```
pip install Pillow==2.9.0
```

创建一个新项目使用以下命令：

```
django-admin startproject educa
```

进入这个新的 *educa* 目录并且创建一个新应用使用以下命令：

```
cd educa
django-admin startapp courese
```

编辑 *educa* 项目的 *settings.py* 文件并且添加 *courses* 到 `INSTALLED_APPS` 设置中如下所示：

```
INSTALLED_APPS = (
    'courses',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
)
```

courses 应用现在已经在这个项目中激活。让我们定义模型给课程以及课程内容。

构建课程模型

我们的在线学习平台将会提供课程在许多主题中。每一个课程都将会划分为一个可配置的模块编号，并且每个模块将会包含一个可配置的内容编号。将会有许多类型的内容：文本，文件，图片，或者视频。下面的例子展示了我们的课程目录的数据结构：

```
Subject 1
    Course 1
        Module 1
            Content 1 (images)
            Content 3 (text)
        Module 2
            Content 4 (text)
            Content 5 (file)
```

```
        Content 6 (video)
        ...
```

让我们来构建课程模型。编辑 *courses* 应用的 *models.py* 文件并且添加如下代码：

```
from django.db import models
from django.contrib.auth.models import User
class Subject(models.Model):
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)
    class Meta:
        ordering = ('title',)
    def __str__(self):
        return self.title

class Course(models.Model):
    owner = models.ForeignKey(User,
                              related_name='courses_created')
    subject = models.ForeignKey(Subject,
                                related_name='courses')
    title = models.CharField(max_length=200)
    slug = models.SlugField(max_length=200, unique=True)
    overview = models.TextField()
    created = models.DateTimeField(auto_now_add=True)
    class Meta:
        ordering = ('-created',)
    def __str__(self):
        return self.title

class Module(models.Model):
    course = models.ForeignKey(Course, related_name='modules')
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    def __str__(self):
        return self.title
```

这些是最初的 *Subject,Course,*以及 *Module* 模型。*Course* 模型字段如下所示：

- owner：创建这个课程的教师。
- subject：这个课程属于的主题。这是一个 *ForeingnKey* 字段指向 *Subject* 模型。
- title：课程标题。
- slug：课程的 slug。之后它将会被用在 URLs 中。
- overview：这是一个 *TextFied* 列用来包含一个关于课程的概述。
- created：课程被创建的日期和时间。它将会被 Django 自动设置当创建一个新的对象，因为 `auto_now_add=True`。

每一个课程都被划分为多个模块。因此，*Module* 模型包含一个 *ForeignKey* 字段用来指向 *Course* 模型。

打开 shell 并且运行一下命令来给这个应用创建最初的迁移：

```
python manange.py makemigrations
```

你将会看到以下输出：

```
Migrations for 'courses':
    0001_initial.py:
      - Create model Course
      - Create model Module
      - Create model Subject
      - Add field subject to course
```

之后，运行一下命令来应用所有的迁移到数据库中：

```
python manage.py migrate
```

你将会看到一个输出包含所有应用的迁移，包括 Django 的那些。这个输出将会包含以下行：

```
Applying courses.0001_initial... OK
```

这告诉我们那个我们的 *courses* 引用模型已经同步到了数据库中。

注册模型到管理平台中

我们将要添加课程模型到管理平台中。编辑 *courses* 应用目录下的 *admin.py* 文件并且添加以下代码：

```
from django.contrib import admin
from .models import Subject, Course, Module

@admin.register(Subject)
class SubjectAdmin(admin.ModelAdmin):
    list_display = ['title', 'slug']
    prepopulated_fields = {'slug': ('title',)}

class ModuleInline(admin.StackedInline):
    model = Module
@admin.register(Course)

class CourseAdmin(admin.ModelAdmin):
    list_display = ['title', 'subject', 'created']
    list_filter = ['created', 'subject']
    search_fields = ['title', 'overview']
    prepopulated_fields = {'slug': ('title',)}
    inlines = [ModuleInline]
```

课程应用的模型现在已经在管理平台中注册。我们使用`@admin.register()`装饰器替代了`admin.site.register()`方法。它们都提供了相同的功能。

提供最初数据给模型

有时候你可能想要预装你的数据库通过使用硬编码数据。这是很有用的，当自动包含最初数据在项目设置中用来替代手工去添加数据。Django 自带一个简单的方法来加载以及转储数据库中的数据到字段中，这被称为 fixtures。

Django 支持 fixtures 在 JSON,XML,或者 YAML 格式中。我们将要创建一个 fixture 用来包含一些最初的 *Subject* 对象给我们的项目。

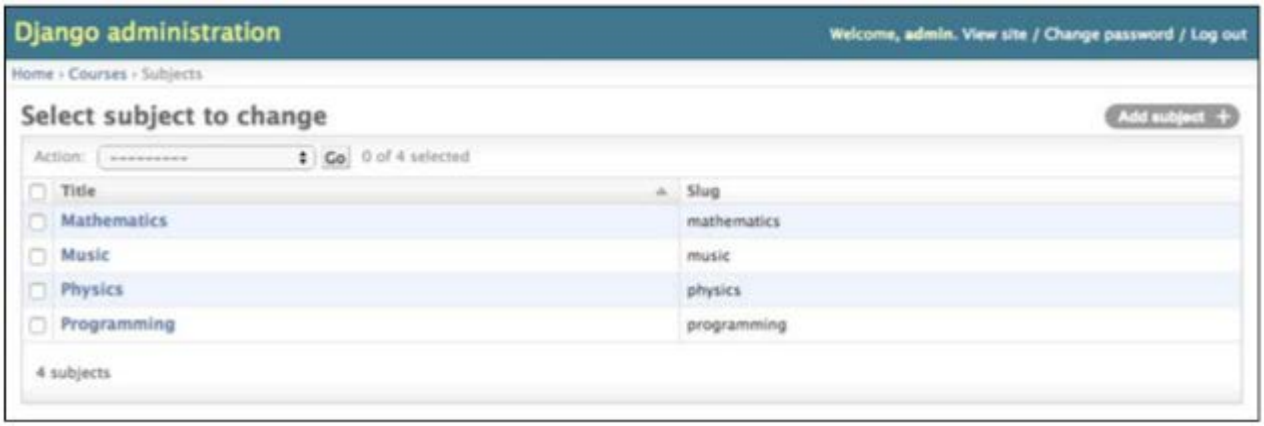
首先，创建一个超级用户使用如下命令：

```
python manage.py createsuperuser
```

之后，运行开发服务器使用以下命令：

```
python manage.py runserver
```

现在，打开 http://127.0.0.1:8000/admin/courses/subject/ 在你的浏览器中。创建一些主题通过使用管理平台。列页面看上去如下所示：



django-10-1

运行一下命令在 shell 中：

```
python manage.py dumpdata courese --indent=2
```

你会看到类似以下的输出：

```
[
{
  "fileld:": {
    "title": "Programming",
    "slug": "programming"
  },
  "model": "courses.subject",
  "pk": 1
},
{
"fields": {
    "title": "Mathematics",
    "slug": "mathematics"
  },
  "model": "courses.subject",
  "pk": 2
},
{
"fields": {
    "title": "Physics",
    "slug": "physics"
  },
  "model": "courses.subject",
  "pk": 3
}, {
  "fields": {
    "title": "Music",
    "slug": "music"
```

```
  },
  "model": "courses.subject",
  "pk": 4
}
]
```

dumpdata 命令从数据库中转储数据到标准输出中，默认序列化为 JSON。这串数据结构包含的信息关于这个模型以及它的字段将会被 Django 用来加载它到数据库中。
你可以提供应用名给这命令或者指定模型给输出数据使用 `app.Model` 格式。你还可以指定格式通过使用 `--format` 标记。默认的，*dumpdata* 输出序列化数据给标准输出。当然，你可以表明一个输出文件通过使用`--output` 标记。`--indent` 标记允许你指定缩进。更多信息关于 *udmpdata* 参数，运行 `python manage.py dumpdata --help`。
保存这个转储为一个 fixtures 文件到 *orders* 应用的 *fixtures/*目录中，通过使用如下命令：

```
mkdir courses/fixtures
python manage.py dumpdata courses --indent=2 --output=courses/fixtures/
subjects.json
```

使用管理平台去移除你之前创建的主题。之后加载 fixture 到数据库中通过使用以下命令：

```
python manage.py loaddata subjects.json
```

所有包含在 fixture 中的 `subject` 对象都会加载到数据库中。
默认的，Django 会寻找每一个应用的 *fixtures/*目录下的文件，但是你可以指定 fixture 文件的完整路径给 *loaddata* 命令。你还可以使用 `FIXTURE_DIRS` 设置来告诉 Django 去额外的目录寻找 fixtures。
Fixtures 并不只对初始化数据有用，还可以提供简单的数据给你的应用或者数据请求给你的测试用例。
你可以找到更多关于如何使用 fixtures 在测试中，通过访问 https://docs.djangoproject.com/en/1.8/topics/testing/tools/#topics-testing-fixtures 。
如果你想要加载 fixturres 在模型迁移中，去看下 Django 的文档关于数据迁移。请记住，我们创建了一个定制迁移在**第九章，扩展你的商店**来迁移存在的数据在修改给翻译的模型之后。你可以找到迁移数据的文档，通过访问 https://docs.djangoproject.com/en/1.8/topics/migrations/#data-migrations 。

给不同的内容创建模型

我们打算添加各种不同的内容类型给课程模块，例如文本，图片，文件以及视屏。我们需要一个通用的数据模型可以允许我们去存储不同的内容。在**第六章，跟踪用户行为**中，你已经学习过有关使用通用关系方便的创建外键能够指向任何模型的对象。我们将要创建一个 *content* 模型相当于模块内容以及定义一个通用关系来连接任意种类的内容。
编辑 *courses* 应用下的 *models.py* 文件并且添加如下导入：

```
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey
```

之后添加如下代码到文件后面：

```
class Content(models.Model):
    module = models.ForeignKey(Module, related_name='contents')
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    item = GenericForeignKey('content_type', 'object_id')
```

这就是一个 *Content* 模型。一个模块包含多种内容，所有我们定义了一个 `ForeignKey` 字段给 *module* 模型。我们还设置了一个通用关系来连接对象从不同的模型中相当于不同的内容类型。请记住，我们需要三种不同的字段来设置一个通用关系。在我们的 *Content* 模型中，它们是：

- content_type：一个 *ForeignKey* 字段指向 *ContentType* 模型
- object_id：这是 *PositiveIntegerField* 用来存储有关联对象的关键字
- item：一个 *GenericForeignKey* 字段指向被关联的对象通过结合前两个字段

只有 `content_type` 和 `object_id` 字段有一个对应列在这个模型的数据库表中。`item` 字段允许你去检索或者直接设置关联对象，并且它的功能是简历在其他两个字段之上。
我们将要使用一个不同的模型给每一种内容。我们的内容模型将会有很多共有的字段，但是它们将会有不同之处在它们存储的真实内容中。

使用模型继承

Django 支持模型继承。类似与 Python 中的标准类继承。Django 提供以下三种方式来使用模型继承：

- Abstract models：非常有用当你想要安置一些公用信息到多个模型中。没有数据库表会被创建给抽象模型。
- Multi-table model inheritance：可适当的利用当每个模型经过慎重考虑都是一个它自身的完整的模型。每个模型都会创建一个数据库表。
- Proxy models：非常有用当你需要改变一个模型行为，比如说，包含额外的方法，修改默认管理器，或者使用不同的元选项。没有数据表会被创建给代理模型。

让我们对以上三者都来一次近距离的实践。

抽象模型

一个抽象模型就是一个基础类，你定义在其中的字段就是你想要包含到所有子模型中的字段。Djnago 不会创建任何数据库表给抽象模型。每个子模型都会创建一张数据库表，包含有继承自抽象类的字段以及在子模型中自己定义的字段。
为了抽象一个模型，你需要在 `Meta` 类中包含 `abstract=True`。Django 将会认出这个模型是一个抽象模型并且不会给它创建数据库表。为了创建子模型，你只需要基于这个抽象模型。下面就是一个例子关于一个抽象的 *Content* 模型和一个子的 `Text` 模型：

```
from django.db import models

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        abstract = True

class Text(BaseContent):
    body = models.TextField()
```

在这个例子中，Django 将只会给 `Text` 模型创建表，包含 `title`,`created` 以及 `body` 字段。

多表模型继承

在多表模型继承中，每个模型对应一个张数据库表。Django 创建一个 *OneToOneField* 字段给子模型创建关系指向它的父模型。
为了使用多表继承，你必须基于一个存在的模型。djnago 将会创建一张数据表给每个源头模型以及子模型。以下例子展示多表继承：

```
from django.db import models
```

```
class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class Text(BaseContent):
    body = models.TextField()
```

Django 将会包含一个自动生成的 *OneToOneField* 字段在 `Text` 模型中并且给每个模型创建一张数据库表。

代理模型

代理模型被用于改变一个模型的行为，举个例子，包含额外的方法或者不同的元选项。每个模型对源头模型的数据库表起作用。为了创建一个代理模型，在这个模型的 `Meta` 类中添加 `proxy=True`。
以下例子说明如何创建一个代理模型：

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

class BaseContent(models.Model):
    title = models.CharField(max_length=100)
    created = models.DateTimeField(auto_now_add=True)

class OrderedContent(BaseContent):
    class Meta:
        proxy = True
        ordering = ['created']

    def created_delta(self):
        return timezone.now() - self.created
```

这里，我们定义了一个 *OrderedContent* 模型这是一个代理模型给 *Content* 模型使用。这个模型提供了一个默认的排序给查询集并且一个额外的 `create_delta()`方法。这两个模型，`Content` 和 `OrderedContent`，对同一个数据库表起作用，并且通过任一一个模型都能通过 ORM 渠道连接到对象。

创建内容模型

我们的 *courses* 应用的 *Content* 模型包含一个通用关系来连接不同类型的内容给该应用。我们将要创建一个不同的模型给每种类型的内容。所有内容模型将会有一些公用的字段，以及额外的字段去存储定制数据。我们将会创建一个抽象模型来提供公用字段给所有内容模型。
编辑 *courses* 应用的 *models.py* 文件，并且添加以下代码：

```
class ItemBase(models.Model):
    owner = models.ForeignKey(User,
                              related_name='%(class)s_related')
    title = models.CharField(max_length=250)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True
    def __str__(self):
        return self.title
```

```
class Text(ItemBase):
    content = models.TextField()

class File(ItemBase):
    file = models.FileField(upload_to='files')

class Image(ItemBase):
    file = models.FileField(upload_to='images')

class Video(ItemBase):
    url = models.URLField()
```

在这串代码中，我们定义了一个抽象模型命名为`ItemBase`。除此以外，我们在`Meta`类中设置`abstract=True`。在这个模型中，我们定义`owner`,`title`,`created`，以及`updated`字段。这些公用字段将会被所有的内容类型使用到。`owner`字段允许我们去存储哪个用户创建了这个内容。因为和这个字段是被定义在一个抽象类中，我们需要不同的`related_name`给每个子模型。Django 允许我们去指定一个占位符给`model`类名在`related_name`属性类似`%(class)s`。为了做到这些，`related_name`对每个子模型都会自动生成。因为我们使用`%(class)s_related`作为`related_name`，给子模型的相对关系将各自是`text_related`,`file_related`,`image_related`,以及`vide0_related`。

我们已经定义了四种不同的内容模型，它们都继承自`ItemBase`抽象模型，它们是：

- Text：用来存储文本内容。
- File：用来存储文件，例如 PDF。
- Image：用来存储图片文件。
- Video：用来存储视频。我们使用一个`URLField`字段来提供一个视频 URL 为了嵌入该视频。

每个子模型包含定义在`ItemBase`类中的字段以及它自己的字段。`text_related`,`file_related`,`image_related`,以及`vide0_related`都会各自创建一张数据库表。不会有数据库表连接到`ItemBase`模型，因为它是一个抽象模型。

编辑你之前创建的 *Content* 模型，修改它的`content_type`字段如下所示：

```
content_type = models.ForeignKey(ContentType,
                  limit_choices_to={'model__in':('text',
                                    'video',
                                    'image',
                                    'file')})
```

我们添加一个`limit_choices_to`参数来限制`ContentType`对象可以被通用关系使用。我们使用`model__in`字段查找过滤这个查询给`ContentType`对象通过一个`model`属性就像'text','video','image',或者'file'。

让我们创建一个迁移来包含这些新的模型我们之前添加的。运行以下命令：

```
python manage.py makemigrations
```

你会看到以下输出：

```
Migrations for 'courses':
    0002_content_file_image_text_video.py:
      - Create model Content
      - Create model File
      - Create model Image
      - Create model Text
```

```
    - Create model Video
```

之后，运行一下命令来应用新的迁移：

```
python manage.py migrate
```

你会在输出结果看到以下内容：

```
Running migrations:
    Rendering model states... DONE
    Applying courses.0002_content_file_image_text_video... OK
```

我们之前创建的模型对于添加不同的内容给课程模块是很合适的。但是，仍然有一些东西是被遗漏的在我们的模型中。课程模块和内容应当跟随一个特定的顺序。我们需要一个字段，这个字段允许我们简单的排序它们。

创建定制模型字段

Django 自带一个完整的模型字段采集能让你用来构建你的模型。当然，你也可以创建你自己的模型字段来存储定制数据或者改变现有字段的行为。

我们需要一个字段允许我们给对象们定义次序。如果你想通过 Djanog 提供的一个字段来方便的处理这点，你大概会想到添加一个 `PositiveIntegerField` 给你的模型。这是一个好的起点。我们可以创建一个定制字段，该字段继承自 `PositiveIntegerField` 并且提供额外的行为。

有两种相关的功能我们将构建到我们的次序字段中：

- 自动分配一个次序值当没有指定的次序被提供的时候。当没有次数被提供的时候存储一个对象，我们的字段将自动分配下一个次序，该次序基于最后存在次序的对象。如果有两个对象，分别是次序 1 和次序 2，当保存第三个对象的时候，我们会自动分配次序 3 给第三个对象如果没有给予指定的次序。
- 次序对象关于其他的字段。课程模块将按照它们所属的课程和相关模块的内容进行排序。

创建一个新的 *fields.py* 文件到 *courses* 应用目录下，然后添加以下代码：

```
from django.db import models
from django.core.exceptions import ObjectDoesNotExist

class OrderField(models.PositiveIntegerField):

    def __init__(self, for_fields=None, *args, **kwargs):
        self.for_fields = for_fields
        super(OrderField, self).__init__(*args, **kwargs)

    def pre_save(self, model_instance, add):
        if getattr(model_instance, self.attname) is None:
            # no current value
            try:
                qs = self.model.objects.all()
                if self.for_fields:
                    # filter by objects with the same field values
                    # for the fields in "for_fields"
                    query = {field: getattr(model_instance, field) for field in self.for_fields}
                    qs = qs.filter(**query)
                # get the order of the last item
                last_item = qs.latest(self.attname)
```

```
                value = last_item.order + 1
            except ObjectDoesNotExist:
                value = 0
            setattr(model_instance, self.attname, value)
            return value
        else:
            return super(OrderField,
                         self).pre_save(model_instance, add)
```

这就是我们的定制 `OrderField`.它继承自 Django 提供的 `PositiveIntegerField` 字段。我们的 `OrderField` 字段需要一个可选的 `for_fields` 参数，这个参数允许我们表明次序根据这些字段进行计算。
我们的字段覆盖 `PositiveIntegerField` 字段的 `pre_save()`方法，这字段会在保存这个字段到数据库之前进行执行。在这个方法中，我们做了以下操作：

- 1 我们检查在模型实例中的字段是否已有一个值。我们是 `self.attname`，它是在这个模型中给予这个字段的属性名。如果在这个属性的值不同于 `None`，我们就会进行如下操作来计算出一个次序给它：
  - 1 我们构建一个查询集去检索所有对象给这个字段的模型。我们通过访问 `self.model` 来检索该字段所属的模型类。
  - 2 我们通过模型字段中的那些被定义在 `for_fields` 参数中的字段的当前值来过滤这个查询集（如果有的话）。为了做到这点，我们通过给予的字段来计算次序。
  - 3 我们从数据库中使用最高的次序来检索对象通过是用 `last_item = qs.latest(self.attname)`。如果没有找到对象，我们假定这个对象是第一个并且分配次序 0 给它。
  - 4 如果找到一个对象，我们给找到的最高次序增加 1。
  - 5 我们分配计算过的次序给在模型实例中的字段的值通过使用 `setattr()`并且返回它。
- 2 如果这个模型实例有一个值给当前的字段，我们不需要做任何事情。

当你创建定制模型字段，使它们通过。避免硬编码数据被依赖一个指定模型或者字段。你的字段才能在任意模型中起效。
你可以找到更多的信息关于编写定制模型字段，通过访问 https://docs.djangoproject.com/en/1.8/howto/custom-model-fields/ 。
让我们添加新的字段给我们的模型。编辑 *courses* 应用的 *models.py* 文件，导入新的字段如下所示：

```
from .fields import OrderField
```

之后，添加以下 `OrderField` 字段给 `Module` 模型：

```
order = OrderField(blank=True, for_fields=['course'])
```

我们命名新的字段为 `order`，并且我们指定该字段的次序根据课程计算通过设置 `for_fields=['course']`。这意味着新的模块的次序将会是最后的同样的 *Course* 对象模块的次序增加 1。现在你可以编辑 `Module` 模型的`__str__()`方法来包含它的次序如下所示：

```
def __str__(self):
    return '{}. {}'.format(self.order, self.title)
```

模块内容也需要跟随一个特定的次序。添加一个 `OrderField` 字段给 `Content` 模型如下所示：

```
order = OrderField(blank=True, for_fields=['module'])
```

这一次，我们指定这个次序根据 `moduel` 字段进行计算。最后，让我们给这两个模型都添加一个默认的序列。添加如下 `Meta` 类给 `Module` 和 `Content` 模型：

```
class Meta:
```

```
    ordering = ['order']
```

`Module` 和 `Content` 模型现在看上去如下所示：

```
class Module(models.Model):
    course = models.ForeignKey(Course,related_name='modules')
    title = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    order = OrderField(blank=True, for_fields=['course'])

    class Meta:
        ordering = ['order']
    def __str__(self):
        return '{}. {}'.format(self.order, self.title)

class Content(models.Model):
    module = models.ForeignKey(Module, related_name='contents')
    content_type = models.ForeignKey(ContentType,
                  limit_choices_to={'model__in':('text',
                                                 'video',
                                                 'file')})
    item = GenericForeignKey('content_type', 'object_id')
    order = OrderField(blank=True, for_fields=['module'])
    class Meta:
        ordering = ['order']
```

让我们创建一个新模型迁移来体现新的次序字段。打开 shell 并且运行如下命令：

```
python manage.py makemigrations courses
```

你会看到如下输出：

```
You are trying to add a non-nullable field 'order' to content without a default; we can't do that (the database
needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows)
 2) Quit, and let me add a default in models.py
Select an option:
```

Django 正在告诉我们由于我们添加了一个新的字段给已经存在的模型，我们必须提供一个默认值给数据库中已经存在的各行记录。如果这个字段有 `null=True`，它将会采用空值并且 Django 将会创建这个迁移而不会找我们要一个默认值。我们可以指定一个默认值或者取消这次迁移然后在创建这个迁移之前去 `models.py` 文件中给 `order` 字段添加一个 `default` 属性。
输入 1 然后按下回车来提供一个默认值给已经存在的记录。你将会看到如下输出：

```
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do e.g. timezone.now()
>>>
```

输入 0 作为给已经存在的记录的默认值然后按下回车。Djanog 将会询问你还需要一个默认值给 `Module` 模型。选择第一个选项然后再次输入 0 作为默认值。最后，你将会看到如下类似的输入：

```
Migrations for 'courses':
 0003_auto_20150701_1851.py:
    - Change Meta options on content
    - Change Meta options on module
    - Add field order to content
    - Add field order to module
```

之后，应用新的迁移通过以下命令：

```
python manage.py migrate
```

这个命令的输出将会通知你这次迁移成功的应用，如下所示：

```
Applying courses.0003_auto_20150701_1851... OK
```

让我们测试我们新的字段。打开 shell 使用 `python manage.py shell` 然后创建一个新的课程如下所示：

```
>>> from django.contrib.auth.models import User
>>> from courses.models import Subject, Course, Module
>>> user = User.objects.latest('id')
>>> subject = Subject.objects.latest('id')
>>> c1 = Course.objects.create(subject=subject, owner=user,
title='Course 1', slug='course1')
```

我们已经在数据库中创建了一个课程。现在让我们给课程添加模块然后看下模块的次序是如何自动计算的。我们创建一个初始模板然后检查它的次序：

```
>>> m1 = Module.objects.create(course=c1, title='Module 1')
>>> m1.order
0
```

`OrderField` 设置这个模块的值为 0，因为这个模块是这个课程的第一个 `Module` 对象。现在我们创建第二个对象给这个课程：

```
>>> m2 = Module.objects.create(course=c1, title='Module 2')
>>> m2.order
1
```

`OrderField` 计算出下一个次序值是已经存在的对象中最高的次序值加上 1。让我们创建第三个模块强制指定一个次序：

```
>>> m3 = Module.objects.create(course=c1, title='Module 3', order=5)
>>> m3.order
5
```

如果我们指定了一个定制次序，`OrderField` 字段将不会进行干涉，然后 `order` 的值将会使用指定的次序。让我们添加第四个模块：

```
>>> m4 = Module.objects.create(course=c1, title='Module 4')
>>> m4.order
6
```

这第四个模块的次序会被自动设置。我们的 `OrderField` 字段不会保证所有的次序值是连续的。无论如何，它会根据已经存在的次序值并且分配下一个次序基于已经存在的最高次序。
让我们创建第二个课程并且添加一个模块给它：

```
>>> c2 = Course.objects.create(subject=subject, title='Course 2', slug='course2', owner=user)
>>> m5 = Module.objects.create(course=c2, title='Module 1')
>>> m5.order
0
```

为了计算这个新模块的次序，该字段只需要考虑基于同一课程的已经存在的模块。由于这是第二个课程的第一个模块，次序的结果值就是 0 。这是因为我们指定 `for_fields=['course']`在 `Module` 模型的 `order` 字段中。
恭喜你！你已经成功的创建了你的第一个定制模型字段。

创建一个内容管理系统

到现在我们已经创建了一个多功能数据模型，我们将要构建一个内容管理系统（CMS）。这个 CMS 将允许教师去创建课程以及管理课程的内容。我们需要提供以下功能：

- 登录 CMS。
- 排列教师创建的课程。
- 创建，编辑以及删除课程。
- 添加模块到一个课程中并且重新排序它们。
- 添加不同类型的内容给每个模块并且重新排序内容。

添加认证系统

我们将要使用 Django 的认证框架到我们的平台中。教师和学生都将会是 Django *User* 模型的一个实例。从而，他们将能够登录这个站点通过使用 `django.contrib.auth` 的认证视图。
编辑 *educa* 项目的主 *urls.py* 文件然后包含 Django 认证框架的 `login` 和 `logout` 视图：

```
from django.conf.urls import include, url
from django.contrib import admin
from django.contrib.auth import views as auth_views

urlpatterns = [
    url(r'^accounts/login/$', auth_views.login, name='login'),
    url(r'^accounts/logout/$', auth_views.logout, name='logout'),
    url(r'^admin/', include(admin.site.urls)),
]
```

创建认证模板

在 *courses* 应用目录下创建如下文件结构：

```
templates/
    base.html
    registration/
        login.html
        logged_out.html
```

在构建认证模板之前，我们需要给我们的项目准备好基础模板。编辑 *base.html* 模板文件然后添加以下内容：

```
{% load staticfiles %}
```

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>{% block title %}Educa{% endblock %}</title>
    <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
    <div id="header">
      <a href="/" class="logo">Educa</a>
       <ul class="menu">
         {% if request.user.is_authenticated %}
           <li><a href="{% url "logout" %}">Sign out</a></li>
         {% else %}
           <li><a href="{% url "login" %}">Sign in</a></li>
         {% endif %}
       </ul>
     </div>
     <div id="content">
       {% block content %}
       {% endblock %}
     </div>
     <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
     <script>
       $(document).ready(function() {
         {% block domready %}
         {% endblock %}
       });
     </script>
   </body>
</html>
```

这个基础模板将会被其他的模板扩展。在这个模板中，我们定义了以下区块：

- title：这个区块是给别的模板用来给每个页面添加定制的标题。
- content：这个是内容的主区块。所有扩展基础模板的模板都可以添加各自的内容到这个区块。
- domready：位于 jQuery 的`$document.ready()`方法里面。它允许我们执行代码当 DOM 完成加载的时候。

这个模板使用的 CSS 样式位于本章实例代码的 *courses* 应用下的 *static/*目录下。你可以拷贝 *static/*目录到你的项目的相同目录下来使用它们。

编辑 *registration/login.html* 模板并且添加以下代码：

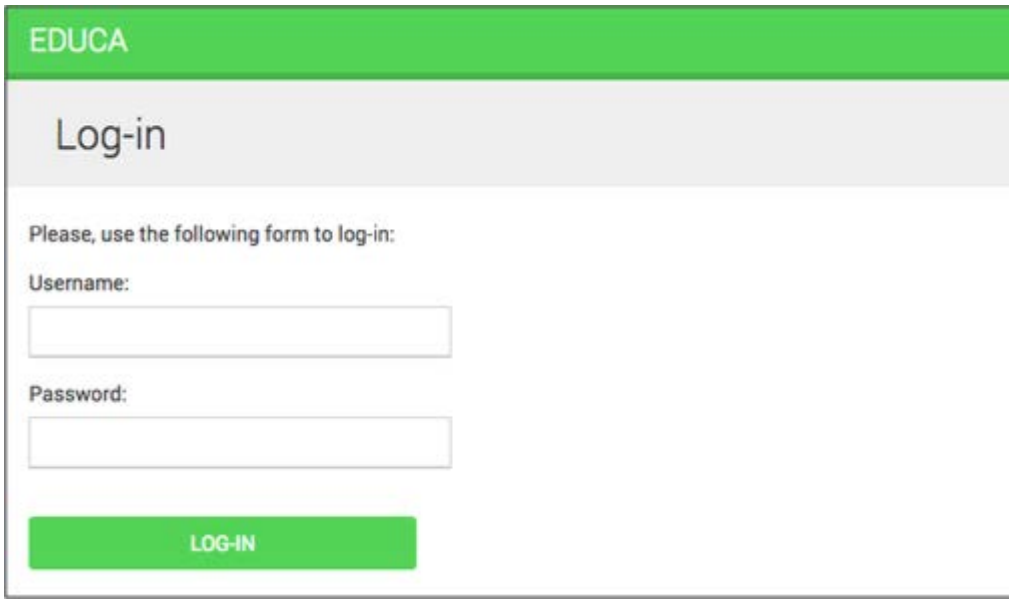
```
{% extends "base.html" %}

{% block title %}Log-in{% endblock %}

{% block content %}
    <h1>Log-in</h1>
    <div class="module">
      {% if form.errors %}
        <p>Your username and password didn't match. Please try again.</p>
```

```
      {% else %}
        <p>Please, use the following form to log-in:</p>
      {% endif %}
      <div class="login-form">
        <form action="{% url 'login' %}" method="post">
          {{ form.as_p }}
          {% csrf_token %}
          <input type="hidden" name="next" value="{{ next }}" />
          <p><input type="submit" value="Log-in"></p>
        </form>
      </div>
    </div>
{% endblock %}
```

这是一个给 Django 的 `login` 视图用的标准登录模板。编辑 *registration/logged_out.html* 模板然后添加以下代码：

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

{% block title %}Logged out{% endblock %}

{% block content %}
    <h1>Logged out</h1>
    <div class="module">
      <p>You have been successfully logged out. You can <a href="{% url"login" %}">log-in again</a>.</p>
    </div>
{% endblock %}
```

这个模板将会在用户登出后展示。通过命令 `python manage.py runserver` 命令运行开发服务器然后在你的浏览器中打开 http://127.0.0.1:8000/accounts/login/ 。你会看到如下登录页面：



django-10-2

创建基于类的视图

我们将要构建一些视图用来创建，编辑，以及删除课程。为了这个目的我们将会使用基于类的视图。编辑 *courses* 应用的 *views.py* 文件并且添加如下代码：

```
from django.views.generic.list import ListView
from .models import Course

class ManageCourseListView(ListView):
    model = Course
    template_name = 'courses/manage/course/list.html'

    def get_queryset(self):
        qs = super(ManageCourseListView, self).get_queryset()
        return qs.filter(owner=self.request.user)
```

以上就是 `ManageCourseListView` 视图。它从 Django 的通用 `ListView` 继承而来。我们重写了这个视图的 `get_queryset()`方法来只对当前用户创建的课程进行检索。为了阻止用户对不是由他们创建的课程进行编辑，更新或者删除操作，我们还需要重写在创建，更新以及删除视图中的 `get_queryse()`方法。当你需要去提供一个指定行为给多个基于类的视图，推荐你使用 `mixins`。

对基于类的视图使用 mixins

mixins 是一种特殊的用于一个类的多重继承。你可以使用它们来提供常见的离散功能，添加到其他的 mixins，允许你去定义一个类的行为。有两种场景要使用 mixins：

- 你想要提供多个可选的特性给一个类
- 你想要使用一个特定的特性在多个类上

你可以找到关于如何在基于类的视图上使用 mixins 的文档，通过访问 https://docs.djangoproject.com/en/1.8/topics/class-based-views/mixins/ 。

Django 自带多个 mixins 用来提供额外的功能给你的基于类的视图。你可以找到所有的 mixins 在 https://docs.djangoproject.com/en/1.8/ref/class-based-views/mixins/ 。

我们将要创建一个 mixin 类来包含一个公用的行为并且将它给课程的视图使用。编辑 *courses* 应用的 *views.py* 文件，把它修改成如下所示：

```
from django.core.urlresolvers import reverse_lazy
from django.views.generic.list import ListView
from django.views.generic.edit import CreateView, UpdateView, \
                                      DeleteView
from .models import Course

class OwnerMixin(object):
    def get_queryset(self):
        qs = super(OwnerMixin, self).get_queryset()
        return qs.filter(owner=self.request.user)

class OwnerEditMixin(object):
    def form_valid(self, form):
        form.instance.owner = self.request.user
        return super(OwnerEditMixin, self).form_valid(form)

class OwnerCourseMixin(OwnerMixin):
    model = Course

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin):
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')
```

```
    template_name = 'courses/manage/course/form.html'

class ManageCourseListView(OwnerCourseMixin, ListView):
    template_name = 'courses/manage/course/list.html'

class CourseCreateView(OwnerCourseEditMixin, CreateView):
    pass

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):
    pass

class CourseDeleteView(OwnerCourseMixin, DeleteView):
    template_name = 'courses/manage/course/delete.html'
    success_url = reverse_lazy('manage_course_list')
```

在上述代码中，我们创建了 `OwnerMixin` 和 `OwnerEditMixin` 这两个 mixin。我们将要使用这些 mixins 与 Django 提供的 `ListView`，`CreateView`，`UpdateView` 以及 `DeleteView` 视图结合。`Ownermixin` 导入了以下方法。

- `get_queryset()`：这个方法被视图用来获取基础查询集。我们的 mixin 将会重写这个方法使用 `owner` 属性对对象进行过滤来检索属于当前用户的对象（request.user)。

`OwnerEditMixin` 导入了以下方法：

- `form_valid()`：这个方法被视图用来使用 Django 的 `ModelFormMixin` mixin，也就是说，带有表单的视图或模型表单的视图比如 `Createview` 和 `UpdateView.form_valid()` 当提交的表单是有效的时候就会被执行。这个方法默认的行为是保存实例（对于模型表单）以及重定向用户到 `success_url`。我们重写了这个方法来自动设置当前的用户到本次会被保存的对象的 `owner` 属性中。为了做到前面所说的，我们设置自动分配一个拥有者给该对象，当该对象被保存的时候。

我们的 `OwnerMixin` 类能够被视图用来和任意模型进行交互使模型包含一个 `owner` 属性。

我们还定义了一个 `OwnercourseMixin` 类，该类继承 `OwnerMixin` 并且提供以下属性给子视图：

- `model`：这个模型给查询集使用。被所有视图使用。

我们定义一个 `OwnerCourseEditMixin` mixin 通过以下属性：

- `fields`：这些模型字段用来从 `CreateView` 和 `UpdateView` 视图中构建模型。
- `success_url`：被 `CreateView` 和 `UpdateView` 使用来在表单成功提交之后重定向用户。我们之后将会创建一个名为 `manage_course_list` 的 URL 来使用。

最后，我们创建以下视图，这些视图都是基于 `OwnerCourseMixin` 的子类：

- `MangeCourselISTvIEW`：排序用户创建的课程。它从 `OwnerCourseMixin` 和 `ListView` 继承而来。
- `CoursecreateView`：使用模型表单来创建一个新的 `Course` 对象。它使用定义在 `OwnerCourseEditMixin` 中的字段来构建一个表单模型并且也是 `CreateView` 的子类。
- `CourseUpdateView`：允许编辑一个现有的 `Course` 对象。它从 `OwnerCourseMixin` 和 `UpdateView` 继承而来。
- `CourseDeleteView`：从 `OwnerCourseMixin` 和通用的 `DeleteView` 继承而来。定义 `success_url` 在对象被删除的时候重定向用户。

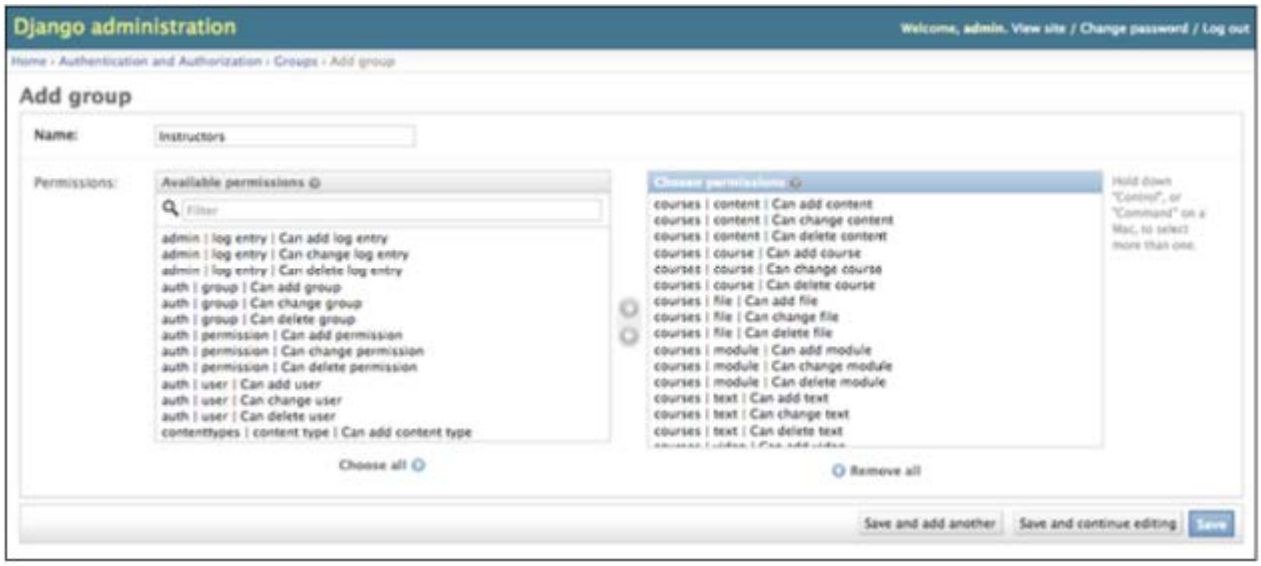
（译者注：上半章结束）

（以下是第十章下半章）

使用组和权限

我们已经创建了基础的视图来管理课程。但是目前所有的用户都可以使用这些视图。我们想要限制这些视图从而只有教师有权限去创建和管理课程。Django 认证框架包含一个权限系统允许你去分配权限给用户和组。我们将要创建一个组给教师用户并且分配权限给他们可以创建，更新以及删除课程。

使用命令 `python manage.py runserver` 命令运行开发服务器并且在你的浏览器中打开 http://127.0.0.1:8000/admin/auth/group/add/ 来创建一个新的 `Group` 对象。添加的组名为 *Instructors*,然后选择 *courses* 应用中的除了 *Subject* 模型的所有权限，如下所示：



django-10-3

如你所见，有三种不同的权限给每个模型：**Can add**，**can change** 以及 **Can delete**。选择好给这个组的权限之后，点击 **Save** 按钮。

Django 会自动给模型创建权限，但是你也可以创建定制的权限。你可以找到更多关于添加定制权限的文档，通过访问 https://docs.djangoproject.com/en/1.8/topics/auth/customizing/#custom-permissions 。

打开 http://127.0.0.1:8000/admin/auth/user/add/ 然后创建一个新用户。编辑这个用户然后添加 **Instructors** 组给这个用户如下所示：

django-10-4

用户会继承他们所在组的权限，但是你也可以在管理平台上添加单独的权限给一个指定的用户。当用户的 *is_superuser* 设置为 *True* 的时候会自动拥有所有的权限。

限制使用基于类的视图

我们将要限制使用基于类的视图从而只有那些拥有合适权限的用户才能添加，修改，或者删除 `Course` 对象认证框架包含一个 `permission_required` 装饰器来限制对视图的使用。Django 1.9 将会包含权限 mixins 给基于类的视图（**译者注：到目前为止，Django 版本已经是 1.10.6**）。然而，Django1.8 还没有包含它们。因此，我们将要第三方模块提供的权限 mixins，该第三方模块名为 django-braces（**译者注：。。。。。。下面我是不是可以不用翻译了。。。。。。**）。

使用 django-braces 的 mixins

Django-braces 是一个第三方的模块，它包含一个通用 mixins 的采集给 Django 使用。这些 mixins 提供额外的特性给基于类的视图。你可以看到 django-braces 提供的所有 mixins 列表，通过访问 http://django-braces.readthedocs.org/en/latest/。

使用 pip 命令安装 django-braces：

```
pip install django-braces==1.8.1
```

我们将要使用以下两个 django-braces 提供的 mixinx 来限制视图的使用：

- *LoginRequiredMixin*：复制 `login_required` 装饰器的功能。
- *PermissionRequiredMixin*：准许拥有指定权限的用户使用该视图。请记住，超级用户自动拥有所有权限。

编辑 *courses* 应用的 *views.py* 文件，添加如下导入：

```
from braces.views import LoginRequiredMixin,
```

```
                        PermissionRequiredMixin
```

像下面一样让 `OwnerCourseEditMixin` 继承 `LoginRequiredMixin`：

```
class OwnerCourseEditMixin(OwnerMixin, LoginRequiredMixin):
    model = Course
    fields = ['subject', 'title', 'slug', 'overview']
    success_url = reverse_lazy('manage_course_list')
```

之后，添加一个 `permission_required` 属性给创建，跟新，以及删除视图，如下所示：

```
class CourseCreateView(PermissionRequiredMixin,
                       OwnerCourseEditMixin,
                       CreateView):
    permission_required = 'courses.add_course'

class CourseUpdateView(PermissionRequiredMixin,
                       OwnerCourseEditMixin,
                       UpdateView):
    template_name = 'courses/manage/course/form.html'
    permission_required = 'courses.change_course'

class CourseDeleteView(PermissionRequiredMixin,
                       OwnerCourseMixin,
                       DeleteView):
    template_name = 'courses/manage/course/delete.html'
    success_url = reverse_lazy('manage_course_list')
    permission_required = 'courses.delete_course'
```

`PermissionRequiredMixin` 会在用户使用视图的时候检查该用户是否有指定在 `permission_required` 属性中的权限。我们的视图现在只准许有适当权限的用户使用。
让我们给以上视图创建 URLs。在 *courses* 应用目录中创建新的文件命名为 *urls.py*。添加以下代码：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^mine/$',
        views.ManageCourseListView.as_view(),
        name='manage_course_list'),
    url(r'^create/$',
        views.CourseCreateView.as_view(),
        name='course_create'),
    url(r'^(?P<pk>\d+)/edit/$',
        views.CourseUpdateView.as_view(),
        name='course_edit'),
    url(r'^(?P<pk>\d+)/delete/$',
        views.CourseDeleteView.as_view(),
        name='course_delete'),
]
```

以上的 URL 模式是给列表，创建，编辑以及删除课程试图使用的。编辑 *educa* 项目的主 *urls.py* 文件然后包含 *courses* 应用的 URL 模式，如下所示：

```
urlpatterns = [
    url(r'^accounts/login/$', auth_views.login, name='login'),
    url(r'^accounts/logout/$', auth_views.logout, name='logout'),
    url(r'^admin/', include(admin.site.urls)),
    url(r'^course/', include('courses.urls')),
]
```

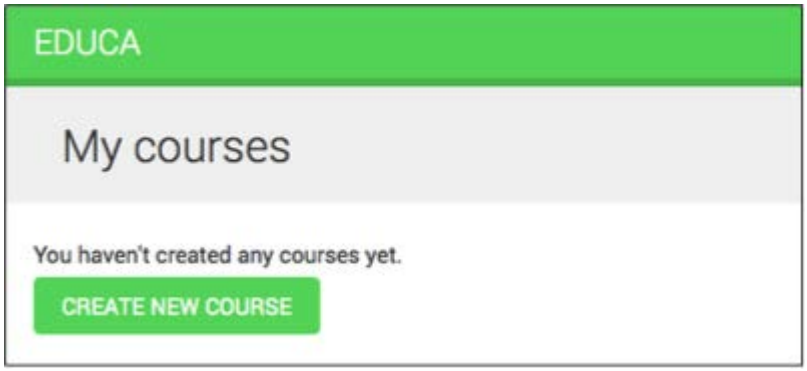
我们需要给这些视图创建模块。在 *courses* 应用中创建以下目录以及文件：

```
courses/
       manage/
           course/
               list.html
               form.html
               delete.html
```

编辑 *courses/manage/course/list.html* 模板并且添加如下代码：

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

{% block title %}My courses{% endblock %}

{% block content %}
     <h1>My courses</h1>
     <div class="module">
       {% for course in object_list %}
         <div class="course-info">
           <h3>{{ course.title }}</h3>
           <p>
             <a href="{% url "course_edit" course.id %}">Edit</a>
             <a href="{% url "course_delete" course.id %}">Delete</a>
           </p>
         </div>
       {% empty %}
         <p>You haven't created any courses yet.</p>
       {% endfor %}
       <p>
         <a href="{% url "course_create" %}" class="button">Create new course</a>
         </p>
    </div>
{% endblock %}
```

这是 `ManageCourseListView` 视图的模板。在这个模板中，我们通过当前用户来排列课程。我们给每个课程都包含了编辑或者删除链接，以及一个创建新课程的链接。
使用命令 `python manage.py runserver` 命令运行开发服务器。在你的浏览器中打开 http://127.0.0.1:8000/accounts/login/?next=/course/mine/ 然后使用 *Instrctors* 组中的一个用户进行登录。登录完成后，你会被重定向到 http://127.0.0.1:8000/course/mine/ 并且你会看到如下页面：

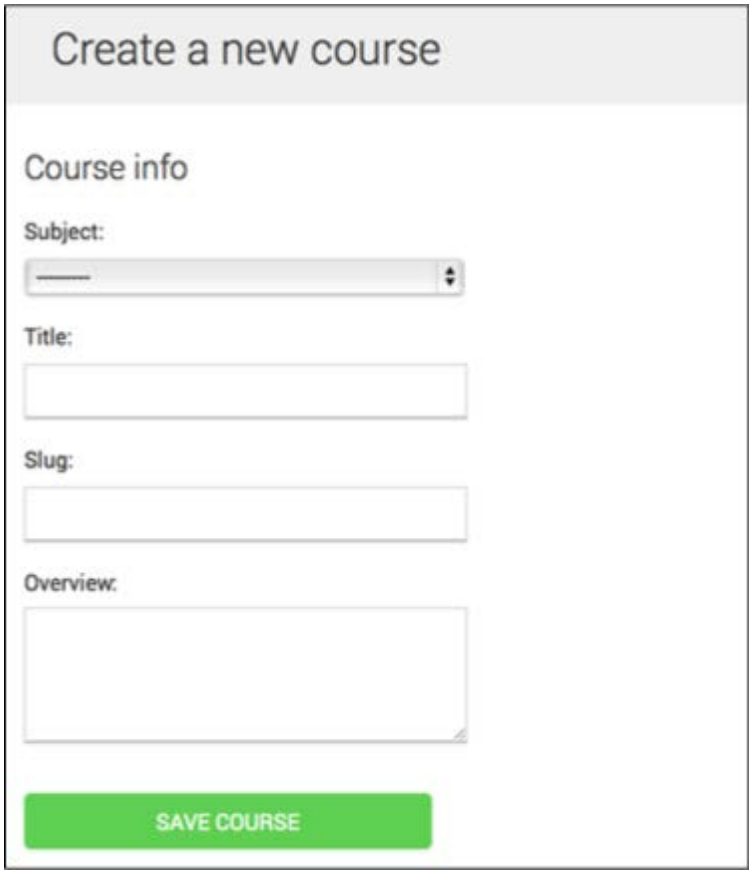


django-10-5

这个页面将会展示所有当前用户创建的课程。
让我们创建一个给创建和更新课程视图使用的模板，该模板用来展示表单。编辑 *courses/manage/course/form.html* 模板并且输入以下代码：

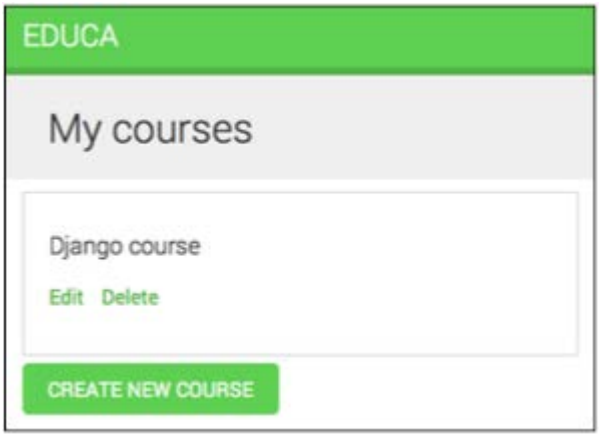
```
{% extends "base.html" %}
{% block title %}
  {% if object %}
    Edit course "{{ object.title }}"
  {% else %}
    Create a new course
  {% endif %}
{% endblock %}
{% block content %}
  <h1>
    {% if object %}
      Edit course "{{ object.title }}"
    {% else %}
      Create a new course
    {% endif %}
  </h1>
  <div class="module">
    <h2>Course info</h2>
    <form action="." method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <p><input type="submit" value="Save course"></p>
    </form>
  </div>
{% endblock %}
```

这个 *form.html* 模板被 `CoursecREATEvIEW` 和 `courseUpdateView` 视图使用。在这个模板中，我们检查是否有个 *object* 变量在上下文环境中。如果 *object* 存在上下文环境中，我们就知道我们正在更新一个存在的课程，并且我们在页面标题中使用它。如果不存在，我们就要创建一个新的 *Course* 对象。
在你的浏览器中打开 http://127.0.0.1:8000/course/mine/ 然后点击 **Create new course** 按钮。你会看到如下页面：



django-10-6

填写好表单内容然后点击 **Save course** 按钮。这个课程将会被保存并且你将会被重定向到课程列表页面。它看上去如下所示：



django-10-7

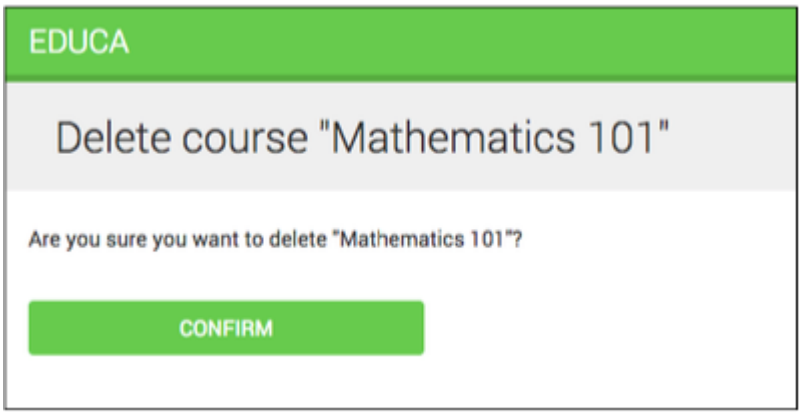
之后，点击你刚才创建的课程的 **Edit** 链接。你将会再次看到表单，但是这一次你将编辑一个已经存在的 *Course* 对象而不是创建新课程。

最后，编辑 *courses/manage/course/delete.html* 模板然后添加以下代码：

```
{% extends "base.html" %}
{% block title %}Delete course{% endblock %}
{% block content %}
  <h1>Delete course "{{ object.title }}"</h1>
  <div class="module">
    <form action="" method="post">
      {% csrf_token %}
      <p>Are you sure you want to delete "{{ object }}"?</p>
      <input type="submit" class"button" value="Confirm">
    </form>
  </div>
```

```
{% endblock %}
```

这个模板是给 `CourseDeleteView` 视图使用的。这个视图从 Django 提供的 `DeleteView` 视图继承而来，`DeleteView` 视图期望用户确认删除一个对象。
打开你的浏览器，点击你的课程的 **Delete** 链接。你会看到如下确认页面：



django-10-8

点击 **CONFIRM** 按钮。这个课程将会被删除并且你再次会被重定向到课程列表页面。
教师们现在可以创建，编辑，以及删除课程。下一步，我们需要提供他们一个内容管理系统来给课程添加模块以及内容。我们将从管理课程模块开始。

使用 formsets

Django 自带一个抽象层用于在同一个页面中使用多个表单。这些表单的组合成为 formsets。formsets 能管理多个 *Form* 或者 *ModelForm* 表单实例。所有的表单都可以一次性提交并且 formset 会照顾到一些事情，例如，表单的初始化数据展示，限制表单能够提交的最大数字，以及所有表单的验证。
formsets 包含一个 `is_valid()`方法来一次性验证所有表单。你还可以提供初始数据给表单以及指定展示任意多的额外的空的表单。
你可以学习到更多关于 formsets，通过访问
https://docs.djangoproject.com/en/1.8/topics/forms/modelforms/#model-formsets 。

管理课程模块

由于课程会被分为可变数量的模块，因此在这里使用 formets 是有意义的。在 *courses* 应用目录下创建一个 *forms.py* 文件，然后添加以下代码：

```
from django import forms
from django.forms.models import inlineformset_factory
from .models import Course, Module

ModuleFormSet = inlineformset_factory(Course,
                                      Module,
                                      fields=['title',
                                              'description'],
                                      extra=2,
                                      can_delete=True)
```

以上就是 `ModuleFormSet` formset。我们使用 Django 提供的 `inlineformset_factory()`函数来构建它。内联 formsets 是在 formsets 之上的一个小抽象，用于方便被关联对象的操作。这个函数允许我们去给关联到一个 *Course* 对象的 *Module* 对象动态的构建一个模型 formset。
我们使用以下参数去构建 formset：

- fields：这个字段将会被 formset 中的每个表单包含。
- extra：允许我们设置在 formset 中显示的空的额外的表单数。

- can_delete：如果你将这个参数设置为 *True*，Django 将会包含一个布尔字段给所有的表单，该布尔字段将会渲染成一个复选框。允许你确定这个对象你真的要进行删除。

编辑 *courses* 应用的 *views.py* 文件并且添加如下代码：

```
from django.shortcuts import redirect, get_object_or_404
from django.views.generic.base import TemplateResponseMixin, View
from .forms import ModuleFormSet

class CourseModuleUpdateView(TemplateResponseMixin, View):
    template_name = 'courses/manage/module/formset.html'
    course = None

    def get_formset(self, data=None):
        return ModuleFormSet(instance=self.course,data=data)

    def dispatch(self, request, pk):
        self.course = get_object_or_404(Course,
                                        id=pk,
                                        owner=request.user)
        return super(CourseModuleUpdateView,
                     self).dispatch(request, pk)

    def get(self, request, *args, **kwargs):
        formset = self.get_formset()
        return self.render_to_response({'course': self.course,
                                        'formset': formset})

    def post(self, request, *args, **kwargs):
        formset = self.get_formset(data=request.POST)
        if formset.is_valid():
            formset.save()
            return redirect('manage_course_list')
        return self.render_to_response({'course': self.course,
                                        'formset': formset})
```

`CourseModuleUpdateView` 视图控制 formset 给一个指定的课程添加，更新，以及删除模块。这个视图继承自以下的 mixins 和视图：

- TemplateResponseMixin：这个 mixins 负责渲染模板以及返回一个 HTTP 响应。它需要一个 template_name 属性，该属性指明要被渲染的模板，并提供 `render_to_ response()`方法来传递上下文并渲染模板。
- View：Django 提供的基础的基于类的视图。

在这个视图中，我们导入以下方法：

- get_formset()：我们定义这个方法去避免重复构建 formset 的代码。我们使用可选数据为给予的 `Course` 对象创建一个 `ModuleFormSet` 对象。
- dispatch()：这个方法由 `View` 类提供。它需要一个 HTTP 请求及其参数并尝试委托一个与使用的 HTTP 方法匹配的小写方法：GET 请求被委派给 `get()`方法和一个 POST 请求到 `post()`。在这种方法中，我们使用 `get_object_or_404()`快捷方式函数获取属于当前用户的给予 id 参数的 Course 对象。我们将这串代码包含在 `dispatch()`方法中是因为我们需要检索所有 GET 和 POST 请求的课程。我们保存该对象到这个视图的 `course` 属性给使它能被别的方法使用。
- get()：给 GET 请求执行。我们构建一个空的 `ModuleFormSet` formset 并且使用 `TemplateResponseMixin` 提供的 `render_to_response()`方法将它与当前的 `Course` 对象一起渲染到模板中。

- post()：给 POST 请求执行。在这个方法中，我们执行以下操作：
  - 1 我们使用提交的数据构建一个 `ModuleFormSet` 实例。
  - 2 我们执行 formset 的 `is_valid()`方法来验证其中的所有表单。
  - 3 如果这个 formset 验证通过，我们通过调用 `save()`方法来保存它。在这点上，任何的修改操作，例如增加，更新或者标记模块用来删除，都会应用到数据库中。之后，我们重定向用户到 `manage_course_list` URL。如果这个 formset 没有通过验证，我们渲染模板展示所有内置的错误信息。

编辑 *courses* 应用的 *urls.py* 文件，添加以下 URL 模式：

```
url(r'^(?P<pk>\d+)/module/$',
    views.CourseModuleUpdateView.as_view(),
    name='course_module_update'),
```

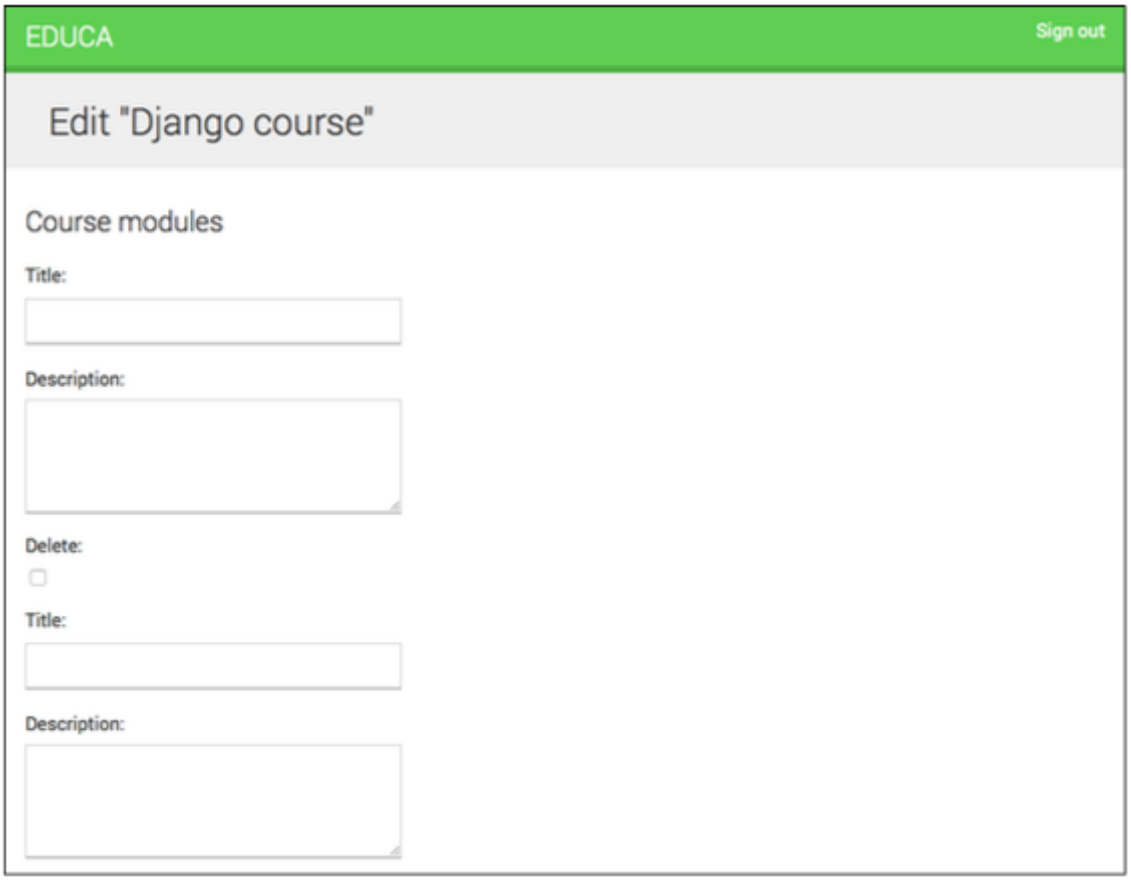
在 *courses/manage/*模板目录中创建一个新的目录命名为 *module*。创建一个 *courses/manage/module/formset.html* 模板并且添加以下代码：

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

{% block title %}
    Edit "{{ course.title }}"
{% endblock %}

{% block content %}
    <h1>Edit "{{ course.title }}"</h1>
    <div class="module">
      <h2>Course modules</h2>
      <form action="" method="post">
        {{ formset }}
        {{ formset.management_form }}
        {% csrf_token %}
        <input type="submit" class="button" value="Save modules">
      </form>
    </div>
{% endblock %}
```

在这个模板中，我们创建一个`<form>`HTML 元素，在其中我们包含我们的 `formset`。我们还通过变量 `{{ formset.management_form }}`包含给 formset 使用的管理表单。这个管理表单包含隐藏的字段去控制保单的初始化，总数，最小值和最大值。如你所见，创建一个 formset 非常容易。
编辑 *courses/manage/course/list.html* 模板并且在课程编辑和删除链接下方添加以下链接给 `course_module_update` 使用：

```
<a href="{% url "course_edit" course.id %}">Edit</a>
<a href="{% url "course_delete" course.id %}">Delete</a>
<a href="{% url "course_module_update" course.id %}">Edit modules</a>
```

我们已经包含了用来编辑课程模板的链接。在你浏览器中打开 http://127.0.0.1:8000/course/mine/ 然后选择一个课程点击对应的 **Edit modules** 链接。你会看到一个如下的 formset：



django-10-9

这个 formset 包含所有在这个课程中存在的 *Module* 对象的表单。在这些表单之后，有两个空的额外的表单会被展示因为我们给 `ModuleFormSet` 设置 `extra=2`。当你保存这个 formset 的时候，Django 将会包含另外两个额外的字段来添加新的模块。

添加内容给课程模块

现在，我们需要一个方法来添加内容给课程模块。我们有四种不同的内容类型：文本，视频，图片以及文件。我们可以考虑创建四个不同的视图去保存内容，给每个模型都对应上一个视图。然而，我们将采取更通用的方法，并创建一个处理创建或更新任何内容模型的对象的视图。
编辑 *courses* 应用的 *views.py* 文件并且添加如下代码：

```
from django.forms.models import modelform_factory
from django.apps import apps
from .models import Module, Content

class ContentCreateUpdateView(TemplateResponseMixin, View):
    module = None
    model = None
    obj = None
    template_name = 'courses/manage/content/form.html'

    def get_model(self, model_name):
        if model_name in ['text', 'video', 'image', 'file']:
            return apps.get_model(app_label='courses',
                                  model_name=model_name)
        return None

    def get_form(self, model, *args, **kwargs):
        Form = modelform_factory(model, exclude=['owner',
```

```
                                        'order',
                                        'created',
                                        'updated'])
        return Form(*args, **kwargs)

    def dispatch(self, request, module_id, model_name, id=None):
        self.module = get_object_or_404(Module,
                                        id=module_id,
                                    course__owner=request.user)
        self.model = self.get_model(model_name)
        if id:
            self.obj = get_object_or_404(self.model,
                                        id=id,
                                        owner=request.user)
        return super(ContentCreateUpdateView,
             self).dispatch(request, module_id, model_name, id)
```

以上是 `ContentCreateUpdateView` 视图的第一部分。这个视图允许我们去创建和更新不同模块的内容。这个视图定义了以下方法：

- get_model()：在这儿，我们会对被给予的模型名是否四种内容模型中的一种：text,video,image 以及 file.之后我们使用 Django 的 `apps` 模块去通过给予的模型名来获取实际的类。如果给予的模型名不是其中的一种，我们返回 `None`。
- get_form()：我们使用表单框架的 `modelform_factory()`函数来构建一个动态的表单。由于我们将要给 *Text*，*Video*，*Image* 以及 *File* 模型构建一个表单，我们使用 `exclude` 参数去指定要从表单中排除的公共字段，并允许自动包含所有其他属性。通过做到这点，我们不必去知道依赖的模型中锁包含的字段。
- dispatch()：它检索以下 URL 参数并且存储相符的模块，模型以及内容对象作为类的属性：
  - module_id：The id for the module that the content is/will be associated with**(译者注：求比较好的翻译)**。
  - model_name：要创建或更新的内容的模型名。
  - id：这是将要更新的对象的 id。在创建新对象的时候它会是 `None`。

添加以下 `get()`和 `post()`方法给 `ContentCreateUpdateView`：

```
def get(self, request, module_id, model_name, id=None):
    form = self.get_form(self.model, instance=self.obj)
    return self.render_to_response({'form': form,
                                    'object': self.obj})

def post(self, request, module_id, model_name, id=None):
    form = self.get_form(self.model,
                         instance=self.obj,
                         data=request.POST,
                         files=request.FILES)
    if form.is_valid():
        obj = form.save(commit=False)
        obj.owner = request.user
        obj.save()
        if not id:
            # new content
            Content.objects.create(module=self.module,
```

```
            return redirect('module_content_list', self.module.id)
        return self.render_to_response({'form': form,
                                        'object': self.obj})
```

以上方法如下所示：

- get()：当收到一个 GET 请求的时候会被执行。我们构建模型表单给 `Text`，`Video`，`Image`,以及 `File` 实例使用当它们被保存的时候。除此以外，我们不会传递实例给创建新的对象，因为 `self.obj` 在没有 id 提供的时候是 `None`。
- post()：当收到一个 POST 请求的时候会被执行。我们构建模型表单会传递所有提交的数据和文件给该表单。之后我们验证该表单。如果这个表单验证通过，我们创建一个新的对象并且在保存该对象到数据库之前分配 `request.user` 作为该对象的拥有者。我们会检查 `id` 参数，如果没有提供 `id`，我们就知道当前用户正在创建一个新的对象而不是更新一个已经存在的对象。如果这是一个新的对象，我们创建一个 `Content` 对象给给予的模块并且关联新的内容给该模块。

编辑 *courses* 应用的 *urls.py* 文件禀帖添加以下 URL 模式：

```
url(r'^module/(?P<module_id>\d+)/content/(?P<model_name>\w+)/create/$',
    views.ContentCreateUpdateView.as_view(),
    name='module_content_create'),
url(r'^module/(?P<module_id>\d+)/content/(?P<model_name>\w+)/(?P<id>\d+)/$',
    views.ContentCreateUpdateView.as_view(),
    name='module_content_update'),
```

以上新的 URL 模式如下：

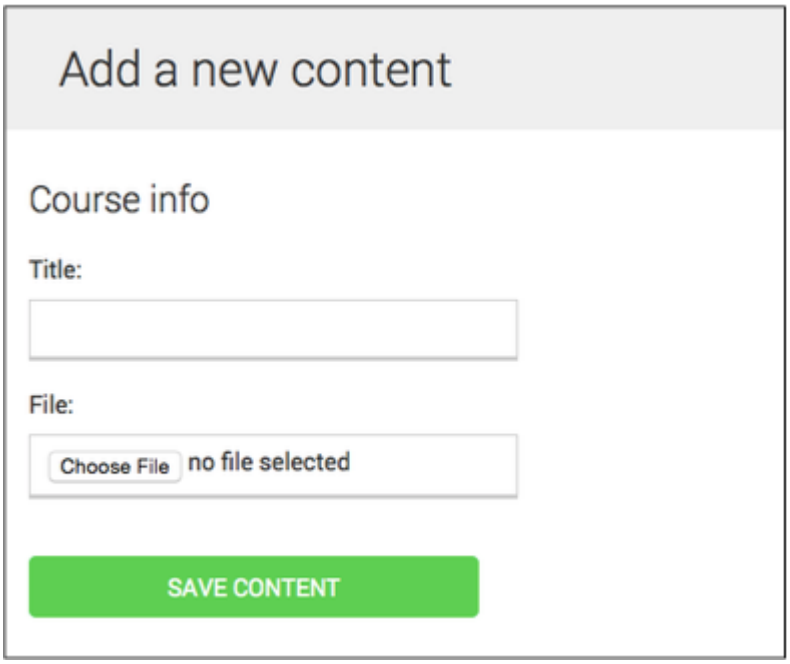
- module_content_create：用来创建新的文本，视频，图片或者文件对象并且给一个模块添加这些对象。它包含 `module_id` 和 `model_name` 参数。前者允许连接新的内容对象给给予的模块。后者指定构建表单使用的内容模型。
- module_content_update：用来更新一个已有的文本，视频，图片或者文件对象。它包含 `module_id` 和 `model_name` 参数，以及一个 `id` 参数来辨明那个需要被更新的内容。

在 *courses/manage/*模板目录下创建新的目录命名为 *content*。创建模板 *courses/manage/content/form.html* 并且添加以下代码：

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

{% block title %}
  {% if object %}
    Edit content "{{ object.title }}"
  {% else %}
    Add a new content
  {% endif %}
{% endblock %}

{% block content %}
  <h1>
    {% if object %}
      Edit content "{{ object.title }}"
    {% else %}
      Add a new content
    {% endif %}
  </h1>
  <div class="module">
    <h2>Course info</h2>
```

```
      <form action="" method="post" enctype="multipart/form-data">
        {{ form.as_p }}
        {% csrf_token %}
        <p><input type="submit" value="Save content"></p>
      </form>
    </div>
  {% endblock %}
```

这个模板是给 `ContentCreateUpdateView` 视图使用的。在这个模板中，我们会检查是否有一个 `object` 变量在上下文环境中。如果 `object` 存在上下文环境中，我们知道我们正在更新一个已经存在的对象。如果没有，我们在创建一个新的对象。
我们在`<form>`HTML 元素中包含 `enctype="multipart/form-data"`，因为这个表单包含一个文件上传用来给 `Field` 和 `Image` 内容模型使用。
运行开发服务器。给存在的课程创建一个模块并且在你的浏览器中打开 http://127.0.0.1:8000/course/module/6/content/image/create/ 。如果有必要，在 ULR 中修改模块 id。你将会看到以下表单用来创建新的 `Image` 对象：



django-10-10

先不要提交表单。如果你想要尝试，它将会是失败的，因为我们还没有定义 `module_content_list` 的 URL。我们一会儿就要去创建它。
我们还需要一个视图去删除内容。编辑 *courses* 应用的 *views.py* 文件，添加以下代码：

```
class ContentDeleteView(View):
    def post(self, request, id):
        content = get_object_or_404(Content,
                            id=id,
                            module__course__owner=request.user)
        module = content.module
        content.item.delete()
        content.delete()
        return redirect('module_content_list', module.id)
```

`ContentDeleteView` 通过给予的 id 检索 `content` 对象，它删除关联的 *Text*，*Video*，*Image* 以及 *File* 对象，并且在最后，它会删除 `Content` 对象并且重定向用户到 `module_content_list`URL 去排列其他模块的内容。
编辑 *courses* 应用的 *urls.py* 文件并且添加以下 URL 模式：

```
url(r'^content/(?P<id>\d+)/delete/$',
    views.ContentDeleteView.as_view(),
    name='module_content_delete'),
```

现在，教师们可以方便的创建，更新以及删除内容。

管理模块和内容

我们已经构建了用来创建，编辑以及删除课程模块和内容的视图。现在，我们需要一个视图去给一个课程显示所有的模块并且给一个指定的模块排列所有的内容。
编辑 *courses* 应用的 *views.py* 文件并且添加以下代码：

```
class ModuleContentListView(TemplateResponseMixin, View):
    template_name = 'courses/manage/module/content_list.html'

    def get(self, request, module_id):
        module = get_object_or_404(Module,
                                   id=module_id,
                                   course__owner=request.user)

        return self.render_to_response({'module': module})
```

以上就是 `ModuleContentListView` 视图。这个视图通过给予的 id 拿到 `Module` 对象该对象是属于当前的用户并且通过给予的模块渲染一个模板。
编辑 *courses* 应用的 *urls.py* 文件，添加以下 URL 模式：

```
url(r'^module/(?P<module_id>\d+)/$',
    views.ModuleContentListView.as_view(),
    name='module_content_list'),
```

在 *templates/courses/manage/module/*目录下创建新的模板命名为 *content_list.html*，添加以下代码：

```
{% extends "base.html" %}
{% block title %}
     Module {{ module.order|add:1 }}: {{ module.title }}
{% endblock %}
{% block content %}
{% with course=module.course %}
  <h1>Course "{{ course.title }}"</h1>
  <div class="contents">
    <h3>Modules</h3>
    <ul id="modules">
      {% for m in course.modules.all %}
        <li data-id="{{ m.id }}" {% if m == module %}
        class="selected"{% endif %}>
          <a href="{% url "module_content_list" m.id %}">
            <span>
              Module <span class="order">{{ m.order|add:1 }}</span>
            </span>
            <br>
            {{ m.title }}
          </a>
```

```
        </li>
      {% empty %}
        <li>No modules yet.</li>
      {% endfor %}
    </ul>
    <p><a href="{% url "course_module_update" course.id %}">Edit modules</a>
    </p>
  </div>
  <div class="module">
    <h2>Module {{ module.order|add:1 }}: {{ module.title }}</h2>
    <h3>Module contents:</h3>
    <div id="module-contents">
      {% for content in module.contents.all %}
        <div data-id="{{ content.id }}">
          {% with item=content.item %}
            <p>{{ item }}</p>
            <a href="#">Edit</a>
            <form action="{% url "module_content_delete" content.id %}" method="post">
              <input type="submit" value="Delete">
              {% csrf_token %}
            </form>
          {% endwith %}
        </div>
      {% empty %}
        <p>This module has no contents yet.</p>
      {% endfor %}
      </div>
       <hr>
       <h3>Add new content:</h3>
       <ul class="content-types">
         <li><a href="{% url "module_content_create" module.id "text" %}">Text</a></li>
         <li><a href="{% url "module_content_create" module.id "image" %}">Image</a></li>
         <li><a href="{% url "module_content_create" module.id "video" %}">Video</a></li>
         <li><a href="{% url "module_content_create" module.id "file" %}">File</a></li>
      </ul>
    </div>
{% endwith %}
{% endblock %}
```

这个模板展示一个课程所有的模块以及被选中的模块的内容。我们迭代课程模块并将它们展示在侧边栏。我们还迭代模块的内容并且通过 `content.item` 去获取关联的 *Text*，*Video*，*Image* 以及 *File* 对象。我们还包含可以创建新的文本，视频，图片以及文件内容的链接。

我们想要知道每个 *item* 对象是哪种类型（文本，视频，图片或者文件）的对象。我们需要模型名用来构建 URL 去编辑对象。除此以外，我们还在模板中展示各式各样的不同的 *item*，基于 *item* 的内容类型。我们可以通过访问对象的 `_meta` 属性来从模型的 `Meta` 类中获取一个对象的模型。尽管如此，Django 不允许访问开头是下划线的变量或者属性在模板中为了编辑检索私有属性或者调用到私有方法。我们可以通过编写一个定制模板过滤器来解决这个问题。

在 *courses* 应用目录下创建以下文件结构：

```
templatetags/
```

```
    __init__.py
    course.py
```

编辑 *course.py* 模块，添加以下代码：

```
from django import template

register = template.Library()

@register.filter
def model_name(obj):
    try:
        return obj._meta.model_name
    except AttributeError:
        return None
```

以上就是 `model_name` 模板过滤器。我们可以在模板中通过 `object|model_name` 应用它来给一个对象获取模型的名字。

编辑 *templates/courses/manage/module/content_list.html* 模板，在`{% extends %}`模板标签下添加以下内容：

```
{% load course %}
```

这样将会加载 *course* 模板标签。之后，将以下内容：

```
<p>{{ item }}</p>
<a href="#">Edit</a>
```

替换成：

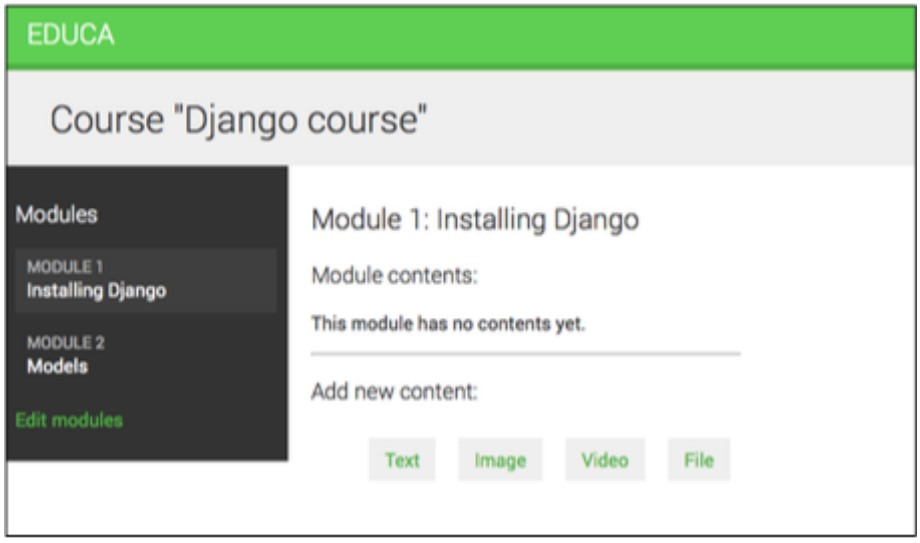
```
<p>{{ item }} ({{ item|model_name }})</p>
<a href="{% url "module_content_update" module.id item|model_name item.id %}">Edit</a>
```

现在，我们在模板中展示 item 模型并且使用模型名曲构建编辑对象的链接。编辑 *courses/manage/course/list.html* 模板，添加一个链接给 `module_content_list` URL，如下所示：

```
<a href="{% url "course_module_update" course.id %}">Edit modules</a>
{% if course.modules.count > 0 %}
 <a href="{% url "module_content_list" course.modules.first.id %}">Manage contents</a>
{% endif %}
```

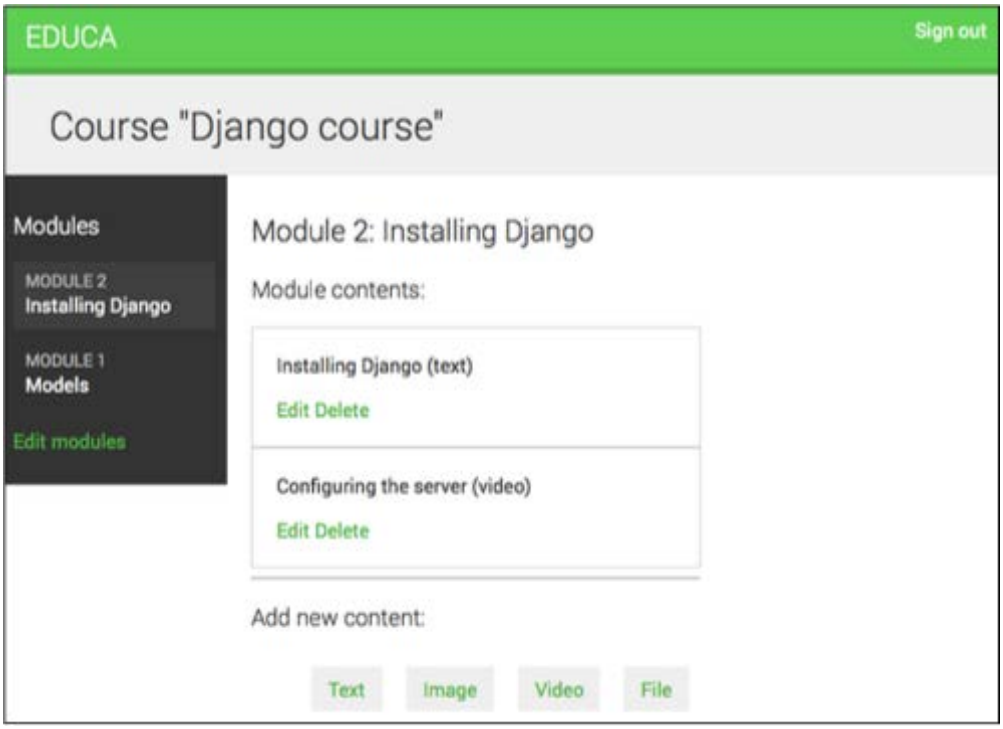
这个新链接允许用户去访问课程的第一个模块的内容，如果有好多内容的话。

打开 http://127.0.0.1:8000/course/mine/ 然后点击一个包含最新模块的课程的 **Manage contents** 链接。你会看到如下页面：



django-10-11

当你点击左方侧边栏的一个模块上，它的内容会在主区域显示。这个模板还包含用来给展示的模块添加一个新的文本，视频，图片或者文件内容。给这个模块添加一堆不同的内容然后看下结果。这个内容将会出现在 **Module contents** 之后，如下所示：



django-10-12

重新整理模块和内容

我们需要提供一个简单的放来去重新排序课程模板和它们的内容。我们将要使用一个 JavaScript drag-n-drop 控件去让我们的用户通过拖拽课程的模块来对课程模块进行重新排序。当用户完成拖拽一个模块，我们将会执行一个异步请求（AJAX）去存储新的模块顺序。

我们需要一个视图，该视图通过编译在 JSON 中的模块的 id 来检索新的对象。编辑 *courses* 应用的 *views.py* 文件，添加以下代码：

```
from braces.views import CsrfExemptMixin, JsonRequestResponseMixin

class ModuleOrderView(CsrfExemptMixin,
                      JsonRequestResponseMixin,
                      View):

    def post(self, request):
```

```
        for id, order in self.request_json.items():
            Module.objects.filter(id=id,
                course__owner=request.user).update(order=order)
        return self.render_json_response({'saved': 'OK'})
```

以上是 `ModuleOrderView`。我们使用以下 django-braces 的 mixins：

- csrfExemptMixin：用来避免在 POST 请求中检查一个 CSRF token。
- JsonRequestResponseMixin：将请求的数据分析为 JSON 并且将相应也序列化成 JSON 并且返回一个 `application/json` 内容类型的 HTTP 响应。

我们可以构建一个类似的视图去排序一个模块的内容。添加以下代码到 *views.py* 中：

```
class ContentOrderView(CsrfExemptMixin,
                       JsonRequestResponseMixin,
                       View):
    def post(self, request):
        for id, order in self.request_json.items():
            Content.objects.filter(id=id,
                       module__course__owner=request.user) \
                       .update(order=order)
        return self.render_json_response({'saved': 'OK'})
```

现在，编辑 *courses* 应用的 *urls.py* 文件，添加以下 URL 模式：

```
url(r'^module/order/$',
       views.ModuleOrderView.as_view(),
       name='module_order'),
url(r'^content/order/$',
       views.ContentOrderView.as_view(),
       name='content_order'),
```

最后，我们在模板中导入 drag-n-drop 功能。我们将要使用 jQuery UI 库来使用这个功能。jQuery UI 基于 jQuery 构建并且它提供了一组界面交互，效果和小部件。我们将要使用它的 *sortable* 元素。首先，我们需要在基础模板中加载 jQuery UI。打开 *courses* 应用下的 *templates/*目录下的 *base.html* 文件，在加载 jQuery 的下方添加 jQuery UI 脚本，如下所示：

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.11.4/jquery-ui.min.js"></script>
```

（译者注：要用以上地址，记得翻墙。。。。。或者自己直接下载）

我们在 jQuery 框架下加载 jQuery UI。现在，编辑 *courses/manage/module/content_list.html* 模板添加以下代码在模板的底部：

```
{% block domready %}
  $('#modules').sortable({
      stop: function(event, ui) {
          modules_order = {};
          $('#modules').children().each(function(){
              // update the order field
              $(this).find('.order').text($(this).index() + 1);
              // associate the module's id with its order
              modules_order[$(this).data('id')] = $(this).index();
              });
```

```
            $.ajax({
             type: 'POST',
             url: '{% url "module_order" %}',
             contentType: 'application/json; charset=utf-8',
             dataType: 'json',
             data: JSON.stringify(modules_order)
             });
    }
});

$('#module-contents').sortable({
    stop: function(event, ui) {
        contents_order = {};
        $('#module-contents').children().each(function(){
            // associate the module's id with its order
            contents_order[$(this).data('id')] = $(this).index();
          });
        $.ajax({
              type: 'POST',
              url: '{% url "content_order" %}',
              contentType: 'application/json; charset=utf-8',
              dataType: 'json',
              data: JSON.stringify(contents_order),
        });
      }
   });
{% endblock %}
```

这个 JavaScripy 代码在`{% block domready %}`区块中，因此它会被包含在我们之前定义在 *base.html* 模板中的 jQuery 的`$(document).ready()`事件中。这将保证我们的 JavaScripy 代码会在页面每次加载的时候都会被执行一次。我们给列在侧边栏的模块定义了一个 *sortable* 元素并且给模块内容列也定义了一个不同的。这两者有着相似的方式。在以上代码中，我们执行以下任务：

- 1 首先，我们给 *modules* HTML 元素定义了一个 *sortable* 元素。请记住，我们使用`#moudles`，因为 jQuery 给选择器使用 CSS 符号。
- 2 我们给 *stop* 事件指定一个函数。这个时间会在用户每次储存一个元素的时候被触发。
- 3 我们创建一个空的 *modules_orders* 目录。给这个目录的键将会是模块的 id，并且给每个模块的值都会被分配次序。
- 4 我们迭代`#module`子元素。我们给每个模块重新计算展示次序并且拿到每个模块的 *data-id* 属性，该属性包含了模块的 id。我们给 *modules_order* 目录添加 id 作为一个键并且模型的新的索引作为值。
- 5 我们运行一个 AJAX POST 请求给 *content_order* URL，在请求中包含 *modules_orders* 的序列化的 JSON 数据。相应的`ModuleOrderView`会负责更新模块的顺序。

sortable 元素排列内容非常类似与上者的方法。回到你的浏览器然后重载页面。现在你将可以点击并且拖动模块和内容，去重新排序它们如下所示：

django-10-13

很好！现在你可以重新排序课程模块和模块内容了。

总结

在这章中，你学习了如何创建一个多功能的内容管理系统。你使用了模型继承以及创建了一个定制模型字段。你还通过基于类的视图和 mixins 工作。你创建了 formsets 以及一个系统去管理不同类型的内容。

在下一章，你将会创建一个学生注册系统。你还将熏染不同类型的内容，并且你还会学习如何使用 Django 的缓存框架。

第十一章 缓存内容

在上一章中，你使用模型继承和一般关系创建了一个灵活的课程内容模型。你也使用基于类的视图，表单集，以及 内容的 AJAX 排序，创建了一个课程管理系统。在这一章中，你将会：

- 创建展示课程信息的公共视图
- 创建一个学生注册系统
- 在 *courses* 中管理学生注册
- 创建多样化的课程内容
- 使用缓存框架缓存内容

我们将会从创建一个课程目录开始，好让学生能够浏览当前的课程以及注册这些课程。

展示课程

对于课程目录，我们需要创建以下的功能：

- 列出所有的可用课程，可以通过可选科目过滤
- 展示一个单独的课程概览

编辑 courses 应用的 `views.py`，添加以下代码：

```
from django.db.models import Count
from .models import Subject

class CourseListView(TemplateResponseMixin, View):
    model = Course
    template_name = 'courses/course/list.html'
    def get(self, request, subject=None):
        subjects = Subject.objects.annotate(
                    total_courses=Count('courses'))
        courses = Course.objects.annotate(
                    total_modules=Count('modules'))
```

```
        if subject:
            subject = get_object_or_404(Subject, slug=subject)
            courses = courses.filter(subject=subject)
        return self.render_to_response({'subjects':subjects,
                                        'subject': subject,
                                        'courses': courses})
```

这是 `CourseListView` 。它继承了 `TemplateResponseMixin` 和 `View` 。在这个视图中，我们实现了下面的功能：

- 1 我们检索所有的课程，包括它们当中的每个课程总数。我们使用 ORM 的 `annotate()` 方法和 `Count()` 聚合方法来实现这一功能。
- 2 我们检索所有的可用课程，包括在每个课程中包含的模块总数。
- 3 如果给了科目的 slug URL 参数，我们就检索对应的课程对象，然后我们将会把查询限制在所给的科目之内。
- 4 我们使用 `TemplateResponseMixin` 提供的 `render_to_response()` 方法来把对象渲染到模板中，然后返回一个 HTTP 响应。

让我们创建一个详情视图来展示单一课程的概览。在 `views.py` 中添加以下代码：

```
from django.views.generic.detail import DetailView

class CourseDetailView(DetailView):
    model = Course
    template_name = 'courses/course/detail.html'
```

这个视图继承了 Django 提供的通用视图 `DetailView` 。我们定义了 `model` 和 `template_name` 属性。Django 的 `DetailView` 期望一个 **主键(pk)** 或者 **slug** URL 参数来检索对应模型的一个单一对象。然后它就会渲染 `template_name` 中的模板，同样也会把上下文中的对象渲染进去。

编辑 `educa` 项目中的主 `urls.py` 文件，添加以下 URL 模式：

```
from courses.views import CourseListView

urlpatterns = [
    # ...
    url(r'^$', CourseListView.as_view(), name='course_list'),
]
```

我们把 `course_list` 的 URL 模式添加进了项目的主 `urls.py` 中，因为我们想要把课程列表展示在 `http://127.0.0.1:8000/` 中，然后 `courses` 应用的所有的其他 URL 都有 `/course/`前缀。

编辑 `courses` 应用的 `urls.py` ，添加以下 URL 模式：

```
url(r'^subject/(?P<subject>[\w-]+)/$',
    views.CourseListView.as_view(),
    name='course_list_subject'),

url(r'^(?P<slug>[\w-]+)/$',
    views.CourseDetailView.as_view(),
    name='course_detail'),
```

我们定义了以下 URL 模式：

- `course_list_subject`：用于展示一个科目的所有课程
- `course_detail`：用于展示一个课程的概览

让我们为 `CourseListView` 和 `CourseDetailView` 创建模板。在 `courses` 应用的 `templates/courses/` 路径下创建以下路径：

- course/
- list.html
- detail.html

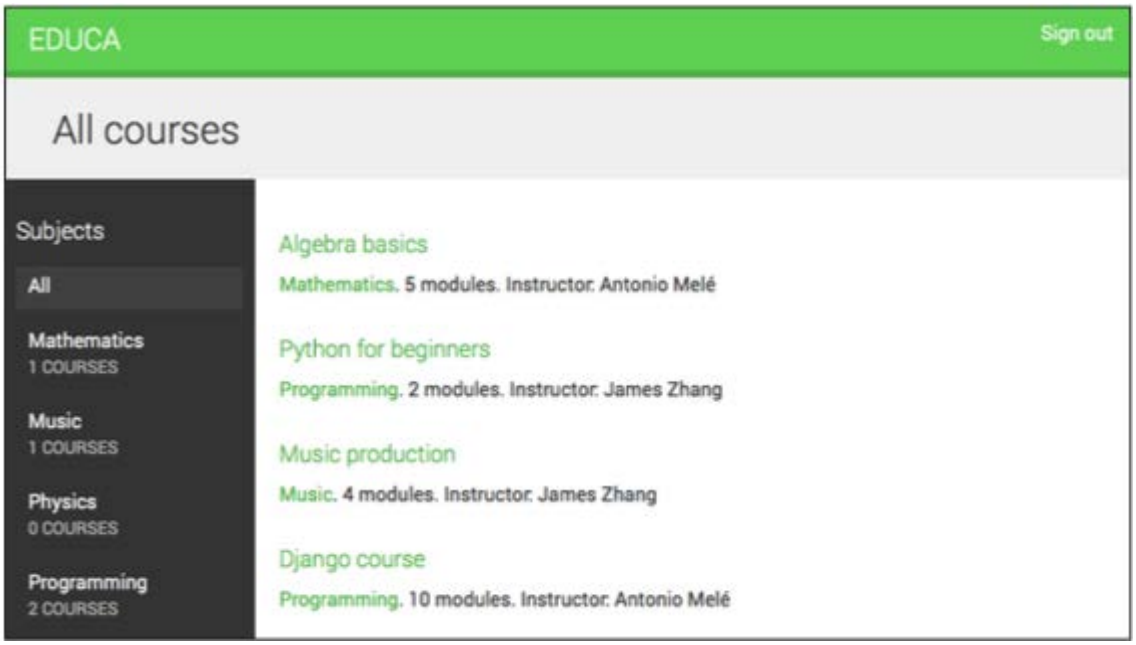
编辑 `courses/course/list.html` 模板，写入以下代码：

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

{% block title %}
    {% if subject %}
        {{ subject.title }} courses
    {% else %}
        All courses
    {% endif %}
{% endblock %}

{% block content %}
<h1>
    {% if subject %}
        {{ subject.title }} courses
    {% else %}
        All courses
    {% endif %}
</h1>
<div class="contents">
    <h3>Subjects</h3>
    <ul id="modules">
        <li {% if not subject %}class="selected"{% endif %}>
            <a href="{% url "course_list" %}">All</a>
        </li>
        {% for s in subjects %}
            <li {% if subject == s %}class="selected"{% endif %}>
                <a href="{% url "course_list_subject" s.slug %}">
                    {{ s.title }}
                    <br><span>{{ s.total_courses }} courses</span>
                </a>
            </li>
        {% endfor %}
    </ul>
</div>
<div class="module">
    {% for course in courses %}
        {% with subject=course.subject %}
            <h3><a href="{% url "course_detail" course.slug %}">{{ course.title }}</a></h3>
            <p>
                <a href="{% url "course_list_subject" subject.slug %}">{{ subject }}</a>.
                {{ course.total_modules }} modules.
                Instructor: {{ course.owner.get_full_name }}
            </p>
        {% endwith %}
```

```
    {% endfor %}
</div>
{% endblock %}
```

这个模板用于展示可用课程列表。我们创建了一个 HTML 列表来展示所有的 `Subject` 对象，然后为它们每一个都创建了一个链接，这个链接链接到 `course_list_subject` 的 URL 。 如果存在当前科目，我们就把 `selected` HTML 类添加到当前科目中高亮显示该科目。我们迭代每个 `Course` 对象，展示模块的总数和教师的名字。

使用 `python manage.py runserver` 打开代发服务器，访问 http://127.0.0.1:8000/ 。你看到的应该是像下面这个样子：



django-11-1

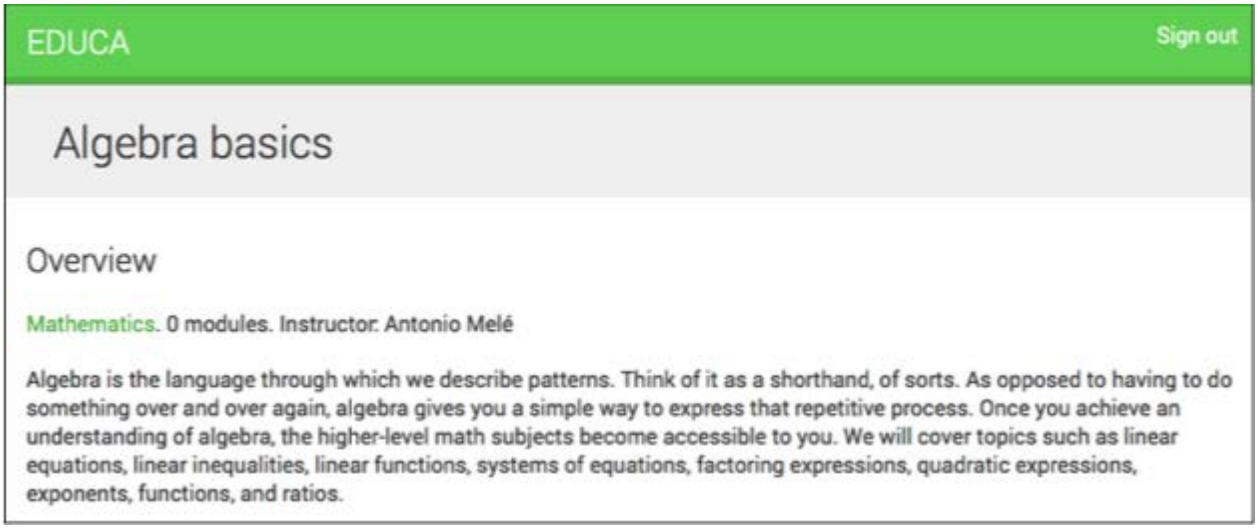
左边的侧边栏包含了所有的科目，包括每个科目的课程总数。你可以点击任意一个科目来筛选展示的课程。

编辑 `courses/course/detail.html` 模板，添加以下代码：

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

{% block title %}
    {{ object.title }}
{% endblock %}

{% block content %}
    {% with subject=course.subject %}
        <h1>
            {{ object.title }}
        </h1>
        <div class="module">
            <h2>Overview</h2>
            <p>
                <a href="{% url "course_list_subject" subject.slug %}">{{ subject.title }}</a>.
                {{ course.modules.count }} modules.
                Instructor: {{ course.owner.get_full_name }}
            </p>
            {{ object.overview|linebreaks }}
```

```
        </div>
    {% endwith %}
{% endblock %}
```

在这个模板中，我们展示了每个单一课程的概览和详情。访问 http://127.0.0.1:8000/ ，点击任意一个课程。你就应该看到有下面结构的页面了：



django-11-2

我们已经创建了一个展示课程的公共区域了。下面，我们需要让用户可以注册为学生以及注册他们的课程。

添加学生注册

使用下面的命令创键一个新的应用：

```
python manage.py startapp students
```

编辑 `educa` 项目的 `settings.py` ，把 `students` 添加进 `INSTALLED_APPS` 设置中：

```
INSTALLED_APPS = (
    # ...
    'students',
)
```

创建一个学生注册视图

编辑 `students` 应用的 `views.py` ，写入下下面的代码：

```
from django.core.urlresolvers import reverse_lazy
from django.views.generic.edit import CreateView
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import authenticate, login

class StudentRegistrationView(CreateView):
    template_name = 'students/student/registration.html'
    form_class = UserCreationForm
    success_url = reverse_lazy('student_course_list')

    def form_valid(self, form):
        result = super(StudentRegistrationView,
```

```
                    self).form_valid(form)
        cd = form.cleaned_data
        user = authenticate(username=cd['username'],
                            password=cd['password1'])
        login(self.request, user)
        return result
```

这个视图让学生可以注册进我们的网站里。我们使用了可以提供创建模型对象功能的通用视图 `CreateView` 。这个视图要求以下属性：

- `template_name`：渲染这个视图的模板路径。
- `form_class`：用于创建对象的表单，我们使用 Django 的 `UserCreationForm` 作为注册表单来创建 `User` 对象。
- `success_url`：当表单成功提交时要将用户重定向到的 URL 。我们逆序了 `student_course_list` URL，我们稍候将会将建它来展示学生已报名的课程。

当合法的表单数据被提交时 `form_valid()` 方法就会执行。它必须返回一个 HTTP 响应。我们覆写了这个方法来让用户在成功注册之后登录。

在 students 应用路径下创建一个新的文件，命名为 `urls.py` ，添加以下代码：

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^register/$',
          views.StudentRegistrationView.as_view(),
          name='student_registration'),
]
```

编辑 `educa` 的主 `urls.py` ，然后把 `students` 应用的 URLs 引入进去：

```
url(r'^students/', include('students.urls')),
```

在 `students` 应用内创建如下的文件结构:

```
templates/
    students/
        student/
            registration.html
```

编辑 `students/student/registration.html` 模板，然后添加以下代码：

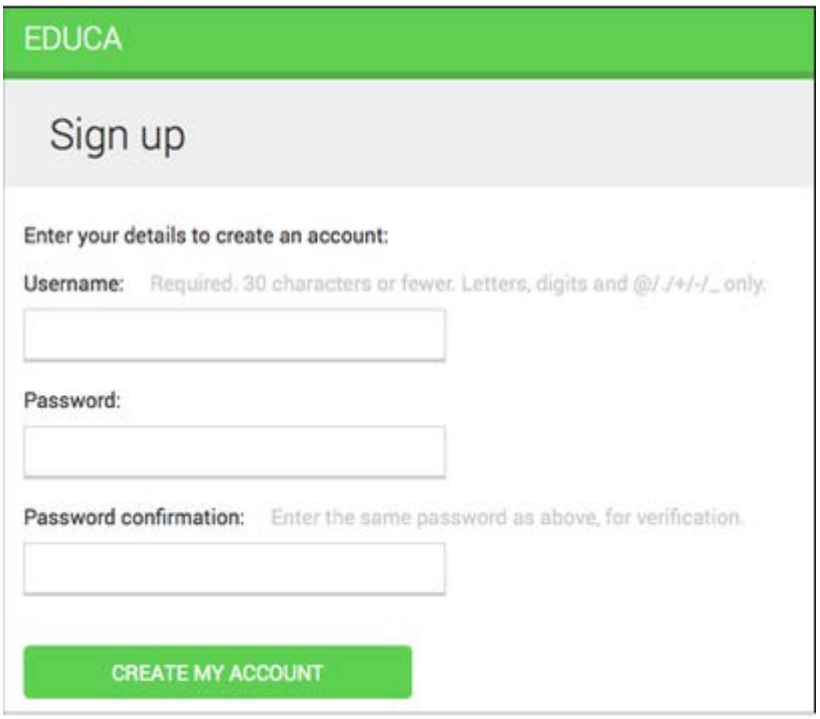
```
{% extends "base.html" %}

{% block title %}
    Sign up
{% endblock %}

{% block content %}
    <h1>
        Sign up
    </h1>
    <div class="module">
        <p>Enter your details to create an account:</p>
```

```
        <form action="" method="post">
            {{ form.as_p }}
            {% csrf_token %}
            <p><input type="submit" value="Create my account"></p>
        </form>
    </div>
{% endblock %}
```

最后编辑 `educa` 的设置文件，添加以下代码：

```
from django.core.urlresolvers import reverse_lazy
LOGIN_REDIRECT_URL = reverse_lazy('student_course_list')
```

这个是由 `auth` 模型用来给用户在成功的登录之后重定向的设置，如果请求中没有 next 参数的话。
打开开发服务器，访问 http://127.0.0.1:8000/students/register/ ，你可以看到像这样的注册表单：



django-11-3

报名

在用户创建一个账号之后，他们应该就可以在 `courses` 中报名了。为了保存报名表，我们需要在 `Course` 和 `User` 模型之间创建一个多对多关系。编辑 `courses` 应用的 `models.py` 然后把下面的字段添加进 `Course` 模型中：

```
students = models.ManyToManyField(User,
                                  related_name='courses_joined',
                                  blank=True)
```

在 shell 中执行下面的命令来创建迁移：

```
python manage.py makemigrations
```

你可以看到类似下面的输出：

```
Migrations for 'courses':
    0004_course_students.py:
```

```
    - Add field students to course
```

接下来执行下面的命令来应用迁移：

```
python manage.py migrate
```

你可以看到以下输出：

```
Operations to perform:
    Apply all migrations: courses
Running migrations:
    Rendering model states... DONE
    Applying courses.0004_course_students... OK
```

我们现在就可以把学生和他们报名的课程相关联起来了。
让我们创建学生报名课程的功能吧。
在 `students` 应用内创建一个新的文件，命名为 `forms.py`，添加以下代码：

```
from django import forms
from courses.models import Course

class CourseEnrollForm(forms.Form):
    course = forms.ModelChoiceField(queryset=Course.objects.all(),
                                    widget=forms.HiddenInput)
```

我们将会把这张表用于学生报名。`course` 字段是学生报名的课程。所以，它是一个 `ModelChoiceField`。我们使用 `HiddenInput` 控件，因为我们不打算把这个字段展示给用户。我们将会在 `CourseDetailView` 视图中使用这个表单来展示一个报名按钮。
编辑 `students` 应用的 `views.py`，添加以下代码：

```
from django.views.generic.edit import FormView
from braces.views import LoginRequiredMixin
from .forms import CourseEnrollForm

class StudentEnrollCourseView(LoginRequiredMixin, FormView):
    course = None
    form_class = CourseEnrollForm

    def form_valid(self, form):
        self.course = form.cleaned_data['course']
        self.course.students.add(self.request.user)
        return super(StudentEnrollCourseView,
                     self).form_valid(form)

    def get_success_url(self):
        return reverse_lazy('student_course_detail',
                            args=[self.course.id])
```

这就是 `StudentEnrollCourseView`。它负责学生在 `courses` 中报名。新的视图继承了 `LoginRequiredMixin`，所以只有登录了的用户才可以访问到这个视图。我们把 `CourseEnrollForm` 表单用在了 `form_class` 属性上，同时我们也定义了一个 `course` 属性来储存所给的 `Course` 对象。当表单合法时，我们把当前用户添加到课程中已报名学生中去。

如果表单提交成功，`get_success_url` 方法就会返回用户将会被重定向到的 URL 。这个方法相当于 `success_url` 属性。我们反序 `student_course_detail` URL ,我们稍候将会创建它来展示课程中的学生。
编辑 `students` 应用的 `urls.py` ， 添加以下 URL 模式：

```
url(r'^enroll-course/$',
    views.StudentEnrollCourseView.as_view(),
    name='student_enroll_course'),
```

让我们把报名按钮表添加进课程概览页。编辑 `course` 应用的 `views.py` ， 然后修改 `CourseDetailView` 让它看起来像这样：

```
from students.forms import CourseEnrollForm

class CourseDetailView(DetailView):
    model = Course
    template_name = 'courses/course/detail.html'

    def get_context_data(self, **kwargs):
        context = super(CourseDetailView,
                        self).get_context_data(**kwargs)
        context['enroll_form'] = CourseEnrollForm(
                        initial={'course':self.object})
        return context
```

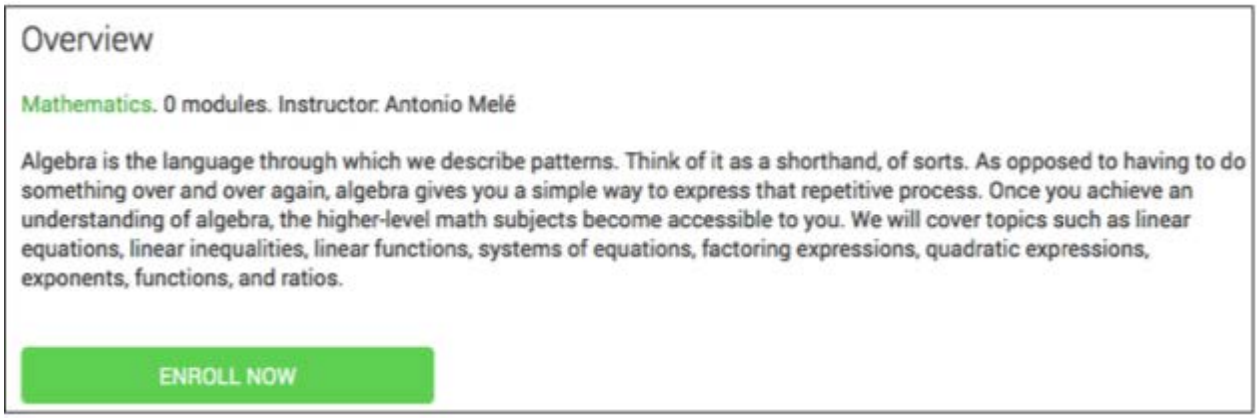
我们使用 `get_context_data()` 方法来在渲染进模板中的上下文里引入报名表。我们初始化带有当前 `Course` 对象的表单的隐藏 course 字段，这样它就可以被直接提交了。
编辑 `courses/course/detail.html` 模板，然后找到下面的这一行：

```
{{ object.overview|linebreaks }}
```

起始行应该被替换为下面的这几行：

```
{{ object.overview|linebreaks }}
{% if request.user.is_authenticated %}
    <form action="{% url "student_enroll_course" %}" method="post">
        {{ enroll_form }}
        {% csrf_token %}
        <input type="submit" class="button" value="Enroll now">
    </form>
{% else %}
    <a href="{% url "student_registration" %}" class="button">
        Register to enroll
    </a>
{% endif %}
```

这个按钮就是用于报名的。如果用户是被认证过的，我们就展示包含了隐藏表单字段的报名按钮，这个表单指向了 `student_enroll_course` URL。如果用户没有被认证，我们将会展示一个注册链接。
确保已经打开了开发服务器，访问 http://127.0.0.1:8000/ ，然后点击一个课程。如果你登录了，你就可以在底部看到 **ENROLL NOW** 按钮，就像这样：



django-11-4

如果你没有登录，你就会看到一个 **Register to enroll** 的按钮。

获取课程内容

我们需要一个视图来展示学生已经报名的课程，和一个获取当前课程内容的视图。编辑 `students` 应用的 `views.py` ，添加以下代码：

```
from django.views.generic.list import ListView
from courses.models import Course

class StudentCourseListView(LoginRequiredMixin, ListView):
    model = Course
    template_name = 'students/course/list.html'

    def get_queryset(self):
        qs = super(StudentCourseListView, self).get_queryset()
        return qs.filter(students__in=[self.request.user])
```

这个是用于列出学生已经报名课程的视图。它继承 `LoginRequiredMixin` 来确保只有登录的用户才可以连接到这个视图。同时它也继承了通用视图 `ListView` 来展示 `Course` 对象列表。我们覆写了 `get_queryset()` 方法来检索用户已经报名的课程。我们通过学生的 `ManyToManyField` 字段来筛选查询集以达到这个目的。把下面的代码添加进 `views.py` 文件中：

```
from django.views.generic.detail import DetailView

class StudentCourseDetailView(DetailView):
    model = Course
    template_name = 'students/course/detail.html'

    def get_queryset(self):
        qs = super(StudentCourseDetailView, self).get_queryset()
        return qs.filter(students__in=[self.request.user])

    def get_context_data(self, **kwargs):
        context = super(StudentCourseDetailView,
                        self).get_context_data(**kwargs)
        # get course object
        course = self.get_object()
        if 'module_id' in self.kwargs:
            # get current module
```

```
            context['module'] = course.modules.get(
                            id=self.kwargs['module_id'])
        else:
            # get first module
            context['module'] = course.modules.all()[0]
        return context
```

这是 `StudentCourseDetailView` 。我们覆写了 `get_queryset` 方法把查询集限制在用户报名的课程之内。我们同样也覆写了 `get_context_data()` 方法来把课程的一个模块赋值在上下文内，如果给了 `model_id` URL 参数的话。否则，我们就赋值课程的第一个模块。这样，学生就可以在课程之内浏览各个模块了。
编辑 `students` 应用的 `urls.py` ，添加以下 URL 模式：

```
url(r'^courses/$',
    views.StudentCourseListView.as_view(),
    name='student_course_list'),

url(r'^course/(?P<pk>\d+)/$',
    views.StudentCourseDetailView.as_view(),
    name='student_course_detail'),

url(r'^course/(?P<pk>\d+)/(?P<module_id>\d+)/$',
    views.StudentCourseDetailView.as_view(),
    name='student_course_detail_module'),
```

在 `students` 应用的 `templates/students/` 路径下创建以下文件结构：

```
course/
    detail.html
    list.html
```

编辑 `students/course/list.html` 模板，然后添加下列代码：

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

{% block title %}My courses{% endblock %}

{% block content %}
    <h1>My courses</h1>

    <div class="module">
        {% for course in object_list %}
            <div class="course-info">
                <h3>{{ course.title }}</h3>
                <p><a href="{% url "student_course_detail" course.id %}">Access contents</a></p>
            </div>
        {% empty %}
            <p>
                You are not enrolled in any courses yet.
                <a href="{% url "course_list" %}">Browse courses</a>to enroll in a course.
            </p>
        {% endfor %}
```

```
    </div>
{% endblock %}
```

这个模板展示了用户报名的课程。编辑 `students/course/detail.html` 模板，添加以下代码：

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

{% block title %}
    {{ object.title }}
{% endblock %}

{% block content %}
    <h1>
        {{ module.title }}
    </h1>
    <div class="contents">
        <h3>Modules</h3>
        <ul id="modules">
        {% for m in object.modules.all %}
            <li data-id="{{ m.id }}" {% if m == module %}
class="selected"{% endif %}>
                <a href="{% url "student_course_detail_module" object.id m.id %}">
                    <span>
                        Module <span class="order">{{ m.order|add:1 }}</span>
                    </span>
                    <br>
                    {{ m.title }}
                </a>
            </li>
        {% empty %}
            <li>No modules yet.</li>
        {% endfor %}
        </ul>
    </div>
    <div class="module">
        {% for content in module.contents.all %}
            {% with item=content.item %}
                <h2>{{ item.title }}</h2>
                {{ item.render }}
            {% endwith %}
        {% endfor %}
    </div>
{% endblock %}
```

这个模板用于报名了的学生连接到课程内容。首先我们创建了一个包含所有课程模块的 HTML 列表且高亮当前模块。然后我们迭代当前的模块内容，之后使用 `{{ item.render }}` 来连接展示内容。接下来我们将会在内容模型中添加 `render()` 方法。这个方法将会负责精准的展示内容。

渲染不同类型的内容

我们需要提供一个方法来渲染不同类型的内容。编辑 `course` 应用的 `models.py`，把 `render()` 方法添加进 `ItemBase` 模型中：

```
from django.template.loader import render_to_string
from django.utils.safestring import mark_safe

class ItemBase(models.Model):
    # ...
    def render(self):
        return render_to_string('courses/content/{}.html'.format(
                        self._meta.model_name), {'item': self})
```

这个方法使用了 `render_to_string()` 方法来渲染模板以及返回一个作为字符串的渲染内容。每种内容都使用以内容模型命名的模板渲染。我们使用 `self._meta.model_name` 来为 `la` 创建合适的模板名。 `render()` 方法提供了一个渲染不同页面的通用接口。
在 `courses` 应用的 `templates/courses/` 路径下创建如下文件结构：

```
content/
    text.html
    file.html
    image.html
    video.html
```

编辑 `courses/content/text.html` 模板，写入以下代码：

```
{{ item.content|linebreaks|safe }}
```

编辑 `courses/content/file.html` 模板，写入以下代码：

```
<p><a href="{{ item.file.url }}" class="button">Download file</a></p>
```

编辑 `courses/content/image.html` 模板，写入以下代码：

```
<p>![]({{ item.file.url }})</p>
```

为了使上传带有 `ImageField` 和 `FielField` 的文件工作，我们需要配置我们的项目以使用开发服务器提供媒体文件服务。编辑你的项目中的 `settings.py`，添加以下代码：

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
```

记住 `MEDIA_URL` 是服务上传文件的基本 URL 路径， `MEDIA_ROOT` 是放置文件的本地路径。
编辑你的项目的主 `urls.py`，添加以下 imports：

```
from django.conf import settings
from django.conf.urls.static import static
```

然后，把下面这几行写入文件的结尾：

```
urlpatterns += static(settings.MEDIA_URL,
               document_root=settings.MEDIA_ROOT)
```

你的项目现在已经准备好使用开发服务器上传和服务文件了。记住开发服务器不能用于生产环境中。我们将会在下一章中学习如何配置生产环境。

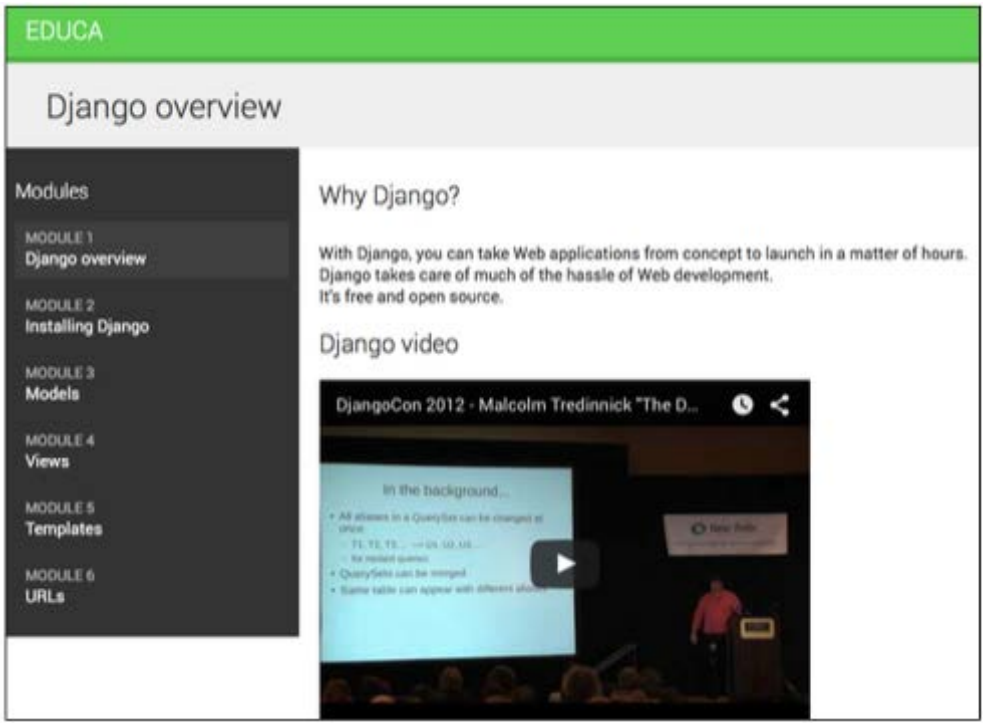
我们也需要创建一个模板来渲染 `Video` 对象。我们将会使用 django-embed-video 来嵌入视频内容。Django-embed-video 是一个第三方 Django 应用，它使你可以通过提供一个视频的公共 URL 来在模板中嵌入视频，类似来自 YouTube 或者 Vimeo 的资源。
使用下面的命令来安装这个包：

```
pip isntall django-embed-video==1.0.0
```

然后编辑项目的 `settings.py` 然后添加 `embed_video` 到 `INSTALLED_APPS` 设置 中。你可以在这里找到 django-embed-video 的文档：
http://django-embed-video.readthedocs.org/en/v1.0.0/ 。
编辑 `courses/content/video.html` 模板，写入以下代码：

```
{% load embed_video_tags %}
{% video item.url 'small' %}
```

现在运行开发服务器，访问 http://127.0.0.1:8000/course/mine/ 。用属于教师组或者超级管理员的用户访问站点，然后添加一些内容到一个课程中。为了引入视频内容，你也可以复制任何一个 YouTube 视频 URL ，比如 ：https://www.youtube.com/watch?n=bgV39DlmZ2U ,然后把它引入到表单的 `url` 字段中。在添加内容到课程中之后，访问 http://127.0.0.1:8000/ ， 点击课程然后点击 **ENROLL NOW** 按钮。你就可以在课程中报名了，然后被重定向到 `student_course_detail` URL 。下面这张图片展示了一个课程内容样本：



django-11-5

真棒！你已经创建了一个渲染课程的通用接口了，它们中的每一个都会被用特定的方式渲染。

使用缓存框架

你的应用的 HTTP 请求通常是数据库链接，数据处理，和模板渲染的。就处理数据而言，它的开销可比服务一个静态网站大多了。
请求开销在你的网站有越来越多的流量时是有意义的。这也使得缓存变得很有必要。通过缓存 HTTP 请求中 的查询结果，计算结果，或者是渲染上下文，你将会避免在接下来的请求中巨大的开销。这使得服务端的响应时间和处理时间变短。
Django 配备有一个健硕的缓存系统，这使得你可以使用不同级别的颗粒度来缓存数据。你可以缓存单一的查询，一个特定的输出视图，部分渲染的模板上下文，或者整个网站。缓存系统中的内容会在默认时间内被储存。你可以指定缓存数据过期的时间。

这是当你的站点收到一个 HTTP 请求时将会通常使用的缓存框架的方法：

```
1. 试着在缓存中寻找缓存数据
2. 如果找到了，就返回缓存数据
3. 如果没有找到，就执行下面的步骤：
    1. 执行查询或者处理请求来获得数据
    2. 在缓存中保存生成的数据
    3. 返回数据
```

你可以在这里找到更多关于 Django 缓存系统的细节信息：
https://docs.djangoproject.com/en/1.8/topics/cache/ 。

激活缓存后端

Django 配备有几个缓存后端，他们是：

- `backends.memcached.MemcachedCache` 或 `backends.memcached.PyLibMCCache`：一个内存缓存后端。内存缓存是一个快速、高效的基于内存的缓存服务器。后端的使用取决于你选择的 Python 绑定（bindings）。
- `backends.db.DatabaseCache`： 使用数据库作为缓存系统。
- `backends.filebased.FileBasedCache`：使用文件储存系统。把每个缓存值序列化和储存为单一的文件。
- `backends.locmem.LocMemCache`：本地内存缓存后端。这是默认的缓存后端
- `backends.dummy.DummyCache`：一个用于开发的虚拟缓存后端。它实现了缓存交互界面而不用真正的缓存任何东西。缓存是独立进程且是线程安全的

对于可选的实现，使用内存的缓存后端吧，比如 `Memcached` 后端。

安装 Memcached

我们将会使用 Memcached 缓存后端。内存缓存运行在内存中，它在 RAM 中分配了指定的数量。当分配的 RAM 满了时，Memcahed 就会移除最老的数据来保存新的数据。
在这个网址下载 Memcached： http://memcached.org/downloads 。如果你使用 Linux，你可以使用下面的命令安装：

```
./configure && make && make test && sudo make install
```

如果你在使用 Mac OS X，你可以使用命令 `brew install Memcached` 通过 Homebrew 包管理器来安装 Memcached 。你可以在这里下载 Homebrew `http://brew.sh`
如果你正在使用 Windwos ，你可以在这里找到一个 Windows 的 Memcached 二进制版本：
http://code.jellycan.com/memcached/ 。
在安装 Memcached 之后，打开 shell ，使用下面的命令运行它：

```
memcached -l 127.0.0.1:11211
```

Memcached 将会默认地在 11211 运行。当然，你也可以通过 `-l` 选项指定一个特定的主机和端口。
你可以在这里找到更多关于 Memcached 的信息：http://memcached.org。
在安装 Memcached 之后，你需要安装它的 Python 绑定（bindings）。使用下面的命令安装：】

```
python install python3-memcached==1.51
```

缓存设置

Django 提供了如下的缓存设置：

- `CACHES`：一个包含所有可用的项目缓存。
- `CACHE_MIDDLEWARE_ALIAS`：用于储存的缓存别名。
- `CACHE_MIDDLEWARE_KEY_PREFIX`：缓存键的前缀。设置一个缓存前缀来避免键的冲突，如果你在几个站点中分享相同的缓存的话。

- CACHE_MIDDLEWARE_SECONDS：默认的缓存页面秒数

项目的缓存系统可以使用 CACHES 设置来配置。这个设置是一个字典，让你可以指定多个缓存的配置。每个 CACHES 字典中的缓存可以指定下列数据：

- BACKEND：使用的缓存后端。
- KEY_FUNCTION：包含一个指向回调函数的点路径的字符，这个函数以 prefix(前缀)、verision(版本)、和 key (键) 作为参数并返回最终缓存键（cache key）。
- KEY_PREFIX：一个用于所有缓存键的字符串，避免冲突。
- LOCATION：缓存的位置。基于你的缓存后端，这可能是一个路径、一个主机和端口，或者是内存中后端的名字。
- OPTIONS：任何额外的传递向缓存后端的参数。
- TIMEOUT：默认的超时时间，以秒为单位，用于储存缓存键。默认设置是 300 秒，也就是五分钟。如果把它设置为 None，缓存键将不会过期。
- VERSION：默认的缓存键的版本。对于缓存版本是很有用的。

把 memcached 添加进你的项目

让我们为我们的项目配置缓存。编辑 educa 项目的 settings.py 文件，添加以下代码：

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.
MemcachedCache',
    'LOCATION': '127.0.0.1:11211',
    }
}
```

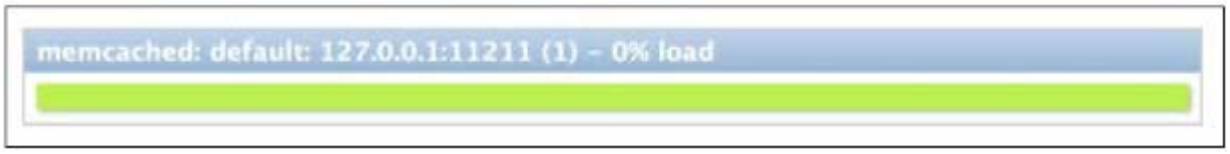
我们正在使用 MemcachedCache 后端。我们使用 address:port 标记指定了它的位置。如果你有多个 memcached 实例，你可以在 LOCATION 中使用列表。

监控缓存

这里有一个第三方包叫做 django-memcached-status ，它可以在管理站点展示你的项目的 memcached 实例的统计数据。为了兼容 Python3（**译者夜夜月注：python3 大法好。**），从下面的分支中安装它：

```
pip install git+git://github.com/zenx/django-memcached-status.git
```

编辑 settings.py，然后把 memcached_status 添加进 INSTALLED_APPS 设置中。确保 memcached 正在运行，在另外一个 shell 中打开开发服务器，然后访问 http://127.0.0.1:8000/adim/，使用超级用户登录进管理站点，你就可以看到如下的区域：



django-11-6

这张图片展示了缓存使用。绿色代表了空闲的缓存，红色的表示使用了的空间。如果你点击方框的标题，它展示了你的 memcached 实例的统计详情。

我们已经为项目安装好了 memcached 并且可以监控它。让我们开始缓存数据吧！

缓存级别

Django 提供了以下几个级别按照颗粒度上升的缓存排列：

- Low-level cache API：提供了最高颗粒度。允许你缓存具体的查询或计算结果。
- Per-view cache：提供单一视图的缓存。

- `Template cache`：允许你缓存模板片段。
- `Per-site cache`：最高级的缓存。它缓存你的整个网站。

在你执行缓存之请仔细考虑下缓存策略。首先要考虑那些不是单一用户为基础的查询和计算开销

使用 low-level cache API （低级缓存 API）

低级缓存 API 让你可以缓存任意颗粒度的对象。它位于 `django.core.cache` 。你可以像这样导入它：

```
from django.core.cache import cache
```

这使用的是默认的缓存。它相当于 `caches['default']` 。通过它的别名来连接一个特定的缓存也是可能的：

```
from django.core.cache import caches
my_cache = caches['alias']
```

让我们看看缓存 API 是如何工作的。使用命令 `python manage.py shell` 打开 shell 然后执行下面的代码：

```
>>> from django.core.cache import cache
>>> cache.set('musician', 'Django Reinhardt', 20)
```

我们连接的是默认的缓存后端，使用 `set{key,value, timeout}` 来保存一个名为 `musician` 的键和它的为字符串 `Django Reinhardt` 的值 20 秒钟。如果我们不指定过期时间，Django 会使在 `CACHES` 设置中缓存后端的默认过期时间。现在执行下面的代码：

```
>>> cache.get('musician')
'Django Reinhardt'
```

我们在缓存中检索键。等待 20 秒然后指定相同的代码：

```
>>> cache.get('musician')
None
```

`musician` 缓存键已经过期了，`get()` 方法返回了 None 因为键已经不在缓存中了。

在缓存键中要避免储存 None 值，因为这样你就无法区分缓存值和缓存过期了

让我们缓存一个查询集：

```
>>> from courses.models import Subject
>>> subjects = Subject.objects.all()
>>> cache.set('all_subjects', subjects)
```

我们执行了一个在 `Subject` 模型上的查询集，然后把返回的对象储存在 `all_subjects` 键中。让我们检索一下缓存数据:

```
>>> cache.get('all_subjects')
[<Subject: Mathematics>, <Subject: Music>, <Subject: Physics>,
<Subject: Programming>]
```

我们将会在视图中缓存一些查询集。编辑 `courses` 应用的 `views.py` ， 添加以下 导入：

```
from django.core.cache import cache
```

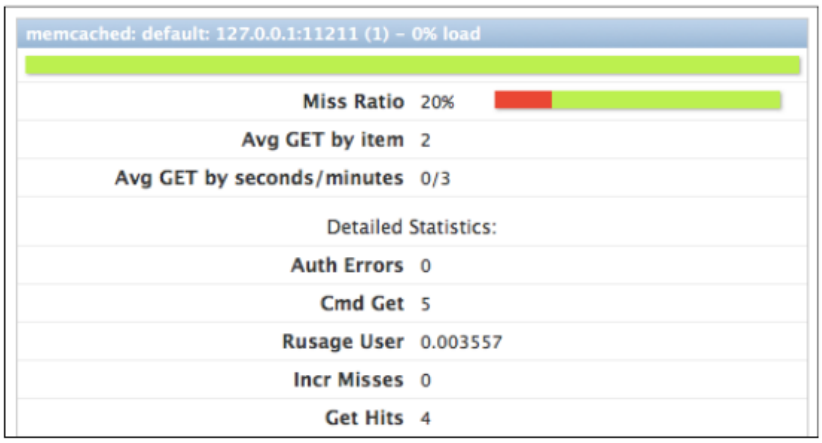
在 `CourseListView` 的 `get()` 方法，把下面这几行：

```
subjects = Subject.objects.annotate(
       total_courses=Count('courses'))
```

替换为：

```
subjects = cache.get('all_subjects')
if not subjects:
    subjects = Subject.objects.annotate(
               total_courses=Count('courses'))
    cache.set('all_subjects', subjects)
```

在这段代码中，我们首先尝试使用 `cache.get()` 来从缓存中得到 `all_students` 键。如果所给的键没有找到，返回的是 None 。如果键没有被找到（没有被缓存，或者缓存了但是过期了），我们就执行查询来检索所有的 `Subject` 对象和它们课程的数量，我们使用 `cache.set()` 来缓存结果。
打开代发服务器，访问 http://127.0.0.1:8000 。当视图被执行的时候，缓存键没有被找到的话查询集就会被执行。访问 http://127.0.0.1:8000/adim/ 然后打开 memcached 统计。你可以看到类似于下面的缓存的使用数据：



django-11-7

看一眼 **Curr Items** 应该是 1 。这表示当前有一个内容被缓存。**Get Hits** 表示有多少的 get 操作成功了，**Get Miss** 表示有多少的请求丢失了。**Miss Ratio** 是使用它们俩来计算的。
现在导航回 http://127.0.0.1:8000/ ，重载页面几次。如果你现在看缓存统计的话，你就会看到更多的（**Get Hits** 和 **Cmd Get** 被执行了）

基于动态数据的缓存

有很多时候你都会想使用基于动态数据的缓存的。基于这样的情况，你必须要创建包含所有要求信息的动态键来特别定以缓存数据。编辑 `courses` 应用的 `views.py` ，修改 `CourseListView` ，让它看起来像这样：

```
class CourseListView(TemplateResponseMixin, View):
    model = Course
    template_name = 'courses/course/list.html'

    def get(self, request, subject=None):
        subjects = cache.get('all_subjects')
        if not subjects:
            subjects = Subject.objects.annotate(
                        total_courses=Count('courses'))
            cache.set('all_subjects', subjects)
        all_courses = Course.objects.annotate(
            total_modules=Count('modules'))
        if subject:
            subject = get_object_or_404(Subject, slug=subject)
            key = 'subject_{}_courses'.format(subject.id)
            courses = cache.get(key)
```

```
            if not courses:
                courses = all_courses.filter(subject=subject)
                cache.set(key, courses)
        else:
            courses = cache.get('all_courses')
            if not courses:
                courses = all_courses
                cache.set('all_courses', courses)
        return self.render_to_response({'subjects': subjects,
                                        'subject': subject,
                                        'courses': courses})
```

在这个场景中，我们把课程和根据科目筛选的课程都缓存了。我们使用 `all_courses` 缓存键来储存所有的课程，如果没有给科目的话。如果给了一个科目的话我们就用 `'subject_()_course'.format(subject.id)` 动态的创建缓存键。
注意到我们不能用一个缓存查询集来创建另外的查询集是很重要的，因为我们已经缓存了当前的查询结果，所以我们不能这样做：

```
courses = cache.get('all_courses')
courses.filter(subject=subject)
```

相反，我们需要创建基本的查询集 `Course.objects.annotate(total_modules=Count('modules'))`，它不会被执行除非你强制执行它，然后用它来更进一步的用 `all_courses.filter(subject=subject)` 限制查询集万一数据没有在缓存中找到的话。

缓存模板片段

缓存模板片段是一个高级别的方法。你需要使用 `{% load cache %}` 在模板中载入缓存模板标签。然后你就可以使用 `{% cache %}` 模板标签来缓存特定的模板片段了。你通常可以像下面这样使用缓存标签：

```
{% cache 300 fragment_name %}
...
{% endcache %}
```

`{% cache %}` 标签要求两个参数：过期时间，以秒为单位，和一个片段名称。如果你需要缓存基于动态数据的内容，你可以通过传递额外的参数给 `{% cache %}` 模板标签来特别的指定片段。
编辑 `students` 应用的 `/students/course/detail.html` 。在顶部添加以下代码，就在 `{% extends %}` 标签的后面：

```
{% load cache %}
```

然后把下面几行：

```
{% for content in module.contents.all %}
    {% with item=content.item %}
        <h2>{{ item.title }}</h2>
        {{ item.render }}
    {% endwith %}
{% endfor %}
```

替换为：

```
{% cache 600 module_contents module %}
```

```
    {% for content in module.contents.all %}
        {% with item=content.item %}
            <h2>{{ item.title }}</h2>
            {{ item.render }}
        {% endwith %}
    {% endfor %}
{% endcache %}
```

我们使用名字 `module_contents` 和传递当前的 `Module` 对象来缓存模板片段。这对于当请求不同的模型是避免缓存一个模型的内容和服务错误的内容来说是很重要的。
如果 `USE_I18N` 设置是为 True，单一站点的中间件缓存将会遵照当前激活的语言。如果你使用了 `{% cache %}` 模板标签以及可用翻译特定的变量中的一个，那么他们的效果将会是一样的，比如：`{% cache 600 name request.LANGUAGE_CODE %}`

缓存视图

你可以使用位于 `django.views.decrators.cache` 的 `cache_page` 装饰器来焕春输出的单个视图。装饰器要求一个过期时间的参数（以秒为单位）。
让我们在我们的视图中使用它。编辑 `students` 应用的 `urls.py`， 添加以下 导入：

```
from django.views.decorators.cache import cache_page
```

然后按照如下在 `student_course_detail_module` URL 模式上应用 `cache_page` 装饰器：

```
url(r'^course/(?P<pk>\d+)/$',
    cache_page(60 * 15)(views.StudentCourseDetailView.as_view()),
    name='student_course_detail'),

url(r'^course/(?P<pk>\d+)/(?P<module_id>\d+)/$',
    cache_page(60 * 15)(views.StudentCourseDetailView.as_view()),
    name='student_course_detail_module'),
```

现在 `StudentCourseDetailView` 的结果就会被缓存 15 分钟了。
单一的视图缓存使用 URL 来创建缓存键。多个指向同一个视图的 URLs 将会被分开储存

使用单一站点缓存

这是最高级的缓存。他让你可以缓存你的整个站点。
为了使用单一站点缓存，你需要编辑项目中的 `settings.py`， 把 `UpdateCacheMiddleware` 和 `FetchFromCacheMiddleware` 添加进 `MIDDLEWARE_CLASSES` 设置中：

```
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.cache.FetchFromCacheMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    # ...
)
```

记住在请求的过程中，中间件是按照所给的顺序来执行的，在相应过程中是逆序执行的。
`UpdateCacheMiddleware` 被放在 `CommonMiddleware` 之前，因为它在相应时才执行，此时中间件是逆序执行的。
`FetchFromCacheMiddleware` 被放在 `CommonMiddleware` 之后，是因为它需要连接后者的的请求数据集。

然后，把下列设置添加进 `settings.py` 文件：

```
CACHE_MIDDLEWARE_ALIAS = 'default'
CACHE_MIDDLEWARE_SECONDS = 60 * 15 # 15 minutes
CACHE_MIDDLEWARE_KEY_PREFIX = 'educa'
```

在这些设置中我们为中间件使用了默认的缓存，然后我们把全局缓存过期时间设置为 15 分钟。我们也指定了所有的缓存键前缀来避免冲突万一我们为多个项目使用了相同的 memcached 后端。我们的站点现在将会为所有的 GET 请求缓存以及返回缓存内容。
我们已经完成了这个来测试单一站点缓存功能。尽管，以站点的缓存对于我们来说是不怎么合适的，因为我们的课程管理视图需要展示更新数据来实时的给出任何的修改。我们项目中的最好的方法是缓存用于展示给学生的课程内容的模板或者视图数据。
我们已经大致体验过了 Django 提供的方法来缓存数据。你应合适的定义你自己的缓存策略，优先考虑开销最大的查询集或者计算。

总结

在这一章中，我们创建了一个用于课程的公共视图，创建了一个用于学生注册和报名课程的系统。我们安装了 memcached 以及实现了不同级别的缓存。
在下一章中，我们将会为你的项目创建 RESTful API。

第十二章 构建一个 API

在上一章中，你构建了一个学生注册系统和课程报名。你创建了用来展示课程内容的视图以及如何使用 Django 的缓存框架。在这章中，你将学习如何做到以下几点：

- 构建一个 RESTful API
- 用 API 视图操作认证和权限
- 创建 API 视图放置和路由

构建一个 RESTful API

你可能想要创建一个接口给其他的服务来与你的 web 应用交互。通过构建一个 API，你可以允许第三方来消费信息以及程序化的操作你的应用程序。
你可以通过很多方法构成你的 API，但是我们最鼓励你遵循 REST 原则。REST 体系结构来自 Representational State Transfer。RESTful API 是基于资源的（resource-based）。你的模型代表资源和 HTTP 方法例如 GET,POST,PUT,以及 DELETE 是被用来取回，创建，更新，以及删除对象的。HTTP 响应代码也可以在上下文中使用。不同的 HTTP 响应代码的返回用来指示 HTTP 请求的结果，例如，2XX 响应代码用来表示成功，4XX 表示错误，等等。
在 RESTful API 中最通用的交换数据是 JSON 和 XML。我们将要为我们的项目构建一个 JSON 序列化的 REST API。我们的 API 会提供以下功能：

- 获取科目
- 获取可用的课程
- 获取课程内容
- 课程报名

我们可以通过创建定制视图从 Django 开始构建一个 API。当然，有很多第三方的模块可以给你的项目简单的创建一个 API，其中最有名的就是 Django Rest Framework。

安装 Django Rest Framework

Django Rest Framework 允许你为你的项目方便的构建 REST APIs。你可以通过访问 http://www.django-rest-framework.org 找到所有 REST Framework 信息。
打开 shell 然后通过以下命令安装这个框架：

```
pip install djangorestframework=3.2.3
```

编辑 *educa* 项目的 *settings.py* 文件，在 *INSTALLED_APPS* 设置中添加 *rest_framework* 来激活这个应用，如下所示：

```
INSTALLED_APPS = (
      # ...
      'rest_framework',
   )
```

之后，添加如下代码到 *settings.py* 文件中：

```
REST_FRAMEWORK = {
      'DEFAULT_PERMISSION_CLASSES': [
'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
      ]
}
```

你可以使用 *REST_FRAMEWORK* 设置为你的 API 提供一个指定的配置。REST Framework 提供了一个广泛的设置去配置默认的行为。*DEFAULT_PERMISSION_CLASSES* 配置指定了去读取，创建，更新或者删除对象的默认权限。我们设置 *DjangoModelPermissionsOrAnonReadOnly* 作为唯一的默认权限类。这个类依赖与 Django 的权限系统允许用户去创建，更新，或者删除对象，同时提供只读的访问给陌生人用户。你会在之后学习更多关于权限的方面。
如果要找到一个完整的 REST 框架可用设置列表，你可以访问 http://www.django-rest-framework.org/api-guide/settings/ 。

定义序列化器

设置好 REST Framework 之后，我们需要指定我们的数据将会如何序列化。输出的数据必须被序列化成指定的格式，并且输出的数据将会给进程去序列化。REST 框架提供了以下类来给单个对象去构建序列化：

- Serializer：给一般的 Python 类实例提供序列化。
- ModelSerializer：给模型实例提供序列化。
- HyperlinkedModelSerializer：类似与 *ModelSerializer*，但是代表与链接而不是主键的对象关系。

让我们构建我们的第一个序列化器。在 *courses* 应用目录下创建以下文件结构：

```
api/
    __init__.py
    serializers.py
```

我们将会在 *api* 目录中构建所有的 API 功能为了保持一切都有良好的组织。编辑 *serializeers.py* 文件，然后添加以下代码：

```
from rest_framework import serializers
from ..models import Subject

class SubjectSerializer(serializers.ModelSerializer):
    class Meta:
        model = Subject
        fields = ('id', 'title', 'slug')
```

以上是给 *Subject* 模型使用的序列化器。序列化器以一种类似的方式被定义给 Django 的 *From* 和 *ModelForm* 类。*Meta* 类允许你去指定模型序列化以及给序列化包含的字段。所有的模型字段都会被包含如果你没有设置一个 *fields* 属性。

让我们尝试我们的序列化器。打开命令行通过`python manage.py shell*开始 Django shell。运行以下代码：

```
from courses.models import Subject
from courses.api.serializers import SubjectSerializer
subject = Subject.objects.latest('id')
serializer = SubjectSerializer(subject)
serializer.data
```

在上面的例子中，我们拿到了一个 *Subject* 对象，创建了一个 *SubjectSerializer* 的实例，并且访问序列化的数据。你会得到以下输出：

```
{'slug': 'music', 'id': 4, 'title': 'Music'}
```

如你所见，模型数据被转换成了 Python 的数据类型。

了解解析器和渲染器

在你在一个 HTTP 响应中返回序列化数据之前，这个序列化数据必须使用指定的格式进行渲染。同样的，当你拿到一个 HTTP 请求，在你使用这个数据操作之前你必须解析传入的数据并且反序列化这个数据。REST Framework 包含渲染器和解析器来执行以上操作。
让我们看下如何解析传入的数据。给予一个 JSON 字符串输入，你可以使用 REST 康佳提供的 *JSONParser* 类来转变它成为一个 Python 对象。在 Python shell 中执行以下代码：

```
from io import BytesIO
from rest_framework.parsers import JSONParser
data = b'{"id":4,"title":"Music","slug":"music"}'
JSONParser().parse(BytesIO(data))
```

你将会拿到以下输出：

```
{'id': 4, 'title': 'Music', 'slug': 'music'}
```

REST Framework 还包含 *Renderer* 类，该类允许你去格式化 API 响应。框架会查明通过的内容使用的是哪种渲染器。它对响应进行检查，根据请求的 *Accept* 头去预判内容的类型。除此以外，渲染器可以通过 URL 的格式后缀进行预判。举个例子，访问将会出发 *JSONRenderer* 为了返回一个 JSON 响应。
回到 shell 中，然后执行以下代码去从提供的序列化器例子中渲染 *serializer* 对象：

```
from rest_framework.renderers import JSONRenderer
JSONRenderer().render(serializer.data)
```

你会看到以下输出：
b'{"id":4,"title":"Music","slug":"music"}'
我们使用 *JSONRenderer* 去渲染序列化数据为 JSON。默认的，REST Framework 使用两种不同的渲染器：*JSONRenderer* 和 *BrowsableAPIRenderer*。后者提供一个 web 接口可以方便的浏览你的 API。你可以通过 *REST_FRAMEWORK* 设置的 *DEFAULT_RENDERER_CLASSES* 选项改变默认的渲染器类。
你可以找到更多关于渲染器和解析器的信息通过访问 http://www.django-rest-framework.org/api-guide/renderers/ 以及 http://www.django-rest- framework.org/api-guide/parsers/ 。

构建列表和详情视图

REST Framework 自带一组通用视图和 mixins，你可以用来构建你自己的 API。它们提供了获取，创建，更新以及删除模型对象的功能。你可以看到所有 REST Framework 提供的通用 mixins 和视图，通过访问 http://www.django-rest-framework.org/api-guide/generic-views/ 。
让我们创建列表和详情视图去取回 *Subject* 对象们。在 *courses/api/*目录下创建一个新的文件并命名为 *views.py*。添加如下代码：

```
from rest_framework import generics
from ..models import Subject
from .serializers import SubjectSerializer

class SubjectListView(generics.ListAPIView):
    queryset = Subject.objects.all()
    serializer_class = SubjectSerializer

class SubjectDetailView(generics.RetrieveAPIView):
    queryset = Subject.objects.all()
    serializer_class = SubjectSerializer
```

在这串代码中，我们使用 REST Framework 提供的 *ListAPIView* 和 *RetrieveAPIView* 视图。我们给给予的关键值包含了一个 *pk* URL 参数给详情视图去取回对象。两个视图都有以下属性：

- queryset：基础查询集用来取回对象。
- serializer_class：这个类用来序列化对象。

让我们给我们的视图添加 URL 模式。在 *courses/api/*目录下创建新的文件并命名为 *urls.py* 并使之看上去如下所示：

```
from django.conf.urls import url
from . import views

urlpatterns = [
      url(r'^subjects/$',
          views.SubjectListView.as_view(),
          name='subject_list'),
      url(r'^subjects/(?P<pk>\d+)/$',
          views.SubjectDetailView.as_view(),
          name='subject_detail'),
]
```

编辑 *educa* 项目的主 *urls.py* 文件并且包含以下 API 模式：

```
urlpatterns = [
    # ...
    url(r'^api/', include('courses.api.urls', namespace='api')),
]
```

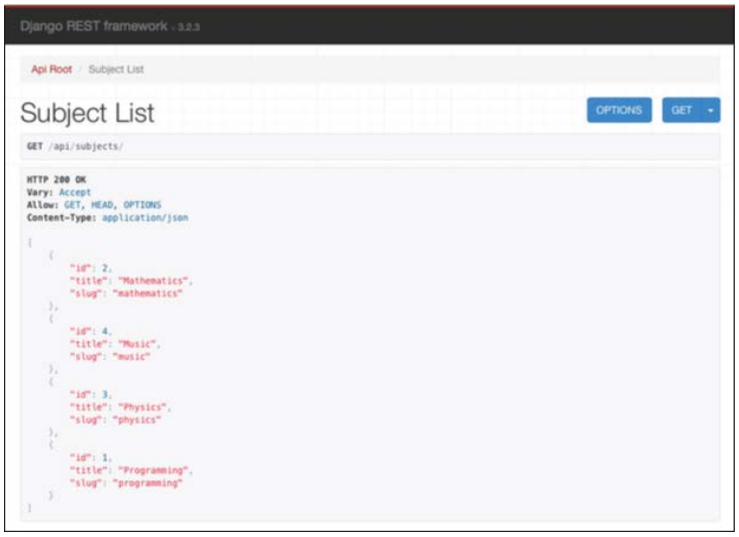
我们给我们的 API URLs 使用 *api* 命名空间。确保你的服务器已经通过命令 `python manange.py runserver` 启动。打开 shell 然后通过 *cURL* 获取 URLhttp://127.0.0.1:8000/api/subjects/ 如下所示：

```
$ curl http://127.0.0.1:8000/api/subjects/
```

你会获取类似以下的响应：

[{"id":2,"title":"Mathematics","slug":"mathematics"},{"id":4,"title":"Music","slug":"music"},{"id":3,"title":"Physics","slug":"physics"},{"id":1,"title":"Programming","slug":"programming"}]

这个 HTTP 响应包含 JSON 格式的一个 *Subject* 对象列。如果你的操作系统没有安装过 *cURL*，你可还可以使用其他的工具去发送定制 HTTP 请求例如一个浏览器扩展 Postman ，这个扩展你可以在 https://www.getpostman.com 找到。
在你的浏览器中打开 http://127.0.0.1:8000/api/subjects/ 。你会看到如下所示的 REST Framework 的可浏览 API：



django-12-1

这个 HTML 界面由 *BrowsableAPIRenderer* 渲染器提供。它展示了结果头和内容并且允许执行请求。你还可以在 URL 包含一个 *Subject* 对象的 id 来访问该对象的 API 详情视图。在你的浏览器中打开 http://127.0.0.1:8000/api/subjects/1/ 。你将会看到一个单独的渲染成 JSON 格式的 *Subject* 对象。

创建嵌套的序列化

我们将要给 *Course* 模型创建一个序列化。编辑 *api/serializers.py* 文件并添加以下代码：

```
from ..models import Course


class CourseSerializer(serializers.ModelSerializer):
    class Meta:
        model = Course
        fields = ('id', 'subject', 'title', 'slug', 'voerview',
                  'created', 'owner', 'modules')
```

让我们看下一个 *Course* 对象是如何被序列化的。打开 shell，运行 `python manage.py shell`，然后运行以下代码：

```
from rest_framework.renderers import JSONRenderer
from courses.models import Course
from courses.api.serializers import CourseSerializer
course = Course.objects.latest('id')
serializer = CourseSerializer(course)
JSONRenderer().render(serializer.data)
```

你将会通过我们包含在 *CourseSerializer* 中的字段获取到一个 JSON 对象。你可以看到 *modules* 管理器的被关联对象呗序列化成一列关键值，如下所示：

```
"modules": [17, 18, 19, 20, 21, 22]
```

我们想要包含个多的信息关于每一个模块，所以我们需要序列化 *Module* 对象以及嵌套它们。修改 *api/serializers.py* 文件提供的代码，使之看上去如下所示：

```
from rest_framework import serializers
from ..models import Course, Module

class ModuleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Module
        fields = ('order', 'title', 'description')

class CourseSerializer(serializers.ModelSerializer):
    modules = ModuleSerializer(many=True, read_only=True)
    class Meta:
        model = Course
        fields = ('id', 'subject', 'title', 'slug', 'overview',
                  'created', 'owner', 'modules')
```

我们给 *Module* 模型定义了一个 *ModuleSerializer* 去提供序列化。之后我们添加一个 *modules* 属性给 *CourseSerializer* 去嵌套 *ModuleSerializer* 序列化器。我们设置 *many=True* 去表明我们正在序列化多个对象。*read_only* 参数表明这个字段是只读的并且不可以被包含在任何输入中去创建或者升级对象。打开 shell 并且再次创建一个 *CourseSerializer* 的实例。使用 *JSONRenderer* 渲染序列化器的 *data* 属性。这一次，被排列的模块会被通过嵌套的 *ModuleSerializer* 序列化器给序列化，如下所示：

```
"modules": [
       {
          "order": 0,
          "title": "Django overview",
          "description": "A brief overview about the Web Framework."
       },
       {
           "order": 1,
           "title": "Installing Django",
           "description": "How to install Django."
       },
       ...
]
```

你可以找到更多关于序列化器的内容，通过访问 http://www.django-rest-framework.org/api-guide/serializers/。

构建定制视图

REST Framework 提供一个 *APIView* 类，这个类基于 Django 的 *View* 类构建 API 功能。*APIView* 类与 *View* 在使用 REST Framework 的定制 *Request* 以及 *Response* 对象时不同，并且操作 *APIException* 例外的返回合适的 HTTP 响应。它还有一个内建的验证和认证系统去管理视图的访问。
我们将要创建一个视图给用户去对课程进行报名。编辑 *api/views.py* 文件并且添加以下代码：

```
from django.shortcuts import get_object_or_404
from rest_framework.views import APIView
```

```
from rest_framework.response import Response
from ..models import Course

class CourseEnrollView(APIView):
    def post(self, request, pk, format=None):
        course = get_object_or_404(Course, pk=pk)
        course.students.add(request.user)
        return Response({'enrolled': True})
```

CourseEnrollView 视图操纵用户对课程进行报名。以上代码解释如下：

- 我们创建了一个定制视图，是 *APIView* 的子类。
- 我们给 POST 操作定义了一个 *post()*方法。其他的 HTTP 方法都不允许放这个这个视图。
- 我们预计一个 *pk*URL 参数会博涵一个课程的 ID。我们通过给予的 *pk* 参数获取这个课程，并且如果这个不存在的话就抛出一个 404 异常。
- 我们添加当前用户给 *Course* 对象的 *students* 多对多关系并放回一个成功响应。

编辑 *api/urls.py* 文件并且给 *CourseEnrollView* 视图添加以下 URL 模式：

```
url(r'^courses/(?P<pk>\d+)/enroll/$',
      views.CourseEnrollView.as_view(),
      name='course_enroll'),
```

理论上，我们现在可以执行一个 POST 请求去给当前用户对一个课程进行报名。但是，我们需要辨认这个用户并且阻止为认证的用户来访问这个视图。让我们看下 API 认证和权限是如何工作的。

操纵认证

REST Framework 提供认证类去辨别用户执行的请求。如果认证成功，这个框架会在 *request.user* 中设置认证的 *User* 对象。如果没有用户被认证，一个 Django 的 *AnonymousUser* 实例会被代替。

REST Framework 提供以下认证后台：

- BasicAuthentication：HTTP 基础认证。用户和密码会被编译为 Base64 并被客户端设置在 *Authorization* HTTP 头中。你可以学习更多关于它的内容，通过访问 https://en.wikipedia.org/wiki/Basic_access_authentication 。
- TokenAuthentication：基于 token 的认证。一个 *Token* 模型被用来存储用户 tokens。用来认证的 *Authorization* HTTP 头里面拥有包含 token 的用户。
- SessionAuthentication：使用 Djnago 的会话后台（session backend）来认证。这个后台从你的网站前端来执行认证 AJAX 请求给 API 是非常有用的。

你可以创建一个通过继承 REST Framework 提供的 *BaseAuthentication* 类的子类以及重写 *authenticate()*方法来构建一个定制的认证后台。

你可以在每个视图的基础上设置认证，或者通过 *DEFAULT_AUTHENTICATION_CLASSES* 设置为全局认证。

认证只能失败用户正在执行的请求。它无法允许或者组织视图的访问。你必须使用权限去限制视图的访问。

你可以找到关于认证的所有信息，通过访问 http://www.django-rest- framework.org/api-guide/authentication/ 。

让我们给我们的视图添加 *BasicAuthentication*。编辑 *courses* 应用的 *api/views.py* 文件，然后给 *CourseEnrollView* 添加一个 *authentication_classes* 属性，如下所示：

```
from rest_framework.authentication import BasicAuthentication

class CourseEnrollView(APIView):
    authentication_classes = (BasicAuthentication,)
```

```
    # ...
```

用户将会被设置在 HTTP 请求中的 *Authorization* 头里面的证书进行识别。

给视图添加权限

REST Framework 包含一个权限系统去限制视图的访问。一些 REST Framework 的内置权限如下所示：

- AllowAny：无限制的访问，无论当前用户是否通过认证。
- IsAuthenticated：只允许通过认证的用户。
- IsAuthenticatedOrReadOnly：通过认证的用户拥有完整的权限。陌生用户只允许去还行可读的方法，例如 GET, HEAD 或者 OPETIONS。
- DjangoModelPermissions：权限与 *django.contrib.auth* 进行了捆绑。视图需要一个 *queryset* 属性。只有分配了模型权限的并经过认证的用户才能获得权限。
- DjangoObjectPermissions：基于每个对象基础上的 Django 权限。

如果用户没有权限，他们通常会获得以下某个 HTTP 错误：

- HTTP 401：无认证。
- HTTP 403：没有权限。

你可以获得更多的关于权限的信息，通过访问 http://www.django-rest-framework.org/api-guide/permissions/ 。

编辑 *courses* 应用的 *api/views.py* 文件然后给 *CourseEnrollView* 添加一个 *permission_classes* 属性，如下所示：

```
from rest_framework.authentication import BasicAuthentication
from rest_framework.permissions import IsAuthenticated

class CourseEnrollView(APIView):
    authentication_classes = (BasicAuthentication,)
    permission_classes = (IsAuthenticated,)
    # ...
```

我们包含了 *IsAuthenticated* 权限。这个权限将会组织陌生用户访问这个视图。现在，我们可以之 sing 一个 POST 请求给我们的新的 API 方法。

确保开发服务器正在运行。打开 shell 然后运行以下命令：

```
curl -i -X POST http://127.0.0.1:8000/api/courses/1/enroll/
```

你将会得到以下响应：

```
HTTP/1.0 401 UNAUTHORIZED
...
{"detail": "Authentication credentials were not provided."}
```

如我们所预料的，我们得到了一个 401 HTTP code，因为我们没有认证过。让我们带上我们的一个用户进行下基础认证。运行以下命令：

```
curl -i -X POST -u student:password http://127.0.0.1:8000/api/courses/1/enroll/
```

使用一个已经存在的用户的证书替换 *student:password*。你会得到以下响应：

```
HTTP/1.0 200 OK
...
{"enrolled": true}
```

你可以额访问管理站点然后检查到上面命令中的用户已经完成了课程的报名。

创建视图设置和路由

ViewSets 允许你去定义你的 API 的交互并且让 REST Framework 通过一个 *Router* 对象动态的构建 URLs。通过使用视图设置，你可以避免给多个视图重复编写相同的逻辑。视图设置包含典型的创建，获取，更新，删除选项操作，它们是 *list()*,*create()*,*retrieve()*,*update()*,*partial_update()*以及 *destroy()*。让我们给 *Course* 模型创建一个视图设置。编辑 *api/views.py* 文件然后添加以下代码：

```
from rest_framework import viewsets
from .serializers import CourseSerializer

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer
```

我们创建了一个继承 *ReadOnlyModelViewSet* 类的子类，被继承的类提供了只读的操作 *list()*和 *retrieve()*，前者用来排列对象，后者用来取回一个单独的对象。编辑 *api/urls.py* 文件并且给我们的视图设置创建一个路由，如下所示：

```
from django.conf.urls import url, include
from rest_framework import routers
from . import views

router = routers.DefaultRouter()
router.register('courses', views.CourseViewSet)

urlpatterns = [
    # ...
    url(r'^', include(router.urls)),
]
```

我们创建两个一个 *DefaultRouter* 对象并且通过 *courses* 前缀注册了我们的视图设置。这个路由负责给我们的视图动态的生成 URLs。
在你的浏览器中打开 http://127.0.0.1:8000/api/ 。你会看到路由排列除了所有的视图设置在它的基础 URL 中，如下图所示：

django-12-2

你可以访问 http://127.0.0.1:8000/api/courses/ 去获取课程的列表。
你可以学习到跟多关于视图设置的内容，通过访问 http://www.django-rest-framework.org/api-guide/viewsets/ 。你也可以找到更多关于路由的信息，通过访问 http://www.django-rest-framework.org/api-guide/routers/ 。

给视图设置添加额外的操作

你可以给视图设置添加额外的操作。让我们修改我们之前的 *CourseEnrollView* 视图成为一个定制的视图设置操作。编辑 *api/views.py* 文件然后修改 *CourseViewSet* 类如下所示：

```
from rest_framework.decorators import detail_route

class CourseViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Course.objects.all()
    serializer_class = CourseSerializer
    @detail_route(methods=['post'],
                  authentication_classes=[BasicAuthentication],
                  permission_classes=[IsAuthenticated])
    def enroll(self, request, *args, **kwargs):
        course = self.get_object()
        course.students.add(request.user)
        return Response({'enrolled': True})
```

我们添加了一个定制 *enroll()*方法相当于给这个视图设置的一个额外的操作。以上的代码解释如下：

- 我们使用框架的 *detail_route* 装饰器去指定这个类是一个在一个单独对象上被执行的操作。
- 这个装饰器允许我们给这个操作添加定制属性。我们指定这个视图只允许 POST 方法，并且设置了认证和权限类。
- 我们使用 *self.get_object()*去获取 *Course* 对象。
- 我们给 *students* 添加当前用户的多对多关系并且返回一个定制的成功响应。

编辑 *api/urls.py* 文件并移除以下 URL，因为我们不再需要它们：

```
url(r'^courses/(?P<pk>[\d]+)/enroll/$',
      views.CourseEnrollView.as_view(),
      name='course_enroll'),
```

之后编辑 *api/views.py* 文件并且移除 *CourseEnrollView* 类。
这个用来在课程中报名的 URL 现在已经是由路由动态的生成。这个 URL 保持不变，因为它使用我们的操作名 *enroll* 动态的进行构建。

创建定制权限

我们想要学生可以访问他们报名过的课程的内容。只有在这个课程中报名过的学生才能访问这个课程的内容。最好的方法就是通过一个定制的权限类。Django 提供了一个 *BasePermission* 类允许你去定制以下功能：

- has_permission()：视图级的权限检查。
- has_object_permission()：实例级的权限检查。

以上方法会返回 *True* 来允许访问，相反就会返回 *False*。在 *courses/api/*中创建一个新的文件并命名为 *permissions.py*。添加以下代码：

```
from rest_framework.permissions import BasePermission

class IsEnrolled(BasePermission):
    def has_object_permission(self, request, view, obj):
        return obj.students.filter(id=request.user.id).exists()
```

我们创建了一个继承 *BasePermission* 类的子类，并且重写了 *has_object_permission()*。我们检查执行请求的用户是否存在 *Course* 对象的 *students* 关系。我们下一步将要使用 *IsEnrolled* 权限。

序列化课程内容

我们需要序列化课程内容。*Content* 模型包含一个通用的外键允许我们去关联不同的内容模型对象。然而，我们给上一章中给所有的内容模型添加了一个公用的 *render()*方法。我们可以使用这个方法去提供渲染过的内容给我们的 API。
编辑 *courses* 应用的 *api/serializers.py* 文件并且添加以下代码：

```
from ..models import Content

class ItemRelatedField(serializers.RelatedField):
    def to_representation(self, value):
        return value.render()

class ContentSerializer(serializers.ModelSerializer):
    item = ItemRelatedField(read_only=True)
    class Meta:
        model = Content
        fields = ('order', 'item')
```

在以上代码中，我们通过子类化 REST Framework 提供的 *RealtedField* 序列化器字段定义了一个定制字段并且重写了 *to_representation()*方法。我们给 *Content* 模型定义了 *ContentSerializer* 序列化器并且使用定制字段给 *item* 生成外键。
我们需要一个替代序列化器给 *Module* 模型来包含它的内容，以及一个扩展的 *Course* 序列化器。编辑 *api/serializers.py* 文件并且添加以下代码：

```
   class ModuleWithContentsSerializer(serializers.ModelSerializer):
       contents = ContentSerializer(many=True)
       class Meta:
           model = Module
           fields = ('order', 'title', 'description', 'contents')

   class CourseWithContentsSerializer(serializers.ModelSerializer):
       modules = ModuleWithContentsSerializer(many=True)
       class Meta:
           model = Course
           fields = ('id', 'subject', 'title', 'slug',
                     'overview', 'created', 'owner', 'modules')
```

让我们创建一个视图来模仿 *retrieve()*操作的行为但是包含课程内容。编辑 *api/views.py* 文件添加以下方法给 *CourseViewSet* 类：

```
   from .permissions import IsEnrolled
   from .serializers import CourseWithContentsSerializer

   class CourseViewSet(viewsets.ReadOnlyModelViewSet):
       # ...
       @detail_route(methods=['get'],
                     serializer_class=CourseWithContentsSerializer,
                     authentication_classes=[BasicAuthentication],
                     permission_classes=[IsAuthenticated,
                                         IsEnrolled])
       def contents(self, request, *args, **kwargs):
           return self.retrieve(request, *args, **kwargs)
```

以上的方法解释如下：

- 我们使用 *detail_route* 装饰器去指定这个操作是在一个单独的对象上进行执行。
- 我们指定只有 GET 方法允许访问这个操作。
- 我们使用新的 *CourseWithContentsSerializer* 序列化器类来包含渲染过的课程内容。
- 我们使用 *IsAuthenticated* 和我们的定制 *IsEnrolled* 权限。只要做到了这点，我们可以确保只有在这个课程中报名的用户才能访问这个课程的内容。
- 我们使用存在的 *retrieve()*操作去返回课程对象。

在你的浏览器中打开 http://127.0.0.1:8000/api/courses/1/contents/ 。如果你使用正确的证书访问这个视图，你会看到这个课程的每一个模块都包含给渲染过的课程内容的 HTML，如下所示：

```
  {
  "order": 0,
  "title": "Installing Django",
  "description": "",
  "contents": [
      {
      "order": 0,
      "item": "<p>Take a look at the following video for installing Django:</p>\n"
      },
      {
      "order": 1,
      "item": "\n<iframe width=\"480\" height=\"360\"
src=\"http://www.youtube.com/embed/bgV39DlmZ2U?wmode=opaque\" frameborder=\"0\"
allowfullscreen></iframe>\n\n"
      }
   ]
   }
```

你已经构建了一个简单的 API 允许其他服务器来程序化的访问课程应用。REST Framework 还允许你通过 *ModelViewSet* 视图设置去管理创建以及编辑对象。我们已经覆盖了 Django Rest Framework 的主要部分，但是你可以找到该框架更多的特性，通过查看它的文档，地址
在 http://www.django-rest-framework.org/ 。

总结

在这章中，你创建了一个 RESTful API 给其他的服务器去与你的 web 应用交互。
一个额外的章节**第十三章，Going Live** 需要在线下载：
https://www.packtpub.com/sites/default/files/downloads/Django_By_Example_GoingLive.pdf 。第十三章将会教你如何使用 uWSGI 以及 NGINX 去构建一个生产环境。你还将学习到如何去导入一个定制中间件以及创建定制管理命令。
你已经到达这本书的结尾。恭喜你！你已经学习到了通过 Django 构建一个成功的 web 应用所需的技能。这本书指导你通过其他的技术与 Django 集合去开发了几个现实生活能用到的项目。现在你已经准备好去创建你自己的 Django 项目了，无论是一个简单的样品还是一个一个强大的 wen 应用。
祝你下一次 Django 的冒险好运！