
Table of Contents

Introduction	1.1
Django入门与实践-第1章：环境搭建	1.2
Django入门与实践-第2章：创建项目	1.3
Django入门与实践-第3章：Hello World	1.4
Django入门与实践-第4章：系统设计	1.5
Django入门与实践-第5章：模型设计	1.6
Django入门与实践-第6章：第一个视图函数	1.7
Django入门与实践-第7章：模板引擎设置	1.8
Django入门与实践-第8章：第一个单元测试	1.9
Django入门与实践-第9章：静态文件设置	1.10
Django入门与实践-第10章：Django Admin 介绍	1.11
Django入门与实践-第11章：URL 分发	1.12
Django入门与实践-第12章：复用模板	1.13
Django入门与实践-第13章：表单处理	1.14
Django入门与实践-第14章：用户注册	1.15
Django入门与实践-第15章：用户注销	1.16
Django入门与实践-第16章：用户登录	1.17
Django入门与实践-第17章：保护视图	1.18
Django入门与实践-第18章：实现主题回复列表	1.19
Django入门与实践-第19章：实现主题回复功能	1.20
Django入门与实践-第20章：查询结果集	1.21
Django入门与实践-第21章：Django数据迁移	1.22
Django入门与实践-第22章：基于类的视图	1.23
Django入门与实践-第23章：分页实现	1.24

Django入门与实践-第24章：我的账户	1.25
Django入门与实践-第25章：Markdown支持	1.26
Django入门与实践-第26章：个性化设置	1.27
Django入门与实践-第27章：项目部署	1.28

Django入门与实践教程

本教程是[A Complete Beginner's Guide to Django](#) 的翻译计划，由公众号「Python之禅」发起，这个教程可能是史上最浅显易懂的 Django 入门教程，适合新手作为练手项目学习。

教程从最基础的软件安装、环境搭建开始，基于测试驱动开发。就算你没有任何实践经验，只要有基本的Python语法知识，你就可以跟着教程一步一步搭建出一个完成的论坛网站出来。

教程总共有7个大章节，我们把它拆分出更多的小节来学习，这样更有利于读者学习。目前还剩最后一个章节没有完成翻译，欢迎认领

参与翻译的成员如下，感谢他们的付出

- [liuzhijun](#)
- [vimiix](#)
- [jiangyanglinlan](#)
- [wzhbingo](#)
- [CasualJi](#)
- [hellodabin](#)

目录

- [Django入门与实践-第1章：环境搭建](#)
- [Django入门与实践-第2章：创建项目](#)
- [Django入门与实践-第3章：Hello World](#)
- [Django入门与实践-第4章：系统设计](#)
- [Django入门与实践-第5章：模型设计](#)
- [Django入门与实践-第6章：第一个视图函数](#)
- [Django入门与实践-第7章：模板引擎设置](#)
- [Django入门与实践-第8章：第一个单元测试](#)
- [Django入门与实践-第9章：静态文件设置](#)

- Django入门与实践-第10章：Django Admin 介绍
- Django入门与实践-第11章：URL 分发
- Django入门与实践-第12章：复用模板
- Django入门与实践-第13章：表单处理
- Django入门与实践-第14章：用户注册
- Django入门与实践-第15章：用户注销
- Django入门与实践-第16章：用户登录
- Django入门与实践-第17章：保护视图
- Django入门与实践-第18章：实现主题回复列表
- Django入门与实践-第19章：实现主题回复功能
- Django入门与实践-第20章：查询结果集
- Django入门与实践-第21章：Django数据迁移
- Django入门与实践-第22章：基于类的视图
- Django入门与实践-第23章：分页实现
- Django入门与实践-第24章：我的账户
- Django入门与实践-第25章：Markdown支持
- Django入门与实践-第26章：个性化设置
- Django入门与实践-第27章：项目部署

扫描下面二维码，在公众号回复「django」可获取本教程的电子版



Django入门指南-第1章：环境搭建

前言



今天我将开始一个关于 Django 基础知识的全新系列教程。这是一个开始学习 Django 的完整入门指南。教程材料一共会被分为七个部分（译注：被我们拆分为27个小章节）。我们将从安装，开发环境的准备、模型、视图、模板、URL到更高级的主题（如迁移、测试和部署）中详细探讨所有基本概念。

我想做一些不一样的事情。一个容易学习，内容丰富且不失趣味的教程。我的想法是在文章中穿插一些漫画的方式来演示说明相应的概念和场景。我希望大家能够享受这种阅读！

但是在我们开始之前...

当年我在一所大学担任代课教授时，我曾经在计算机科学专业给新来的学生讲授网络开发学科。那时我总是会用下面这个孔夫子的名言开始新的课程：



(译者注：不确定是孔子讲的，但这句话早在中国古代就有所提到，出自荀子《儒效篇》“不闻不若闻之，闻之不若见之，见之不若知之，知之不若行之；学至于行之而止矣”)

所以，请动起手来！不要只是阅读教程。我们一起来练习！通过实践和练习你会收获的更多。

为什么要学习Django？

Django是一个用python编写的Web框架。Web框架是一种软件，基于web框架可以开发动态网站，各种应用程序以及服务。它提供了一系列工具和功能，可以解决许多与Web开发相关的常见问题，比如：安全功能，数据库访问，会话，模板处理，URL路由，国际化，本地化，等等。

使用诸如 Django 之类的网络框架，使我们能够以标准化的方式快速开发安全可靠的Web应用程序，而无需重新发明轮子。

那么，Django有什么特别之处呢？对于初学者来说，它是一个Python Web框架，这意味着你可以受益于各种各样的开源库包。[python软件包资料库](#)

[\(pypi\)](#) 拥有超过11.6万个软件包（2017年9月6日的数据）。如果当你想要解决一个特定的问题的时候，可能有人已经为它实现了一个库来供你使用。

Django是用python编写的最流行的web框架之一。它绝对是最完整的，提供了各种各样的开箱即用的功能，比如用于开发和测试的独立Web服务器，缓存，中间件系统，ORM，模板引擎，表单处理，基于Python单元测试的工具

接口。Django还自带内部电池，提供内置应用程序，比如一个认证系统，一个可用于 CRUD (增删改查) 操作并且自动生成页面的后台管理界面，生成订阅文档 (RSS/Atom) ,站点地图等。甚至在django中内建了一个地理信息系统 (GIS) 框架。

Django的开发得到了[Django软件基金会](#)的支持，并且由jetbrains和instagram等公司赞助。Django现在已经存在了相当长的一段时间了。到现在为止，活跃的项目开发时间已经超过12年，这也证明了它是一个成熟，可靠和安全的网络框架。

谁在使用Django？

知道谁在使用Django是很好的，同时也想一想你可以用它来做些什么。在使用Django的大型网站

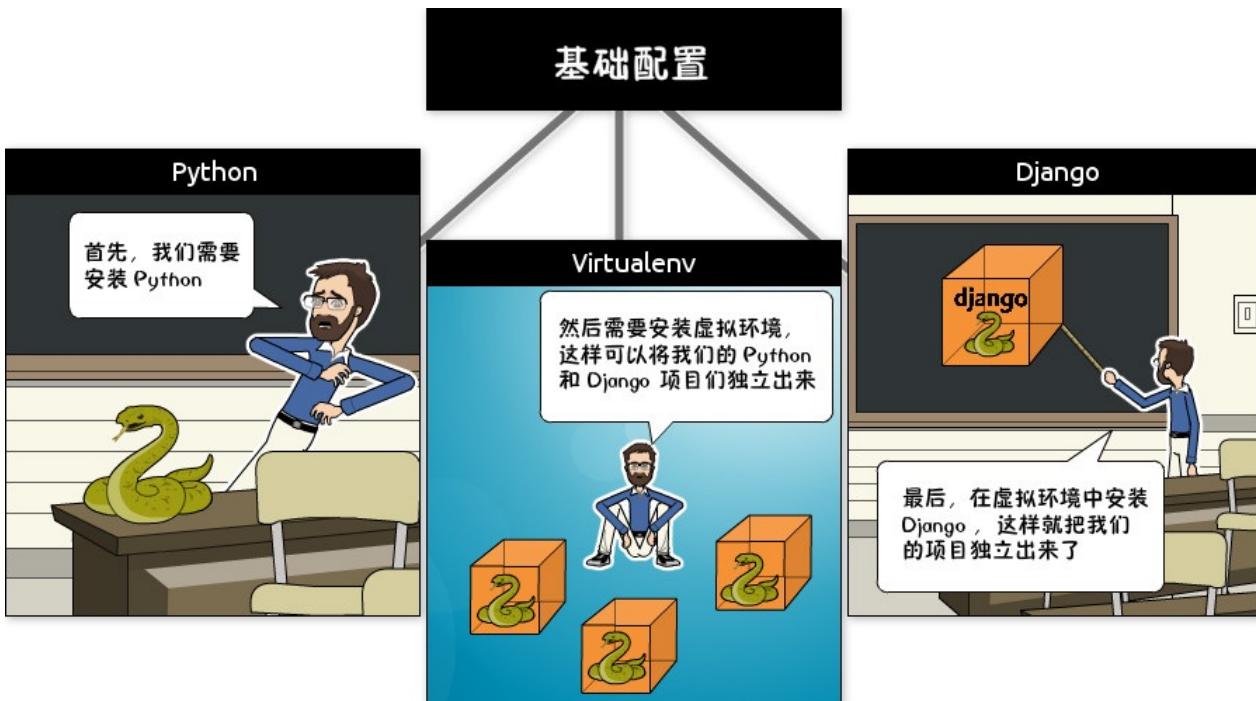
有：[Instagram](#), [Disqus](#), [Mozilla](#), [Bitbucket](#), [Last.fm](#), [国家地理](#)。

想知道更多的示例，你可以到[Django Sites](#)数据库中查看，它提供超过五千个Django驱动的网站列表。

顺便说一下，去年在Django 2016年发布会上，Django核心开发人员，Instagram员工 carl meyer，就[Instagram如何大规模使用Django以及它如何支持他们的用户增长](#)做过一次分享。这是个一小时的演讲，如果你有兴趣学习了解更多的话，这是一次很有趣的演讲。

安装

我们需要做的第一件事是在我们的电脑上安装一些程序，以便能够开始使用django。基本的设置包括安装[Python](#), [Virtualenv](#)和[Django](#)。



使用虚拟环境不是强制性的，但是我还是强烈建议大家这样做。如果你是一个初学者，那么最好形成一个良好的开端。

当你在用 Django 开发一个网站或者一个Web项目的时候，不得不安装外部库以支持开发是非常常见的事情。使用虚拟环境，你开发的每个项目都会有其独立的环境。这样的话，包之间的依赖关系不会发生冲突。同时也使得你能在不同Django版本上运行的本地机器的项目。

在后面你会看到使用它是非常简单的！

安装 Python 3.6.2

我们想要做的第一件事是安装最新版的Python，那就是**Python 3.6.2**。至少是在我写这篇教程的时候。如果有更新的版本，请使用新版。接下来的步骤也应该保持大致相同的做法。

我们将使用Python 3，因为大部分主要的Python库已经被移植到python 3，并且下一个主要的django版本（2.x）也将不再支持python 2。所以Python 3是正确的选择。

最好的方法是通过[Homebrew](#)安装。如果你的Mac还没有安装Homebrew的话，在终端中执行下面的命令：

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

如果你没有安装命令行工具（**Command Line Tools**），Homebrew的安装可能需要稍长一点的时间。但它会帮助你处理好一切，所以不用担心。只需要坐下来等到安装完成即可。

当你看到以下消息时，就代表安装完成了：

```
==> Installation successful!  
  
==> Homebrew has enabled anonymous aggregate user behaviour analytics.  
Read the analytics documentation (and how to opt-out) here:  
  https://docs.brew.sh/Analytics.html  
  
==> Next steps:  
- Run `brew help` to get started  
- Further documentation:  
  https://docs.brew.sh
```

执行下面的命令来安装Python 3：

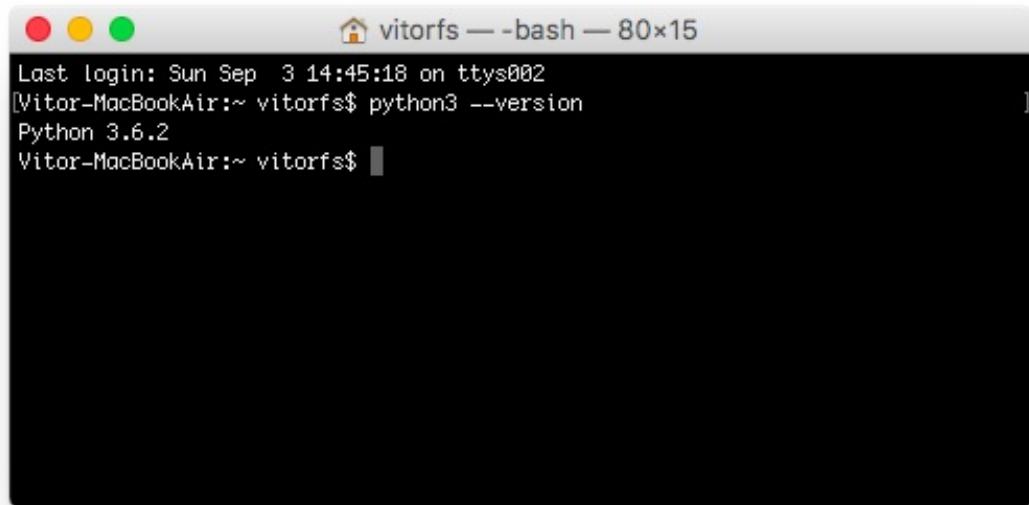
```
brew install python3
```

由于macOS原本已经安装了python 2，所以在安装python 3之后，你将可以同时使用这两个版本。

需要运行Python 2的话，在终端中通过命令 `python` 启动。如果想运行Python 3，则使用 `python3` 来启动。

我们可以在终端中测试一下：

```
python3 --version  
Python 3.6.2
```



很棒，python已经启动并正在运行。下一步：虚拟环境！

安装 Virtualenv

接下来这一步，我们将通过**pip**(一个管理和安装*Python*包的工具)来安装**Virtualenv**。

请注意，Homebrew已经为你安装好了**pip**，在python 3.6.2下的名称为**pip3**。

在终端中，执行下面的命令：

```
sudo pip3 install virtualenv
```

```
Last login: Sun Sep  3 15:53:20 on ttys002
[Vitor-MacBookAir:~ vitorfs$ sudo pip3 install virtualenv
>Password:
The directory '/Users/vitorfs/Library/Caches/pip/http' or its parent directory is not owned by the current user and the cache has been disabled. Please check the permissions and owner of that directory. If executing pip with sudo, you may want sudo's -H flag.
The directory '/Users/vitorfs/Library/Caches/pip' or its parent directory is not owned by the current user and caching wheels has been disabled. Check the permissions and owner of that directory. If executing pip with sudo, you may want sudo's -H flag.
Collecting virtualenv
  Downloading virtualenv-15.1.0-py2.py3-none-any.whl (1.8MB)
    100% |██████████| 1.8MB 449kB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-15.1.0
Vitor-MacBookAir:~ vitorfs$
```

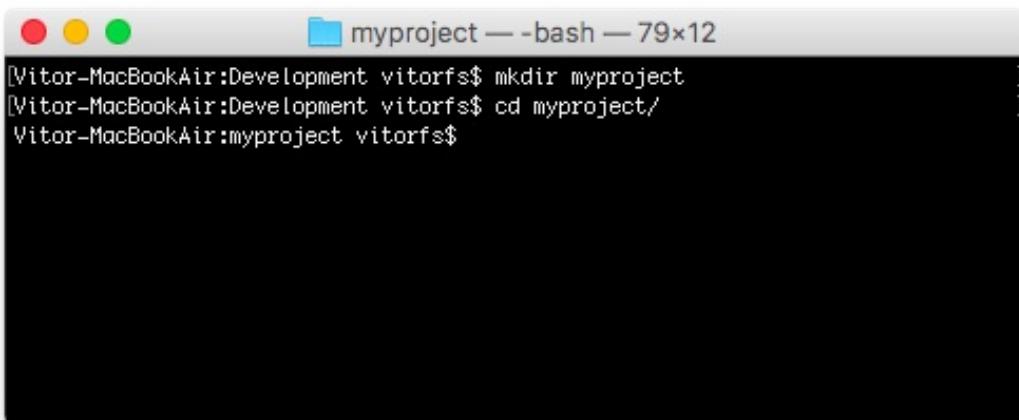
到目前为止，我们执行的安装都是在操作系统环境下运行的。从现在开始，我们安装的所有东西，包括django本身，都将安装在虚拟环境中。

这样想一下：对于你开始的每个Django项目，你首先会为它创建一个虚拟环境。这就像每个Django项目都有一个沙盒。所以你随意运行，安装软件包，卸载软件包而不会破坏任何东西。

我习惯在电脑上创建一个名为**Development**的文件夹。然后，我用它来组织我所有的项目和网站。但你也可以按照接下来的步骤创建适合你自己的目录。

通常，我首先在**Development**文件夹中创建一个项目名称的新文件夹。既然这将是我们第一个项目，我们没必要挑选一个独特的名字。现在，我们可以称之为**myproject**。

```
mkdir myproject
cd myproject
```



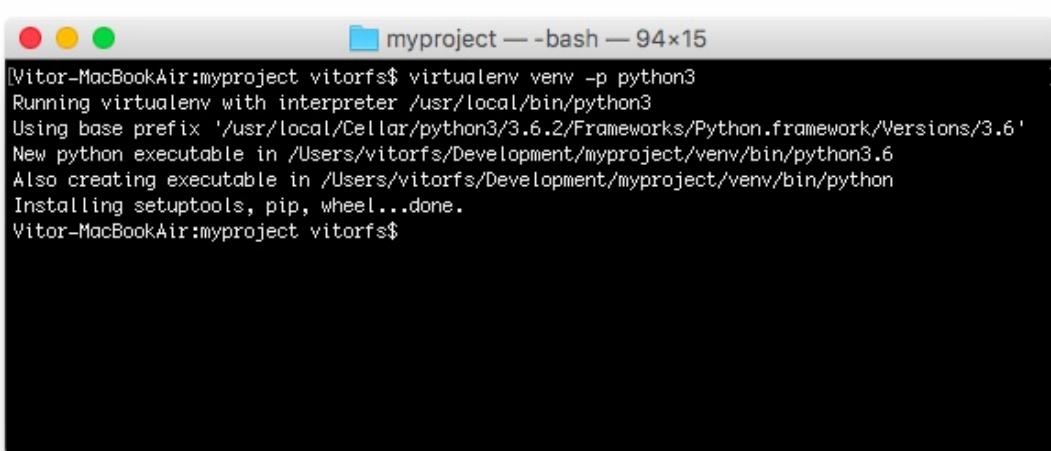
```
[Vitor-MacBookAir:Development vitorfs$ mkdir myproject
[Vitor-MacBookAir:Development vitorfs$ cd myproject/
Vitor-MacBookAir:myproject vitorfs$
```

这个文件夹是级别较高的目录，将存储与我们的Django项目相关的所有文件和东西，包括它的虚拟环境。

所以让我们开始创建我们的第一个虚拟环境并安装django。

在**myproject**文件夹中：

```
virtualenv venv -p python3
```

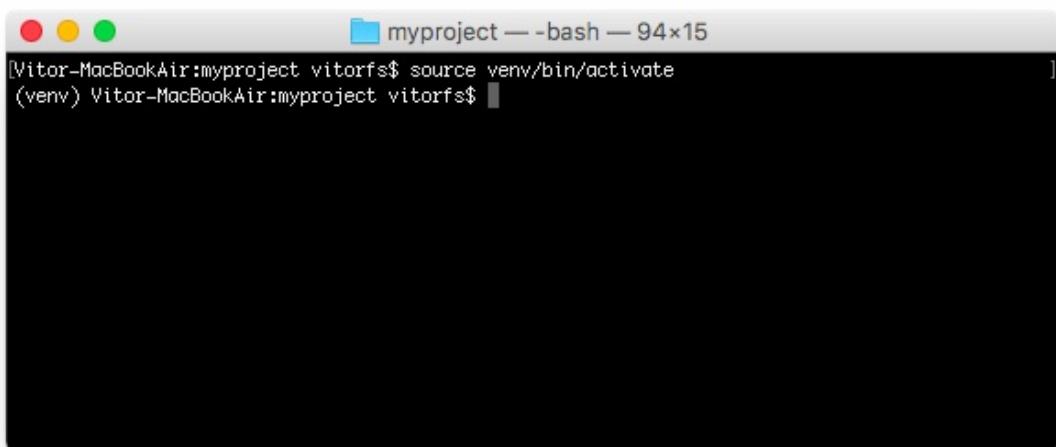


```
[Vitor-MacBookAir:myproject vitorfs$ virtualenv venv -p python3
Running virtualenv with interpreter /usr/local/bin/python3
Using base prefix '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/Versions/3.6'
New python executable in /Users/vitorfs/Development/myproject/venv/bin/python3.6
Also creating executable in /Users/vitorfs/Development/myproject/venv/bin/python
Installing setuptools, pip, wheel...done.
Vitor-MacBookAir:myproject vitorfs$
```

这样我们的虚拟环境就创建好了。在开始使用它之前，我们需要先激活一下环境：

```
source venv/bin/activate
```

如果你在命令行前面看到 (**venv**)，就代表激活成功了，就像这样：



让我们试着了解一下这里发生了什么。我们创建了一个名为**venv**的特殊文件夹。该文件夹内包含了一个python的副本。在我们激活了**venv**环境之后，当我们运行 `Python` 命令时，它将使用我们存储在**venv**里面的本地副本，而不是我们之前在操作系统中安装的那个。

另一个重要的事情是，`pip`程序也已经安装好了，当我们使用它来安装Python的软件包（比如Django）时，它将被安装在**venv**环境中。

请注意，当我们启用**venv**时，我们将使用命令 `python`（而不是 `python3`）来调用Python 3.6.2，并且仅使用 `pip`（而不是 `pip3`）来安装软件包。

顺便说一句，要想退出**venv**环境，运行下面的命令：

```
deactivate
```

但是，我们现在先保持激活状态来进行下一步。

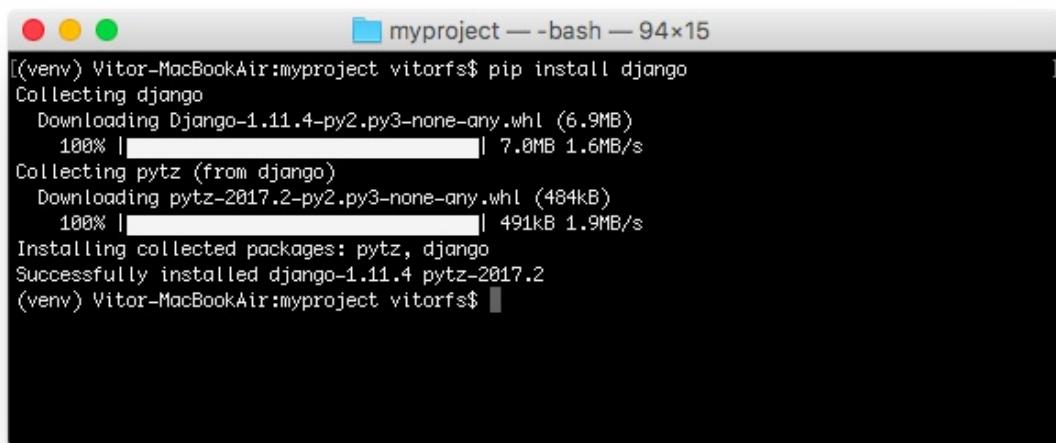
安装 Django 1.11.4

很简单，现在我们已经启动了venv，运行以下命令来安装django：

```
pip install django==1.11.4
```

译注：目前django已经升级到2.x版本，这里为了跟后续教程内容保持一致，所以必须指定版本号进行安装

除非你有能力 debug，否则不建议你使用django 2.x，等你熟悉Django后，再升级都Django2.0 也不迟，因为两个版本的差异非常小



```
(venv) Vitor-MacBookAir:myproject vitorfs$ pip install django
Collecting django
  Downloading Django-1.11.4-py2.py3-none-any.whl (6.9MB)
    100% |████████████████████████████████| 7.0MB 1.6MB/s
Collecting pytz (from django)
  Downloading pytz-2017.2-py2.py3-none-any.whl (484kB)
    100% |███████████████████████████████| 491kB 1.9MB/s
Installing collected packages: pytz, django
Successfully installed django-1.11.4 pytz-2017.2
(venv) Vitor-MacBookAir:myproject vitorfs$
```

现在一切就绪！



Django入门指南-第2章：创建项目

启动一个新项目

执行下面的命令来创建一个新的 Django 项目：

```
django-admin startproject myproject
```

命令行工具**django-admin**会在安装Django的时候一起自动安装好。

执行了上面的命令以后，系统会为Django项目生成基础文件夹结构。

现在，我们的**myproject**目录结构如下所示：

```
myproject/                                <-- 高级别的文件夹
| -- myproject/                            <-- Django项目文件夹
|   | -- myproject/
|   |   | -- __init__.py
|   |   | -- settings.py
|   |   | -- urls.py
|   |   | -- wsgi.py
|   |   +-- manage.py
+-- venv/                                    <-- 虚拟环境文件夹
```

我们最初的项目结构由五个文件组成：

- **manage.py**: 使用**django-admin**命令行工具的快捷方式。它用于运行与我们项目相关的管理命令。我们将使用它来运行开发服务器，运行测试，创建迁移等等。
- **__init__.py**: 这个空文件告诉python这个文件夹是一个python包。
- **settings.py**: 这个文件包含了所有的项目配置。将来我们会一直提到这个文件！
- **urls.py**: 这个文件负责映射我们项目中的路由和路径。例如，如果你想在访问URL / about/ 时显示某些内容，则必须先在这里做映射关系。

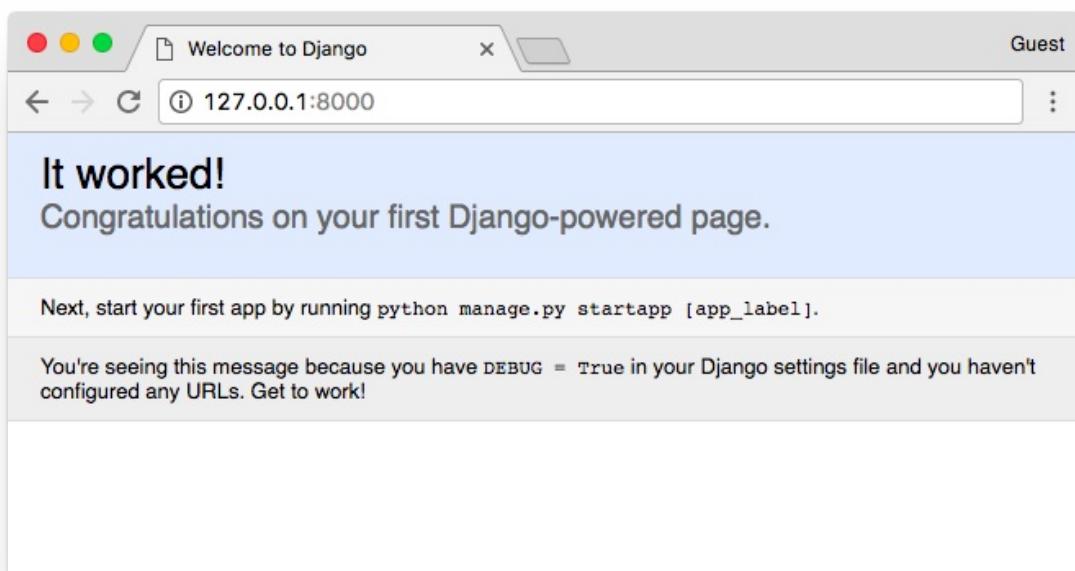
- **wsgi.py**: 该文件是用于部署的简单网关接口。你可以暂且先不用关心她的内容，就先让他在那里就好了。

django自带了一个简单的网络服务器。在开发过程中非常方便，所以我们无需安装任何其他软件即可在本地运行项目。我们可以通过执行命令来测试一下它：

```
python manage.py runserver
```

现在，你可以忽略终端中出现的迁移错误;我们将在稍后讨论。

现在在Web浏览器中打开URL: <http://127.0.0.1:8000>，你应该看到下面的页面：



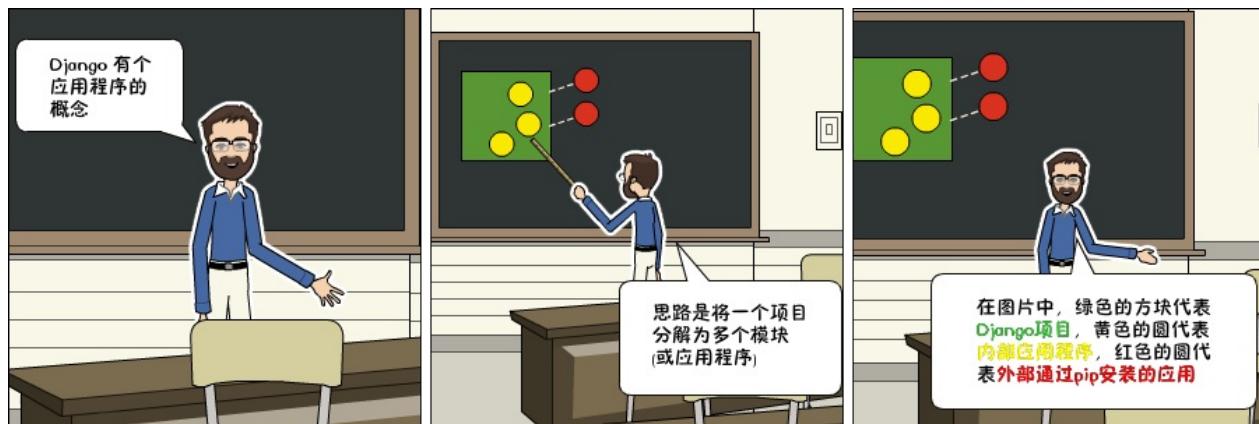
使用组合键 `Control + C` 来终止开发服务器。

Django 应用

在Django的哲学中，我们有两个重要的概念：

- **app**: 是一个可以做完某件事情的Web应用程序。一个应用程序通常由一组**models(数据库表)**, **views(视图)**, **templates(模板)**, **tests(测试)**组成。
- **project**: 是配置和应用程序的集合。一个项目可以由多个应用程序或一个应用程序组成。

请注意，如果没有一个**project**，你就无法运行Django应用程序。像博客这样的简单网站可以完全在单个应用程序中编写，例如可以将其命名为**blog**或**weblog**。



这是组织源代码的一种方式。现在刚开始，判断什么是或不是应用程序这些还不太重要。包括如何组织代码等。现在不用担心那些问题！首先让我们对Django的API和基础知识进行梳理一遍。

好的！那么，为了方便说明，我们来创建一个简单的网络论坛或讨论区。要创建我们的第一个应用程序，请跳转到**manage.py**文件所在的目录并执行以下命令：

```
django-admin startapp boards
```

注意！我们这次使用的命令是**startapp**。

通过这条指令，系统会给我们创建以下目录结构：

```
myproject/
| -- myproject/
|   | -- boards/           <-- 我们新的Django应用 (app) !
|   |   | -- migrations/
|   |   |   +-- __init__.py
|   |   |   | -- __init__.py
|   |   |   | -- admin.py
|   |   |   | -- apps.py
|   |   |   | -- models.py
|   |   |   | -- tests.py
|   |   |   +-- views.py
|   |   | -- myproject/
|   |   |   | -- __init__.py
|   |   |   | -- settings.py
|   |   |   | -- urls.py
|   |   |   | -- wsgi.py
|   |   +-- manage.py
+-- venv/
```

下面，我们来探讨每个文件的作用：

- **migrations/**: 在这个文件夹里，Django会存储一些文件以跟踪你在**models.py**文件中创建的变更，用来保持数据库和**models.py**的同步。
- **admin.py**: 这个文件为一个django内置的应用程序**Django Admin**的配置文件。
- **apps.py**: 这是应用程序本身的配置文件。
- **models.py**: 这里是我们定义Web应用程序数据实例的地方。models会由Django自动转换为数据库表。
- **tests.py**: 这个文件用来写当前应用程序的单元测试。
- **views.py**: 这是我们处理Web应用程序请求(request)/响应(resopnse)周期的文件。

现在我们创建了我们的第一个应用程序，让我们来配置一下项目以便启用这个应用程序。

要做到这一点，打开**settings.py**并尝试找到 `INSTALLED_APPS` 变量：

settings.py

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

如你所见，Django默认已经安装了6个内置应用程序。它们提供大多数Web应用程序所需的常用功能，如身份验证，会话，静态文件管理（图像，JavaScript，CSS等）等。

我们将会在本系列教程中探索这些应用程序。但现在，先不管它们，只需将我们的应用程序**boards**添加到 `INSTALLED_APPS` 列表即可：

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
  
    'boards',  # 译者注：建议和作者一样空一行来区别内置app和自定义的app  
]
```

使用前面漫画正方形和圆圈的比喻，黄色的圆圈就是我们的**boards**应用程序，**django.contrib.admin**, **django.contrib.auth**等就是红色的圆圈。

Django入门指南-第3章：Hello World

现在来写我们的第一个视图(view)。我们将在下一篇教程中详细探讨它。但现在，让我们试试看看如何用Django创建一个新页面。

打开**boards**应用程序中的**views.py**文件，并添加以下代码：

views.py

```
from django.http import HttpResponse

def home(request):
    return HttpResponse('Hello, World!')
```

视图是接收 `httprequest` 对象并返回一个 `httpresponse` 对象的Python函数。接收 `request` 作为参数并返回 `response` 作为结果。这个流程你必须记住！

我们在这里定义了一个简单的视图，命名为**home**，它只是简单地返回一个信息，一个字符串**hello, world!**。

现在我们必须告诉Django什么时候会调用这个view。这需要在**urls.py**文件中完成：

urls.py

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

from boards import views

urlpatterns = [
    url(r'^$', views.home, name='home'),
    url(r'^admin/', admin.site.urls),
]
```

如果你将上面的代码片段与你的**urls.py**文件进行比较，你会注意到我添加了以下新代码：`url(r'^$', views.home, name='home')` 并通过 `from boards import views` 从我们的应用程序**boards**中导入了**views**模块。

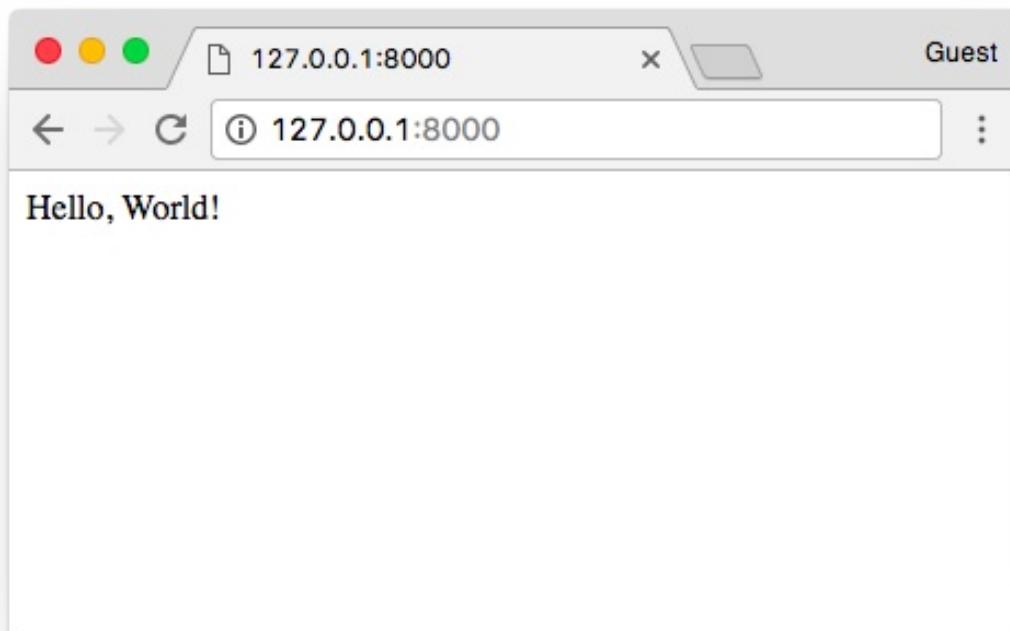
和我之前提到的一样，我们将在稍后详细探讨这些概念。

现在，Django使用正则表达式来匹配请求的URL。对于我们的**home**视图，我使用 `^$` 正则，它将匹配一个空路径，也就是主页（这个URL：<http://127.0.0.1:8000>）。如果我想匹配的URL是 <http://127.0.0.1:8000/homepage/>，那么我的URL正则表达式就会是：`url(r'^homepage/$', views.home, name='home')`。

我们来看看会发生什么：

```
python manage.py runserver
```

在一个Web浏览器中，打开 <http://127.0.0.1:8000> 这个链接：



就是这样！你刚刚成功创建了你的第一个视图。

总结

这是本系列教程的第一部分。在本教程中，我们学习了如何安装最新的Python版本以及如何设置开发环境。我们还介绍了虚拟环境，开始了我们的第一个django项目，并已经创建了我们的初始应用程序。

我希望你会喜欢第一部分！第二部分将于2017年9月11日下周发布。它将涉及模型，视图，模板和URLs。我们将一起探索Django所有的基础知识！如果您希望在第二部分发布时收到通知，可以订阅我们的[邮件列表](#)。

为了让我们能够保持学习过程中页面同步，我在Github上提供了源代码。这个项目的当前状态可以在**release tag v0.1-lw**下找到。下面是直达链接：

<https://github.com/sibtc/django-beginners-guide/tree/v0.1-lw>

Django入门指南-第4章：系统设计

前言

欢迎来到 Django 教程的第二节，在第一节中，我们安装了项目所需要的一切，希望你安装的是 Python3.6，并且在虚拟环境中运行 Django1.11，这节课我们继续在这个项目上编写代码。

咱们先讨论一些项目的背景知识，然后再学习 Django 的基础，包括：模型（models），管理后台（admin），视图（views），模板（templates），和路由（URLs）

动手吧！

论坛项目

我不知道你是怎样认为的，个人觉得，通过看实际例子和代码片段可以学到东西，但就我个人而言，当你在例子中读到诸如 class A 和 class B 这样的代码，或者看到诸如 foo(bar) 这样的例子时，是很难解释清楚这些概念的，所以，我不想让你这样做。（译注：作者要表达的意思是光写些 demo 例子意义并不大，而是要做些实际的项目才有帮助）

所以，在进入模型，视图等其它有趣的部分之前，先让我们花点时间，简要地讨论我们将要开发的这个项目。

如果你已经有了 Web 开发的经验并且觉得它太繁琐了，那么你可以浏览一下图片以了解我们将要构建的内容，然后直接跳转到本教程的模型部分。

但是如果你对 Web 开发不熟悉，我强烈建议你继续阅读下去。我将为你提供关于 Web 应用程序建模和设计上的一些见解。Web 开发和软件开发可不仅仅是编码。



我们的项目是一个论坛系统，整个项目的构思是维护几个论坛版块（boards），每个版块像一个分类一样。在指定的版块里面，用户可以通过创建新主题（Topic）开始讨论，其他用户可以参与讨论回复。

我们需要找到一种方法来区分普通用户和管理员用户，因为只有管理员可以创建版块。下图概述了主要的用例和每种类型的用户角色：

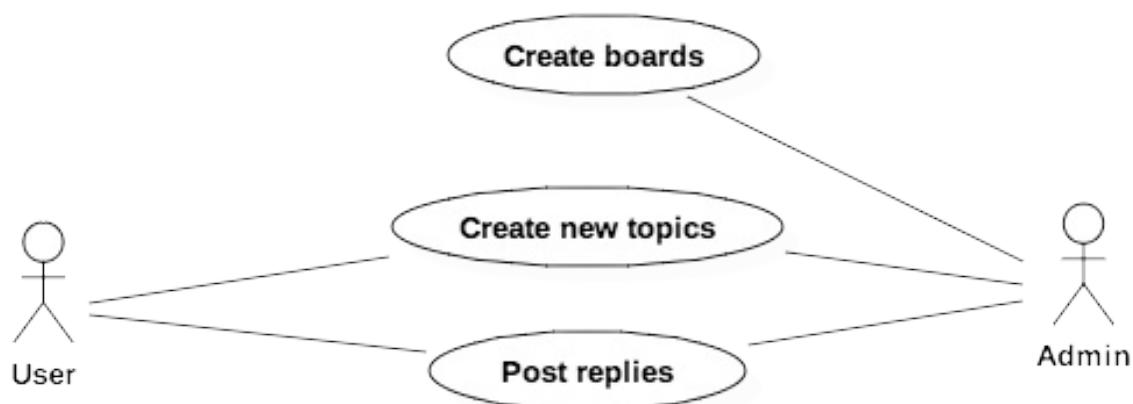


图1：Web Board提供的核心功能用例图

类图

从用例图中，我们可以开始思考项目所需的实体类有哪些。这些实体就是我们要创建的模型，它与我们的Django应用程序处理的数据非常密切。

为了能够实现上面描述的用例，我们需要至少实现下面几个模型：Board, Topic, Post和User。

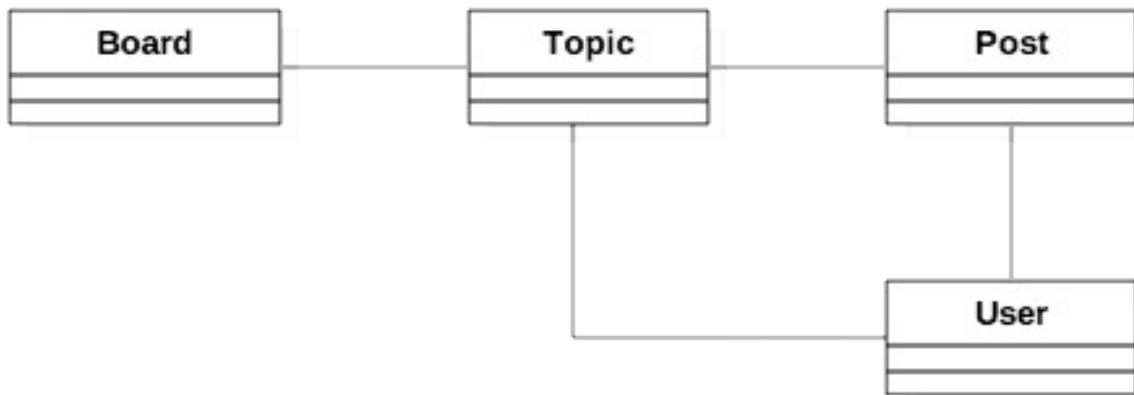


图2：Web Board类图

- Board: 版块
- Topic: 主题
- Post: 帖子（译注：其实就是主题的回复或评论）

花点时间考虑模型之间如何相互关联也很重要。类与类之间的实线告诉我们，在一个主题（Topic）中，我们需要有一个字段（译注：其实就是通过外键来关联）来确定它属于哪个版块（Board）。同样，帖子（Post）也需要一个字段来表示它属于哪个主题，这样我们就可以列出在特定主题内创建的帖子。最后，我们需要一个字段来表示主题是谁发起的，帖子是谁发的。

用户和版块之间也有联系，谁创建的版块。但是这些信息与应用程序无关。还有其他方法可以跟踪这些信息，稍后您会看到。

现在我们的类图有基本的表现形式，我们还要考虑这些模型将承载哪些信息。这很容易让事情变得复杂，所以试着先把重要的内容列出来，这些内容是我们启动项目需要的信息。后面我们再使用 Django 的迁移（Migrations）功能来改进模型，您将在下一节中详细了解这些内容。

但就目前而言，这是模型最基本的内容：

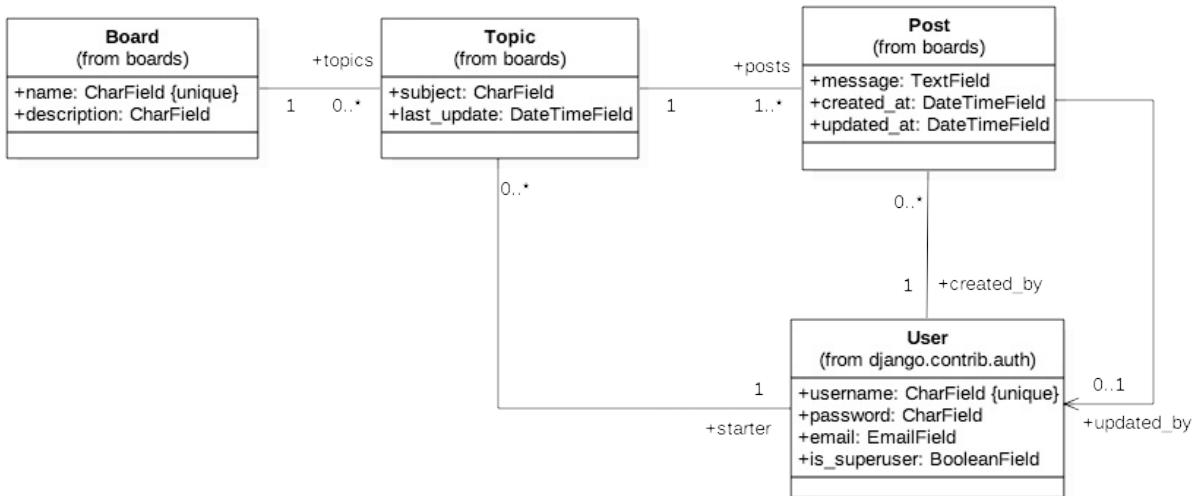


图3：强调类（模型）之间关系的类图

这个类图强调的是模型之间的关系，这些线条和箭头最终会在稍后转换为字段。

对于 **Board** 模型，我们将从两个字段开始：name 和 description。name 字段必须是唯一的，为了避免有重复的名称。description 用于说明这个版块是做什么用的。

Topic 模型包括四个字段：subject 表示主题内容，last_update 用来定义话题的排序，starter 用来识别谁发起的话题，board 用于指定它属于哪个版块。

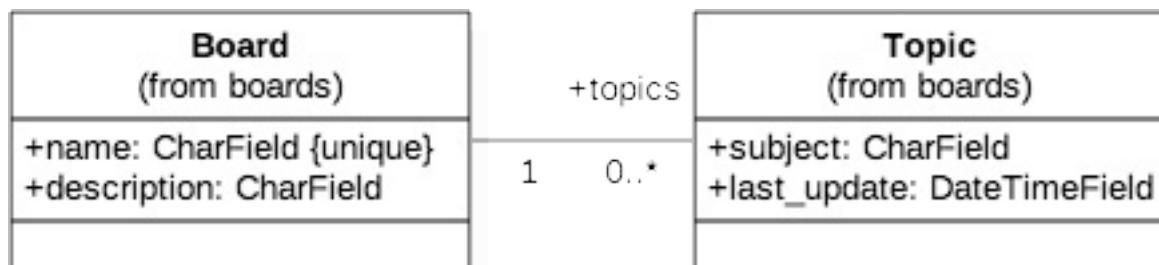
Post 模型有一个 message 字段，用于存储回复的内容，created_at 在排序时候用（最先发表的帖子排最前面），updated_at 告诉用户是否更新了内容，同时，还需要有对应的 User 模型的引用，Post 由谁创建的和谁更新的。

最后是 **User** 模型。在类图中，我只提到了字段 username, password, email, is_superuser 标志，因为这几乎是我们现在要使用的所有东西。

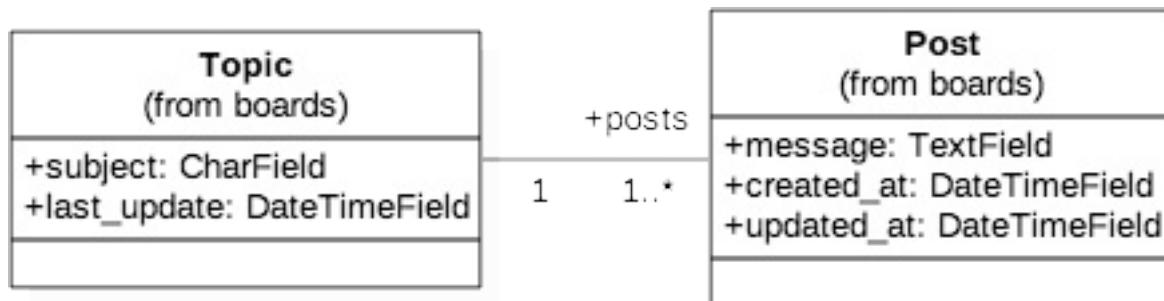
需要注意的是，我们不需要创建 User 模型，因为 Django 已经在 contrib 包中内置了 User 模型，我们将直接拿来用。

关于类图之间的对应关系（数字 1, 0..* 等等），这里教你如何阅读：

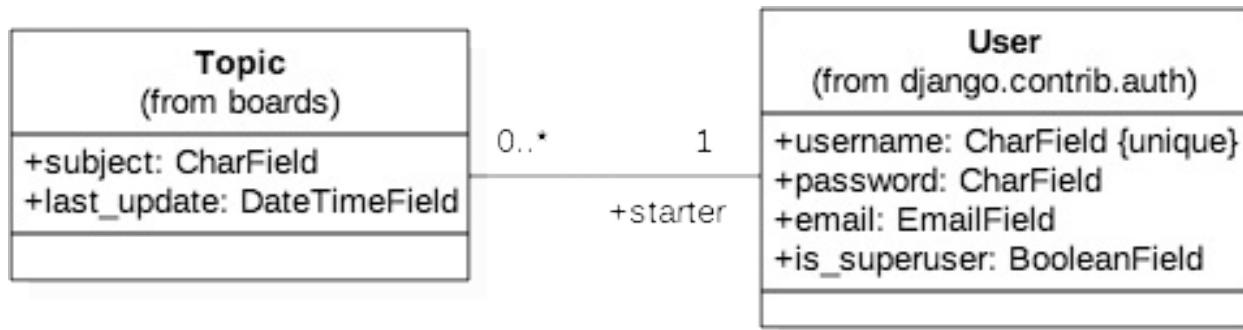
一个topic 必须与一个 (1) Board (这意味着它不能为空) 相关联，但是 Board 下面可能与许多个或者0个 topic 关联 (0..*)。这意味着 Board 下面可能没有主题。 (译注：一对多关系)



一个 Topic 至少有一个 Post (发起话题时，同时会发布一个帖子)，并且它也可能有许多 Post (1..*)。一个Post 必须与一个并且只有一个Topic (1) 相关联。



一个 Topic 必须有一个且只有一个 User 相关联，topic 的发起者是 (1) 。而一个用户可能有很多或者没有 topic (0..*) 。



Post 必须有一个并且只有一个与之关联的用户，用户可以有许多或没有 Post (0..*) 。Post 和 User之间的第二个关联是直接关联 (参见该行最后的箭头)，就是 Post 可以被用户修改 (updated_by) ，updated_by 有可能是空 (Post 没有被修改)

画这个类图的另一种方法是强调字段而不是模型之间的关系：

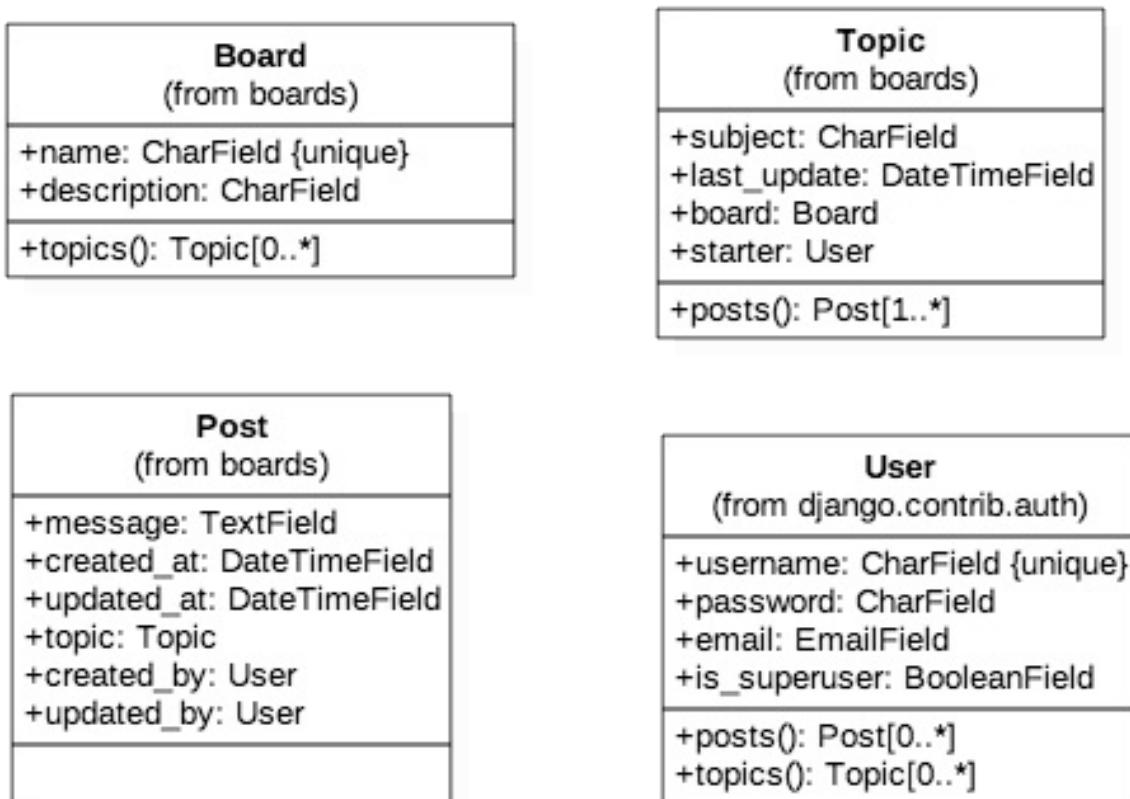


图4：强调类（模型）与属性（字段）的类图

上面的表示方式与前面的表示方式是对等的，不过这种方式更接近我们将要使用 Django Models API 设计的内容。在这种表示方式中，我们可以更清楚地看到，在 Post 模型中，关联了 Topic, created_by (创建者) 和 updated_by (更新者) 字段。另一个值得注意的事情是，在 Topic 模型中，有一个名为 posts () 的操作 (一个类方法)。我们将通过反向关系来实现这一目标，Django 将自动在数据库中执行查询以返回特定主题的所有帖子列表。

好了，现在已经够UML了！为了绘制本节介绍的图表，我使用了 [StarUML](#) 工具。

线框图（原型图）

花了一些时间来设计应用程序的模型后，我喜欢创建一些线框来定义需要完成的工作，并且清楚地了解我们将要做什么。



基于线框图，我们可以更深入地了解应用程序中涉及的实体。

首先，我们需要在主页上显示所有版块：

A wireframe of a web page titled 'Boards'. At the top, there's a header with a 'Boards' button, a search bar containing 'https://www.example.com', and three circular icons. The main content area is titled 'Boards' and contains a table with three rows. The table has columns for 'Board', 'Posts', 'Topics', and 'Last Post'.

Board	Posts	Topics	Last Post
Python Everything related to Python goes here.	287	112	2017-08-05 18:02 by user1
Django Board dedicated to Django and its libraries.	398	276	2017-08-05 17:42 by user1
Random You can post about everything here! Really!	24	5	2017-08-04 23:23 by user2

图5：论坛项目线框主页列出所有可用的版块。

如果用户点击一个链接，比如点击Django版块，它应该列出所有Django相关的主题：

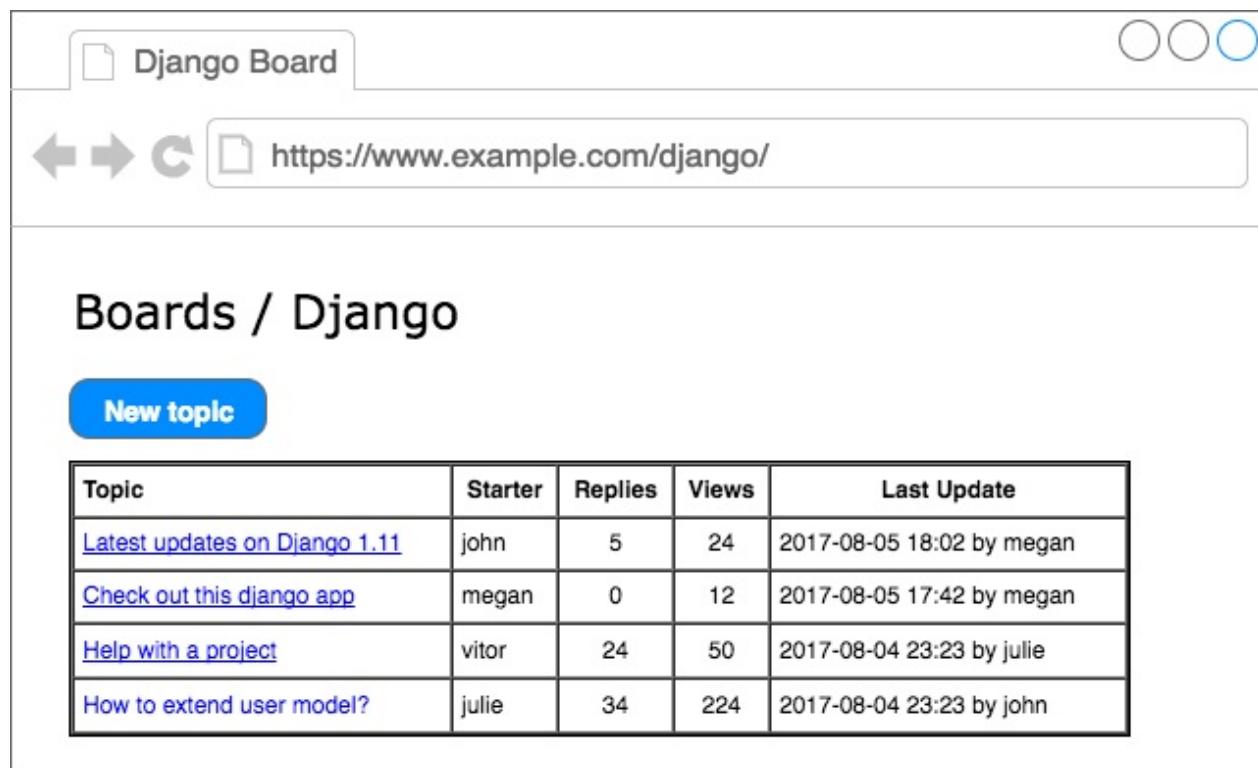
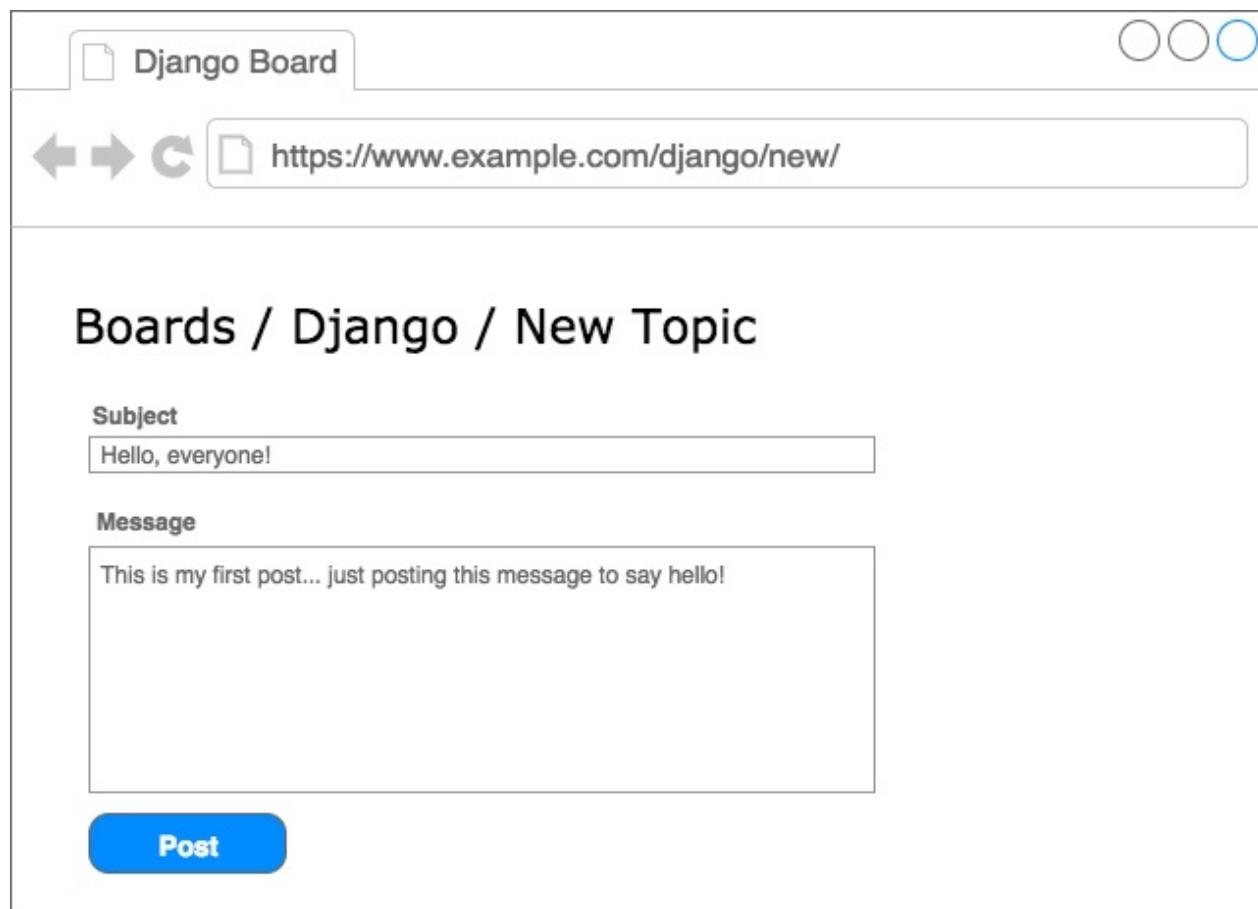


图6：论坛项目线框图列出了Django版块中的所有主题

这里有两个入口：用户点击“new topic”按钮创建新主题，或者点击主题链接查看或参与讨论。

“new topic” 页面：

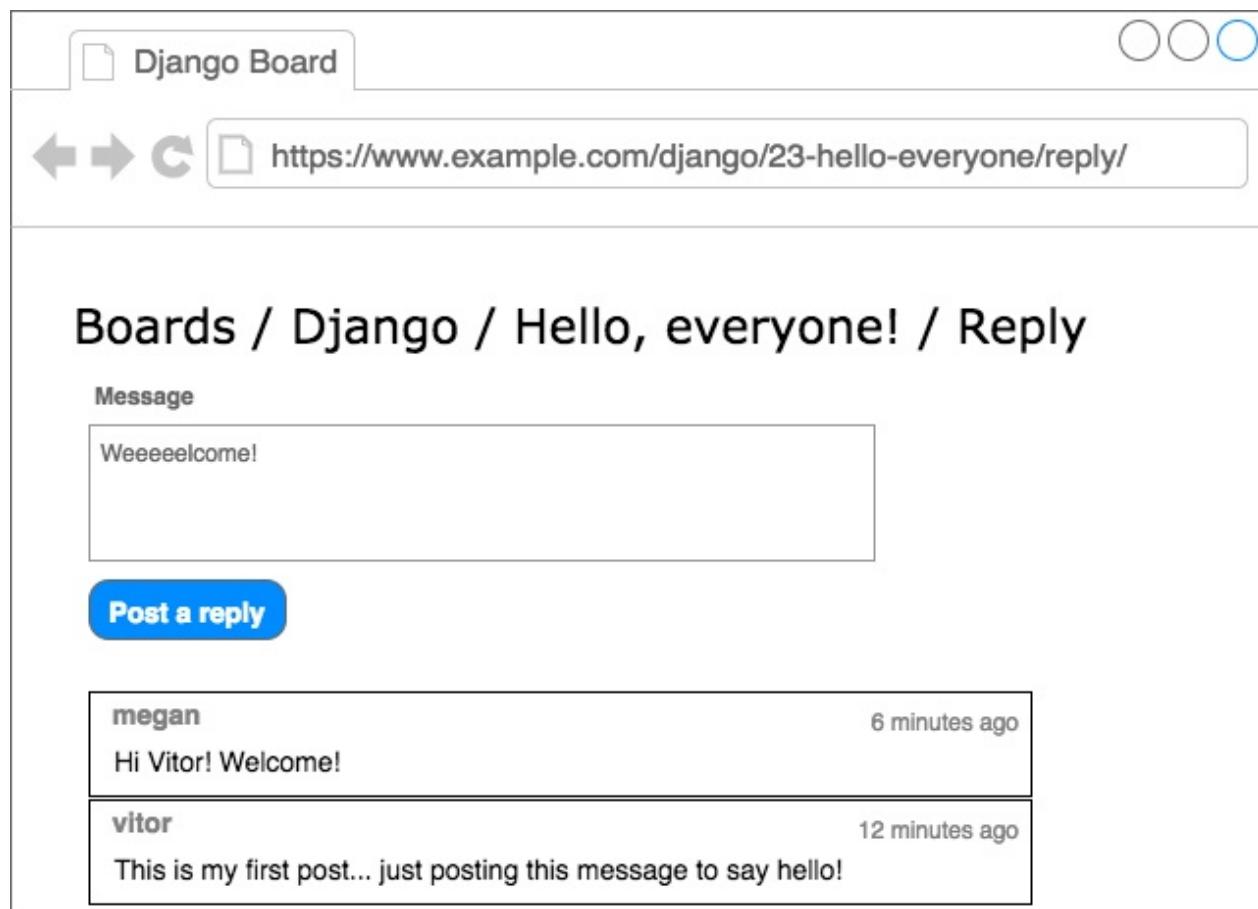


现在，主题页面显示了帖子和讨论：

The screenshot shows a web browser window with the title "Django Board". The URL in the address bar is "https://www.example.com/django/23-hello-everyone/". The main content area displays a board titled "Boards / Django / Hello, everyone!". A blue "Reply" button is visible. Two posts are listed:

- vitor** (12 minutes ago)
This is my first post... just posting this message to say hello!
Posts: 1 [EDIT](#)
- megan** (6 minutes ago)
Hi Vitor! Welcome!
Posts: 2k

如果用户点击回复按钮，将看到下面这个页面，并以倒序的方式（最新的在第一个）显示帖子列表：



绘制这些线框，你可以使用[draw.io](#)服务，它是免费的。

Django入门指南-第5章：模型设计



模型

这些模型基本上代表了应用程序的数据库设计。我们在本节中要做的是创建 Django 所表示的类，这些类就是在上一节中建模的类：Board, Topic 和 Post。User 模型被命名为内置应用叫 **auth**，它以命名空间 `django.contrib.auth` 的形式出现在 `INSTALLED_APPS` 配置中。

我们要做的工作都在 `boards/models.py` 文件中。以下是在 Django 应用程序中如何表示类图的代码：

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

class Board(models.Model):
    name = models.CharField(max_length=30, unique=True)
    description = models.CharField(max_length=100)

class Topic(models.Model):
    subject = models.CharField(max_length=255)
    last_updated = models.DateTimeField(auto_now_add=True)
    board = models.ForeignKey(Board, related_name='topics')
    starter = models.ForeignKey(User, related_name='topics')

class Post(models.Model):
    message = models.TextField(max_length=4000)
    topic = models.ForeignKey(Topic, related_name='posts')
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(null=True)
    created_by = models.ForeignKey(User, related_name='posts')
    updated_by = models.ForeignKey(User, null=True, related_name='+' + '')
```

所有模型都是**django.db.models.Model**类的子类。每个类将被转换为数据库表。每个字段由 **django.db.models.Field**子类（内置在Django core）的实例表示，它们并将被转换为数据库的列。

字段 CharField, DateTimeField等等，都是 **django.db.models.Field** 的子类，包含在Django的核心里面-随时可以使用。

在这里，我们仅使用 CharField, TextField, DateTimeField, 和ForeignKey 字段来定义我们的模型。不过在Django提供了更广泛的选择来代表不同类型的数据，例如 IntegerField, BooleanField, DecimalField和其它一些字段。

我们会在需要的时候提及它们。

有些字段需要参数，例如CharField。我们应该始终设定一个 `max_length`。这些信息将用于创建数据库列。Django需要知道数据库列需要多大。该 `max_length` 参数也将被Django Forms API用来验证用户输入。

在 `Board` 模型定义中，更具体地说，在 `name` 字段中，我们设置了参数 `unique=True`，顾名思义，它将强制数据库级别字段的唯一性。

在 `Post` 模型中，`created_at` 字段有一个可选参数，`auto_now_add` 设置为 `True`。这将告诉Django创建 `Post` 对象时为当前日期和时间。

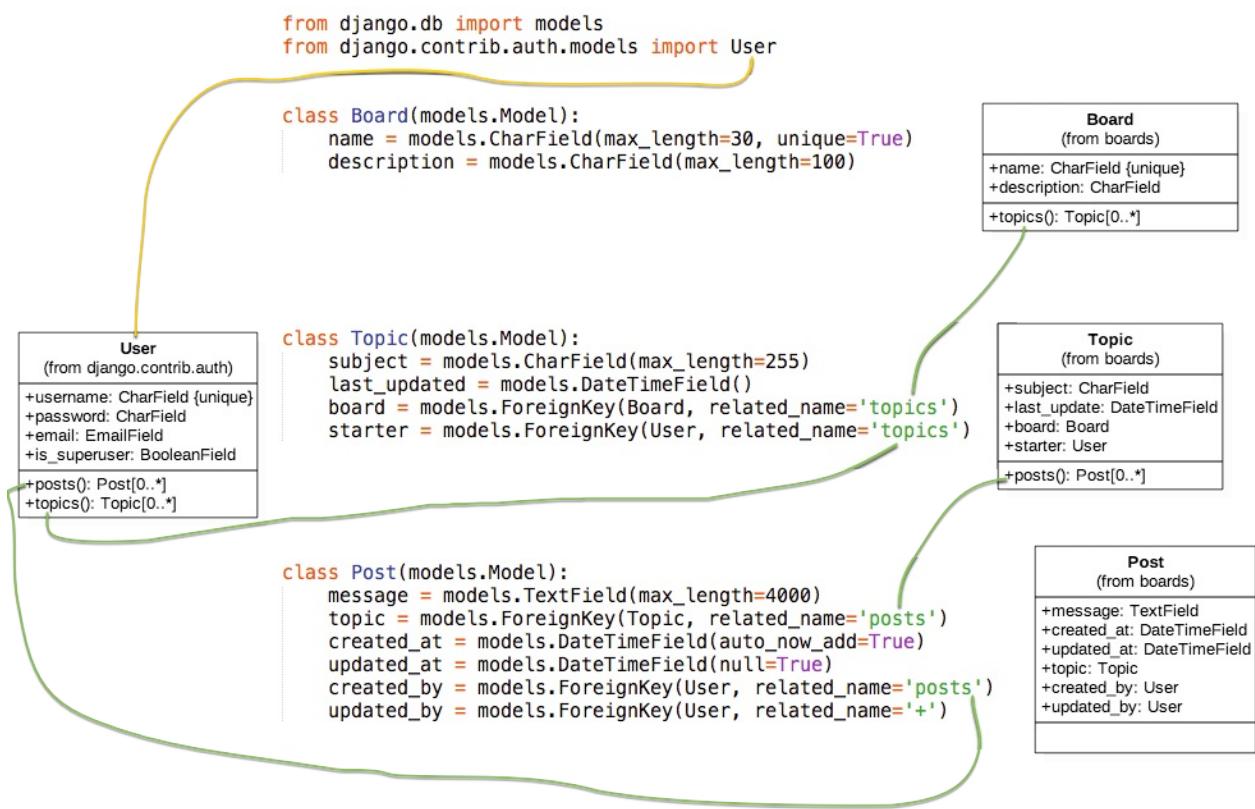
模型之间的关系使用 `ForeignKey` 字段。它将在模型之间创建一个连接，并在数据库级别创建适当的关系（译注：外键关联）。该 `ForeignKey` 字段需要一个位置参数 `related_name`，用于引用它关联的模型。（译注：例如 `created_by` 是外键字段，关联的User模型，表明这个帖子是谁创建的，`related_name=posts` 表示在 User 那边可以使用 `user.posts` 来查看这个用户创建了哪些帖子）

例如，在 `Topic` 模型中，`board` 字段是 `Board` 模型的 `ForeignKey`。它告诉Django，一个 `Topic` 实例只涉及一个 `Board` 实例。`related_name` 参数将用于创建反向关系，`Board` 实例通过属性 `topics` 访问属于这个版块下的 `Topic` 列表。

Django自动创建这种反向关系，`related_name` 是可选项。但是，如果我们不为它设置一个名称，Django会自动生成它：`(class_name)_set`。例如，在 `Board` 模型中，所有 `Topic` 列表将用 `topic_set` 属性表示。而这里我们将其重新命名为 `topics`，以使其感觉更自然。

在 `Post` 模型中，该 `updated_by` 字段设置 `related_name='+'`。这指示 Django我们不需要这种反向关系，所以它会被忽略（译注：也就是说我们不需要关系用户修改过哪些帖子）。

下面您可以看到类图和Django模型的源代码之间的比较，绿线表示我们如何处理反向关系。



这时，你可能会问自己：“主键/ ID呢？”？如果我们没有为模型指定主键，Django会自动为我们生成它。所以现在一切正常。在下一节中，您将看到它是如何工作的。

迁移模型

下一步是告诉Django创建数据库，以便我们可以开始使用它。

打开终端，激活虚拟环境，转到 manage.py文件所在的文件夹，然后运行以下命令：

```
python manage.py makemigrations
```

你会看到输出的内容是：

```
Migrations for 'boards':  
  boards/migrations/0001_initial.py  
    - Create model Board  
    - Create model Post  
    - Create model Topic  
    - Add field topic to post  
    - Add field updated_by to post
```

此时，Django 在 boards/migrations 目录创建了一个名为 0001_initial.py 的文件。它代表了应用程序模型的当前状态。在下一步，Django 将使用该文件创建表和列。

迁移文件将被翻译成SQL语句。如果您熟悉SQL，则可以运行以下命令来检验将是要被数据库执行的SQL指令

```
python manage.py sqlmigrate boards 0001
```

如果你不熟悉SQL，也不要担心。在本系列教程中，我们不会直接使用SQL。所有的工作都将使用Django ORM来完成，它是一个与数据库进行通信的抽象层。

下一步是将我们生成的迁移文件应用到数据库：

```
python manage.py migrate
```

输出内容应该是这样的：

```
Operations to perform:
```

```
  Apply all migrations: admin, auth, boards, contenttypes, sessions
```

```
Running migrations:
```

```
  Applying contenttypes.0001_initial... OK
```

```
  Applying auth.0001_initial... OK
```

```
  Applying admin.0001_initial... OK
```

```
  Applying admin.0002_logentry_remove_auto_add... OK
```

```
  Applying contenttypes.0002_remove_content_type_name... OK
```

```
  Applying auth.0002_alter_permission_name_max_length... OK
```

```
  Applying auth.0003_alter_user_email_max_length... OK
```

```
  Applying auth.0004_alter_user_username_opts... OK
```

```
  Applying auth.0005_alter_user_last_login_null... OK
```

```
  Applying auth.0006_require_contenttypes_0002... OK
```

```
  Applying auth.0007_alter_validators_add_error_messages... 0
```

```
K
```

```
  Applying auth.0008_alter_user_username_max_length... OK
```

```
  Applying boards.0001_initial... OK
```

```
  Applying sessions.0001_initial... OK
```

因为这是我们第一次迁移数据库，所以 `migrate` 命令把Django contrib app 中现有的迁移文件也执行了，这些内置app列在了 `INSTALLED_APPS`。这是预料之中的。

`Applying boards.0001_initial... OK` 是我们在上一步中生成的迁移脚本。

好了！我们的数据库已经可以使用了。



需要注意的是SQLite是一个产品级数据库。SQLite被许多公司用于成千上万的产品，如所有Android和iOS设备，主流的Web浏览器，Windows 10，MacOS等。

但这不适合所有情况。SQLite不能与MySQL，PostgreSQL或Oracle等数据库进行比较。大容量的网站，密集型写入的应用程序，大的数据集，高并发性的应用使用SQLite最终都会导致问题。

我们将在开发项目期间使用SQLite，因为它很方便，不需要安装其他任何东西。当我们将项目部署到生产环境时，再将切换到PostgreSQL（译注：后续，我们后面可能使用MySQL）。对于简单的网站这种做法没什么问题。但对于复杂的网站，建议在开发和生产中使用相同的数据库。

试验 Models API

使用Python进行开发的一个重要优点是交互式shell。我一直在使用它。这是一种快速尝试和试验API的方法。

您可以使用manage.py工具加载我们的项目来启动Python shell：

```
python manage.py shell
```

```
Python 3.6.2 (default, Jul 17 2017, 16:44:45)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

这与直接输入`python`指令来调用交互式控制台是非常相似的，除此之外，项目将被添加到`sys.path`并加载Django。这意味着我们可以在项目中导入我们的模型和其他资源并使用它。

让我们从导入Board类开始：

```
from boards.models import Board
```

要创建新的 board 对象，我们可以执行以下操作：

```
board = Board(name='Django', description='This is a board about Django.')
```

为了将这个对象保存在数据库中，我们必须调用save方法：

```
board.save()
```

save 方法用于创建和更新对象。这里Django创建了一个新对象，因为这时 Board 实例没有id。第一次保存后，Django会自动设置ID：

```
board.id  
1
```

您可以将其余的字段当做Python属性访问：

```
board.name  
'Django'
```

```
board.description  
'This is a board about Django.'
```

要更新一个值，我们可以这样做：

```
board.description = 'Django discussion board.'  
board.save()
```

每个Django模型都带有一个特殊的属性；我们称之为**模型管理器(Model Manager)**。你可以通过属性 `objects` 来访问这个管理器，它主要用于数据库操作。例如，我们可以使用它来直接创建一个新的Board对象：

```
board = Board.objects.create(name='Python', description='General discussion about Python.')
```

```
board.id
```

```
2
```

```
board.name
```

```
'Python'
```

所以，现在我们有两个版块了。我们可以使用 `objects` 列出数据库中所有现有的版块：

```
Board.objects.all()  
<QuerySet [<Board: Board object>, <Board: Board object>]>
```

结果是一个QuerySet。稍后我们会进一步了解。基本上，它是从数据库中查询的对象列表。我们看到有两个对象，但显示的名称是 Board object。这是因为我们尚未实现 Board 的 `__str__` 方法。

`__str__` 方法是对象的字符串表示形式。我们可以使用版块的名称来表示它。

首先，退出交互式控制台：

```
exit()
```

现在编辑boards app 中的 `models.py` 文件：

```
class Board(models.Model):
    name = models.CharField(max_length=30, unique=True)
    description = models.CharField(max_length=100)

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

让我们重新查询，再次打开交互式控制台：

```
python manage.py shell

from boards.models import Board

Board.objects.all()
<QuerySet [<Board: Django>, <Board: Python>]>
```

好多了，对吧？

我们可以将这个QuerySet看作一个列表。假设我们想遍历它并打印每个版块的描述：

```
boards_list = Board.objects.all()
for board in boards_list:
    print(board.description)
```

结果是：

```
Django discussion board.
General discussion about Python.
```

同样，我们可以使用模型的 **管理器（Manager）** 来查询数据库并返回单个对象。为此，我们要使用 `get` 方法：

```
django_board = Board.objects.get(id=1)

django_board.name
'Django'
```

但我们必须小心这种操作。如果我们试图查找一个不存在的对象，例如，查找id=3的版块，它会引发一个异常：

```
board = Board.objects.get(id=3)

boards.models.DoesNotExist: Board matching query does not exist.
```

get 方法的参数可以是模型的任何字段，但最好使用可唯一标识对象的字段来查询。否则，查询可能会返回多个对象，这也会导致异常。

```
Board.objects.get(name='Django')
<Board: Django>
```

请注意，查询区分大小写，小写“django”不匹配：

```
Board.objects.get(name='django')
boards.models.DoesNotExist: Board matching query does not exist.
```

总结

下面是我们在本节中关于模型学到的方法和操作，使用Board模型作为参考。大写的 **Board**指的是类，小写的**board**指**Board**的一个实例（或对象）

操作	代码示例
创建一个对象而不保存	board = Board()
保存一个对象（创建或更新）	board.save()
数据库中创建并保存一个对象	Board.objects.create(name='...', description='...')
列出所有对象	Board.objects.all()
通过字段标识获取单个对象	Board.objects.get(id=1)

在下一小节中，我们将开始编写视图并在HTML页面中显示我们的版块。

Django入门指南-第6章：第一个视图函数

目前我们已经有一个视图函数叫 `home` ,这个视图在我们的应用程序主页上显示为“Hello, World! ”

myproject/urls.py

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

from boards import views

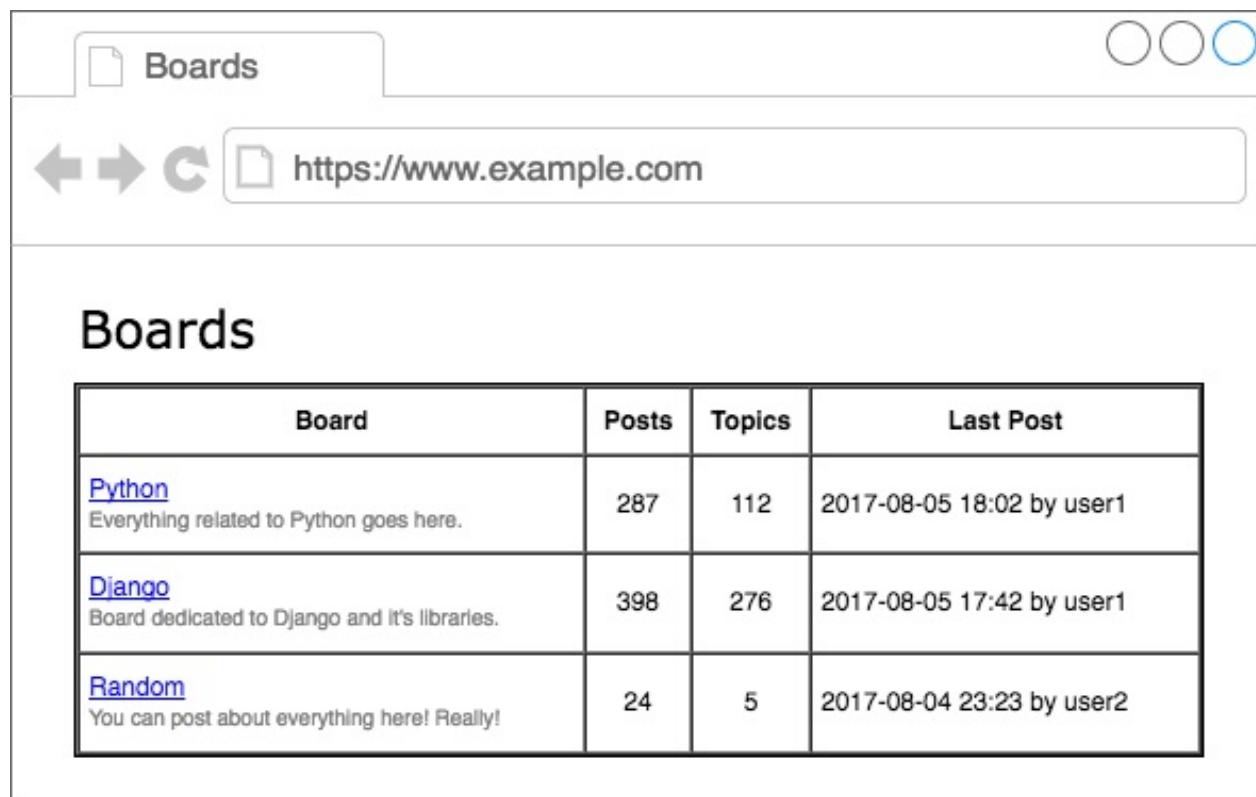
urlpatterns = [
    url(r'^$', views.home, name='home'),
    url(r'^admin/', admin.site.urls),
]
```

boards/views.py

```
from django.http import HttpResponse

def home(request):
    return HttpResponse('Hello, World!')
```

我们可以从这里开始写。如果你回想起我们的原型图，图5显示了主页应该是什么样子。我们想要做的是在表格中列出一些版块的名单以及它们的描述信息。



首先要做的是导入Board模型并列出所有的版块

boards/views.py

```
from django.http import HttpResponse
from .models import Board

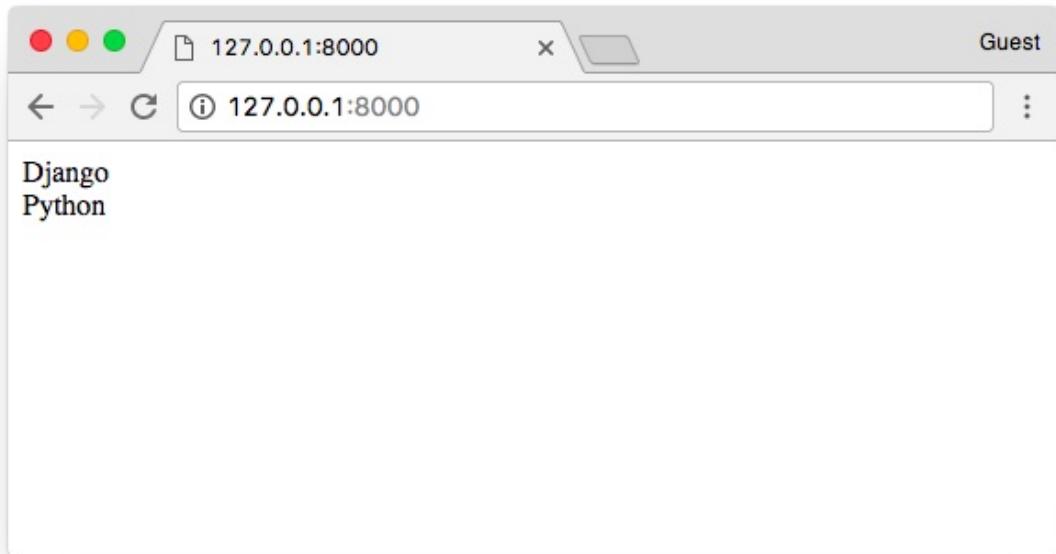
def home(request):
    boards = Board.objects.all()
    boards_names = list()

    for board in boards:
        boards_names.append(board.name)

    response_html = '<br>'.join(boards_names)

    return HttpResponse(response_html)
```

结果就是这个简单的HTML页面：



等等，我们在这里先停一下。真正的项目里面我们不会这样去渲染HTML。对于这个简单视图函数，我们做的就是列出所有版块，然后渲染部分是Django模板引擎的职责。

Django入门指南-第7章：模板引擎设置

在manage.py所在的目录创建一个名为 **templates**的新文件夹：

```
myproject/
| -- myproject/
|   | -- boards/
|   | -- myproject/
|   | -- templates/    <-- 这里
|   +-- manage.py
+-- venv/
```

在templates文件夹中， 创建一个名为home.html的HTML文件：

templates/home.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Boards</title>
  </head>
  <body>
    <h1>Boards</h1>

    {% for board in boards %}
      {{ board.name }} <br>
    {% endfor %}

  </body>
</html>
```

在上面的例子中，我们混入了原始HTML和一些特殊标签 `{% for ... in ... %}` 和 `{{ variable }}`。它们是Django模板语言的一部分。上面的例子展示了如何使用 `for` 遍历列表对象。`{{ board.name }}` 会在 HTML 模板中会被渲染成版块的名称，最后生成动态HTML文档。

在我们可以使用这个HTML页面之前，我们必须告诉Django在哪里可以找到我们应用程序的模板。

打开**myproject**目录下面的**settings.py**文件，搜索 `TEMPLATES` 变量，并设置 `DIRS` 的值为 `os.path.join(BASE_DIR, 'templates')`：

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

本质上，刚添加的这一行所做的事情就是找到项目的完整路径并在后面附加“/templates”

我们可以使用Python shell进行调试：

```
python manage.py shell
```

```
from django.conf import settings

settings.BASE_DIR
'/Users/vitorfs/Development/myproject'

import os

os.path.join(settings.BASE_DIR, 'templates')
'/Users/vitorfs/Development/myproject/templates'
```

看到了吗？它只是指向我们在前面步骤中创建的**templates**文件夹。

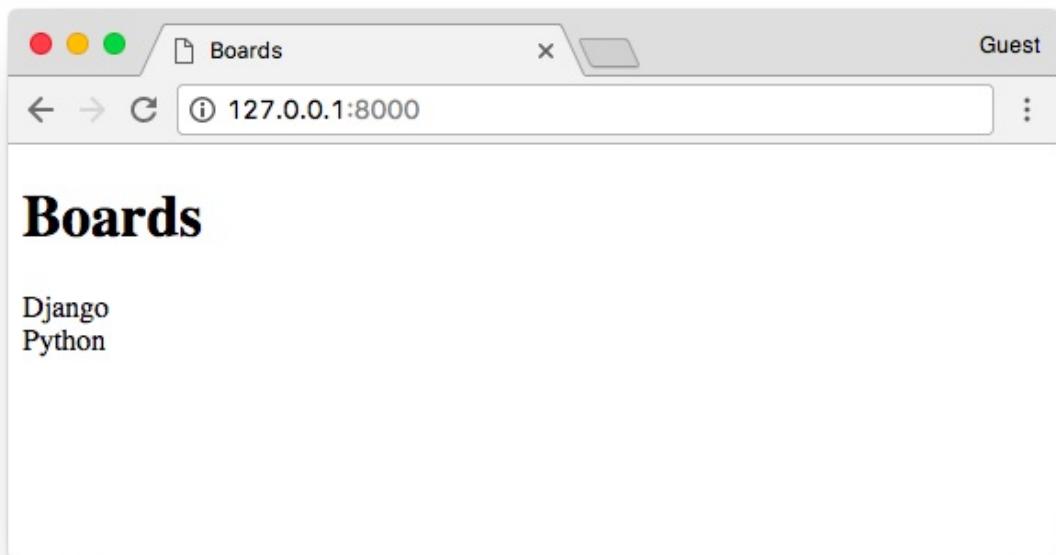
现在我们可以更新**home**视图：

boards/views.py

```
from django.shortcuts import render
from .models import Board

def home(request):
    boards = Board.objects.all()
    return render(request, 'home.html', {'boards': boards})
```

生成的HTML：

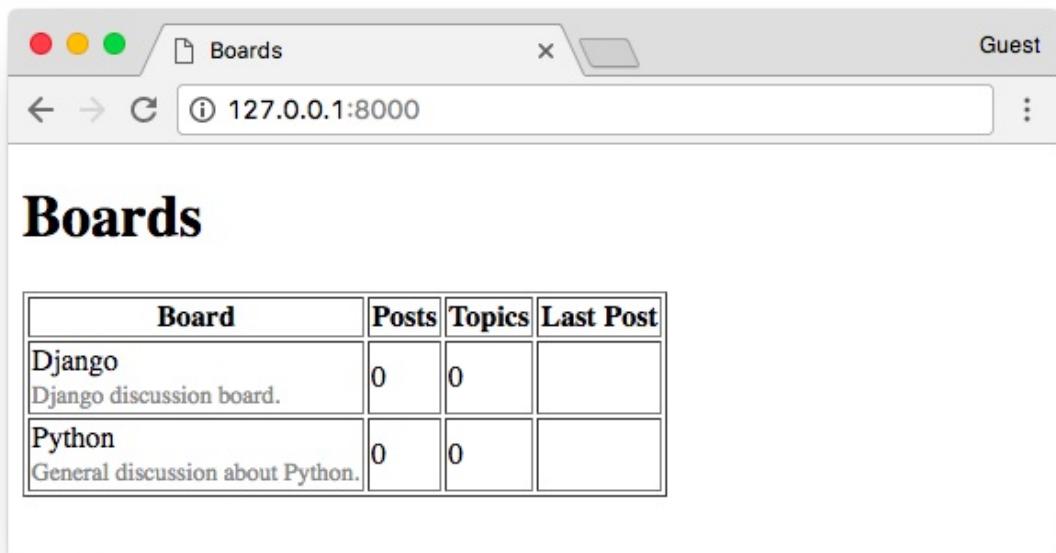


我们可以用一个更漂亮的表格来替换，改进HTML模板：

templates/home.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Boards</title>
    </head>
    <body>
        <h1>Boards</h1>

        <table border="1">
            <thead>
                <tr>
                    <th>Board</th>
                    <th>Posts</th>
                    <th>Topics</th>
                    <th>Last Post</th>
                </tr>
            </thead>
            <tbody>
                {% for board in boards %}
                    <tr>
                        <td>
                            {{ board.name }}<br>
                            <small style="color: #888">{{ board.description }}</small>
                        </td>
                        <td>0</td>
                        <td>0</td>
                        <td></td>
                    </tr>
                {% endfor %}
            </tbody>
        </table>
    </body>
</html>
```



Django入门指南-第8章：第一个测试用例

测试主页



测试将是一个反复出现的主题，我们将在整个教程系列中一起探讨不同的概念和策略。

我们来开始写第一个测试。现在，我们将在**boards**应用程序内的**tests.py**文件中操作

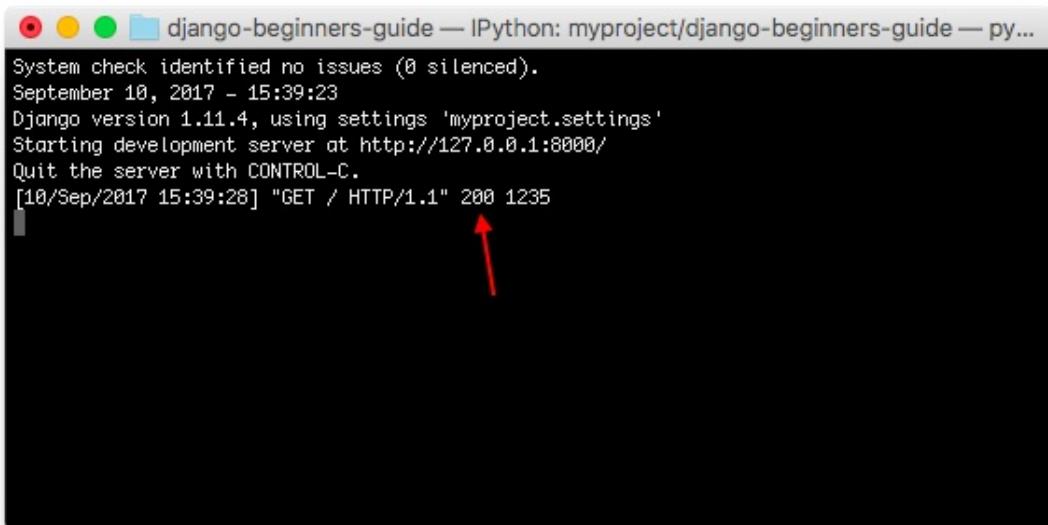
boards/tests.py

```
from django.core.urlresolvers import reverse
from django.test import TestCase

class HomeTests(TestCase):
    def test_home_view_status_code(self):
        url = reverse('home')
        response = self.client.get(url)
        self.assertEqual(response.status_code, 200)
```

这是一个非常简单但非常有用的测试用例，我们测试的是请求该URL后返回的响应状态码。状态码200意味着成功。

请求一下主页后，我们可以在控制台中看到响应的状态代码：



```
System check identified no issues (0 silenced).
September 10, 2017 - 15:39:23
Django version 1.11.4, using settings 'myproject.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
[10/Sep/2017 15:39:28] "GET / HTTP/1.1" 200 1235
```

如果出现未捕获的异常，语法错误或其他任何情况，Django会返回状态代码500，这意味着是内部服务器错误。现在，想象我们的应用程序有100个视图函数。如果我们为所有视图编写这个简单的测试，只需一个命令，我们就能够测试所有视图是否返回成功代码，因此用户在任何地方都看不到任何错误消息。如果没有自动化测试，我们需要逐一检查每个页面是否有错误。

执行Django的测试套件：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.
-----
.
Ran 1 test in 0.041s

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

现在我们可以测试Django是否在请求的URL的时候返回了正确的视图函数。这也是一个有用的测试，因为随着开发的进展，您会发现urls.py模块可能变得非常庞大而复杂。URL conf 全部是关于解析正则表达式的。有些情况下有一个非常宽容的URL（译注：本来不应该匹配的，却因为正则表达式写得过于宽泛而错误的匹配了），所以Django最终可能返回错误的视图函数。

我们可以这样做：

boards/tests.py

```
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase
from .views import home

class HomeTests(TestCase):
    def test_home_view_status_code(self):
        url = reverse('home')
        response = self.client.get(url)
        self.assertEqual(response.status_code, 200)

    def test_home_url_resolves_home_view(self):
        view = resolve('/')
        self.assertEqual(view.func, home)
```

在第二个测试中，我们使用了 `resolve` 函数。Django使用它来将浏览器发起请求的URL与urls.py模块中列出的URL进行匹配。该测试用于确定URL / 返回 home 视图。

再次测试：

```
python manage.py test
```

```
```sh
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

..

Ran 2 tests in 0.027s

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

要查看有关测试执行时更详细的信息，可将**verbosity**的级别设置得更高一点：

```
python manage.py test --verbosity=2
```

```
Creating test database for alias 'default' ('file:memorydb_default?mode=memory&cache=shared')...
Operations to perform:
 Synchronize unmigrated apps: messages, staticfiles
 Apply all migrations: admin, auth, boards, contenttypes, sessions
Synchronizing apps without migrations:
 Creating tables...
 Running deferred SQL...
Running migrations:
 Applying contenttypes.0001_initial... OK
 Applying auth.0001_initial... OK
 Applying admin.0001_initial... OK
 Applying admin.0002_logentry_remove_auto_add... OK
 Applying contenttypes.0002_remove_content_type_name... OK
 Applying auth.0002_alter_permission_name_max_length... OK
 Applying auth.0003_alter_user_email_max_length... OK
 Applying auth.0004_alter_user_username_opts... OK
 Applying auth.0005_alter_user_last_login_null... OK
```

```
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... 0
K
Applying auth.0008_alter_user_username_max_length... OK
Applying boards.0001_initial... OK
Applying sessions.0001_initial... OK
System check identified no issues (0 silenced).
test_home_url_resolves_home_view (boards.tests.HomeTests) ...
ok
test_home_view_status_code (boards.tests.HomeTests) ... ok

Ran 2 tests in 0.017s

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

Verbosity决定了将要打印到控制台的通知和调试信息量；0是无输出，1是正常输出，2是详细输出。

# Django入门指南-第9章：静态文件设置

静态文件是指 CSS, JavaScript, 字体, 图片或者是用来组成用户界面的任何其他资源。

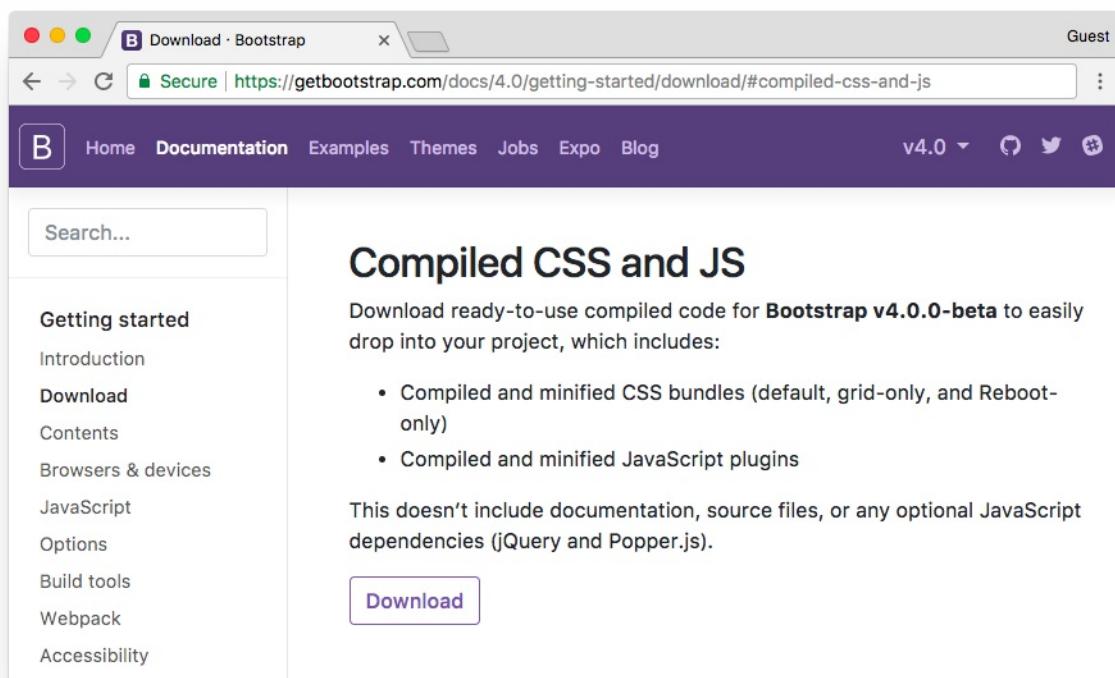
实际上, Django 本身是不负责处理这些文件的, 但是为了让我们的开发过程更轻松, Django 提供了一些功能来帮助我们管理静态文件。这些功能可在 `INSTALLED_APPS` 的 `django.contrib.staticfiles` 应用程序中找到（译者：Django为了使得开发方便, 也可以处理静态文件, 而在生产环境下, 静态文件一般直接由 Nginx 等反向代理服务器处理, 而应用工服务器专心负责处理它擅长的业务逻辑）。

市面上很多优秀前端组件框架, 我们没有理由继续用简陋的HTML文档来渲染】。我们可以轻松地将Bootstrap 4添加到我们的项目中。Bootstrap是一个用HTML, CSS和JavaScript开发的前端开源工具包。

在项目根目录中, 除了 `boards`, `templates` 和`myproject`文件夹外, 再创建一个名为`static`的新文件夹, 并在`static`文件夹内创建另一个名为`css`的文件夹:

```
myproject/
| -- myproject/
| | -- boards/
| | -- myproject/
| | -- templates/
| | -- static/ <-- here
| | +-- css/ <-- and here
| +-- manage.py
+-- venv/
```

转到[getbootstrap.com](http://getbootstrap.com)并下载最新版本:



## 下载编译版本的CSS和JS

在你的计算机中，解压 bootstrap-4.0.0-beta-dist.zip 文件，将文件 css/bootstrap.min.css 复制到我们项目的css文件夹中：

```
myproject/
| -- myproject/
| | -- boards/
| | -- myproject/
| | -- templates/
| | -- static/
| | +-- css/
| | +-- bootstrap.min.css <-- here
| +-- manage.py
+-- venv/
```

下一步是告诉Django在哪里可以找到静态文件。打开settings.py，拉到文件的底部，在`STATIC_URL`后面添加以下内容：

```
STATIC_URL = '/static/'

STATICFILES_DIRS = [
 os.path.join(BASE_DIR, 'static'),
]
```

还记得 **TEMPLATES** 目录吗， 和这个配置是一样的

现在我们必须在模板中加载静态文件（Bootstrap CSS文件）：

**templates/home.html**

```
{% load static %}<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>Boards</title>
 <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
 </head>
 <body>
 <h1>Boards</h1>

 <table border="1">
 <thead>
 <tr>
 <th>Board</th>
 <th>Posts</th>
 <th>Topics</th>
 <th>Last Post</th>
 </tr>
 </thead>
 <tbody>
 {% for board in boards %}
 <tr>
 <td>
 {{ board.name }}

 <small style="color: #888">{{ board.description }}</small>
 </td>
 <td>0</td>
 <td>0</td>
 <td></td>
 </tr>
 {% endfor %}
 </tbody>
 </table>
 </body>
</html>
```

首先，我们在模板的开头使用了 Static Files App 模板标签 `{% load static %}`。

模板标签 `{% static %}` 用于构成资源文件完整URL。在这种情况下，`{% static 'css/bootstrap.min.css' %}` 将返回 **/static/css/bootstrap.min.css**，它相当于 <http://127.0.0.1:8000/static/css/bootstrap.min.css>。

`{% static %}` 模板标签使用 settings.py 文件中的 `STATIC_URL` 配置来组成最终的URL，例如，如果您将静态文件托管在像 <https://static.example.com/> 这样的子域中，那么我们将设置 `STATIC_URL=https://static.example.com/`，然后 `{% static 'css/bootstrap.min.css' %}` 返回的是 <https://static.example.com/css/bootstrap.min.css>

如果目前这些对你来说搞不懂也不要担心。只要记得但凡是需要引用CSS, JavaScript或图片文件的地方就使用 `{% static %}`。稍后，当我们开始部署项目到正式环境时，我们将讨论更多。现在都设置好了。

刷新页面 <http://127.0.0.1:8000>，我们可以看到它可以正常运行：



现在我们可以编辑模板，以利用Bootstrap CSS：

```
{% load static %}<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>Boards</title>
 <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
 </head>
 <body>
 <div class="container">
 <ol class="breadcrumb my-4">
 <li class="breadcrumb-item active">Boards
```

```

<table class="table">
 <thead class="thead-inverse">
 <tr>
 <th>Board</th>
 <th>Posts</th>
 <th>Topics</th>
 <th>Last Post</th>
 </tr>
 </thead>
 <tbody>
 {% for board in boards %}
 <tr>
 <td>
 {{ board.name }}
 <small class="text-muted d-block">{{ board.description }}</small>
 </td>
 <td class="align-middle">0</td>
 <td class="align-middle">0</td>
 <td></td>
 </tr>
 {% endfor %}
 </tbody>
</table>
</div>
</body>
</html>
```

显示效果：



到目前为止，我们使用交互式控制台（`python manage.py shell`）添加了几个新的版块。但我们需要一个更好的方式来实现。在下一节中，我们将为网站管理员实现一个管理界面来管理这些数据。

# Django入门指南-第10章：Django Admin 介绍



到目前为止，我们使用交互式控制台（`python manage.py shell`）添加新的版块。但我们需要一个更好的方式来实现。在这一节中，我们将为网站管理员实现一个管理界面来管理这些数据。

## Django Admin 简介

当我们开始一个新项目时，Django已经配置了Django Admin，这个应用程序列出的INSTALLED\_APPS。



使用 Django Admin的一个很好的例子就是用在博客中；它可以被作者用来编写和发布文章。另一个例子是电子商务网站，工作人员可以创建，编辑，删除产品。

现在，我们将配置 Django Admin 来维护我们应用程序的版块。

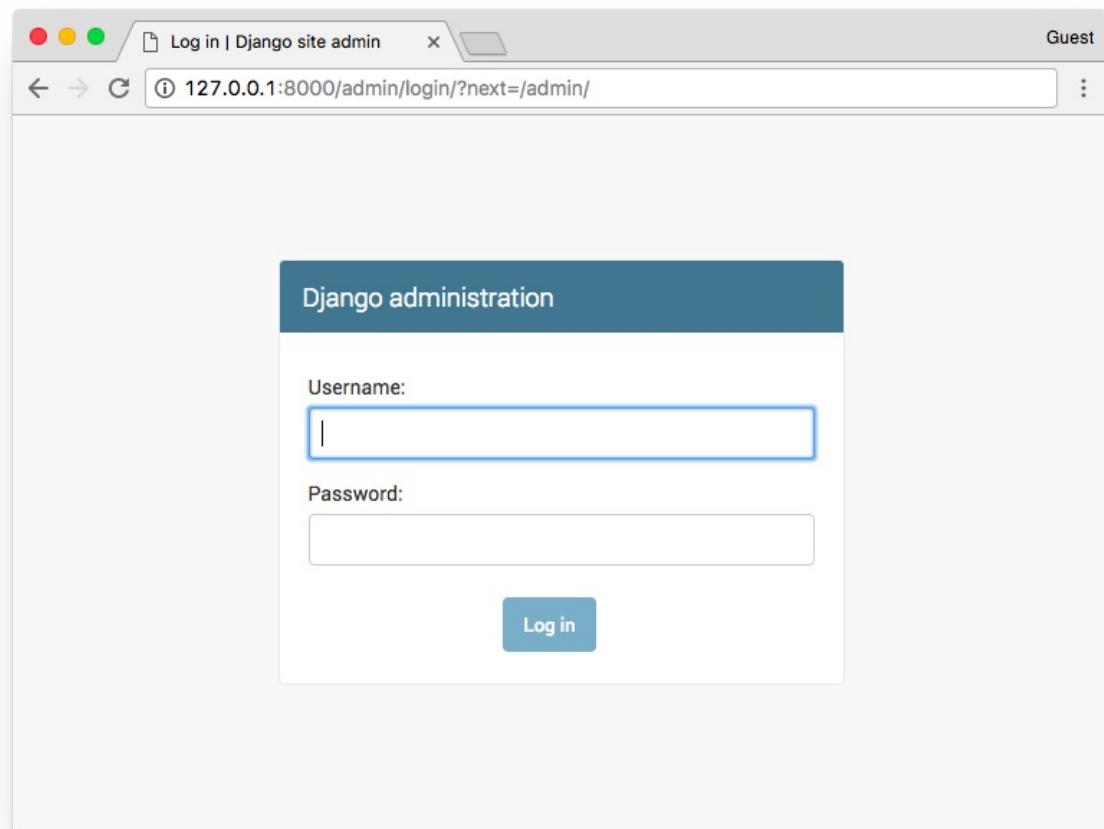
我们首先创建一个管理员帐户：

```
python manage.py createsuperuser
```

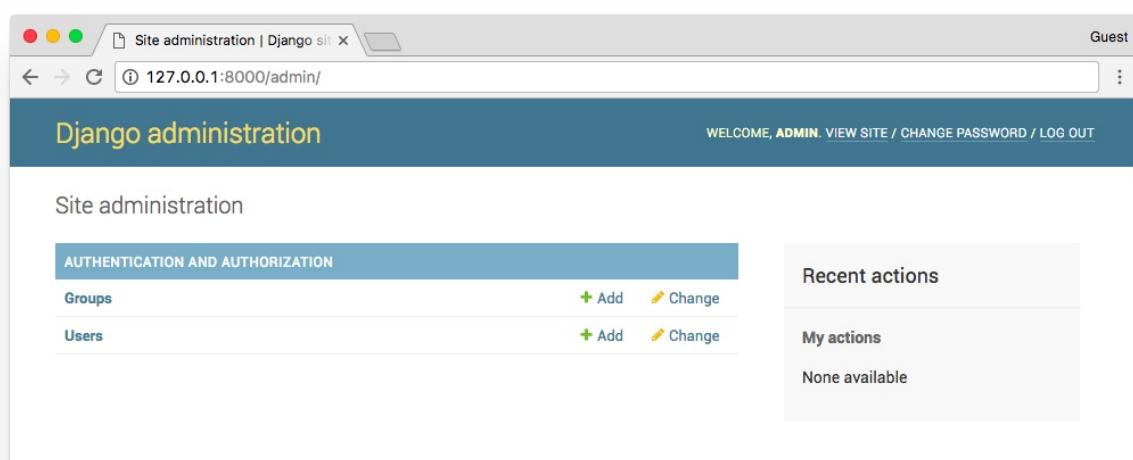
按照说明操作：

```
Username (leave blank to use 'vitorfs'): admin
Email address: admin@example.com
Password:
Password (again):
Superuser created successfully.
```

在浏览器中打开该URL：<http://127.0.0.1:8000/admin/>



输入用户名和密码登录到管理界面：



它已经配置了一些功能。在这里，我们可以添加用户和组的权限管理，这些概念在后面我们将探讨更多。

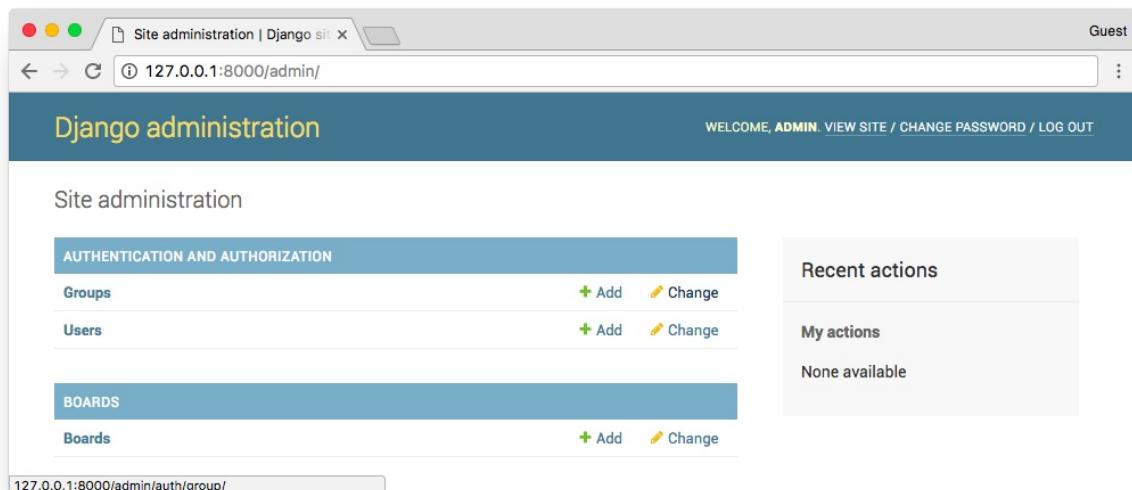
添加Board模型非常简单。打开boards目录中的admin.py文件，并添加以下代码：

### boards/admin.py

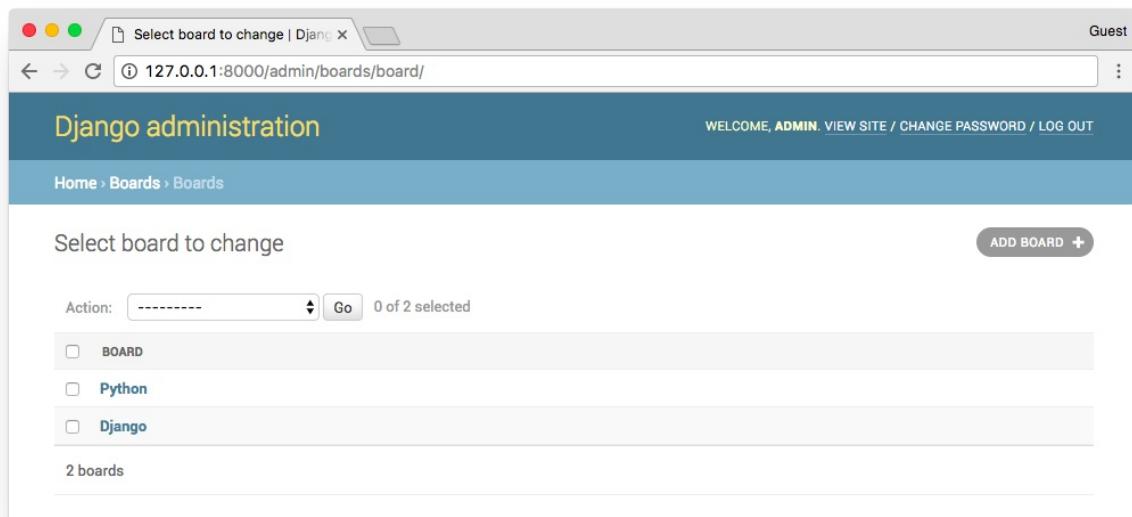
```
from django.contrib import admin
from .models import Board

admin.site.register(Board)
```

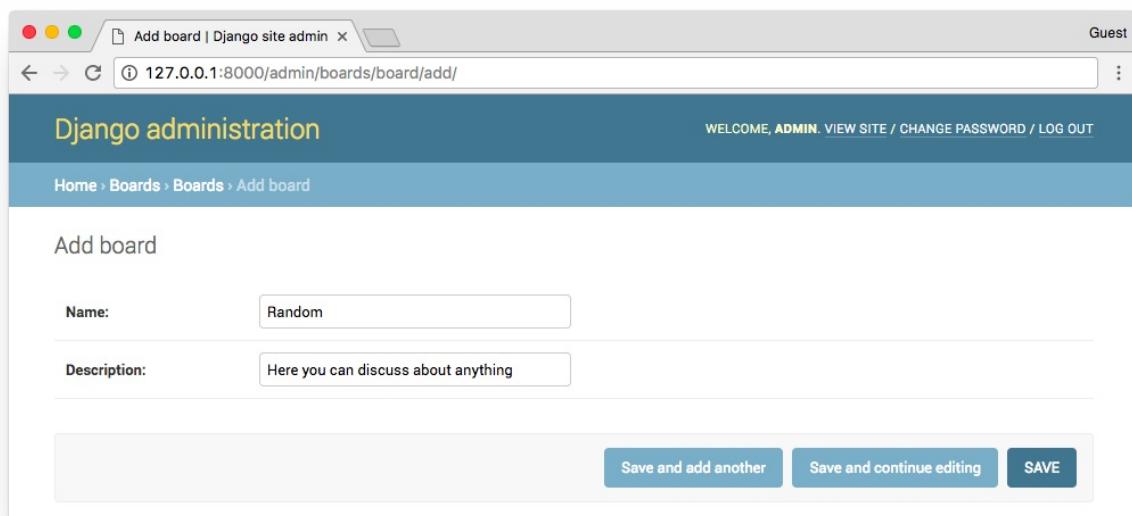
保存admin.py文件，然后刷新网页浏览器中的页面：



对！它已准备好被使用了。点击Boards链接查看现有版块列表：



我们可以通过点击 Add Board 按钮添加一个新的版块：



点击保存按钮：

The screenshot shows the Django Admin interface for the 'Boards' model. At the top, there's a success message: 'The board "Random" was added successfully.' Below it, a list of boards is displayed with checkboxes for selecting them. The boards listed are 'BOARD', 'Random', 'Python', and 'Django'. A button labeled 'ADD BOARD +' is visible at the top right.

我们可以检查一切是否正常，打开URL <http://127.0.0.1:8000>

The screenshot shows the application's main page titled 'Boards'. It lists three boards: 'Django' (with a note 'Django discussion board.'), 'Python' (with a note 'General discussion about Python.'), and 'Random' (with a note 'Here you can discuss about anything'). Each board entry includes columns for 'Posts' and 'Topics' (both currently 0) and a 'Last Post' link.

## 总结

在本教程中，我们探讨了许多新概念。我们为项目定义了一些需求，创建了第一个模型，迁移了数据库，开始玩 Models API。我们创建了第一个视图并编写了一些单元测试。同时我们还配置了Django模板引擎，静态文件，并将Bootstrap 4库添加到项目中。最后，我们简要介绍了Django Admin界面。

我希望你喜欢本系列教程的第二部分！下一部分，我们将探索Django的URL路由，表单API，可重用模板以及更多测试。

该项目的源代码在GitHub上可用。本来的代码可以在发布标签v0.2-lw下找到。下面的链接将带你到正确的地方：

<https://github.com/sibtc/django-beginners-guide/tree/v0.2-lw>

# Django入门与实践-第11章：URL 分发

## 前言

在本节课中，我们将深入理解两个基本概念：URLs 和 Forms。在这个过程中，我们还将学习其它很多概念，如创建可重用模板和安装第三方库。同时我们还将编写大量单元测试。

如果你是从这个系列教程的 part 1 跟着这个教程一步步地编写项目，你可能需要在开始之前更新 **models.py**：

### boards/models.py

```
class Topic(models.Model):
 # other fields...
 # Add `auto_now_add=True` to the `last_updated` field
 last_updated = models.DateTimeField(auto_now_add=True)

class Post(models.Model):
 # other fields...
 # Add `null=True` to the `updated_by` field
 updated_by = models.ForeignKey(User, null=True, related_name='+')
```

现在在已经激活的 virtualenv 环境中执行命令：

```
python manage.py makemigrations
python manage.py migrate
```

如果在你的程序中 `update_by` 字段已经有了 `null=True` 且 `last_updated` 字段中有了 `auto_now_add=True`，你可以放心地忽略上面这步操作。

如果你更喜欢使用我的代码作为出发点，你可以在 GitHub 上找到它。本项目现在的代码，可以在 **v0.2-lw** 标签下找到。下面是链接：

[<https://github.com/sibtc/django-beginners-guide/tree/v0.2-lw>][1]

我们的开发就从这里开始

## URLs

随着我们项目的开发，我们需要实现一个新的功能，就是列出某个板块下的所有主题列表，再来看看一下，你可以看到上一节中我们画的线框图。

Topic	Starter	Replies	Views	Last Update
<a href="#">Latest updates on Django 1.11</a>	john	5	24	2017-08-05 18:02 by megan
<a href="#">Check out this django app</a>	megan	0	12	2017-08-05 17:42 by megan
<a href="#">Help with a project</a>	vitor	24	50	2017-08-04 23:23 by julie
<a href="#">How to extend user model?</a>	julie	34	224	2017-08-04 23:23 by john

我们将从 **myproject** 目录中编写 **urls.py** 开始：

**myproject/urls.py**

```

from django.conf.urls import url
from django.contrib import admin

from boards import views

urlpatterns = [
 url(r'^$', views.home, name='home'),
 url(r'^boards/(?P<pk>\d+)/$', views.board_topics, name='board_topics'),
 url(r'^admin/', admin.site.urls),
]

```

现在我们花点时间来分析一下 `urlpatterns` 和 `url`。

**URL 调度器 (dispatcher)** 和 **URLconf (URL configuration)** 是 Django 应用中的基础部分。在开始的时候，这个看起来让人很困惑；我记得我第一次开始使用 Django 开发的时候也有一段时间学起来很困难。

事实上，Django开发团队正在致力于将路由语法简化（译注：就是将原来`url`函数替换成`path`函数，目前django2.0已经正式使用新的路由语法）

一个项目可以有很多 **urls.py** 分布在多个应用（app）中。Django 需要一个 **url.py** 作为入口。这个特殊的 **urls.py** 叫做 **根路由配置 (root URLconf)**。它被定义在 **settings.py** 中。

### myproject/settings.py

```
ROOT_URLCONF = 'myproject.urls'
```

它已经自动配置好了，你不需要去改变它任何东西。

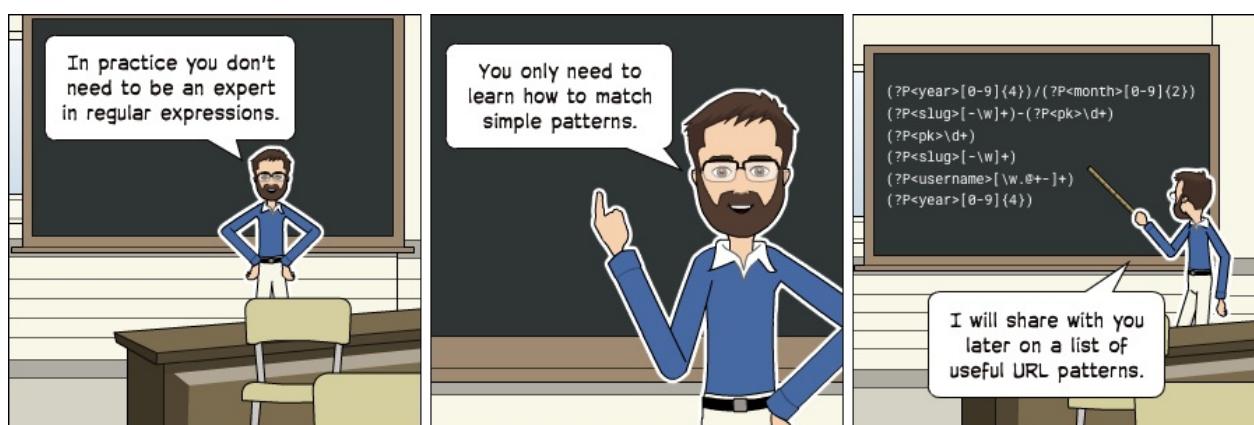
当 Django 接受一个请求(request)，它就会在项目的 URLconf 中寻找匹配项。他从 `urlpatterns` 变量的第一条开始，然后在每个 `url` 中去匹配请求的 URL。

如果 Django 找到了一个匹配路径，他会把请求(request)发送给 `url` 的第二个参数 **视图函数（view function）**。`urlpatterns` 中的顺序很重要，因为 Django 一旦找到匹配就会停止往后搜索。如果 Django 在 URLconf 中没有找到匹配项，他会通过 **Page Not Found** 的错误处理代码抛出一个 **404** 异常。

这是 `url` 函数的剖析：

```
def url(regex, view, kwargs=None, name=None):
 # ...
```

- **regex**: 匹配 URL patterns 的正则表达式。注意：正则表达式会忽略掉 **GET** 或者 **POST** 后面的参数。在一个 <http://127.0.0.1:8000/boards/?page=2> 的请求中，只有 **/boards/** 会被处理。
- **view**: 视图函数被用来处理用户请求，同时它还可以是 `django.conf.urls.include` 函数的返回值，它将引用一个外部的 `urls.py` 文件，例如，你可以使用它来定义一组特定于应用的 URLs，使用前缀将其包含在根 URLconf 中。我们会在后面继续探讨这个概念。
- **kwargs**: 传递给目标视图函数的任意关键字参数，它通常用于在可重用视图上进行一些简单的定制，我们不是经常使用它。
- **name**:: 该 URL 的唯一标识符。这是一个非常重要的特征。要始终记得为你的 URLs 命名。所以，很重要的一点是：不要在 `views(视图)` 或者 `templates(模板)` 中硬编码 URL，而是通过它的名字去引用 URL。



## 基础 URLs 路由

基础URL创建起来很容易。就只是个匹配字符串的问题。比如说，我们想创建一个 "about" 页面，可以这样定义：

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

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

我们也可以创建更深层一点的 URL 结构

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

urlpatterns = [
 url(r'^$', views.home, name='home'),
 url(r'^about/$', views.about, name='about'),
 url(r'^about/company/$', views.about_company, name='about_company'),
 url(r'^about/author/$', views.about_author, name='about_author'),
 url(r'^about/author/vitor/$', views.about_vitor, name='about_vitor'),
 url(r'^about/author/erica/$', views.about_erica, name='about_erica'),
 url(r'^privacy/$', views.privacy_policy, name='privacy_policy'),
]
```

这是一些简单的 URL 路由的例子，对于上面所有的例子，视图函数都是下面这个结构：

```

def about(request):
 # do something...
 return render(request, 'about.html')

def about_company(request):
 # do something else...
 # return some data along with the view...
 return render(request, 'about_company.html', {'company_name': 'Simple Complex'})

```

## 高级 URLs 路由

更高级的URL路由使用方法是通过正则表达式来匹配某些类型的数据并创建动态 URL

例如，要创建一个个人资料的页面，诸如 `github.com/vitorfs` or `twitter.com/vitorfs`(`vitorfs` 是我的用户名) 这样，我们可以像以下几点这样做：

```

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

urlpatterns = [
 url(r'^$', views.home, name='home'),
 url(r'^(?P<username>[\w.+-]+)/$', views.user_profile, name='user_profile'),
]

```

它会匹配 Django 用户模型里面所有有效的用户名。

现在我们可以看到上面的例子是一个很宽松的 URL。这意味大量的 URL patterns 都会被它匹配，因为它定义在 URL 的根，而不像 `/profile//` 这样。在这种情况下，如果我们想定义一个 `/about/` 的URL，我们要把它定义在这个 username URL pattern 的前面：

```

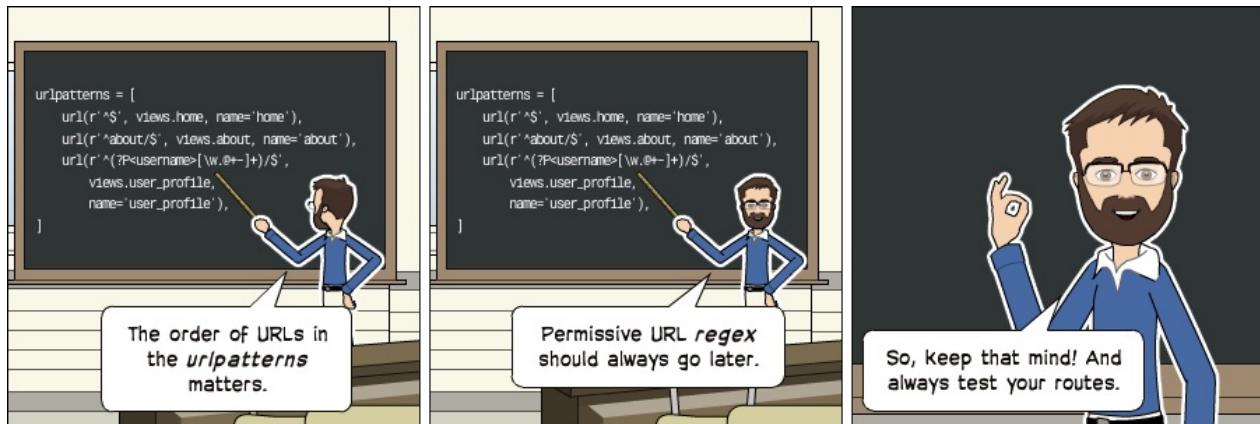
from django.conf.urls import url
from boards import views

urlpatterns = [
 url(r'^$', views.home, name='home'),
 url(r'^about/$', views.about, name='about'),
 url(r'^(?P<username>[\w.@+-]+)/$', views.user_profile, name='user_profile'),
]

```

如果这个 "about" 页面定义在 `username` URL pattern 后面，Django 将永远找不到它，因为 "about" 这个单词会先被 `username` 的正则表达式所匹配到，视图函数 `user_profile` 将会被执行而不是执行 `about`。

此外，这有一些副作用。例如，从现在开始，我们要把 "about" 视为禁止使用的 `username`，因为如果有用户将 "about" 作为他们的 `username`，他们将永远不能看到他们的个人资料页面，而看到的 `about` 页面。



如果你想给用户个人主页设置一个很酷的主页的 URL，那么避免冲突最简单的方法是添加一个前缀，例如：/u/vitorfs，或者像 Medium 一样使用 @ 作为前缀 /@vitorfs/。

这些 URL 路由的主要思想是当 URL 的一部分被当作某些资源(这些资源用来构成某个页面)的标识的时候就去创建一个动态页面。比如说，这个标识可以是一个整数的 ID 或者是一个字符串。

开始的时候，我们使用 **Board** ID 去创建 **Topics**列表的动态页面。让我们再来看一下我在 **URLs** 开头的部分给出的例子：

```
url(r'^boards/(\d+)/$', views.board_topics, name='board_topics')
```

正则表达式中的 `\d+` 会匹配一个任意大小的整数值。这个整数值用来从数据库中取到 指定的 **Board**。现在注意我们这样写这个正则表达式 `(?P<pk>\d+)`，这是告诉 Django 将捕获到的值放入名为 **pk** 的关键字参数中。

这时我们为它写的一个视图函数：

```
def board_topics(request, pk):
 # do something...
```

因为我们使用了 `(?P<pk>\d+)` 正则表达式，在 `board_topics` 函数中，关键字参数必须命名为 **pk**。

如果你想在视图函数使用任意名字的参数，那么可以这样定义：

```
url(r'^boards/(\d+)/$', views.board_topics, name='board_topics')
```

然后在视图函数可以这样定义：

```
def board_topics(request, board_id):
 # do something...
```

或者这样：

```
def board_topics(request, id):
 # do something...
```

名字无关紧要，但是使用命名参数是一个很好的做法，因为，当我们有个更大的URL去捕获多个 ID 和变量时，这会更便于我们阅读。

PK or ID? PK 表示主键（Primary key），这是访问模型的主键ID的简写方法，所有Django模型都有这个属性，更多的时候，使用pk属性和使用id是一样的，这是因为如果我们没有给model定义主键时，Django将自动创建一个 AutoField 类型的字段，名字叫做 id，它就是主键。如果你给model定义了一个不同的主键，例如，假设 email 是你的主键，你就可以这样访问：obj.email 或者 obj.pk，二者是等价的。

## 使用 URLs API

现在到了写代码的时候了。我们来实现我在开头提到的主题列表页面

首先，编辑 `urls.py`，添加新的 URL 路由

### `myproject/urls.py`

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

from boards import views

urlpatterns = [
 url(r'^$', views.home, name='home'),
 url(r'^boards/(?P<pk>\d+)/$', views.board_topics, name='board_topics'),
 url(r'^admin/', admin.site.urls),
]
```

现在创建视图函数 `board_topics`：

### `boards/views.py`

```
from django.shortcuts import render
from .models import Board

def home(request):
 # code suppressed for brevity

def board_topics(request, pk):
 board = Board.objects.get(pk=pk)
 return render(request, 'topics.html', {'board': board})
```

在 **templates** 目录中，创建一个名为 **topics.html** 的模板：

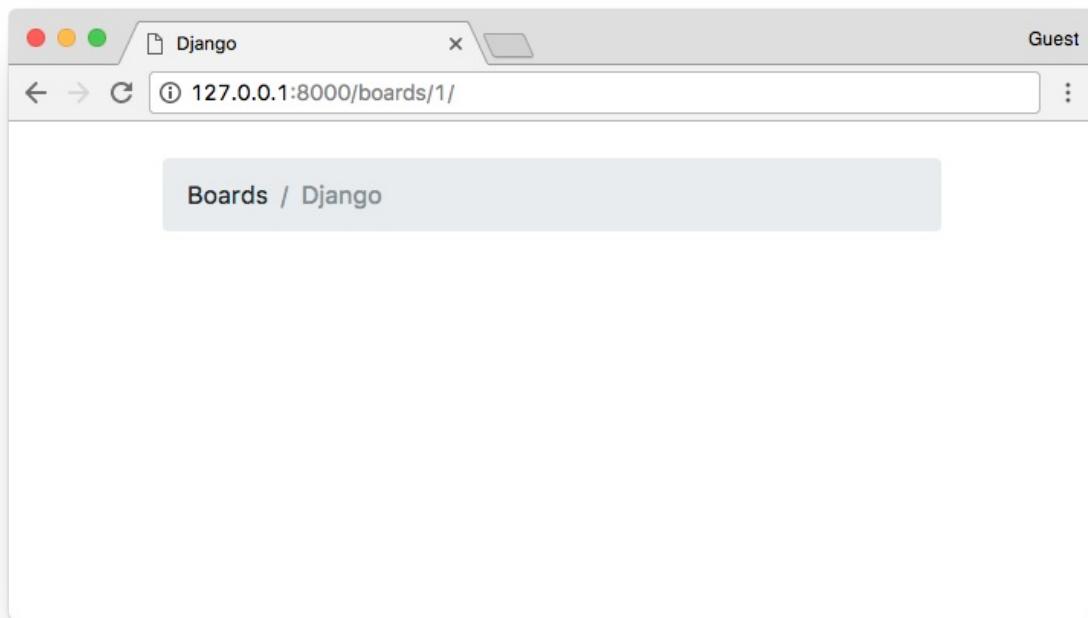
### templates/topics.html

```
{% load static %}<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>{{ board.name }}</title>
 <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
 </head>
 <body>
 <div class="container">
 <ol class="breadcrumb my-4">
 <li class="breadcrumb-item">Boards
 <li class="breadcrumb-item active">{{ board.name }}

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

注意：我们现在只是创建新的 HTML 模板。不用担心，在下一节中我会向你展示如何创建可重用模板。

现在在浏览器中打开 URL <http://127.0.0.1:8000/boards/1/>，结果应该是下面这个页面：



现在到了写一些测试的时候了！编辑 **test.py**，在文件底部添加下面的测试：

**boards/tests.py**

```
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase
from .views import home, board_topics
from .models import Board

class HomeTests(TestCase):
 # ...

class BoardTopicsTests(TestCase):
 def setUp(self):
 Board.objects.create(name='Django', description='Django board.')

 def test_board_topics_view_success_status_code(self):
 url = reverse('board_topics', kwargs={'pk': 1})
 response = self.client.get(url)
 self.assertEqual(response.status_code, 200)

 def test_board_topics_view_not_found_status_code(self):
 url = reverse('board_topics', kwargs={'pk': 99})
 response = self.client.get(url)
 self.assertEqual(response.status_code, 404)

 def test_board_topics_url_resolves_board_topics_view(self):
 :
 view = resolve('/boards/1/')
 self.assertEqual(view.func, board_topics)
```

这里需要注意几件事情。这次我们使用了 `setUp` 方法。在这个方法中，我们创建了一个 **Board** 实例来用于测试。我们必须这样做，因为 Django 的测试机制不会针对当前数据库跑你的测试。运行 Django 测试时会即时创建一个新的数据库，应用所有的model(模型)迁移，运行测试完成后会销毁这个用于测试的数据库。

因此在 `setUp` 方法中，我们准备了运行测试的环境，用来模拟场景。

- `test_board_topics_view_success_status_code` 方法：测试 Django 是否对于现有的 **Board** 返回 status code(状态码) 200(成功)。
- `test_board_topics_view_not_found_status_code` 方法：测试 Django 是否对于不存在于数据库的 **Board** 返回 status code 404(页面未找到)。
- `test_board_topics_url_resolves_board_topics_view` 方法：测试 Django 是否使用了正确的视图函数去渲染 topics。

现在来运行一下测试：

```
python manage.py test
```

输出如下：

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.E...
=====
=====
ERROR: test_board_topics_view_not_found_status_code (boards.tests.BoardTopicsTests)

Traceback (most recent call last):
...
boards.models.DoesNotExist: Board matching query does not exist.

Ran 5 tests in 0.093s

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

测试 `test_board_topics_view_not_found_status_code` 失败。我们可以在 Traceback 中看到返回了一个 exception(异常)  
“boards.models.DoesNotExist: Board matching query does not exist.”

```
DoesNotExist at /boards/99/
Board matching query does not exist.

Request Method: GET
Request URL: http://127.0.0.1:8000/boards/99/
Django Version: 1.11.4
Exception Type: DoesNotExist
Exception Value: Board matching query does not exist.
Exception Location: /Users/vitorfs/Development/myproject/venv/lib/python3.6/site-packages/django/db/models/query.py in get, line 380
Python Executable: /Users/vitorfs/Development/myproject/venv/bin/python
Python Version: 3.6.2
Python Path: ['/Users/vitorfs/Development/myproject/django-beginners-guide',
 '/Users/vitorfs/Development/myproject/venv/lib/python36.zip',
 '/Users/vitorfs/Development/myproject/venv/lib/python3.6',
 '/Users/vitorfs/Development/myproject/venv/lib/python3.6/lib-dynload',
 '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/Versions/3.6/lib/python3.6',
 '/Users/vitorfs/Development/myproject/venv/lib/python3.6/site-packages']
Server time: Tue, 12 Sep 2017 14:56:22 +0000
```

在 `DEBUG=False` 的生产环境中，访问者会看到一个 **500 Internal Server Error** 的页面。但是这不是我们希望得到的。

我们想要一个 **404 Page Not Found** 的页面。让我们来重写我们的视图函数。

## boards/views.py

```
from django.shortcuts import render
from django.http import Http404
from .models import Board

def home(request):
 # code suppressed for brevity

def board_topics(request, pk):
 try:
 board = Board.objects.get(pk=pk)
 except Board.DoesNotExist:
 raise Http404
 return render(request, 'topics.html', {'board': board})
```

重新测试一下：

```
python manage.py test
```

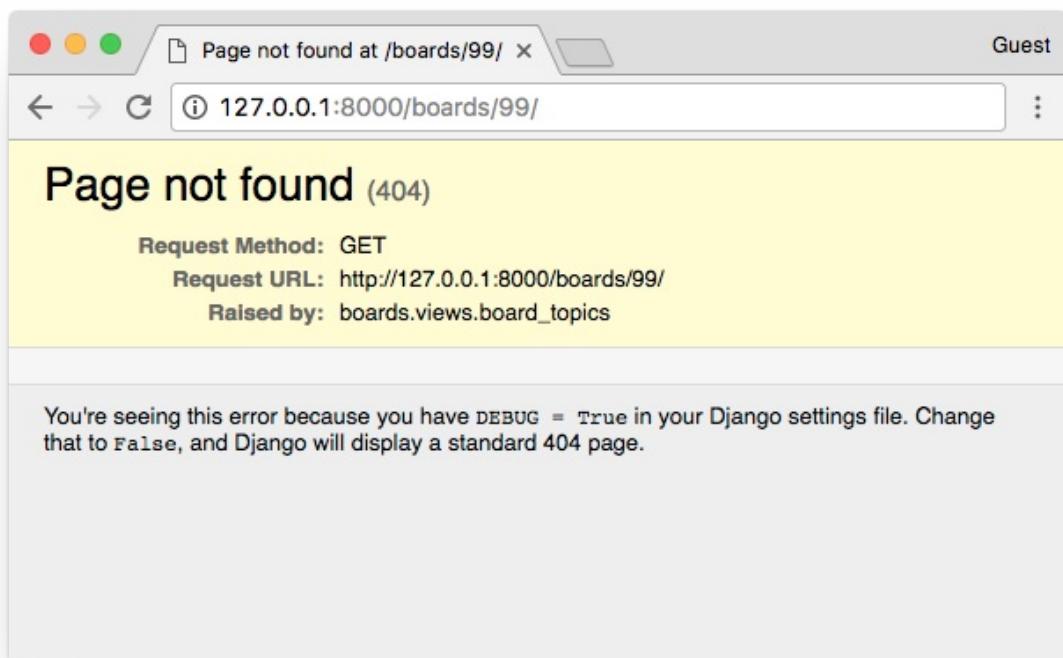
```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.....

Ran 5 tests in 0.042s

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

好极了！现在它按照预期工作。



这是 Django 在 `DEBUG=False` 的情况下显示的默认页面。稍后，我们可以自定义 404 页面去显示一些其他的东西。

这是一个常见的用法。事实上，Django 有一个快捷方式去得到一个对象，或者返回一个不存在的对象 404。

因此让我们再来重写一下 `board_topics` 函数：

```
from django.shortcuts import render, get_object_or_404
from .models import Board

def home(request):
 # code suppressed for brevity

def board_topics(request, pk):
 board = get_object_or_404(Board, pk=pk)
 return render(request, 'topics.html', {'board': board})
```

修改了代码，测试一下。

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

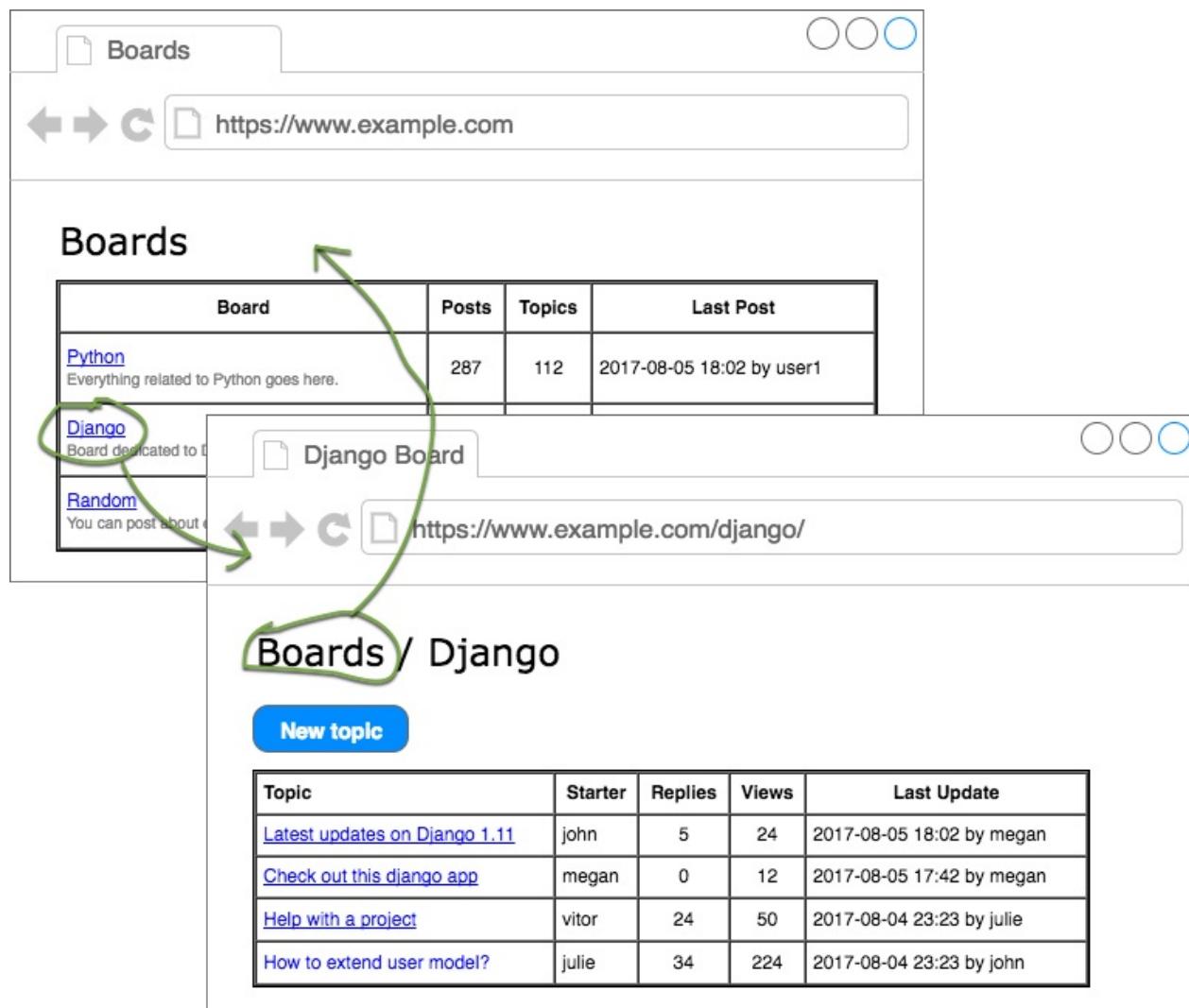
.....

Ran 5 tests in 0.052s

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

没有破坏任何东西。我们可以继续我们的开发。

下一步是在屏幕上创建一个导航链接。主页应该有一个链接指引访问者去访问指定板块下面的主题列表页面。同样地，topics 页面也应当有一个返回主页的链接。



我们可以先为 `HomeTests` 类编写一些测试：

**boards/test.py**

```
class HomeTests(TestCase):
 def setUp(self):
 self.board = Board.objects.create(name='Django', description='Django board.')
 url = reverse('home')
 self.response = self.client.get(url)

 def test_home_view_status_code(self):
 self.assertEqual(self.response.status_code, 200)

 def test_home_url_resolves_home_view(self):
 view = resolve('/')
 self.assertEqual(view.func, home)

 def test_home_view_contains_link_to_topics_page(self):
 board_topics_url = reverse('board_topics', kwargs={'pk': self.board.pk})
 self.assertContains(self.response, 'href="{0}"'.format(board_topics_url))
```

注意到现在我们同样在 **HomeTests** 中添加了 **setUp** 方法。这是因为我们现在需要一个 **Board** 实例，并且我们将 **url** 和 **response** 移到了 **setUp**，所以我们能在新测试中重用相同的 **response**。

这里的新测试是 **test\_home\_view\_contains\_link\_to\_topics\_page**。我们使用 **assertContains** 方法来测试 **response** 主体部分是否包含给定的文本。我们在测试中使用的文本是 `a` 标签的 `href` 部分。所以基本上我们是在测试 **response** 主体是否包含文本 `href="/boards/1/"`。

让我们运行这个测试：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....F.
=====
=====
FAIL: test_home_view_contains_link_to_topics_page (boards.tests.HomeTests)

...

AssertionError: False is not true : Couldn't find 'href="/boards/1/"' in response

Ran 6 tests in 0.034s

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

现在我们可以编写能通过这个测试的代码。

编写 **home.html** 模板：

**templates/home.html** {**% raw %**}

```

<!-- code suppressed for brevity -->
<tbody>
 {% for board in boards %}
 <tr>
 <td>
 {{ board.
name }}
 <small class="text-muted d-block">{{ board.descriptio
n }}</small>
 </td>
 <td class="align-middle">0</td>
 <td class="align-middle">0</td>
 <td></td>
 </tr>
 {% endfor %}
</tbody>
<!-- code suppressed for brevity -->

```

我们只改动了这一行：

```
 {{ board.name }}
```

变为：

```
{{ board.name }}</
a>
```

始终使用 `{% url %}` 模板标签去写应用的 URL。第一个参数是 URL 的名字(定义在 URLconf，即 `urls.py`)，然后你可以根据需求传递任意数量的参数。

如果是一个像主页这种简单的 URL，那就是 `{% url 'home' %}`。保存文件然后再运行一下测试：

```
python manage.py test
```

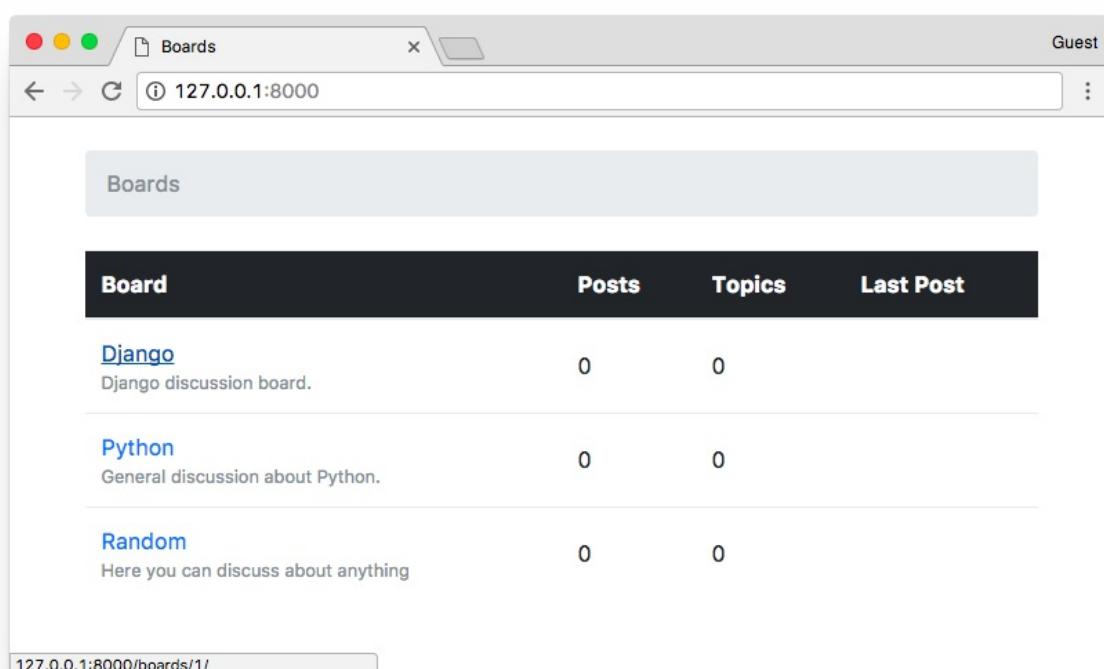
```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.....

Ran 6 tests in 0.037s

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

很棒！现在我们可以看到它在浏览器是什么样子。



现在轮到返回的链接了，我们可以先写测试：

**boards/tests.py**

```
class BoardTopicsTests(TestCase):
 # code suppressed for brevity...

 def test_board_topics_view_contains_link_back_to_homepage(self):
 board_topics_url = reverse('board_topics', kwargs={'pk': 1})
 response = self.client.get(board_topics_url)
 homepage_url = reverse('home')
 self.assertContains(response, 'href="{0}"'.format(homepage_url))
```

运行测试：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.F.....
=====
=====
FAIL: test_board_topics_view_contains_link_back_to_homepage (
boards.tests.BoardTopicsTests)

Traceback (most recent call last):
...

AssertionError: False is not true : Couldn't find 'href="/"'
in response

Ran 7 tests in 0.054s

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

更新主题列表模版：

**templates/topics.html**

```
{% load static %}<!DOCTYPE html>
<html>
 <head><!-- code suppressed for brevity --></head>
 <body>
 <div class="container">
 <ol class="breadcrumb my-4">
 <li class="breadcrumb-item">Boards
 <li class="breadcrumb-item active">{{ board.name }}

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

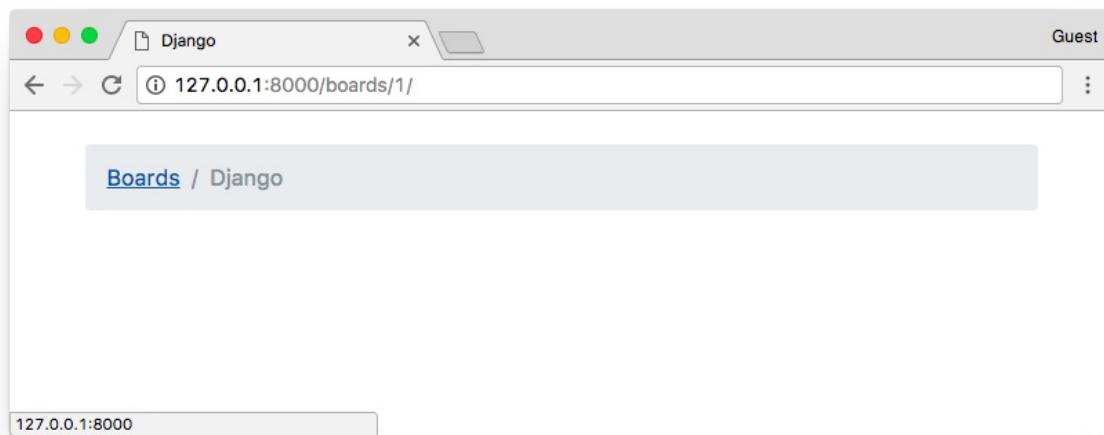
运行测试：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
.
.

Ran 7 tests in 0.061s

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



就如我之前所说的， URL 路由是一个 web 应用程序的基本组成部分。有了这些知识，我们才能继续开发。下一步是完成 URL 的部分，你会看到一些使用 URL patterns 的总结。

## 实用URL模式列表

技巧部分是正则表达式。我准备了一个最常用的 URL patterns 的列表。当你需要一个特定的 URL 时你可以参考这个列表。

主键自增字段	
正则表达式	(?P<pk>\d+)
举例	url(r'^questions/(?P<pk>\d+)/\$', views.question, name='question')
有效 URL	/questions/934/
捕获数据	{'pk': '934'}

Slug 字段	
正则表达式	(?P<slug>[-\w]+)
举例	url(r'^posts/(?P<slug>[-\w]+)/\$', views.post, name='post')
有效 URL	/posts/hello-world/
捕获数据	{'slug': 'hello-world'}

有主键的 Slug 字段	
正则表达式	(?P<slug>[-\w+])- (?P<pk>\d+)
举例	url(r'^blog/(?P<slug>[-\w+])- (?P<pk>\d+)/\$', views.blog_post, name='blog_post')
有效 URL	/blog/hello-world-159/
捕获数据	{'slug': 'hello-world', 'pk': '159'}

Django 用户名	
正则表达式	(?P<username>[\w.@+-]+)
举例	url(r'^profile/(?P<username>[\w.@+-]+)\$', views.user_profile, name='user_profile')
有效 URL	/profile/vitorfs/
捕获数据	{'username': 'vitorfs'}

Year	
正则表达式	(?P<year>[0-9]{4})
举例	url(r'^articles/(?P<year>[0-9]{4})/\$', views.year_archive, name='year')
有效 URL	/articles/2016/
捕获数据	{'year': '2016'}

Year / Month	
正则表达式	(?P<year>[0-9]{4})/(?P<month>[0-9]{2})
举例	url(r'^articles/(?P<year>[0-9]{4})/(?P<month>[0-9]{2})/\$', views.month_archive, name='month')
有效 URL	/articles/2016/01/
捕获数据	{'year': '2016', 'month': '01'}

你可以在这篇文章中看到更多关于正则表达式匹配的细节：[List of Useful URL Patterns](#)。

{% endraw %}

# Django入门与实践-第12章：复用模板

到目前为止，我们一直在复制和粘贴 HTML 文档的多个部分。从长远来看是不可行的。这也是一个坏的做法。

在这一节我们将重写 HTML 模板，创建一个 **master page(母版页)**，其他模板添加它所独特的部分。

在 **templates** 文件夹中创建一个名为 **base.html** 的文件：

**templates/base.html**

```
{% load static %}<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>{% block title %}Django Boards{% endblock %}</title>
 >
 <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
 </head>
 <body>
 <div class="container">
 <ol class="breadcrumb my-4">
 {% block breadcrumb %}
 {% endblock %}

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

这是我们的母版页。每个我们创建的模板都 **extend(继承)** 这个特殊的模板。现在我们介绍 `{% block %}` 标签。它用于在模板中保留一个空间，一个"子"模板(继承这个母版页的模板)可以在这个空间中插入代码和 HTML。

在 `{% block title %}` 中我们还设置了一个默认值 "Django Boards."。如果我们在子模板中未设置 `{% block title %}` 的值它就会被使用。

现在让我们重写我们的两个模板：**home.html** 和 **topics.html**。

**templates/home.html**

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

{% block breadcrumb %}
 <li class="breadcrumb-item active">Boards
{% endblock %}

{% block content %}
 <table class="table">
 <thead class="thead-inverse">
 <tr>
 <th>Board</th>
 <th>Posts</th>
 <th>Topics</th>
 <th>Last Post</th>
 </tr>
 </thead>
 <tbody>
 {% for board in boards %}
 <tr>
 <td>
 {{ board.name }}
 <small class="text-muted d-block">{{ board.description }}</small>
 </td>
 <td class="align-middle">0</td>
 <td class="align-middle">0</td>
 <td></td>
 </tr>
 {% endfor %}
 </tbody>
 </table>
{% endblock %}
```

**home.html** 的第一行是 `{% extends 'base.html' %}`。这个标签用来告诉 Django 使用 **base.html** 作为母版页。之后，我们使用 *blocks* 来放置这个页面独有的部分。

## templates/topics.html

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

{% block title %}
 {{ board.name }} - {{ block.super }}
{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boar
ds
 <li class="breadcrumb-item active">{{ board.name }}
{% endblock %}

{% block content %}
 <!-- just leaving it empty for now. we will add core here
soon. -->
{% endblock %}
```

在 **topics.html** 中，我们改变了 `{% block title %}` 的默认值。注意我们可以通过调用 `{{ block.super }}` 来重用 block 的默认值。这里我们使用的网页标题是 **base.html** 中定义的 "Django Boards"。所以对于 "Python" 的 board 页面，这个标题是 "Python - Django Boards", 对于 "Random" board 标题会是 "Random - Django Boards"。

现在我们运行测试，然后会看到我们没有破坏任何东西：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.....

Ran 7 tests in 0.067s

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

棒极了！一切看起来都很成功。

现在我们有了 **base.html** 模板，我们可以很轻松地在顶部添加一个菜单块：

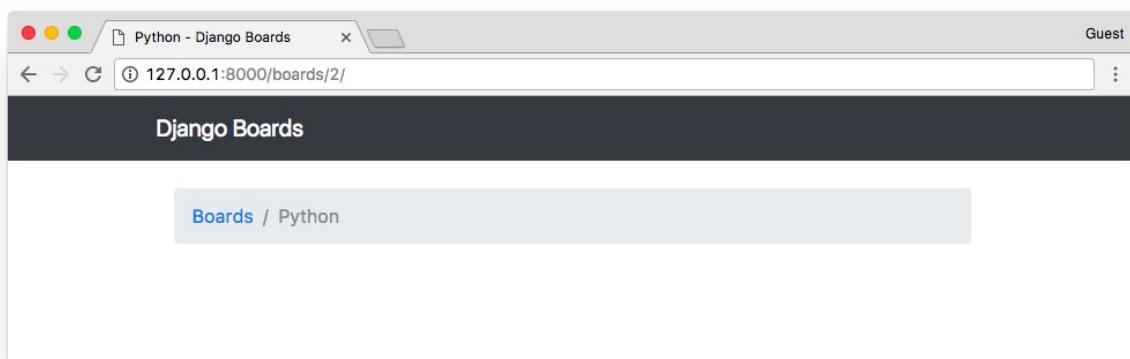
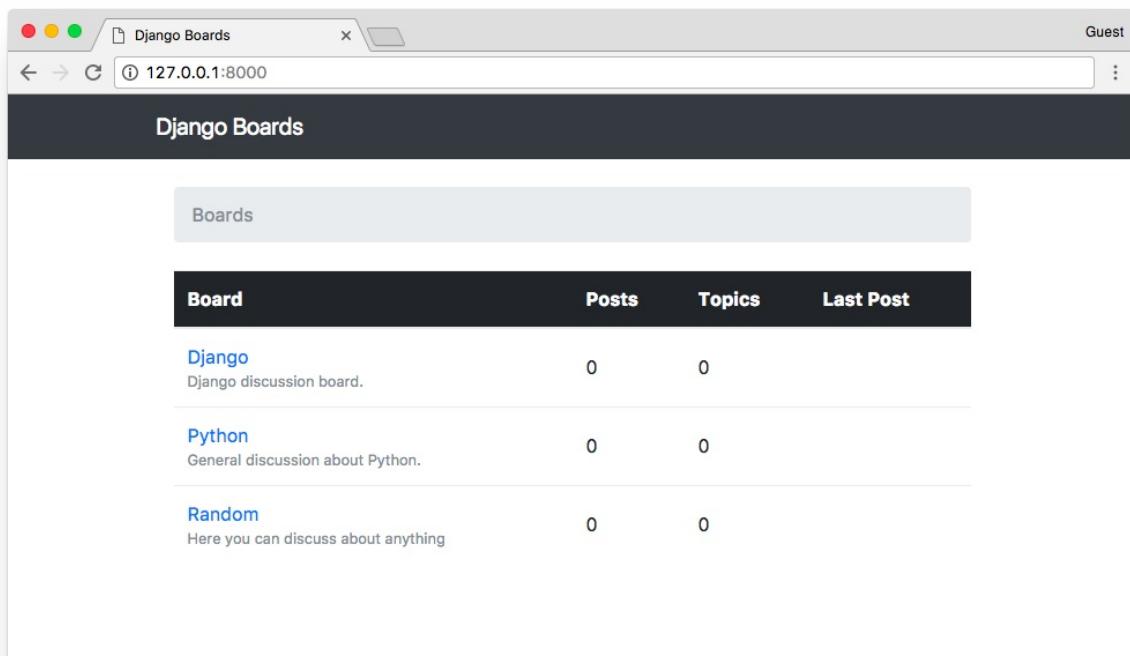
**templates/base.html**

```
{% load static %}<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>{% block title %}Django Boards{% endblock %}</title>
 >
 <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
 </head>
 <body>

 <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
 <div class="container">
 Django Boards
 </div>
 </nav>

 <div class="container">
 <ol class="breadcrumb my-4">
 {% block breadcrumb %}
 {% endblock %}

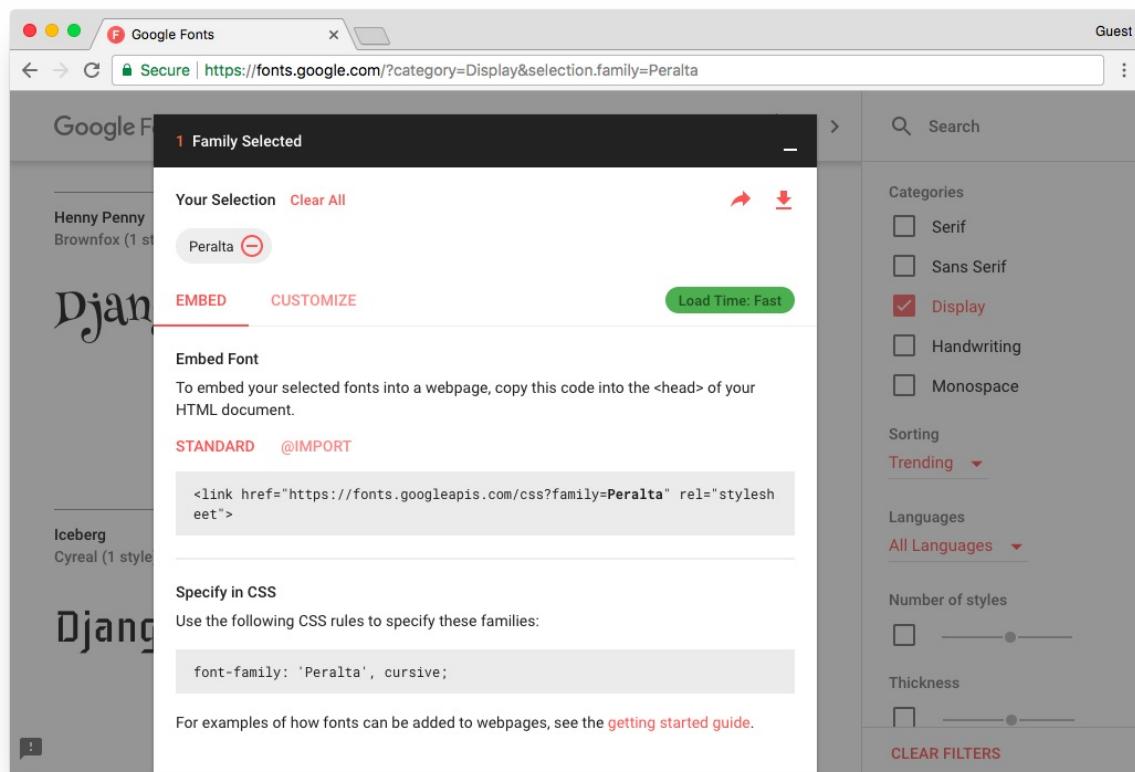
 {% block content %}
 {% endblock %}
 </div>
 </body>
</html>
```



我使用的 HTML 是 [Bootstrap 4 Navbar 组件](#) 的一部分。

我喜欢的一个比较好的改动是改变页面的 "logo"( `.navbar-brand` )。

前往 [fonts.google.com](https://fonts.google.com)，输入 "Django Boards" 或者任何你项目给定的名字然后点击 **apply to all fonts(应用于所有字体)**。浏览一下，找到一个你喜欢的字体。



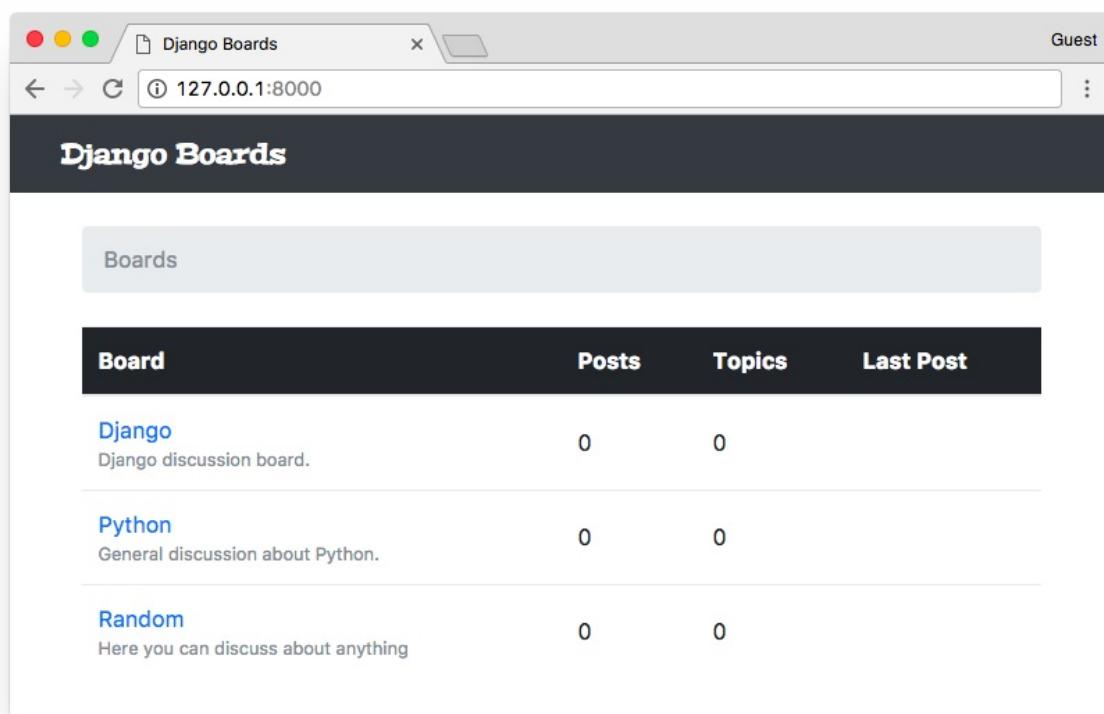
在 **base.html** 模板中添加这个字体：

```
{% load static %}<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>{% block title %}Django Boards{% endblock %}</title>
 <link href="https://fonts.googleapis.com/css?family=Peralta" rel="stylesheet">
 <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
 <link rel="stylesheet" href="{% static 'css/app.css' %}">
 </head>
 <body>
 <!-- code suppressed for brevity -->
 </body>
</html>
```

现在在 **static/css** 文件夹中创建一个新的 CSS 文件命名为 **app.css**:

### static/css/app.css

```
.navbar-brand {
 font-family: 'Peralta', cursive;
}
```



# Django入门与实践-第13章：表单处理

Forms(表单)用来处理我们的输入。这在任何 web 应用或者网站中都是很常见的任务。标准的做法是通过 HTML 表单实现，用户输入一些数据，将其提交给服务器，然后服务器处理它。



表单处理是一项非常复杂的任务，因为它涉及到与应用多个层面的交互。有很多需要关心的问题。例如，提交给服务器的所有数据都是字符串的形式，所以在使用它之前需要将其转换为需要的数据类型(整形，浮点型，日期等)。我们必须验证有关应用程序业务逻辑的数据。我们还需要妥善地清理和审查数据，以避免一些诸如 SQL 注入和 XSS 攻击等安全问题。

好消息是，Django Forms API 使整个过程变的更加简单，从而实现了大量工作的自动化。而且，最终的结果比大多数程序员自己去实现的代码更加安全。所以，不管 HTML 的表单多么简单，总是使用Form API。

## 自己实现表单

起初，我想直接跳到表单 API。但是我觉得花点时间去了解一下表单处理的基本细节是一个不错的主意。否则，这玩意儿将会看起来像魔术一样，这是一件坏事，因为当出现错误时，你将不知道怎么去找到问题所在。

随着对一些编程概念的深入理解，我们可以感觉到自己能更好地掌控一些情况。掌控是很重要的，因为它让我们写代码的时候更有信心。一旦我们能确切地知道发生了什么，实现可预见行为的代码就容易多了。调试和查找错误

也变得很容易，因为你知道去哪里查找。

无论如何，让我们开始实现下面的表单：

The form consists of the following fields:

- Subject**: Hello, everyone!
- Message**: This is my first post... just posting this message to say hello!

**Post**

这是我们在前一个教程绘制的线框图。我现在意识到这个可能是一个不好的例子，因为这个特殊的表单涉及到处理两个不同模型数据：**Topic(subject)** 和 **Post(message)**。

还有一点很重要的我们到现在为止还没讨论过，就是用户认证。我们应该只为登录认证过的用户去显示这个页面。通过这种方式，我们才能知道是谁创建了 **Topic** 或者 **Post**。

现在让我们抽象一些细节，重点了解一下怎么在数据库中保存用户的输入。

首先，先创建一个新的 URL 路由，命名为 **new\_topic**：

**myproject/urls.py**

```

from django.conf.urls import url
from django.contrib import admin

from boards import views

urlpatterns = [
 url(r'^$', views.home, name='home'),
 url(r'^boards/(?P<pk>\d+)/$', views.board_topics, name='board_topics'),
 url(r'^boards/(?P<pk>\d+)/new/$', views.new_topic, name='new_topic'),
 url(r'^admin/', admin.site.urls),
]

```

我们创建的这个 URL 能帮我们标识正确的 **Board**

现在来创建 **new\_topic** 的 视图函数：

### **boards/views.py**

```

from django.shortcuts import render, get_object_or_404
from .models import Board

def new_topic(request, pk):
 board = get_object_or_404(Board, pk=pk)
 return render(request, 'new_topic.html', {'board': board})

```

目前为止， **new\_topic** 的视图函数看起来和 **board\_topics** 恰好相同。这是故意的，让我们一步步地来。

现在我们需要一个名为 **new\_topic.html** 的模板：

### **templates/new\_topic.html**

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

{% block title %}Start a New Topic{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boar
ds
 <li class="breadcrumb-item"><a href="{% url 'board_topics'
board.pk %}">{{ board.name }}
 <li class="breadcrumb-item active">New topic
{% endblock %}

{% block content %}

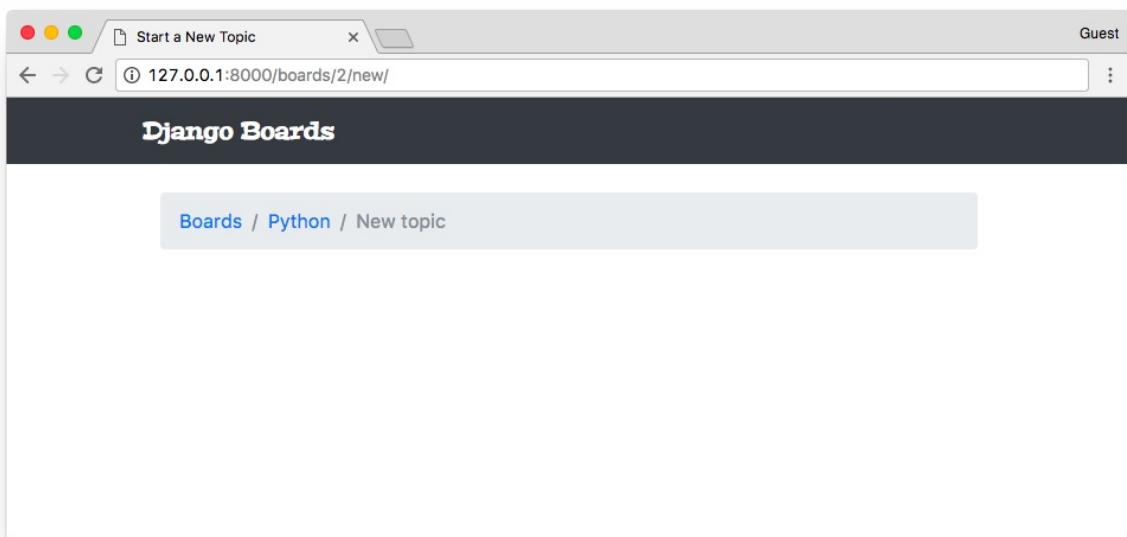
{% endblock %}
```

现在我们只有 breadcrumb 导航。注意我们包含了返回到 **board\_topics** 视图 URL。

打开 URL <http://127.0.0.1:8000/boards/1/new/>。显示结果是下面这个页面：

!(3-17.png)[./statics/3-17.png]

我们依然还没有实现到达这个新页面的方法，但是如果我们将 URL 改为 <http://127.0.0.1:8000/boards/2/new/>，它会把我们带到 **Python Board** 的页面：



注意：如果你没有跟着上一节课一步一步地做，你的结果和我的可能有些不一样。在我这个例子中，我的数据库有 3 个 **Board** 实例，分别是 Django = 1, Python = 2, 和 Random = 3。这些数字是数据库中的 ID，用来找到正确的资源。

我们可以增加一些测试了：

### boards/tests.py

```
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase
from .views import home, board_topics, new_topic
from .models import Board

class HomeTests(TestCase):
 # ...

class BoardTopicsTests(TestCase):
 # ...

class NewTopicTests(TestCase):
 def setUp(self):
 Board.objects.create(name='Django', description='Djan
```

```

go board.')

def test_new_topic_view_success_status_code(self):
 url = reverse('new_topic', kwargs={'pk': 1})
 response = self.client.get(url)
 self.assertEqual(response.status_code, 200)

def test_new_topic_view_not_found_status_code(self):
 url = reverse('new_topic', kwargs={'pk': 99})
 response = self.client.get(url)
 self.assertEqual(response.status_code, 404)

def test_new_topic_url_resolves_new_topic_view(self):
 view = resolve('/boards/1/new/')
 self.assertEqual(view.func, new_topic)

def test_new_topic_view_contains_link_back_to_board_topics_view(self):
 new_topic_url = reverse('new_topic', kwargs={'pk': 1})
 board_topics_url = reverse('board_topics', kwargs={'pk': 1})
 response = self.client.get(new_topic_url)
 self.assertContains(response, 'href="{0}"'.format(board_topics_url))

```

关于我们的测试中新的 NewTopicTests 类的快速总结：

- **setUp**: 创建一个测试中使用的 **Board** 实例
- **test\_new\_topic\_view\_success\_status\_code**: 检查发给 view 的请求是否成功
- **test\_new\_topic\_view\_not\_found\_status\_code**: 检查当 **Board** 不存在时 view 是否会抛出一个 404 的错误
- **test\_new\_topic\_url\_resolves\_new\_topic\_view**: 检查是否正在使用正确的 view
- **test\_new\_topic\_view\_contains\_link\_back\_to\_board\_topics\_view**

：确保导航能回到 topics 的列表

运行测试：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
```

```
.....
```

```

```

```
Ran 11 tests in 0.076s
```

```
OK
```

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

成功，现在我们可以去开始创建表单了。

**templates/new\_topic.html**

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

{% block title %}Start a New Topic{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boar
ds
 <li class="breadcrumb-item"><a href="{% url 'board_topics'
board.pk %}">{{ board.name }}
 <li class="breadcrumb-item active">New topic
{% endblock %}

{% block content %}
 <form method="post">
 {% csrf_token %}
 <div class="form-group">
 <label for="id_subject">Subject</label>
 <input type="text" class="form-control" id="id_subject"
name="subject">
 </div>
 <div class="form-group">
 <label for="id_message">Message</label>
 <textarea class="form-control" id="id_message" name="me
ssage" rows="5"></textarea>
 </div>
 <button type="submit" class="btn btn-success">Post</button
>
 </form>
{% endblock %}
```

这是一个使用 Bootstrap 4 提供的 CSS 类手动创建的 HTML 表单。它看起来是这个样子：

![3-19.png][./statics/3-19.png]

在 `<form>` 标签中，我们定义了 `method` 属性。它会告诉浏览器我们想如何与服务器通信。HTTP 规范定义了几种 request methods(请求方法)。但是在大部分情况下，我们只需要使用 **GET** 和 **POST** 两种 request(请求)类型。

**GET** 可能是最常见的请求类型了。它用于从服务器请求数据。每当你点击了一个链接或者直接在浏览器中输入了一个网址时，你就创建一个 **GET** 请求。

**POST** 用于当我们想更改服务器上的数据的时候。一般来说，每次我们发送数据给服务器都会导致资源状态的变化，我们应该使用 **POST** 请求发送数据。

Django 使用 **CSRF Token**(Cross-Site Request Forgery Token) 保护所有的 **POST** 请求。这是一个避免外部站点或者应用程序向我们的应用程序提交数据的安全措施。应用程序每次接收一个 **POST** 时，都会先检查 **CSRF Token**。如果这个 request 没有 token，或者这个 token 是无效的，它就会抛弃提交的数据。

**csrf\_token** 的模板标签：

```
{% csrf_token %}
```

它是与其他表单数据一起提交的隐藏字段：

```
<input type="hidden" name="csrfmiddlewaretoken" value="jG2o6a
Wj65YGaqzCpl0TYTg5jn6SctjzRZ9Kmluifvx0IVax1wh97YarZKs54Y32">
```

另外一件事是，我们需要设置 HTML 输入的 `name`，`name` 将被用来在服务器获取数据。

```
<input type="text" class="form-control" id="id_subject" name="subject">
<textarea class="form-control" id="id_message" name="message" rows="5"></textarea>
```

下面是示范我们如何检索数据：

```
subject = request.POST['subject']
message = request.POST['message']
```

所以，从 HTML 获取数据并且开始一个新的 topic 视图的简单实现可以这样写：

```
from django.contrib.auth.models import User
from django.shortcuts import render, redirect, get_object_or_404
from .models import Board, Topic, Post

def new_topic(request, pk):
 board = get_object_or_404(Board, pk=pk)

 if request.method == 'POST':
 subject = request.POST['subject']
 message = request.POST['message']

 user = User.objects.first() # TODO: 临时使用一个账号作为登录用户

 topic = Topic.objects.create(
 subject=subject,
 board=board,
 starter=user
)

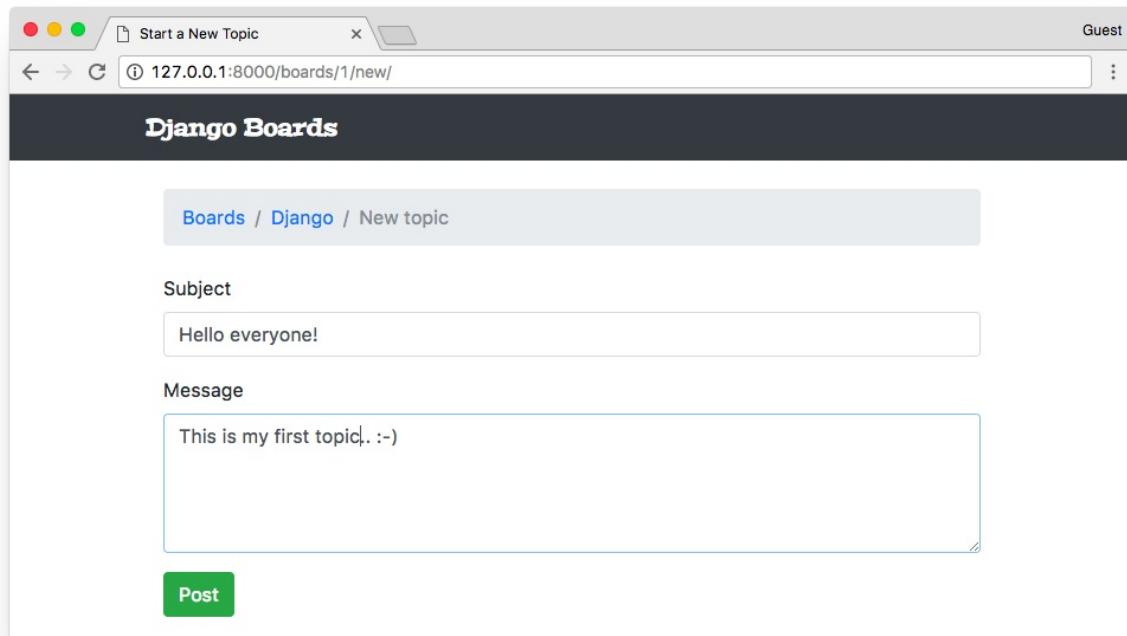
 post = Post.objects.create(
 message=message,
 topic=topic,
 created_by=user
)

 return redirect('board_topics', pk=board.pk) # TODO: redirect to the created topic page

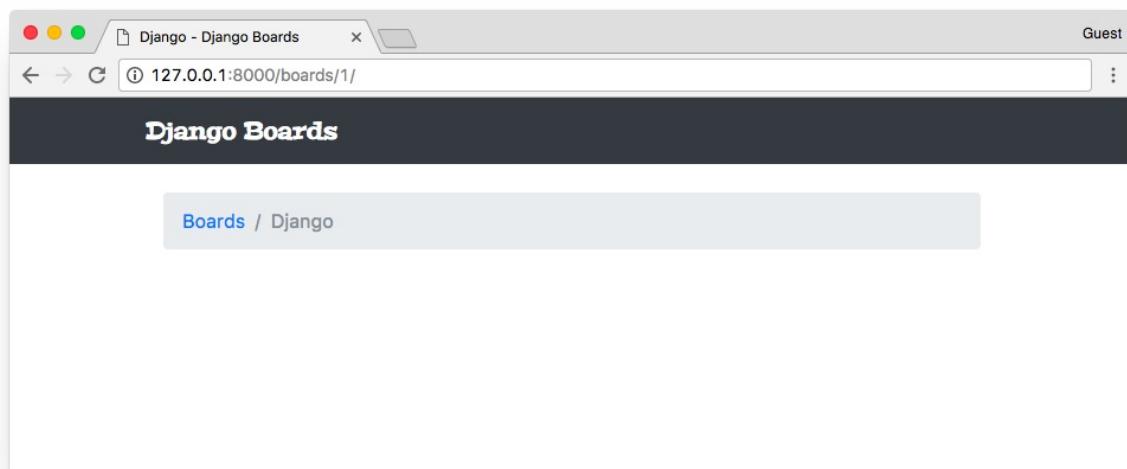
return render(request, 'new_topic.html', {'board': board})
```

这个视图函数只考虑能接收数据并且保存进数据库的乐观合法的 path，但是还缺少一些部分。我们没有验证数据。用户可以提交空表单或者提交一个大于 255 个字符的 **subject**。

到目前为止我们都在对 **User** 字段进行硬编码，因为我们还没有实现身份验证。有一个简单的方法来识别登录的用户。我们会在下一个课程将这一块。此外，我们还没有实现列出 topic 的所有 posts 的视图，实现了它，我们就可以将用户重定向到列出所有主题的列表页面。



点击 **Post** 按钮提交表单：



看起来成功了。但是我们还没有实现主题的列表页面，所以没有东西可以看。让我们来编辑 **templates/topics.html** 来实现一个合适的列表：

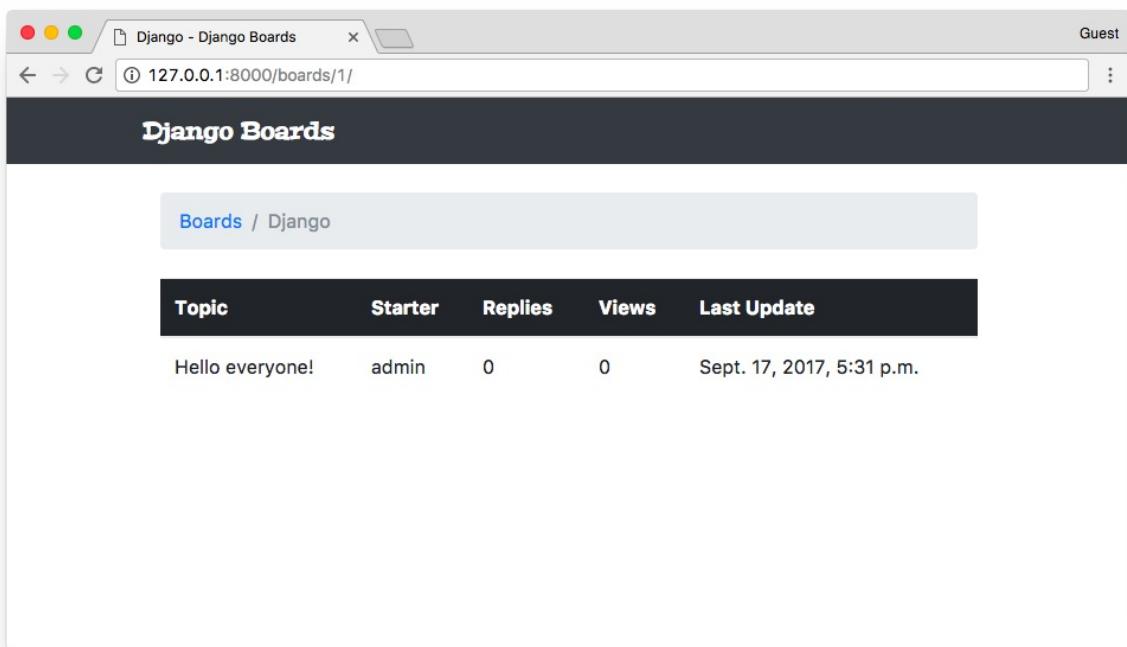
## **templates/topics.html**

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

{% block title %}
 {{ board.name }} - {{ block.super }}
{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boar
ds
 <li class="breadcrumb-item active">{{ board.name }}
{% endblock %}

{% block content %}
 <table class="table">
 <thead class="thead-inverse">
 <tr>
 <th>Topic</th>
 <th>Starter</th>
 <th>Replies</th>
 <th>Views</th>
 <th>Last Update</th>
 </tr>
 </thead>
 <tbody>
 {% for topic in board.topics.all %}
 <tr>
 <td>{{ topic.subject }}</td>
 <td>{{ topic.starter.username }}</td>
 <td>0</td>
 <td>0</td>
 <td>{{ topic.last_updated }}</td>
 </tr>
 {% endfor %}
 </tbody>
 </table>
{% endblock %}
```



我们创建的 **Topic** 显示在这上面了。

这里有两个新概念。

我们首次使用 **Board** 模型中的 **topics** 属性。**topics** 属性由 Django 使用反向关系自动创建。在之前的步骤中，我们创建了一个 **Topic** 实例：

```
def new_topic(request, pk):
 board = get_object_or_404(Board, pk=pk)

 #
 # ...

 topic = Topic.objects.create(
 subject=subject,
 board=board,
 starter=user
)
```

在 `board=board` 这行，我们设置了 **Topic** 模型中的 `board` 字段，它是 `ForeignKey(Board)`。因此，我们的 **Board** 实例就知道了与它关联的 **Topic** 实例。

之所以我们使用 `board.topics.all` 而不是 `board.topics`，是因为 `board.topics` 是一个 **Related Manager**，它与 **Model Manager** 很相似，通常在 `board.objects` 可得到。所以，要返回给定 board 的所有 topic 我们必须使用 `board.topics.all()`，要过滤一些数据，我们可以这样用 `board.topics.filter(subject__contains='Hello')`。

另一个需要注意的是，在 Python 代码中，我们必须使用括号：`board.topics.all()`，因为 `all()` 是一个方法。在使用 Django 模板语言写代码的时候，在一个 HTML 模板文件里面，我们不使用括号，就只是 `board.topics.all`。

第二件事是我们在使用 `ForeignKey`：

```
{% topic.starter.username %}
```

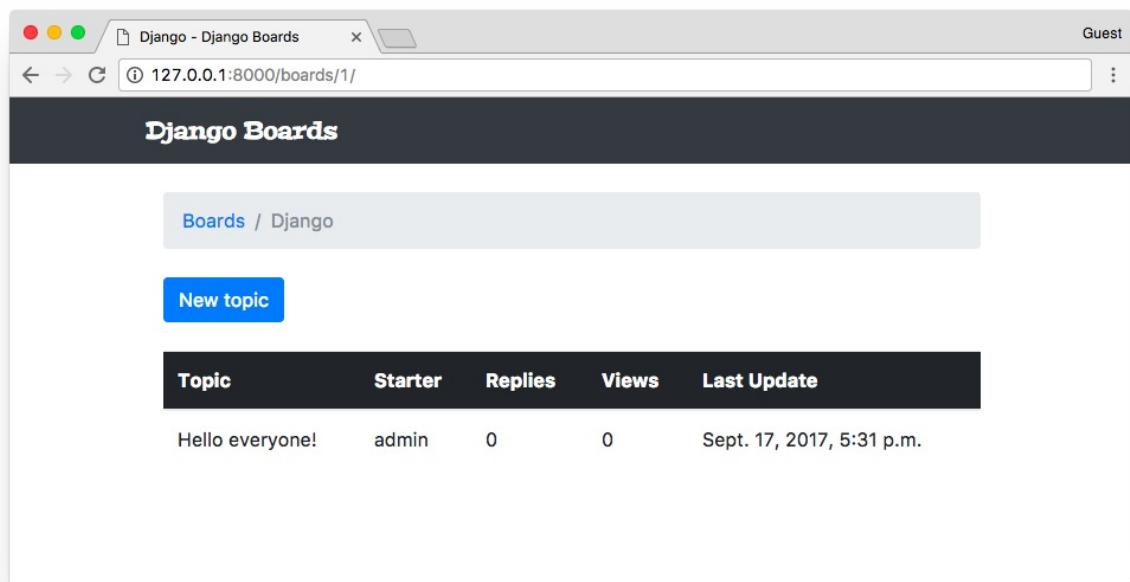
使用一个点加上属性这种写法，我们几乎可以访问 **User** 模型的所有属性。如果我们想得到用户的 email，我们可以使用 `topic.starter.email`。

我们已经修改了 `topics.html` 模板，让我们创建一个能让我们转到 `new topic` 页面的按钮：

### templates/topics.html

```
{% block content %}
<div class="mb-4">
 New topic
</div>

<table class="table">
 <!-- code suppressed for brevity -->
</table>
{% endblock %}
```



我们可以写一个测试以确保用户可以通过此页面访问到 **New Topic** 页面：

### boards/tests.py

```
class BoardTopicsTests(TestCase):
 # ...

 def test_board_topics_view_contains_navigation_links(self):
 board_topics_url = reverse('board_topics', kwargs={'pk': 1})
 homepage_url = reverse('home')
 new_topic_url = reverse('new_topic', kwargs={'pk': 1})

 response = self.client.get(board_topics_url)

 self.assertContains(response, 'href="{0}"'.format(homepage_url))
 self.assertContains(response, 'href="{0}"'.format(new_topic_url))
```

我在这里基本上重命名了

**test\_board\_topics\_view\_contains\_link\_back\_to\_homepage** 方法并添加了一个额外的 `assertContains`。这个测试现在负责确保我们的 view 包含所需的导航链接。

## 测试表单

在我们使用 Django 的方式编写之前的表单示例之前，让我们先为表单处理写一些测试：

### boards/tests.py

```
''' new imports below '''
from django.contrib.auth.models import User
from .views import new_topic
from .models import Board, Topic, Post

class NewTopicTests(TestCase):
 def setUp(self):
 Board.objects.create(name='Django', description='Django board.')
 User.objects.create_user(username='john', email='john@doe.com', password='123') # <- included this line here

 # ...

 def test_csrf(self):
 url = reverse('new_topic', kwargs={'pk': 1})
 response = self.client.get(url)
 self.assertContains(response, 'csrfmiddlewaretoken')

 def test_new_topic_valid_post_data(self):
 url = reverse('new_topic', kwargs={'pk': 1})
 data = {
 'subject': 'Test title',
 'message': 'Lorem ipsum dolor sit amet'
 }
```

```

 response = self.client.post(url, data)
 self.assertTrue(Topic.objects.exists())
 self.assertTrue(Post.objects.exists())

 def test_new_topic_invalid_post_data(self):
 """
 Invalid post data should not redirect
 The expected behavior is to show the form again with
 validation errors
 """

 url = reverse('new_topic', kwargs={'pk': 1})
 response = self.client.post(url, {})
 self.assertEqual(response.status_code, 200)

 def test_new_topic_invalid_post_data_empty_fields(self):
 """
 Invalid post data should not redirect
 The expected behavior is to show the form again with
 validation errors
 """

 url = reverse('new_topic', kwargs={'pk': 1})
 data = {
 'subject': '',
 'message': ''
 }
 response = self.client.post(url, data)
 self.assertEqual(response.status_code, 200)
 self.assertFalse(Topic.objects.exists())
 self.assertFalse(Post.objects.exists())

```

首先， **test.py** 文件变的越来越大。我们会尽快改进它，将测试分为几个文件。但现在，让我们先保持这个状态。

- **setUp**: 包含 `User.objects.create_user` 以创建用于测试的 **User** 实例。
- 
- **test\_csrf**: 由于 **CSRF Token** 是处理 **Post** 请求的基本部分，我们需要

保证我们的 HTML 包含 token。

- 
- **test\_new\_topic\_valid\_post\_data**: 发送有效的数据并检查视图函数是否创建了 **Topic** 和 **Post** 实例。
- 
- **test\_new\_topic\_invalid\_post\_data**: 发送一个空字典来检查应用的行为。
- 
- **test\_new\_topic\_invalid\_post\_data\_empty\_fields**: 类似于上一个测试，但是这次我们发送一些数据。预期应用程序会验证并且拒绝空的 subject 和 message。

运行这些测试：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....EF.....
=====
=====
ERROR: test_new_topic_invalid_post_data (boards.tests.NewTopicTests)

Traceback (most recent call last):
...
django.utils.datastructures.MultiValueDictKeyError: "'subject'"
=====

=====
FAIL: test_new_topic_invalid_post_data_empty_fields (boards.tests.NewTopicTests)

Traceback (most recent call last):
 File "/Users/vitorfs/Development/myproject/django-beginners-guide/boards/tests.py", line 115, in test_new_topic_invalid_post_data_empty_fields
 self.assertEqual(response.status_code, 200)
AssertionError: 302 != 200

Ran 15 tests in 0.512s

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

有一个失败的测试和一个错误。两个都与验证用户的输入有关。不要试图用当前的实现来修复它，让我们通过使用 Django Forms API 来通过这些测试

## 创建表单正确的姿势

自从我们开始使用 Forms，我们已经走了很长一段路。终于，是时候使用 Forms API 了。

Forms API 可在模块 `django.forms` 中得到。Django 使用两种类型的 form：`forms.Form` 和 `forms.ModelForm`。`Form` 类是通用的表单实现。我们可以使用它来处理与应用程序 model 没有直接关联的数据。`ModelForm` 是 `Form` 的子类，它与 model 类相关联。

在 **boards** 文件夹下创建一个新的文件 `forms.py`：

### boards/forms.py

```
from django import forms
from .models import Topic

class NewTopicForm(forms.ModelForm):
 message = forms.CharField(widget=forms.Textarea(), max_length=4000)

 class Meta:
 model = Topic
 fields = ['subject', 'message']
```

这是我们的第一个 form。它是一个与 **Topic** model 相关联的 `ModelForm`。**Meta** 类里面 `fields` 列表中的 `subject` 引用 **Topic** 类中的 `subject` field(字段)。现在注意到我们定义了一个叫做 `message` 的额外字段。它用来引用 **Post** 中我们想要保存的 `message`。

现在我们需要重写我们的 `views.py`：

### boards/views.py

```
from django.contrib.auth.models import User
from django.shortcuts import render, redirect, get_object_or_404
from .forms import NewTopicForm
from .models import Board, Topic, Post

def new_topic(request, pk):
 board = get_object_or_404(Board, pk=pk)
 user = User.objects.first() # TODO: get the currently logged in user
 if request.method == 'POST':
 form = NewTopicForm(request.POST)
 if form.is_valid():
 topic = form.save(commit=False)
 topic.board = board
 topic.starter = user
 topic.save()
 post = Post.objects.create(
 message=form.cleaned_data.get('message'),
 topic=topic,
 created_by=user
)
 return redirect('board_topics', pk=board.pk) # TODO: redirect to the created topic page
 else:
 form = NewTopicForm()
 return render(request, 'new_topic.html', {'board': board, 'form': form})
```

这是我们在 view(视图) 中处理 form(表单) 的方式。让我们去掉一些多余的部分，只看表单处理的核心部分：

```
if request.method == 'POST':
 form = NewTopicForm(request.POST)
 if form.is_valid():
 topic = form.save()
 return redirect('board_topics', pk=board.pk)
else:
 form = NewTopicForm()
return render(request, 'new_topic.html', {'form': form})
```

首先我们判断请求是 **POST** 还是 **GET**。如果请求是 **POST**，这意味着用户向服务器提交了一些数据。所以我们实例化一个将 **POST** 数据传递给 form 的 form 实例：`form = NewTopicForm(request.POST)`。

然后，我们让 Django 验证数据，检查 form 是否有效，我们能否将其存入数据库：`if form.is_valid():`。如果表单有效，我们使用 `form.save()` 将数据存入数据库。`save()` 方法返回一个存入数据库的 Model 实例。所以，因为这是一个 **Topic** form，所以它会返回 `topic = form.save()` 创建的 **Topic**。然后，通用的路径是把用户重定向到其他地方，以避免用户通过按 F5 重新提交表单，并且保证应用程序的流程走向。

现在，如果数据是无效的，Django 会给 form 添加错误列表。然后，视图函数不会做任何处理并且返回最后一句：`return render(request, 'new_topic.html', {'form': form})`。这意味着我们需要更新 **new\_topic.html** 以显示错误。

如果请求是 **GET**，我们只需要使用 `form = NewTopicForm()` 初始化一个新的空表单。

让我们运行测试并观察情况：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.....

Ran 15 tests in 0.522s

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

我们甚至修复了最后两个测试。

Django Forms API 不仅仅是处理和验证数据。它还为我们生成 HTML。

### templates/new\_topic.html

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

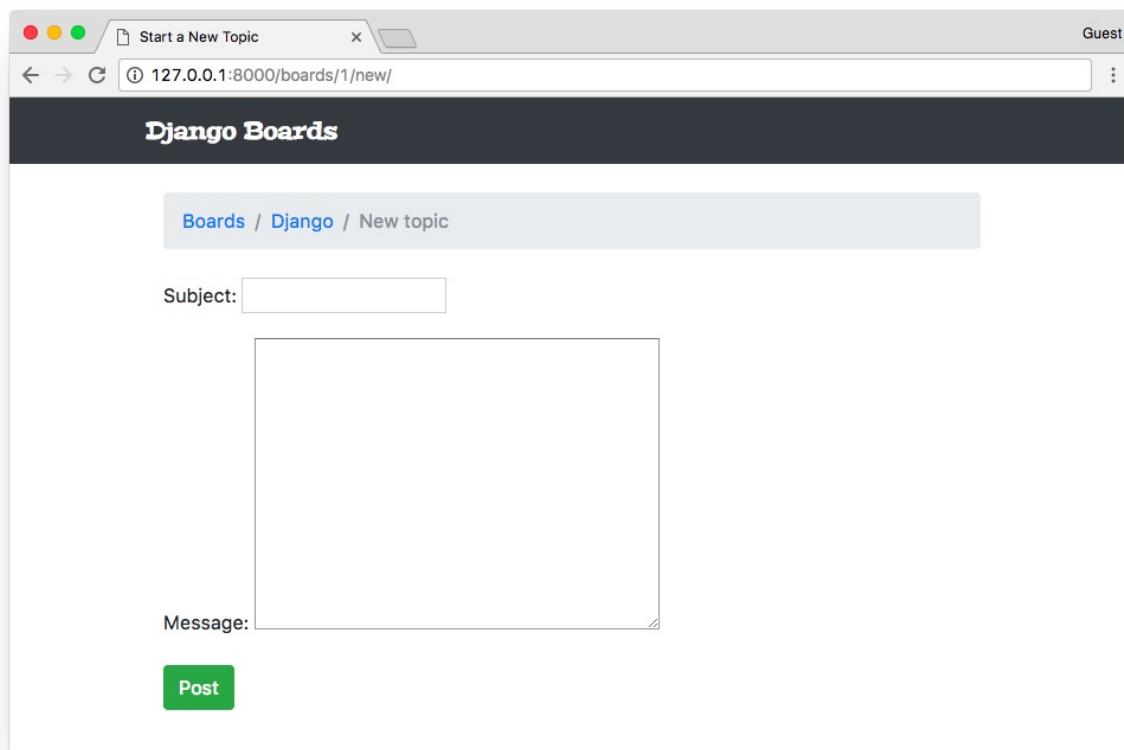
{% block title %}Start a New Topic{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boar
ds
 <li class="breadcrumb-item"><a href="{% url 'board_topics'
board.pk %}">{{ board.name }}
 <li class="breadcrumb-item active">New topic
{% endblock %}

{% block content %}
 <form method="post">
 {% csrf_token %}
 {{ form.as_p }}
 <button type="submit" class="btn btn-success">Post</button
>
 </form>
{% endblock %}
```

这个 `form` 有三个渲染选项：`form.as_table`，`form.as_ul` 和 `form.as_p`。这是一个快速的渲染表单所有字段的方法。顾名思义，`as_table` 使用 `table` 标签来格式化输入，`as_ul` 使用 `li` 标签。

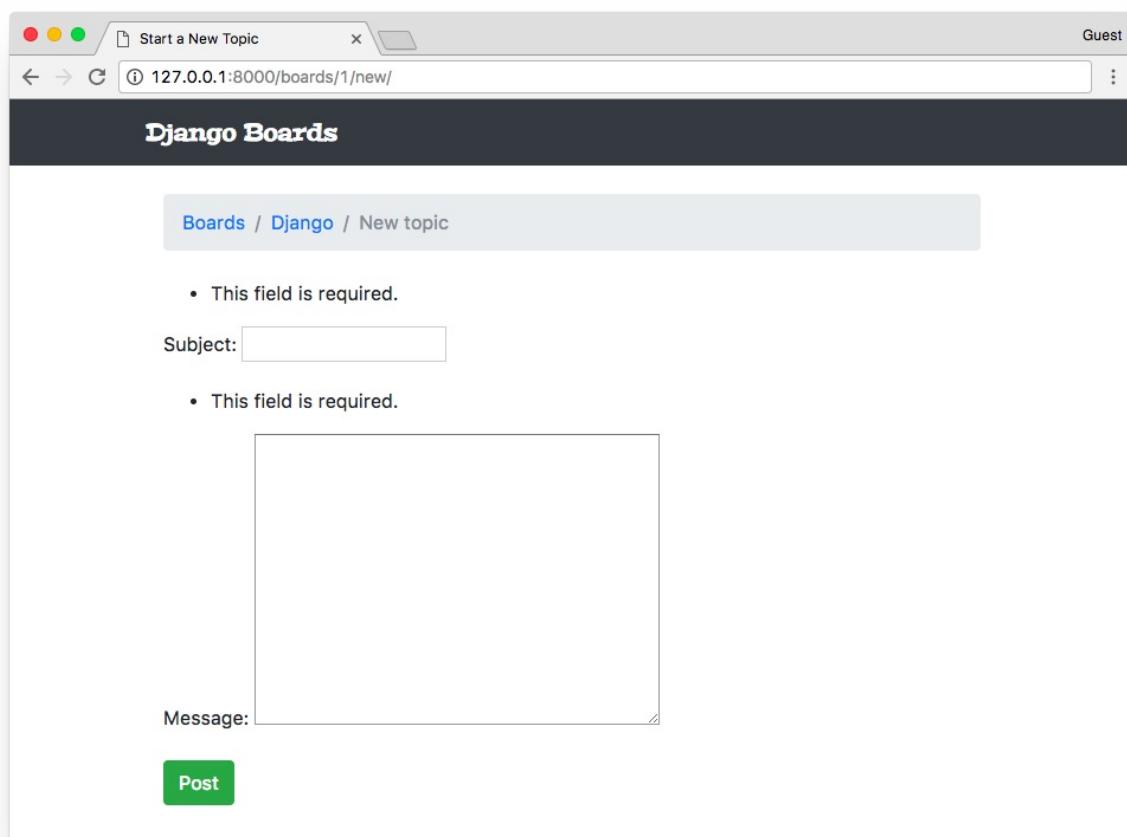
看看效果：



我们以前的 `form` 看起来更好，是吧？我们将立即修复它。

它看起来很破，但是相信我；它背后有很多东西。它非常强大。比如，如果我们的表单有 50 个字段，我们可以通过键入 `{{ form.as_p }}` 来显示所有字段。

此外，使用 Forms API，Django 会验证数据并且向每个字段添加错误消息。让我们尝试提交一个空的表单：



注意：如果你提交表单时看到类似这样的东西：

 Please fill out this field. , 这不是 Django 导致的，而是你的浏览器进行预验证。要禁用它可以在你的表单标签中添加 **novalidate** 属性：

你可以不修改它，不会有问题是。这只是因为我们的表单现在非常简单，而且我们没有太多的数据验证可以看到。

另外一件需要注意的事情是：没有“只客户端验证”这样的事情。JavaScript 验证或者浏览器验证仅用于可用性目的。同时也减少了对服务器的请求数量。数据验证应该始终在服务器端完成，这样我们可以完全掌控数据。

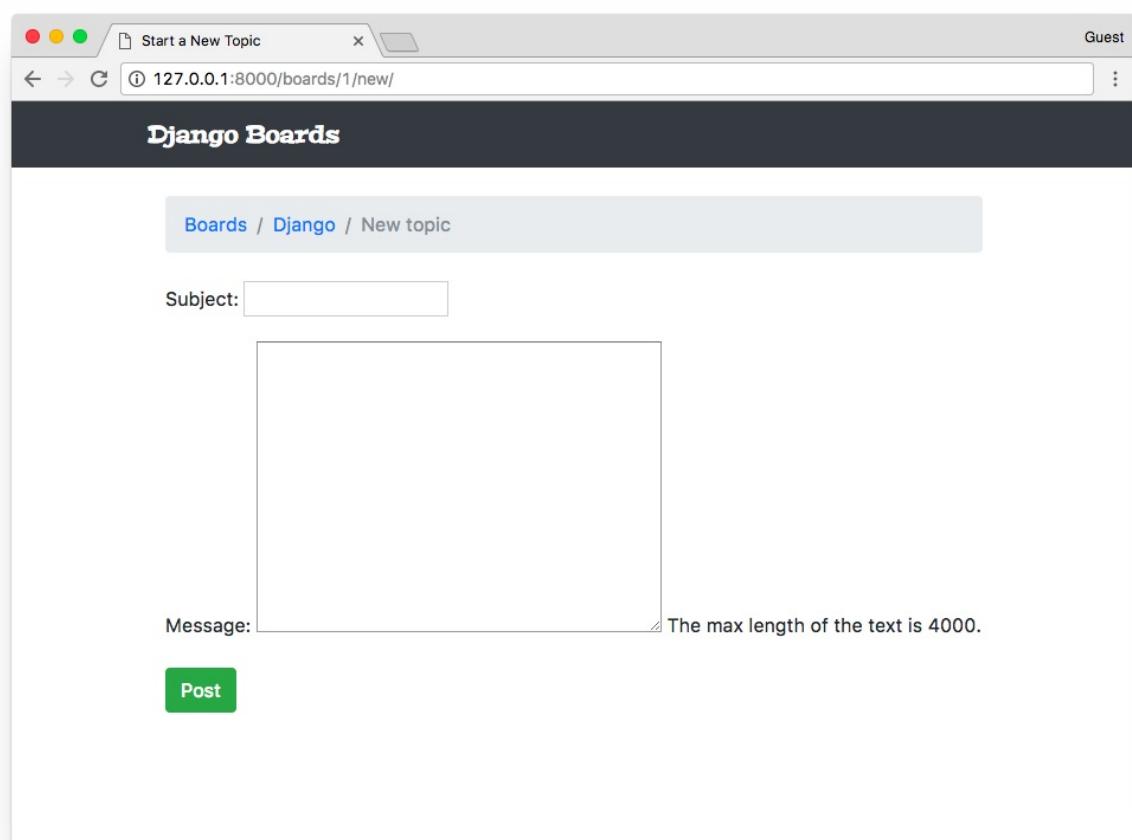
它还可以处理在 **Form** 类或者 **Model** 类中定义的帮助文本。

## boards/forms.py

```
from django import forms
from .models import Topic

class NewTopicForm(forms.ModelForm):
 message = forms.CharField(
 widget=forms.Textarea(),
 max_length=4000,
 help_text='The max length of the text is 4000.'
)

 class Meta:
 model = Topic
 fields = ['subject', 'message']
```



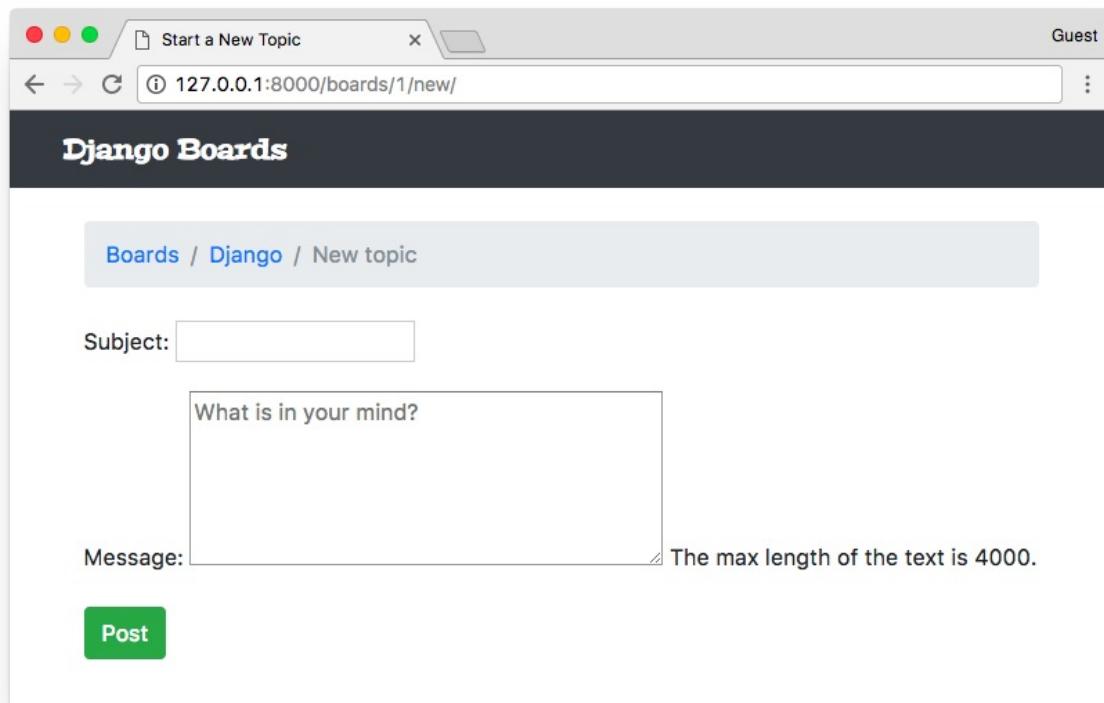
我们也可以为表单字段设置额外的属性：

## boards/forms.py

```
from django import forms
from .models import Topic

class NewTopicForm(forms.ModelForm):
 message = forms.CharField(
 widget=forms.Textarea(
 attrs={'rows': 5, 'placeholder': 'What is on your
mind?'}),
 max_length=4000,
 help_text='The max length of the text is 4000.')
)

 class Meta:
 model = Topic
 fields = ['subject', 'message']
```



## 用Bootstrap 表单渲染

现在让我们把事情做得更完善。

当使用 Bootstrap 或者其他的前端库时，我比较喜欢使用一个叫做 **django-widget-tweaks** 的 Django 库。它可以让我们更好地控制渲染的处理，在保证默认值的情况下，只需在上面添加额外的自定义设置。

开始安装它：

```
pip install django-widget-tweaks
```

添加到 `INSTALLED_APPS`：

**myproject/settings.py**

```
INSTALLED_APPS = [
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',

 'widget_tweaks',

 'boards',
]
```

现在可以使用它了：

**templates/new\_topic.html**

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

{% load widget_tweaks %}

{% block title %}Start a New Topic{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boards
 <li class="breadcrumb-item">{{ board.name }}
 <li class="breadcrumb-item active">New topic
{% endblock %}

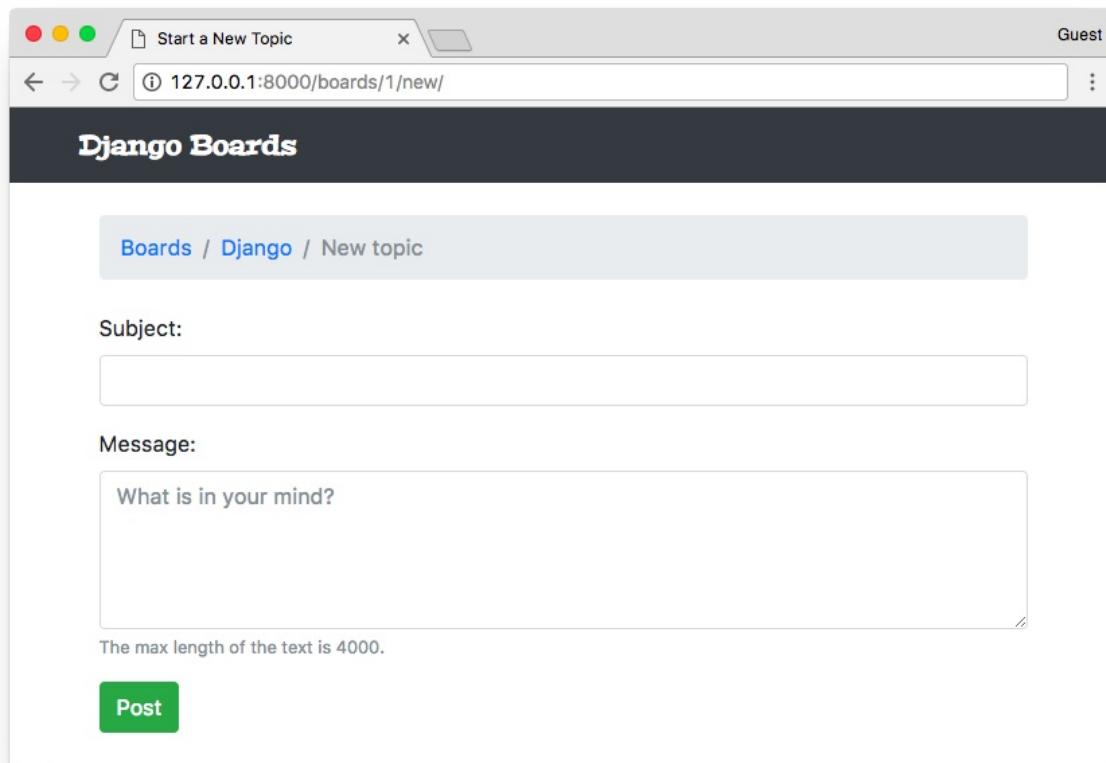
{% block content %}
 <form method="post" novalidate>
 {% csrf_token %}

 {% for field in form %}
 <div class="form-group">
 {{ field.label_tag }}

 {% render_field field class="form-control" %}

 {% if field.help_text %}
 <small class="form-text text-muted">
 {{ field.help_text }}
 </small>
 {% endif %}
 </div>
 {% endfor %}

 <button type="submit" class="btn btn-success">Post</button>
 </form>
{% endblock %}
```



这就是我们使用的 **django-widget-tweaks** 的效果。首先，我们使用 `{% load widget_tweaks %}` 模板标签将其加载到模板。然后这样使用它：

```
{% render_field field class="form-control" %}
```

`render_field` 不属于 Django；它存在于我们安装的包里面。要使用它，我们需要传递一个表单域实例作为第一个参数，然后我们可以添加任意的 HTML 属性去补充它。这很有用因为我们可以根据特定的条件指定类。

一些 `render_field` 模板标签的例子：

```
{% render_field form.subject class="form-control" %}
{% render_field form.message class="form-control" placeholder=form.message.label %}
{% render_field field class="form-control" placeholder="Write a message!" %}
{% render_field field style="font-size: 20px" %}
```

现在要实现 Bootstrap 4 验证标签，我们可以修改 **new\_topic.html** 模板。

**templates/new\_topic.html**

```
<form method="post" novalidate>
 {% csrf_token %}

 {% for field in form %}
 <div class="form-group">
 {{ field.label_tag }}

 {% if form.is_bound %}
 {% if field.errors %}

 {% render_field field class="form-control is-invalid" %}

 {% for error in field.errors %}
 <div class="invalid-feedback">
 {{ error }}
 </div>
 {% endfor %}

 {% else %}
 {% render_field field class="form-control is-valid" %}

 {% endif %}
 {% else %}
 {% render_field field class="form-control" %}

 {% endif %}

 {% if field.help_text %}
 <small class="form-text text-muted">
 {{ field.help_text }}
 </small>
 {% endif %}
 </div>
 {% endfor %}

 <button type="submit" class="btn btn-success">Post</button>
</form>
```

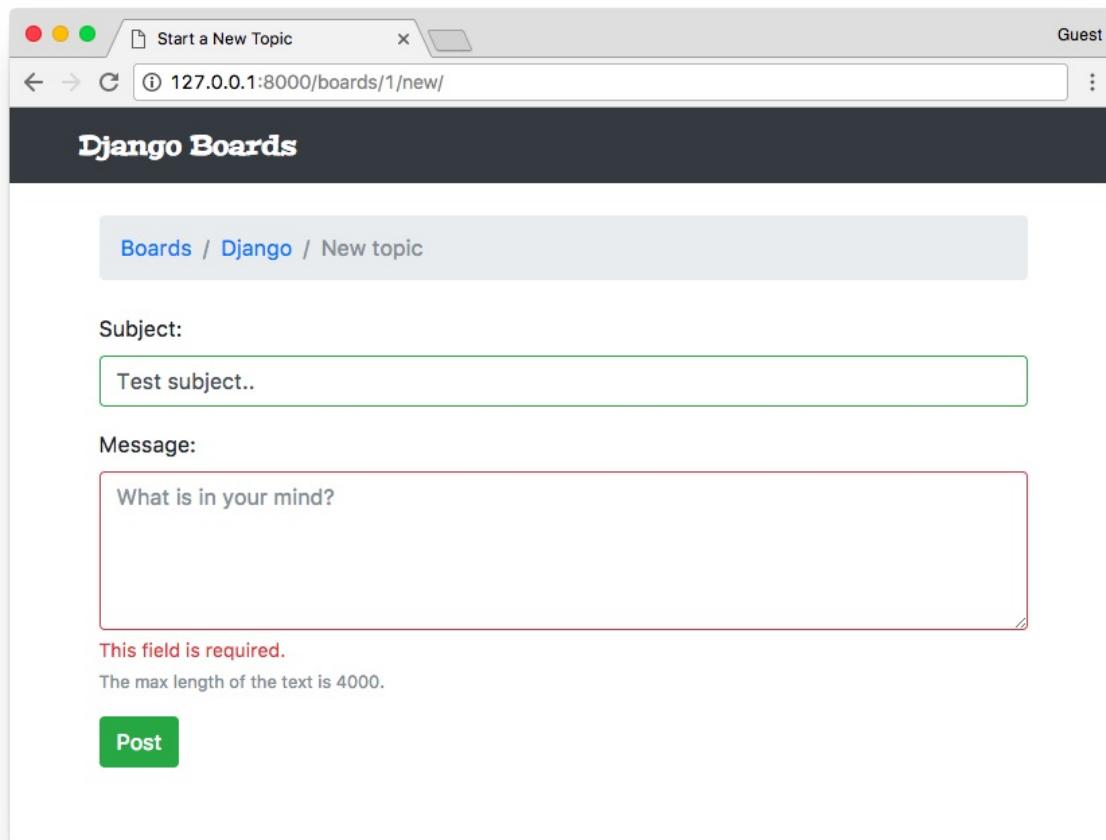
效果是：

The screenshot shows a web browser window titled "Start a New Topic". The URL in the address bar is "127.0.0.1:8000/boards/1/new/". The page header says "Guest". The main content area is titled "Django Boards" and shows a breadcrumb trail: "Boards / Django / New topic".

The form has two fields:

- Subject:** A text input field with a red border. Below it, the error message "This field is required." is displayed in red.
- Message:** A larger text input field with a red border. Below it, the error message "This field is required." is displayed in red, followed by the note "The max length of the text is 4000."

A green "Post" button is located at the bottom left of the form.



所以，我们有三种不同的渲染状态：

- **Initial state**: 表单没有数据(不受约束)
- **Invalid**: 我们添加了 `.is-invalid` 这个 CSS class 并将错误消息添加到具有 `.invalid-feedback` class 的元素中
- **Valid**: 我们添加了 `.is-valid` 的 CSS class, 以绿色绘制表单域, 并向用户反馈它是否可行。

## 复用表单模板

模板看起来有点复杂, 是吧? 有个好消息是我们可以在项目中重复使用它。

在 **templates** 文件夹中, 创建一个新的文件夹命名为 **includes**:

```
myproject/
| -- myproject/
| | -- boards/
| | -- myproject/
| | -- templates/
| | | -- includes/ <-- here!
| | | -- base.html
| | | -- home.html
| | | -- new_topic.html
| | +-- topics.html
| +-- manage.py
+-- venv/
```

在 **includes** 文件夹中，创建一个 **form.html**:

**templates/includes/form.html**

```
{% load widget_tweaks %}

{% for field in form %}
<div class="form-group">
{{ field.label_tag }}

{% if form.is_bound %}
{% if field.errors %}
 {% render_field field class="form-control is-invalid" %}
%
 {% for error in field.errors %}
 <div class="invalid-feedback">
 {{ error }}
 </div>
 {% endfor %}
 {% else %}
 {% render_field field class="form-control is-valid" %}
 %
 {% endif %}
 {% else %}
 {% render_field field class="form-control" %}
 {% endif %}

 {% if field.help_text %}
 <small class="form-text text-muted">
 {{ field.help_text }}
 </small>
 {% endif %}
</div>
{% endfor %}
```

现在来修改我们的 `new_topic.html` 模板：

`templates/new_topic.html`

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

{% block title %}Start a New Topic{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boar
ds
 <li class="breadcrumb-item"><a href="{% url 'board_topics'
board.pk %}">{{ board.name }}
 <li class="breadcrumb-item active">New topic
{% endblock %}

{% block content %}
 <form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-success">Post</button
>
 </form>
{% endblock %}
```

顾名思义，`{% include %}` 用来在其他的模板中包含 HTML 模板。这是在项目中重用 HTML 组件的常用方法。

在下一个我们实现的表单，我们可以简单地使用 `{% include 'includes/form.html' %}` 去渲染它。

## Adding More Tests

现在我们在使用 **Django 表单**；我们可以添加更多的测试以确保它能运行顺利

### boards/tests.py

```
... other imports
from .forms import NewTopicForm

class NewTopicTests(TestCase):
 # ... other tests

 def test_contains_form(self): # <- new test
 url = reverse('new_topic', kwargs={'pk': 1})
 response = self.client.get(url)
 form = response.context.get('form')
 self.assertIsInstance(form, NewTopicForm)

 def test_new_topic_invalid_post_data(self): # <- updated
 this one
 """
 Invalid post data should not redirect
 The expected behavior is to show the form again with
 validation errors
 """
 url = reverse('new_topic', kwargs={'pk': 1})
 response = self.client.post(url, {})
 form = response.context.get('form')
 self.assertEqual(response.status_code, 200)
 self.assertTrue(form.errors)
```

这是我们第一次使用 `assertIsInstance` 方法。基本上我们的处理是抓取上下文的表单实例，检查它是否是一个 `NewTopicForm`。在最后的测试中，我添加了 `self.assertTrue(form.errors)` 以确保数据无效的时候表单会显示错误。

## 总结

在这个课程，我们学习了 URLs, 可重用模板和表单。像往常一样，我们也实现了几个测试用例。这能使我们开发中更自信。

我们的测试文件变的越来越大，所以在下一节中，我们重构它以提高它的可维护性，从而维持我们代码的增加。

我们也达到了我们需要与登录的用户进行交互的目的。在下一节，我们学习了关于认证的一切知识和怎么去保护我们的视图和资源。

该项目的源代码在 GitHub 可用。项目的当前状态可以在发布标签 **v0.3-lw** 下找到。下面是链接：

<https://github.com/sibtc/django-beginners-guide/tree/v0.3-lw>

# Django入门与实践-第14章：用户注册



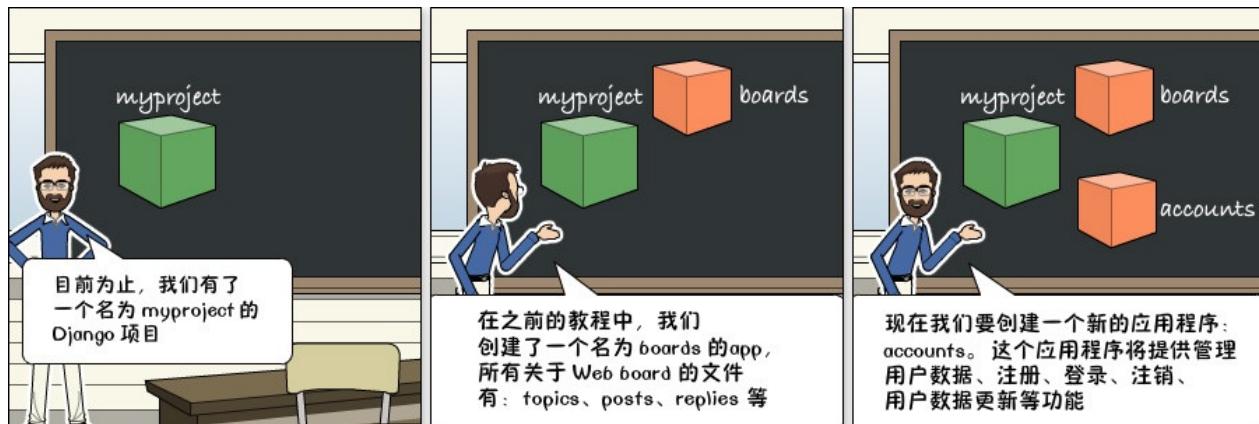
## 前言

这一章节将会全面介绍 Django 的身份认证系统，我们将实现注册、登录、注销、密码重置和密码修改的整套流程。

同时你还会了解到如何保护某些试图以防未授权的用户访问，以及如何访问已登录用户的个人信息。

在接下来的部分，你会看到一些和身份验证有关线框图，将在本教程中实现。之后是一个全新Django 应用的初始化设置。至今为止我们一直在一个名叫 boards 的应用中开发。不过，所有身份认证相关的内容都将在另一个应用中，这样能更良好的组织代码。

在接下来的部分，你会看到一些和身份验证有关线框图，将在本教程中实现。之后是一个全新Django 应用的初始化设置。至今为止我们一直在一个名叫 boards 的应用中开发。不过，所有身份认证相关的内容都将在另一个应用中，这样能更良好的组织代码。



## 线框图

我们必须更新一下应用的线框图。首先，我们需要在顶部菜单添加一些新选项，如果用户未通过身份验证，应该有两个按钮：分别是注册和登录按钮。

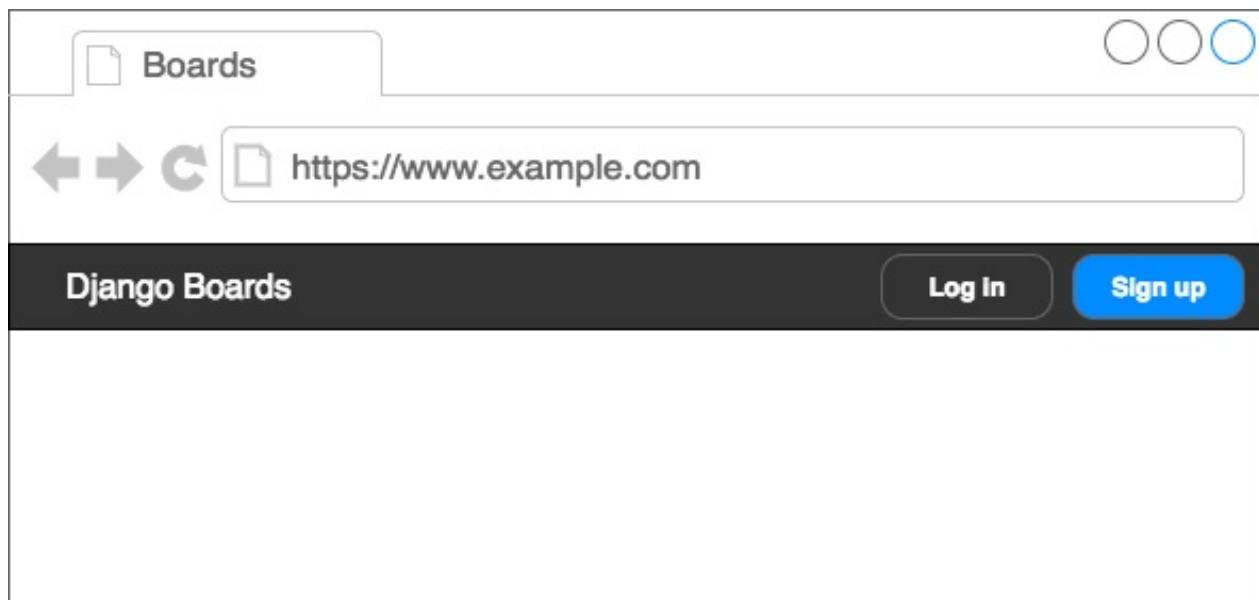


图1: 未认证用户的菜单顶部

如果用户已经通过身份验证，我们应该显示他们的名字，和带有“我的账户”，“修改密码”，“登出”这三个选项的下拉框

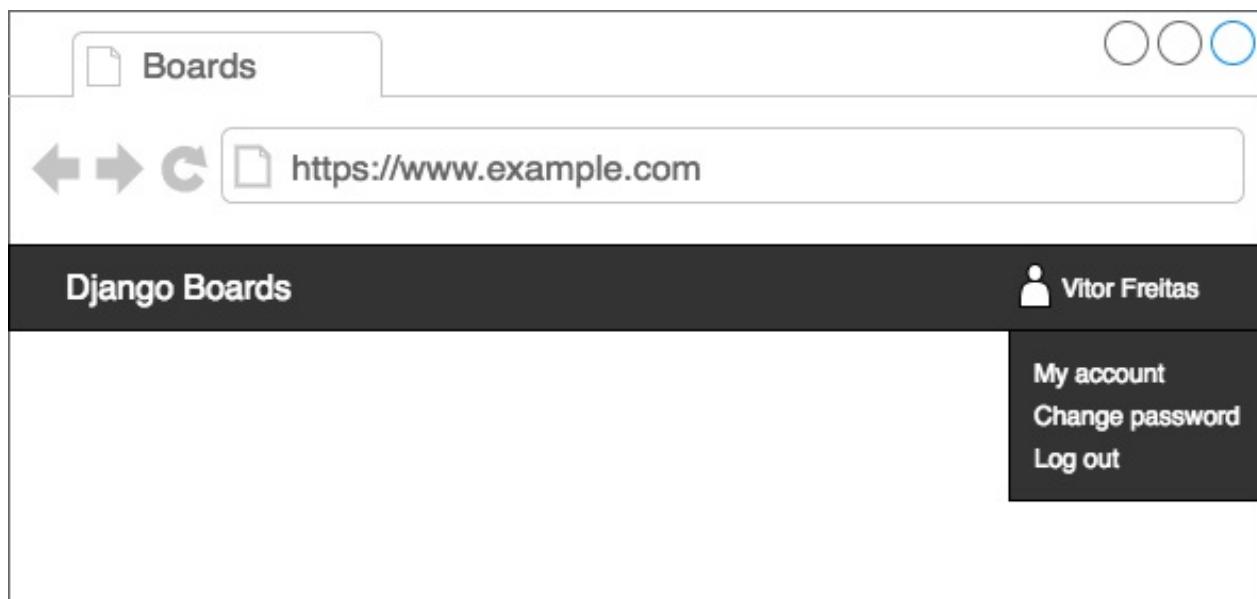


图2: 认证用户的顶部菜单

在登录页面，我们需要一个带有**username**和**password**的表单，一个登录的按钮和可跳转到注册页面和密码重置页面的链接。

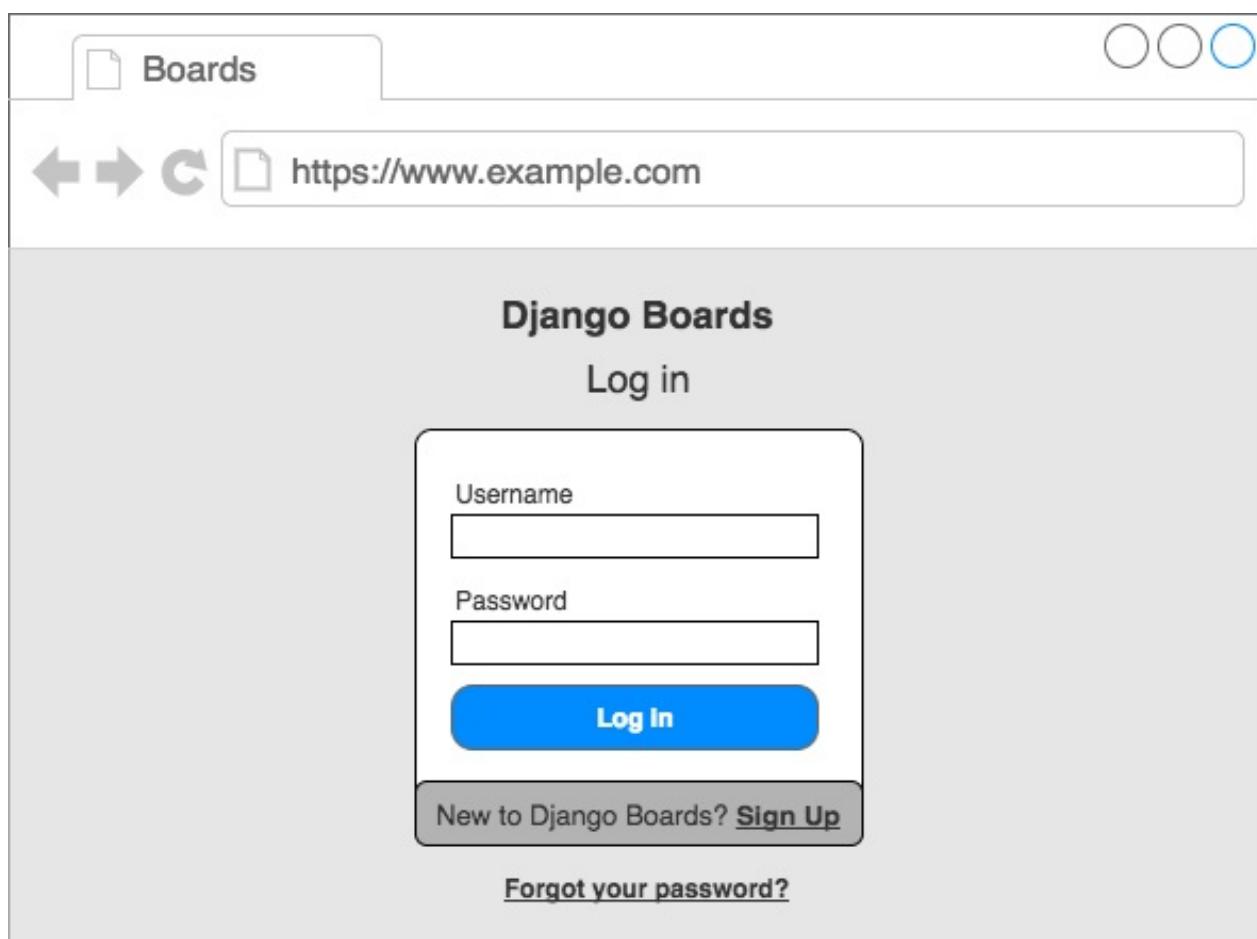


图3：登录页面

在注册页面，我们应该有包含四个字段的表单：**username**, **email address**, **password**和**password confirmation**。同时，也应该有一个能够访问登录页面链接。

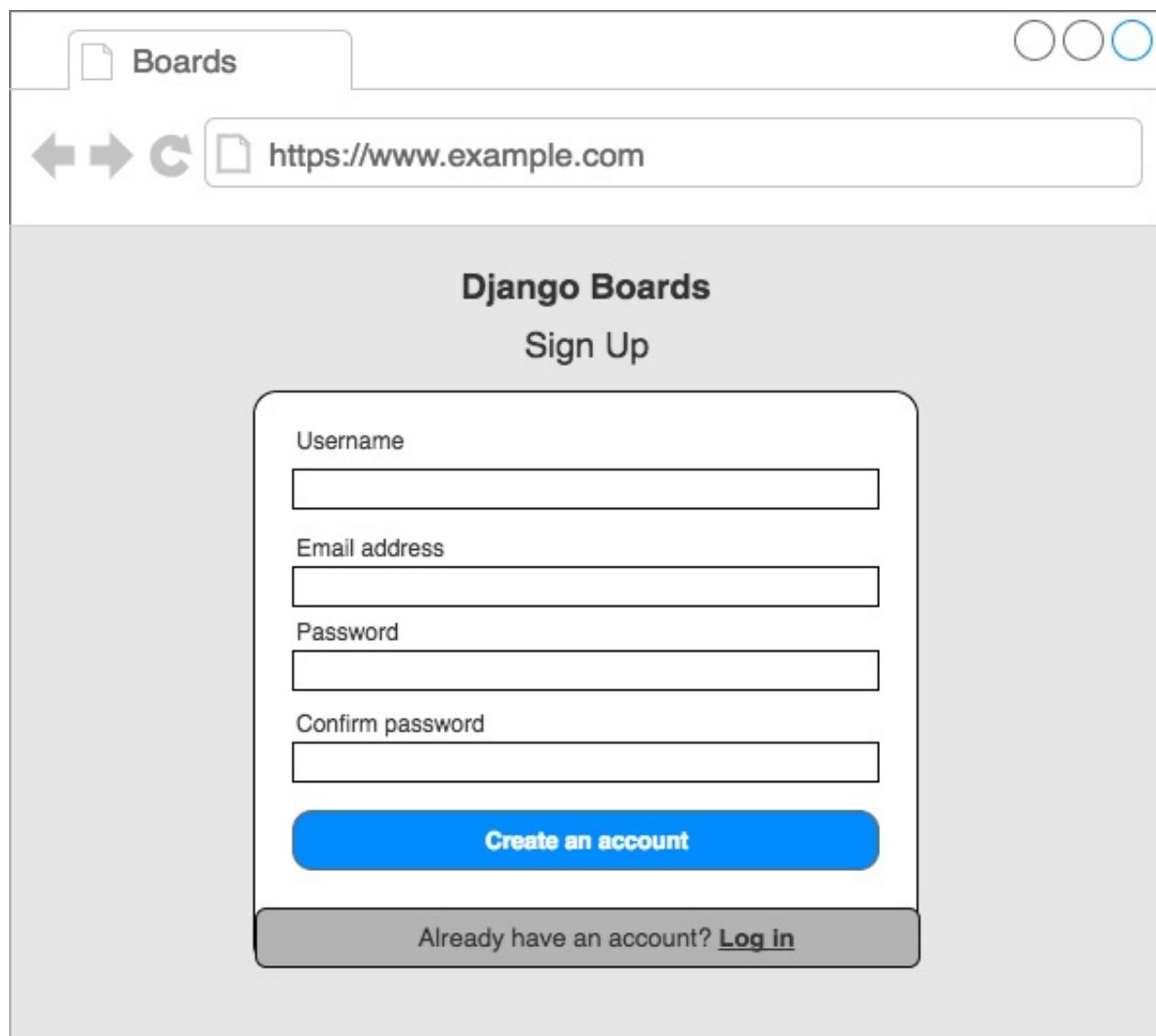


图4：注册页面

在密码重置页面上，只有**email address**字段的表单。

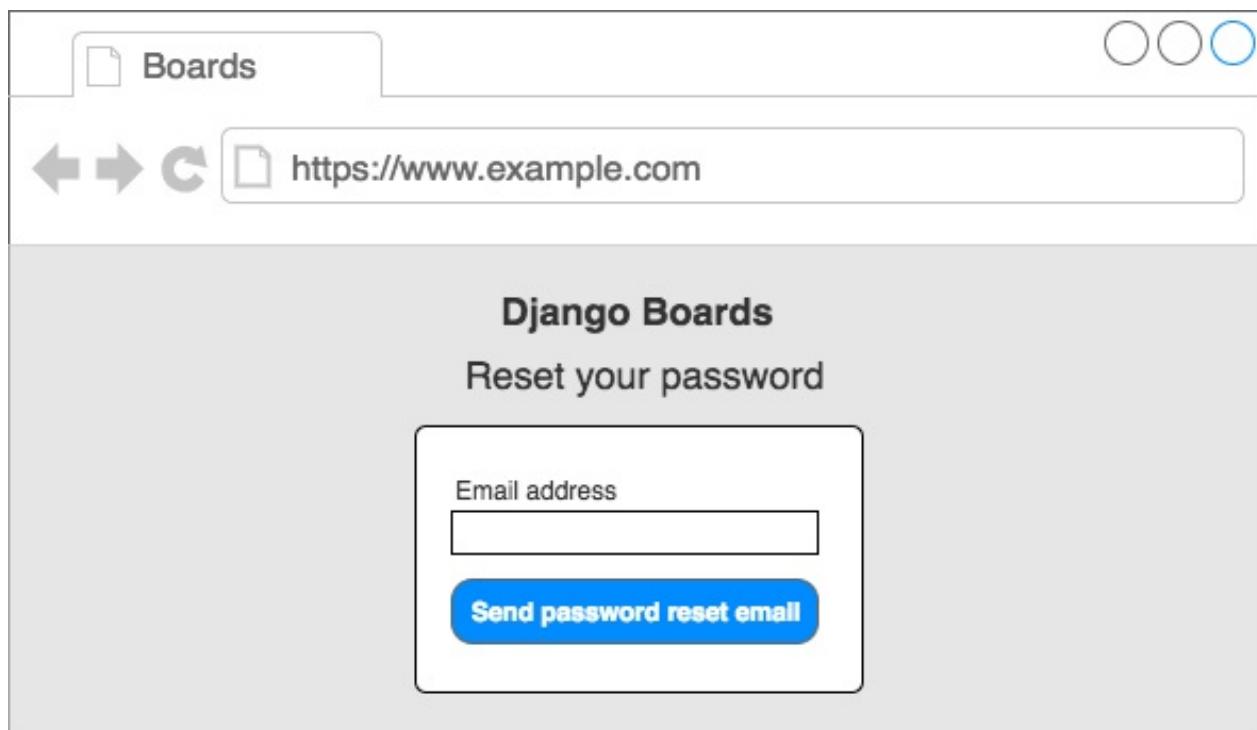


图5: 密码重置

之后，用户在点击带有特殊token的重置密码链接以后，用户将被重定向到一个页面，在那里他们可以设置新的密码。

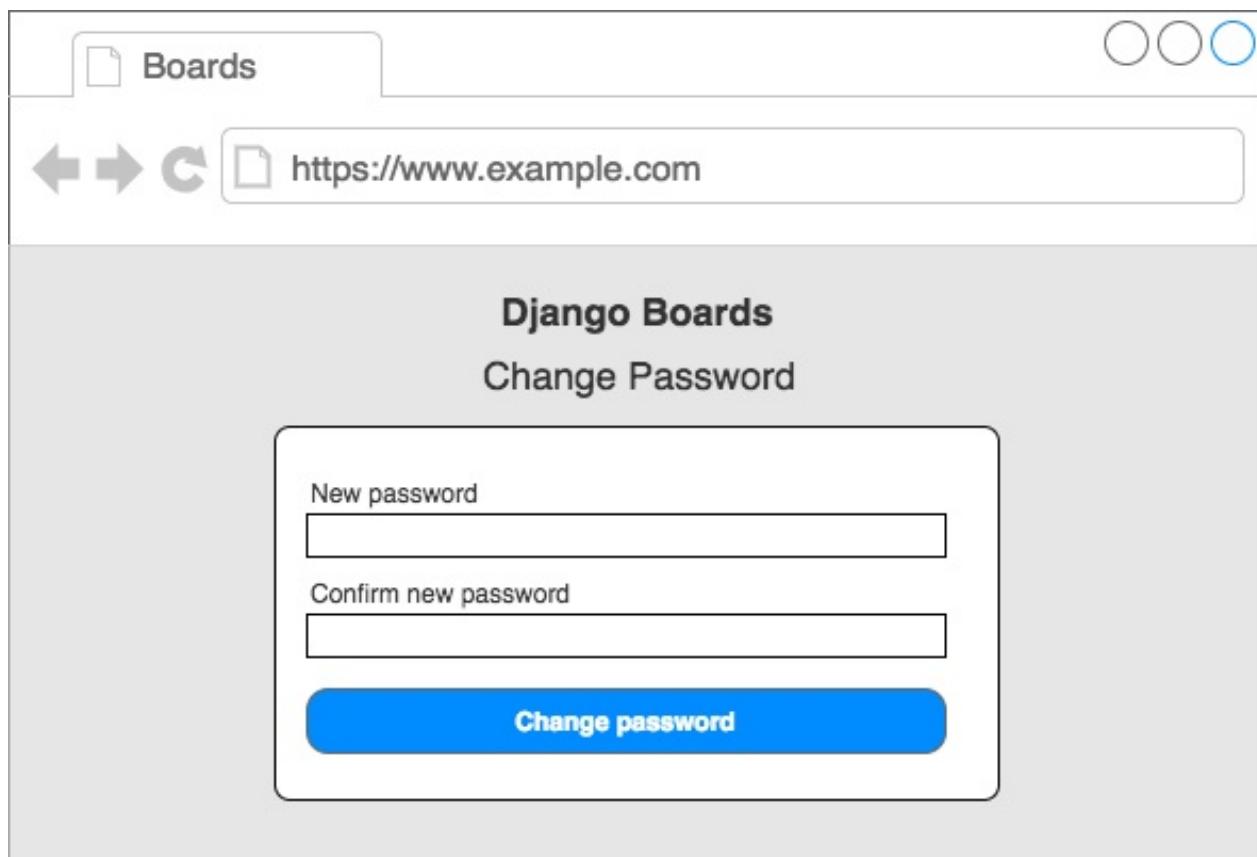


图6: 修改密码

## 初始设置

要管理这些功能，我们可以在另一个应用（app）中将其拆解。在项目根目录中的 manage.py 文件所在的同一目录下，运行以下命令以创建一个新的 app：

```
django-admin startapp accounts
```

项目的目录结构应该如下：

```
myproject/
| -- myproject/
| | -- accounts/ <-- 新创建的app
| | -- boards/
| | -- myproject/
| | -- static/
| | -- templates/
| | -- db.sqlite3
| +-- manage.py
+-- venv/
```

下一步，在 settings.py 文件中将 **accounts** app 添加到 `INSTALLED_APPS`：

```
INSTALLED_APPS = [
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',

 'widget_tweaks',

 'accounts',
 'boards',
]
```

现在开始，我们将会在 **accounts** 这个app下操作。



## 注册

我们从创建注册视图开始。首先，在 `urls.py` 文件中创建一个新的路由：

**myproject/urls.py**

```

from django.conf.urls import url
from django.contrib import admin

from accounts import views as accounts_views
from boards import views

urlpatterns = [
 url(r'^$', views.home, name='home'),
 url(r'^signup/$', accounts_views.signup, name='signup'),
 url(r'^boards/(?P<pk>\d+)/$', views.board_topics, name='board_topics'),
 url(r'^boards/(?P<pk>\d+)/new/$', views.new_topic, name='new_topic'),
 url(r'^admin/', admin.site.urls),
]

```

注意，我们以不同的方式从 `accounts` app 导入了 `views` 模块

```
from accounts import views as accounts_views
```

我们给 `accounts` 的 `views` 指定了别名，否则它会与 `boards` 的 `views` 模块发生冲突。稍后我们可以改进 `urls.py` 的设计，但现在，我们只关注身份验证功能。

现在，我们在 `accounts` app 中编辑 **`views.py`**，新创建一个名为**`signup`**的视图函数：

### accounts/views.py

```

from django.shortcuts import render

def signup(request):
 return render(request, 'signup.html')

```

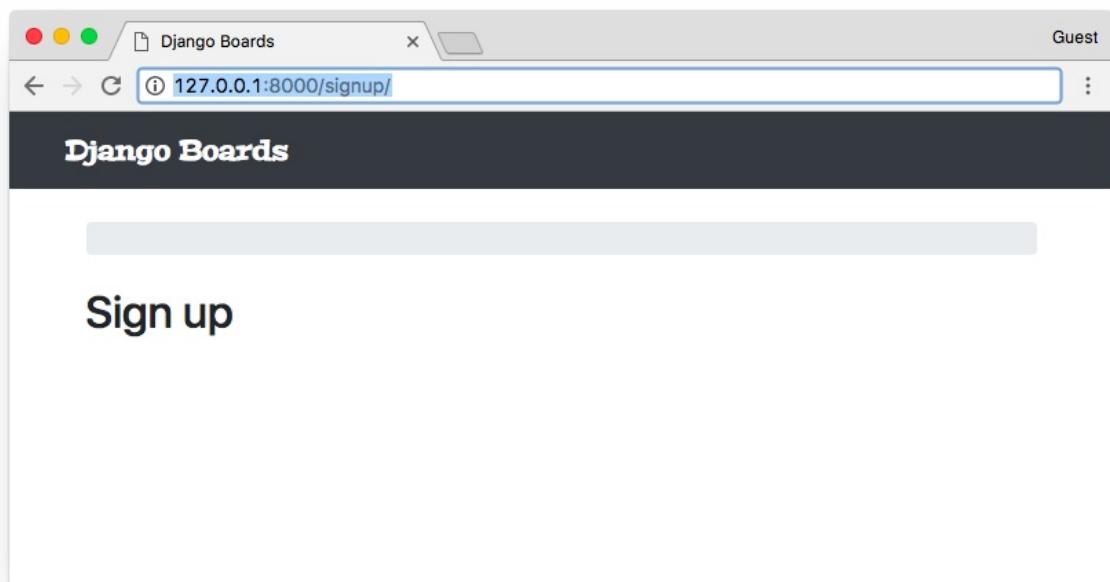
接着创建一个新的模板，取名为**`signup.html`**：

## templates/signup.html

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

{% block content %}
 <h2>Sign up</h2>
{% endblock %}
```

在浏览器中打开 <http://127.0.0.1:8000/signup/>，看看是否程序运行了起来：



接下来写点测试用例：

## accounts/tests.py

```

from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase
from .views import signup

class SignUpTests(TestCase):
 def test_signup_status_code(self):
 url = reverse('signup')
 response = self.client.get(url)
 self.assertEqual(response.status_code, 200)

 def test_signup_url_resolves_signup_view(self):
 view = resolve('/signup/')
 self.assertEqual(view.func, signup)

```

测试状态码（200=success）以及 URL /**signup/** 是否返回了正确的视图函数。

```
python manage.py test
```

```

Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.
.
.
Ran 18 tests in 0.652s

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

```

对于认证视图（注册、登录、密码重置等），我们不需要顶部条和 breadcrumb 导航栏，但仍然能够复用**base.html** 模板，不过我们需要对它做出一些修改，只需要微调：

**templates/base.html**

```
{% load static %}<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>{% block title %}Django Boards{% endblock %}</title>
 >
 <link href="https://fonts.googleapis.com/css?family=Peralta" rel="stylesheet">
 <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
 <link rel="stylesheet" href="{% static 'css/app.css' %}">
 {% block stylesheet %}{% endblock %} <!-- 这里 -->
 </head>
 <body>
 {% block body %} <!-- 这里 -->
 <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
 >
 <div class="container">
 Django Boards
 </div>
 </nav>
 <div class="container">
 <ol class="breadcrumb my-4">
 {% block breadcrumb %}
 {% endblock %}

 {% block content %}
 {% endblock %}
 </div>
 {% endblock body %} <!-- 这里 -->
 </body>
 </html>
```

我在 **base.html** 模板中标注了注释，表示新加的代码。块代码 `{% block stylesheet %}{% endblock %}` 表示添加一些额外的CSS，用于某些特定的页面。

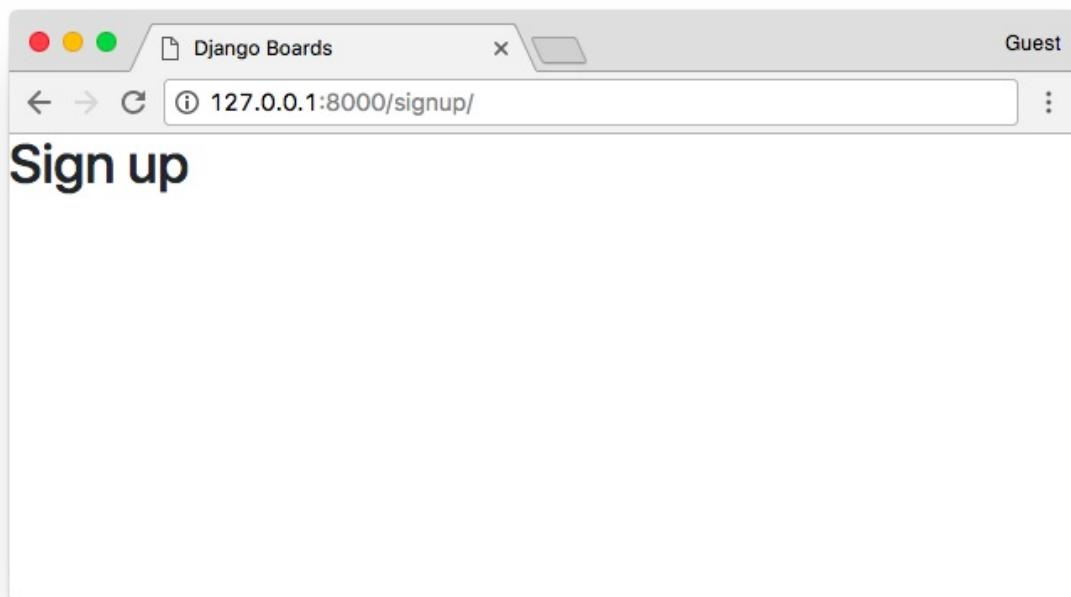
代码块 `{% block body %}` 包装了整个HTML文档。我们可以只有一个空的文档结构，以充分利用**base.html**头部。注意，还有一个结束的代码块 `{% endblock body %}`，在这种情况下，命名结束标签是一种很好的实践方法，这样更容易确定结束标记的位置。

现在，在**signup.html**模板中，我们使用 `{% block body %}` 代替了 `{% block content %}`

### templates/signup.html

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

{% block body %}
 <h2>Sign up</h2>
{% endblock %}
```





是时候创建注册表单了。Django有一个名为 **UserCreationForm** 的内置表单，我们就使用它吧：

### accounts/views.py

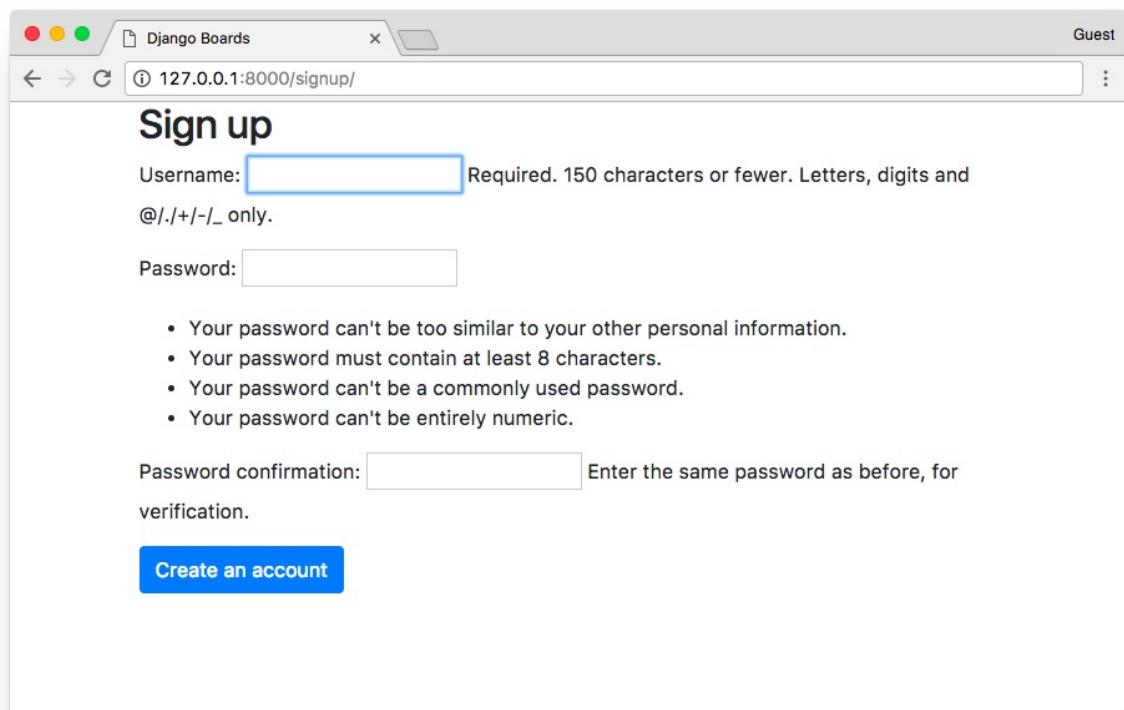
```
from django.contrib.auth.forms import UserCreationForm
from django.shortcuts import render

def signup(request):
 form = UserCreationForm()
 return render(request, 'signup.html', {'form': form})
```

### templates/signup.html

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

{% block body %}
 <div class="container">
 <h2>Sign up</h2>
 <form method="post" novalidate>
 {% csrf_token %}
 {{ form.as_p }}
 <button type="submit" class="btn btn-primary">Create an
 account</button>
 </form>
 </div>
{% endblock %}
```

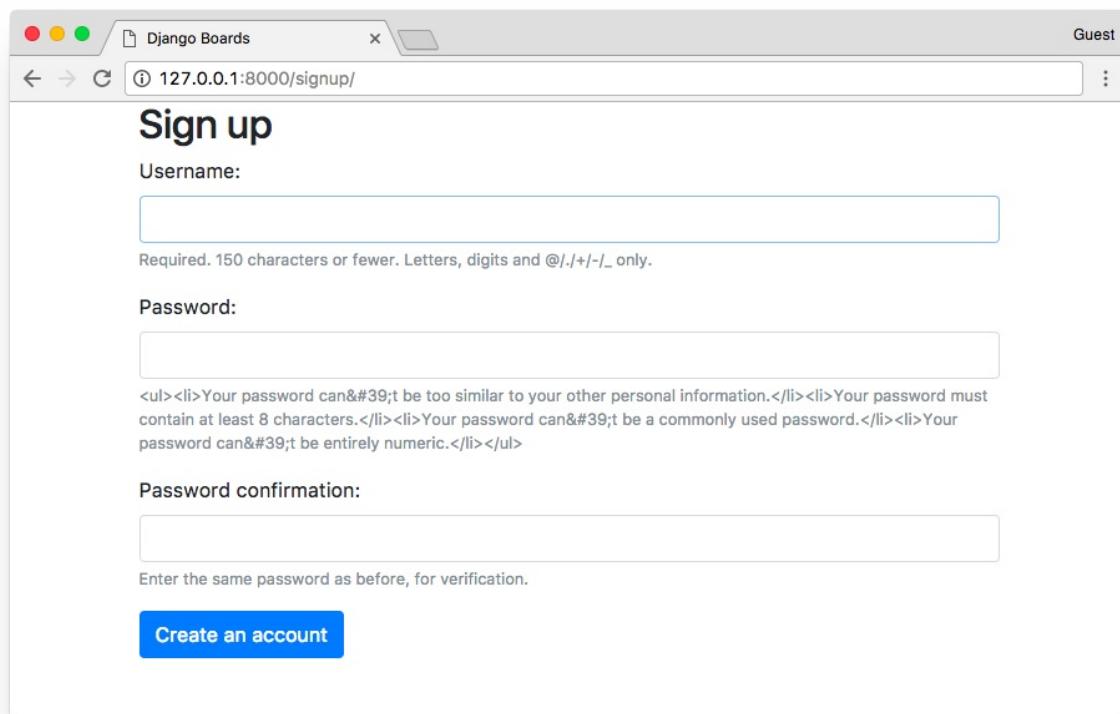


看起来有一点乱糟糟,是吧? 我们可以使用**form.html**模板使它看起来更好:

### templates/signup.html

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

{% block body %}
<div class="container">
 <h2>Sign up</h2>
 <form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-primary">Create an
account</button>
 </form>
</div>
{% endblock %}
```



哈？非常接近目标了，目前，我们的**form.html**部分模板显示了一些原生的HTML代码。这是django出于安全考虑的特性。在默认的情况下，Django将所有字符串视为不安全的，会转义所有可能导致问题的特殊字符。但在这种情况下，我们可以信任它。

## templates/includes/form.html

```
{% load widget_tweaks %}

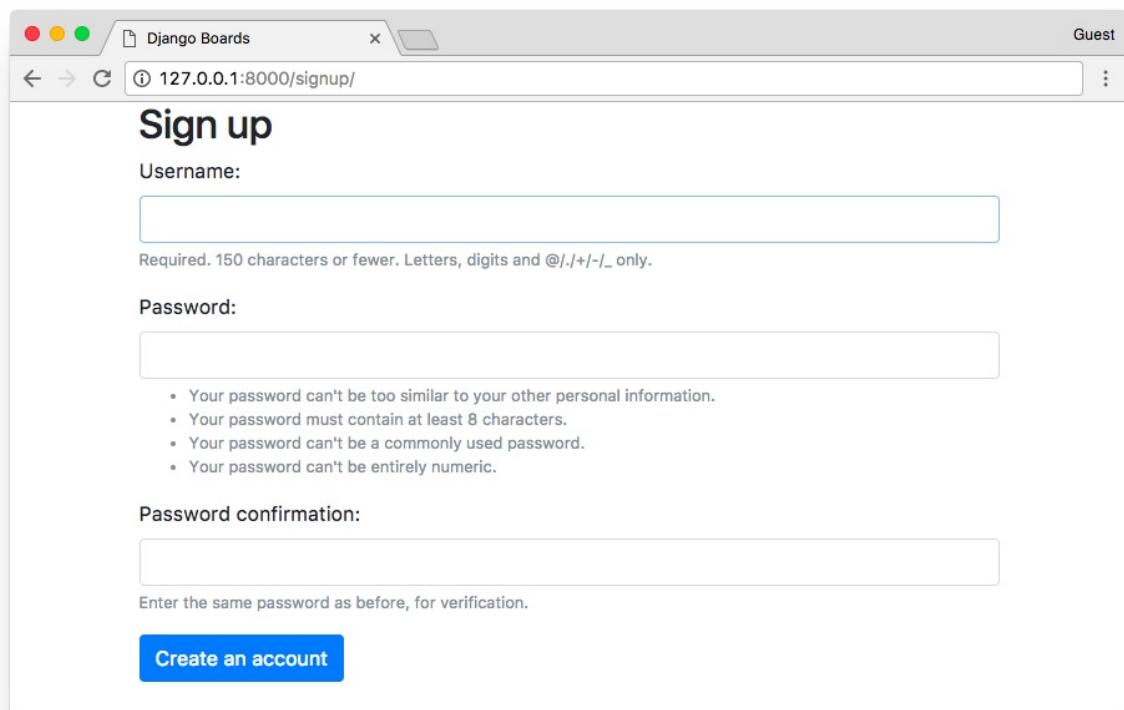
{% for field in form %}
<div class="form-group">
{{ field.label_tag }}

<!-- code suppressed for brevity -->

{% if field.help_text %}
<small class="form-text text-muted">
{{ field.help_text|safe }} <!-- 新的代码 -->
</small>
{% endif %}
</div>
{% endfor %}
```

我们主要在之前的模板中，将选项 `safe` 添加到 `field.help_text : {{ field.help_text|safe }}`。

保存**form.html**文件，然后再次检测注册页面：



现在，让我们在**signup**视图中实现业务逻辑：

### accounts/views.py

```
from django.contrib.auth import login as auth_login
from django.contrib.auth.forms import UserCreationForm
from django.shortcuts import render, redirect

def signup(request):
 if request.method == 'POST':
 form = UserCreationForm(request.POST)
 if form.is_valid():
 user = form.save()
 auth_login(request, user)
 return redirect('home')
 else:
 form = UserCreationForm()
 return render(request, 'signup.html', {'form': form})
```

表单处理有一个小细节：`login`函数重命名为`auth_login`以避免与内置`login`视图冲突）。

（编者注：我重命名了 `login` 函数重命名为 `auth_login`，但后来我意识到Django1.11对登录视图`LoginView`具有基于类的视图，因此不存在与名称冲突的风险。在比较旧的版本中，有一个 `auth.login` 和 `auth.view.login`，这会导致一些混淆，因为一个 是用户登录的功能，另一个是视图。

简单来说：如果你愿意，你可以像 `login` 一样导入它，这样做不会造成任何问题。）

如果表单是有效的，那么我们通过 `user=form.save()` 创建一个User实例。然后将创建的用户作为参数传递给 `auth_login` 函数，手动验证用户。之后，视图将用户重定向到主页，保持应用程序的流程。

让我们来试试吧，首先，提交一些无效数据，无论是空表单，不匹配的字段还是已有的用户名。

Django Boards

Guest

127.0.0.1:8000/signup/

## Sign up

Username:

admin

A user with that username already exists.

Required. 150 characters or fewer. Letters, digits and @./+/-/\_ only.

Password:

This field is required.

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

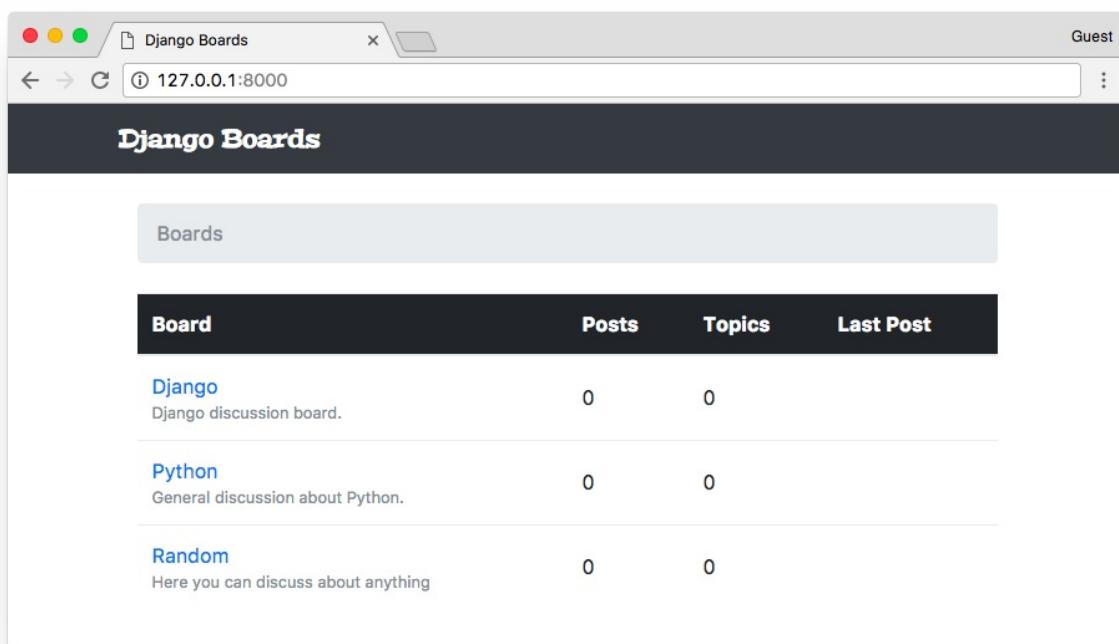
Password confirmation:

This field is required.

Enter the same password as before, for verification.

Create an account

现在填写表单并提交，检查用户是否已创建并重定向到主页。



## 在模板中引用已认证的用户

我们要怎么才能知道上述操作是否有效呢？我们可以编辑**base.html**模板来在顶部栏上添加用户名：

**templates/base.html**

```
{% block body %}

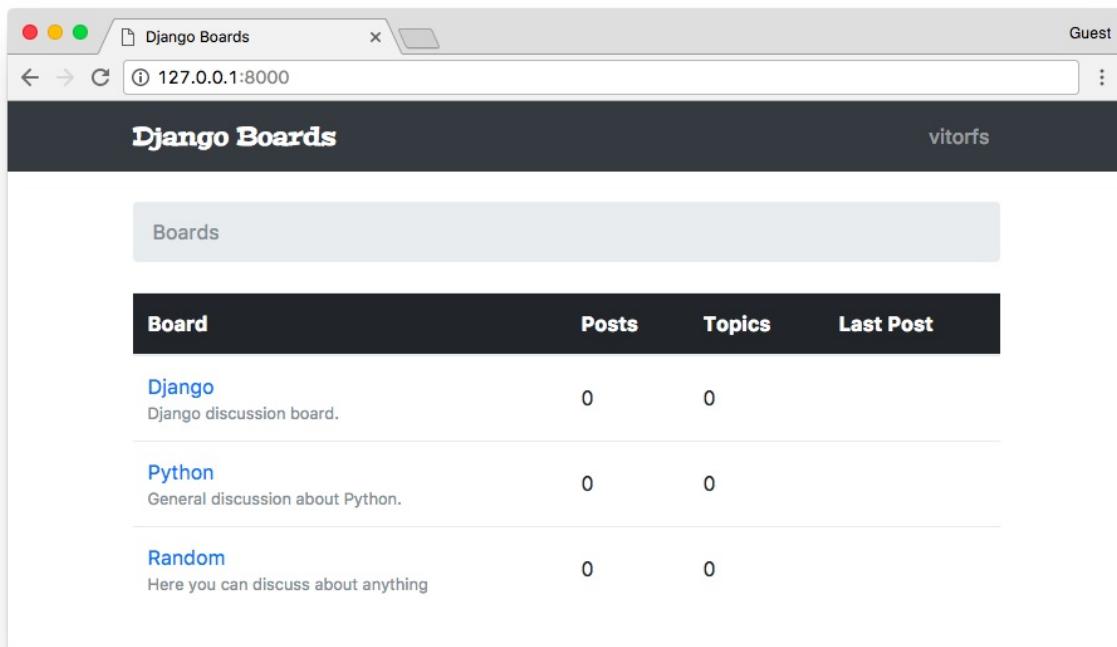
<nav class="navbar navbar-expand-sm navbar-dark bg-dark">
 <div class="container">
 Django
Boards
 <button class="navbar-toggler" type="button" data-toggle
="collapse" data-target="#mainMenu" aria-controls="mainMenu"
aria-expanded="false" aria-label="Toggle navigation">

 </button>
 <div class="collapse navbar-collapse" id="mainMenu">
 <ul class="navbar-nav ml-auto">
 <li class="nav-item">
 {{ user.username }}</
a>

 </div>
 </div>
</nav>

<div class="container">
 <ol class="breadcrumb my-4">
 {% block breadcrumb %}
 {% endblock %}

 {% block content %}
 {% endblock %}
</div>
{% endblock body %}
```



## 测试注册视图

我们来改进测试用例：

**accounts/tests.py**

```

from django.contrib.auth.forms import UserCreationForm
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase
from .views import signup

class SignUpTests(TestCase):
 def setUp(self):
 url = reverse('signup')
 self.response = self.client.get(url)

 def test_signup_status_code(self):
 self.assertEqual(self.response.status_code, 200)

 def test_signup_url_resolves_signup_view(self):
 view = resolve('/signup/')
 self.assertEqual(view.func, signup)

 def test_csrf(self):
 self.assertContains(self.response, 'csrfmiddlewaretoken')

 def test_contains_form(self):
 form = self.response.context.get('form')
 self.assertIsInstance(form, UserCreationForm)

```

我们稍微改变了**SighUpTests**类， 定义了一个**setUp**方法， 将**response**对象移到那里， 现在我们测试响应中是否有表单和CSRF token。

现在我们要测试一个成功的注册功能。这次， 让我们来创建一个新类， 以便于更好地组织测试。

## accounts/tests.py

```

from django.contrib.auth.models import User
from django.contrib.auth.forms import UserCreationForm
from django.core.urlresolvers import reverse

```

```
from django.urls import resolve
from django.test import TestCase
from .views import signup

class SignUpTests(TestCase):
 # code suppressed...

class SuccessfulSignUpTests(TestCase):
 def setUp(self):
 url = reverse('signup')
 data = {
 'username': 'john',
 'password1': 'abcdef123456',
 'password2': 'abcdef123456'
 }
 self.response = self.client.post(url, data)
 self.home_url = reverse('home')

 def test_redirection(self):
 """
 A valid form submission should redirect the user to
 the home page
 """
 self.assertRedirects(self.response, self.home_url)

 def test_user_creation(self):
 self.assertTrue(User.objects.exists())

 def test_user_authentication(self):
 """
 Create a new request to an arbitrary page.
 The resulting response should now have a `user` to its
 context,
 after a successful sign up.
 """
 response = self.client.get(self.home_url)
 user = response.context.get('user')
 self.assertTrue(user.is_authenticated)
```

运行这个测试用例。

使用类似地策略，创建一个新的类，用于数据无效的注册用例

```
from django.contrib.auth.models import User
from django.contrib.auth.forms import UserCreationForm
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase
from .views import signup

class SignUpTests(TestCase):
 # code suppressed...

class SuccessfulSignUpTests(TestCase):
 # code suppressed...

class InvalidSignUpTests(TestCase):
 def setUp(self):
 url = reverse('signup')
 self.response = self.client.post(url, {}) # submit an empty dictionary

 def test_signup_status_code(self):
 ...
 An invalid form submission should return to the same page
 ...
 self.assertEquals(self.response.status_code, 200)

 def test_form_errors(self):
 form = self.response.context.get('form')
 self.assertTrue(form.errors)

 def test_dont_create_user(self):
 self.assertFalse(User.objects.exists())
```

## 将Email字段添加到表单

一切都正常，但还缺失 **email address** 字段。**UserCreationForm** 不提供 email 字段，但是我们可以对它进行扩展。

在**accounts** 文件夹中创建一个名为**forms.py** 的文件：

### accounts/forms.py

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

class SignUpForm(UserCreationForm):
 email = forms.CharField(max_length=254, required=True, widget=forms.EmailInput())
 class Meta:
 model = User
 fields = ('username', 'email', 'password1', 'password2')
```

现在，我们不需要在 `views.py` 中使用 `UserCreationForm`，而是导入新的表单 **SignUpForm**，然后使用它：

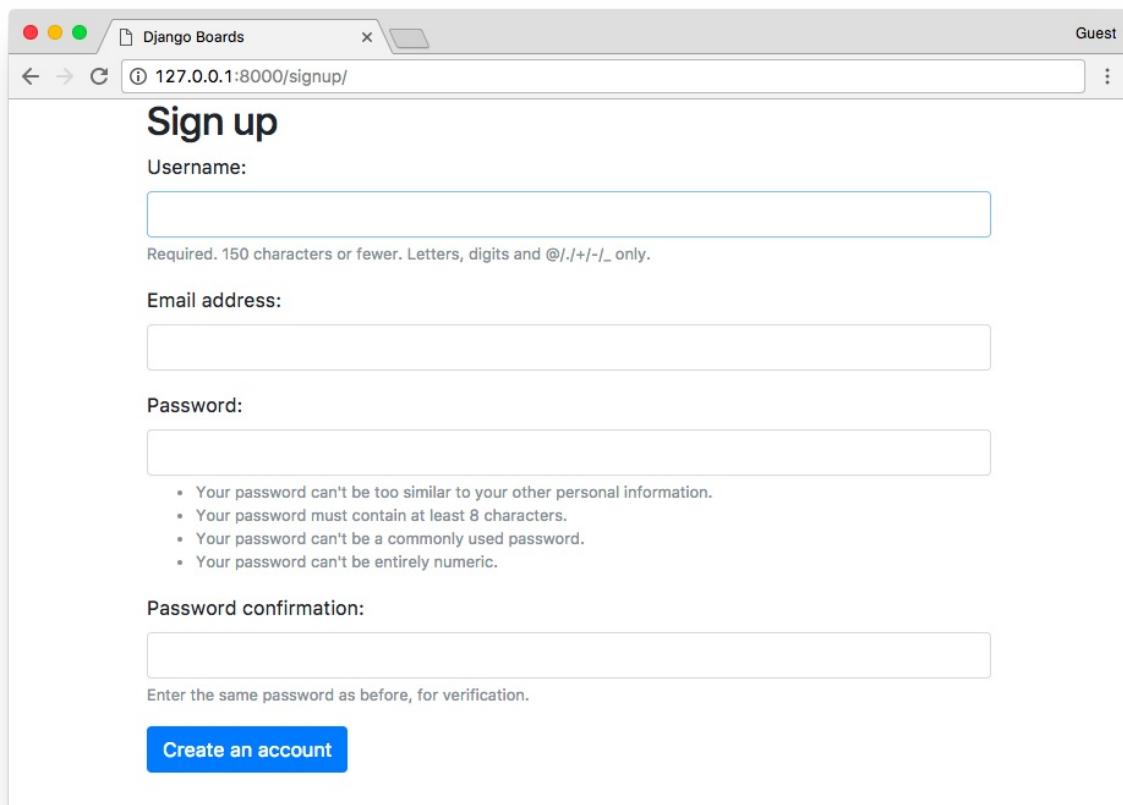
### accounts/views.py

```
from django.contrib.auth import login as auth_login
from django.shortcuts import render, redirect

from .forms import SignUpForm

def signup(request):
 if request.method == 'POST':
 form = SignUpForm(request.POST)
 if form.is_valid():
 user = form.save()
 auth_login(request, user)
 return redirect('home')
 else:
 form = SignUpForm()
 return render(request, 'signup.html', {'form': form})
```

只用这个小小的改变，可以运作了：



请记住更改测试用例以使用SignUpForm而不是UserCreationForm：

```
from .forms import SignUpForm

class SignUpTests(TestCase):
 # ...

 def test_contains_form(self):
 form = self.response.context.get('form')
 self.assertIsInstance(form, SignUpForm)

class SuccessfulSignUpTests(TestCase):
 def setUp(self):
 url = reverse('signup')
 data = {
 'username': 'john',
 'email': 'john@doe.com',
 'password1': 'abcdef123456',
 'password2': 'abcdef123456'
 }
 self.response = self.client.post(url, data)
 self.home_url = reverse('home')

 # ...
```

之前的测试用例仍然会通过，因为SignUpForm扩展了UserCreationForm，它是UserCreationForm的一个实例。

添加了新的表单后，让我们想想发生了什么：

```
fields = ('username', 'email', 'password1', 'password2')
```

它会自动映射到HTML模板中。这很好吗？这要视情况而定。如果将来会有新的开发人员想要重新使用SignUpForm来做其他事情，并为其添加一些额外的字段。那么这些新的字段也会出现在signup.html中，这可能不是所期望的行为。这种改变可能会被忽略，我们不希望有任何意外。

那么让我们来创建一个新的测试，验证模板中的HTML输入：

### accounts/tests.py

```
class SignUpTests(TestCase):
 # ...

 def test_form_inputs(self):
 """
 The view must contain five inputs: csrf, username, email,
 password1, password2
 """

 self.assertContains(self.response, '<input', 5)
 self.assertContains(self.response, 'type="text"', 1)
 self.assertContains(self.response, 'type="email"', 1)
 self.assertContains(self.response, 'type="password"', 2)
```

## 改进测试代码的组织结构

好的，现在我们正在测试输入和所有的功能，但是我们仍然必须测试表单本身。不要只是继续向 `accounts/tests.py` 文件添加测试，我们稍微改进一下项目设计。

在**accounts**文件夹下创建一个名为**tests**的新文件夹。然后在**tests**文件夹中，创建一个名为 `init.py` 的空文件。

现在，将 `test.py` 文件移动到**tests**文件夹中，并将其重命名为 `test_view_signup.py`

最终的结果应该如下：

```
myproject/
| -- myproject/
| | -- accounts/
| | | -- migrations/
| | | -- tests/
| | | | -- __init__.py
| | | +-- test_view_signup.py
| | | -- __init__.py
| | | -- admin.py
| | | -- apps.py
| | | -- models.py
| | +-- views.py
| | -- boards/
| | -- myproject/
| | -- static/
| | -- templates/
| | -- db.sqlite3
| +-- manage.py
+-- venv/
```

注意到，因为我们在应用程序的上下文使用了相对导入，所以我们需要在 **test\_view\_signup.py** 中修复导入：

### accounts/tests/test\_view\_signup.py

```
from django.contrib.auth.models import User
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase

from ..views import signup
from ..forms import SignUpForm
```

我们在应用程序模块内部使用相对导入，以便我们可以自由地重新命名 Django 应用程序，而无需修复所有绝对导入。

现在让我们创建一个新的测试文件来测试SignUpForm，添加一个名为**test\_form\_signup.py**的新测试文件：

### accounts/tests/test\_form\_signup.py

```
from django.test import TestCase
from ..forms import SignUpForm

class SignUpFormTest(TestCase):
 def test_form_has_fields(self):
 form = SignUpForm()
 expected = ['username', 'email', 'password1', 'password2']
 actual = list(form.fields)
 self.assertSequenceEqual(expected, actual)
```

它看起来非常严格对吧，例如，如果将来我们必须更改SignUpForm，以包含用户的名字和姓氏，那么即使我们没有破坏任何东西，我们也可能最终不得不修复一些测试用例。



这些警报很有用，因为它们有助于提高认识，特别是新手第一次接触代码，它可以帮助他们自信地编码。

## 改进注册模板

让我们稍微讨论一下，在这里，我们可以使用Bootstrap4 组件来使它看起来不错。

访问：<https://www.toptal.com/designers/subtlepatterns/> 并找到一个很好地背景图案作为账户页面的背景，下载下来再静态文件夹中创建一个名为img的新文件夹，并将图像放置再那里。

之后，再static/css中创建一个名为accounts.css的新CSS文件。结果应该如下：

```
myproject/
| -- myproject/
| | -- accounts/
| | -- boards/
| | -- myproject/
| | -- static/
| | | -- css/
| | | | -- accounts.css <-- here
| | | | -- app.css
| | | +-- bootstrap.min.css
| | +-- img/
| | | +-- shattered.png <-- here (the name may be
different, depending on the pattern you downloaded)
| | -- templates/
| | -- db.sqlite3
| +-- manage.py
+-- venv/
```

现在编辑accounts.css这个文件：

**static/css/accounts.css**

```
body {
 background-image: url(..../img/shattered.png);
}

.logo {
 font-family: 'Peralta', cursive;
}

.logo a {
 color: rgba(0,0,0,.9);
}

.logo a:hover,
.logo a:active {
 text-decoration: none;
}
```

在signup.html模板中，我们可以将其改为使用新的CSS，并使用Bootstrap4组件：

**templates/signup.html**

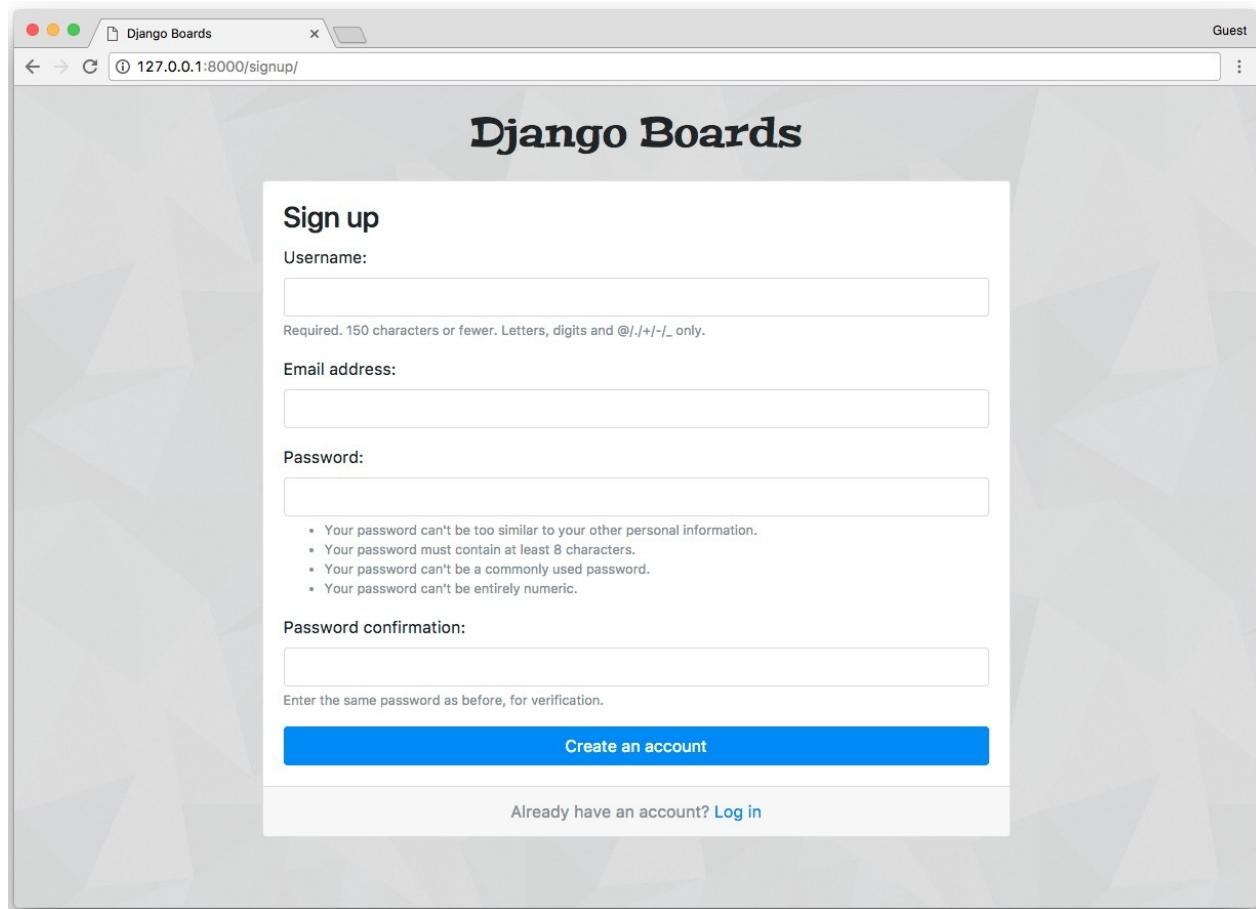
```
{% extends 'base.html' %}

{% load static %}

{% block stylesheet %}
 <link rel="stylesheet" href="{% static 'css/accounts.css' %}">
{% endblock %}

{% block body %}
 <div class="container">
 <h1 class="text-center logo my-4">
 Django Boards
 </h1>
 <div class="row justify-content-center">
 <div class="col-lg-8 col-md-10 col-sm-12">
 <div class="card">
 <div class="card-body">
 <h3 class="card-title">Sign up</h3>
 <form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-primary button-block">Create an account</button>
 </form>
 </div>
 <div class="card-footer text-muted text-center">
 Already have an account? Log in
 </div>
 </div>
 </div>
 </div>
 {% endblock %}
```

这就是我们现在的注册页面：



# Django入门与实践-第15章：用户注销

为了在实现过程保持完整自然流畅的功能，我们还添加注销视图，编辑 `urls.py` 以添加新的路由：

## myproject/urls.py

```
from django.conf.urls import url
from django.contrib import admin
from django.contrib.auth import views as auth_views

from accounts import views as accounts_views
from boards import views

urlpatterns = [
 url(r'^$', views.home, name='home'),
 url(r'^signup/$', accounts_views.signup, name='signup'),
 url(r'^logout/$', auth_views.LogoutView.as_view(), name='logout'),
 url(r'^boards/(?P<pk>\d+)/$', views.board_topics, name='board_topics'),
 url(r'^boards/(?P<pk>\d+)/new/$', views.new_topic, name='new_topic'),
 url(r'^admin/', admin.site.urls),
]
```

我们从Django的contrib模块导入了**views**，我们将其更名为**auth\_views**以避免与**boards.views**发生冲突。注意这个视图有点不同：

`LogoutView.as_view()`。这是一个Django的“基于类”的视图，到目前为止，我们只将类实现为python函数。基于类的视图提供了一种更加灵活的方式来扩展和重用视图。稍后我们将讨论更多这个主题。

打开**settings.py**文件，然后添加 `LOGOUT_REDIRECT_URL` 变量到文件的底部：

## myproject/settings.py

```
LOGOUT_REDIRECT_URL = 'home'
```

在这里我们给变量指定了一个URL模型的名称，以告诉Django当用户退出登录之后跳转的地址。

在这之后，这次重定向就算完成了。只需要访问URL **127.0.0.1:8000/logout/** 然后您就将被注销。但是再等一下，在你注销之前，让我们为登录用户创建下拉菜单。

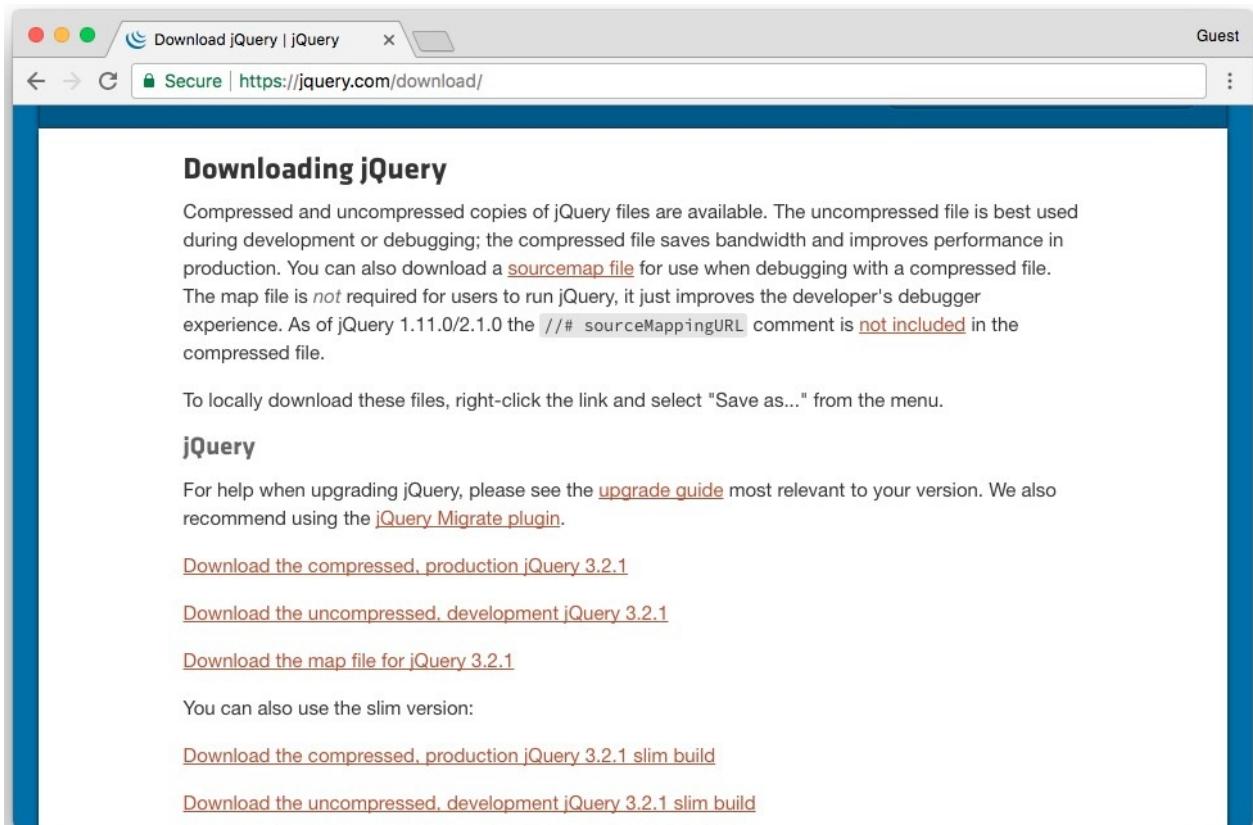
---

## 为登录用户显示菜单

现在我们需要在 **base.html** 模板中进行一些调整。我们必须添加一个带注销链接的下拉菜单。

Bootstrap 4 下拉组件需要jQuery才能工作。

首先，我们前往 [jquery.com/download/](http://jquery.com/download/)，然后下载压缩的 jQuery 3.2.1 版本。



在静态文件夹中，创建一个名为js的新文件夹。将jquery-3.2.1.min.js文件复制到那里。

Bootstrap4还需要一个名为**Popper**的库才能工作，前往[popper.js.org](https://popper.js.org)下载它的最新版本。

在**popper.js-1.12.5**文件夹中，转到**dist/umd**并将文件**popper.min.js**复制到我们的**js**文件夹。这里注意，敲黑板！Bootstrap 4只能与**umd/popper.min.js**协同工作。所以请确保你正在复制正确的文件。

如果您不再拥有 Bootstrap 4文件，请从[getbootstrap.com](https://getbootstrap.com)再次下载它。

同样，将**bootstrap.min.js**文件复制到我们的js文件夹中。最终的结果应该是：

```
myproject/
| -- myproject/
| | -- accounts/
| | -- boards/
| | -- myproject/
| | -- static/
| | | -- css/
| | +- js/
| | | -- bootstrap.min.js
| | | -- jquery-3.2.1.min.js
| | +- popper.min.js
| | -- templates/
| | -- db.sqlite3
| +- manage.py
+- venv/
```

在**base.html**文件底部，在`{% endblock body %}`后面添加脚本：

**templates/base.html**

```
{% load static %}<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8">
 <title>{% block title %}Django Boards{% endblock %}</title>
 >
 <link href="https://fonts.googleapis.com/css?family=Peralta" rel="stylesheet">
 <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
 <link rel="stylesheet" href="{% static 'css/app.css' %}">
 {% block stylesheet %}{% endblock %}
 </head>
 <body>
 {% block body %}
 <!-- code suppressed for brevity -->
 {% endblock body %}
 <script src="{% static 'js/jquery-3.2.1.min.js' %}"></script>
 <script src="{% static 'js/popper.min.js' %}"></script>
 <script src="{% static 'js/bootstrap.min.js' %}"></script>

 </body>
</html>
```

如果你发现上面的说明很模糊，只需要直接在下面的链接下载文件

- <https://code.jquery.com/jquery-3.2.1.min.js>
- <https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/umd/popper.min.js>
- <https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/js/bootstrap.min.js>

打开链接，右键另存为

现在我们可以添加Bootstrap4下拉菜单了：

**templates/base.html**

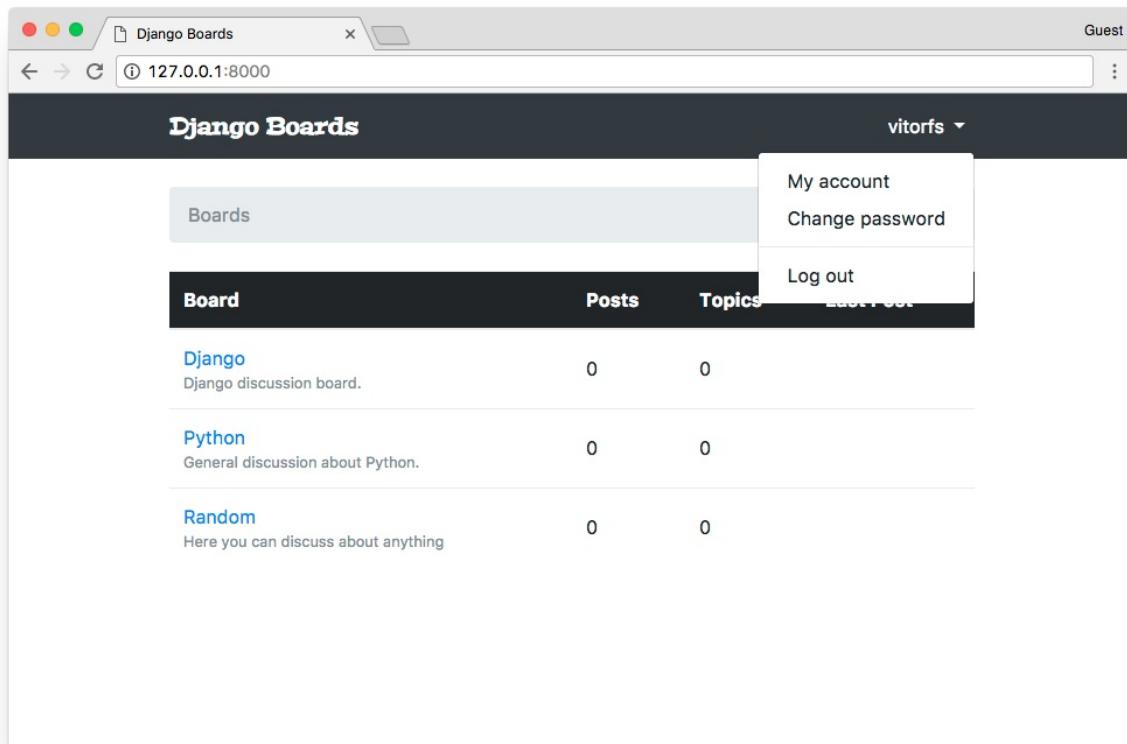
```
<nav class="navbar navbar-expand-sm navbar-dark bg-dark">
 <div class="container">
 Django Boards
 <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#mainMenu" aria-controls="mainMenu" aria-expanded="false" aria-label="Toggle navigation">

 </button>
 <div class="collapse navbar-collapse" id="mainMenu">
 <ul class="navbar-nav ml-auto">
 <li class="nav-item dropdown">

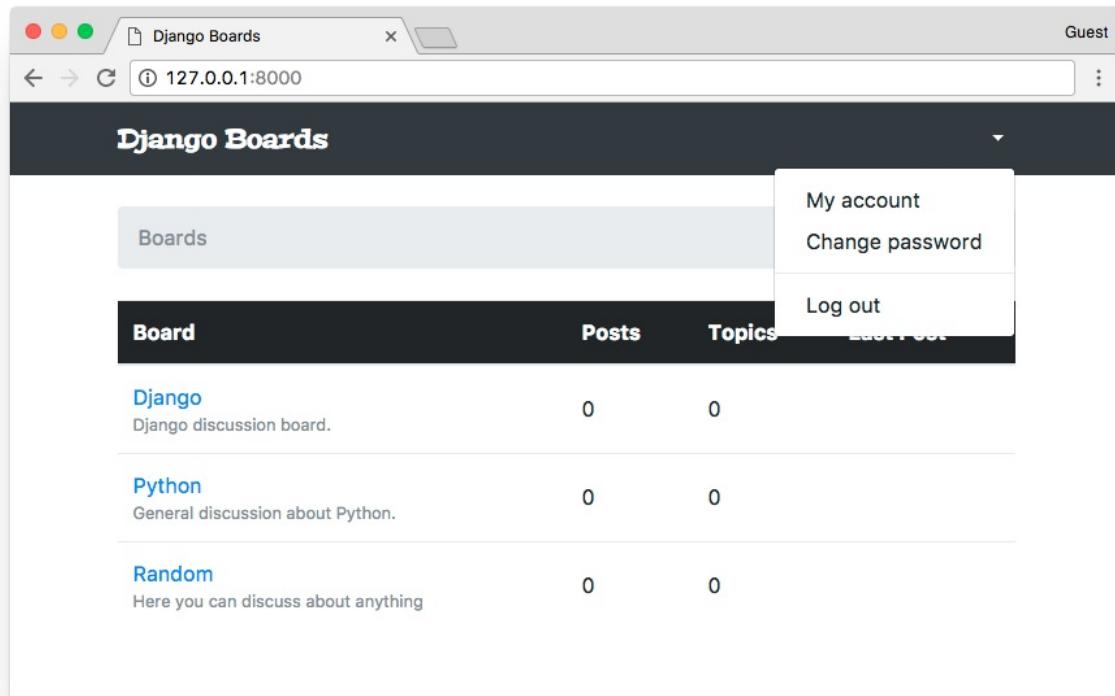
 {{ user.username }}

 <div class="dropdown-menu dropdown-menu-right" aria-labelledby="userMenu">
 My account
 Change password
 </div>
 <div class="dropdown-divider"></div>
 Log out

 </div>
 </div>
</nav>
```



我们来试试吧，点击注销：



现在已经成功显示出来了，但是无论用户登录与否，下拉菜单都会显示。不同的是在未登录时用户名显示是空的，我们只能看到一个箭头。

我们可以改进一点：

```
<nav class="navbar navbar-expand-sm navbar-dark bg-dark">
 <div class="container">
 Django Bo
ards
 <button class="navbar-toggler" type="button" data-toggle=
"collapse" data-target="#mainMenu" aria-controls="mainMenu" a
ria-expanded="false" aria-label="Toggle navigation">

 </button>
 <div class="collapse navbar-collapse" id="mainMenu">
 {% if user.is_authenticated %}
 <ul class="navbar-nav ml-auto">
 <li class="nav-item dropdown">
 <a class="nav-link dropdown-toggle" href="#" id="
userMenu" data-toggle="dropdown" aria-haspopup="true" aria-ex
panded="false">
 {{ user.username }}

 <div class="dropdown-menu dropdown-menu-right" ar
ia-labelledby="userMenu">
 My account

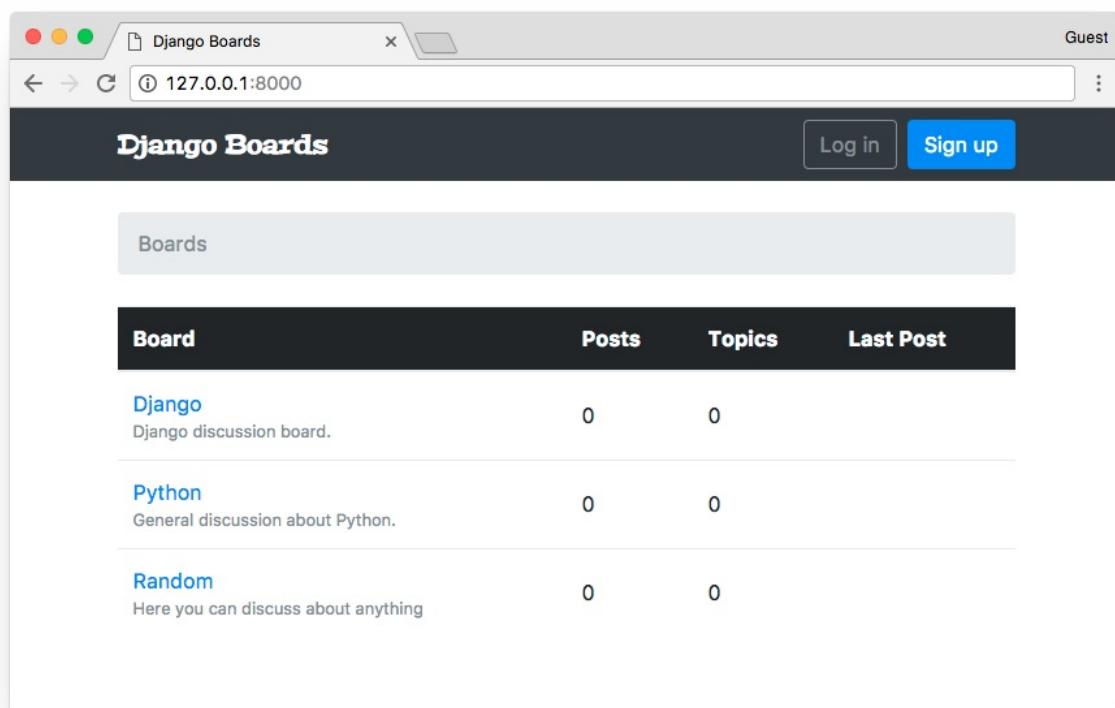
 Change passwo
rd
 <div class="dropdown-divider"></div>
 <a class="dropdown-item" href="{% url 'logou
t' %}">Log out
 </div>

 {% else %}
 <form class="form-inline ml-auto">
 Log in
 </form>
 {% endif %}
 </div>
 </div>
</nav>
```

```

 <a href="{% url 'signup' %}" class="btn btn-primary
ml-2">Sign up
 </form>
 {% endif %}
</div>
</div>
</nav>
```

现在，我们告诉Django程序，要在用户登录时显示下拉菜单，如果没有，则显示登录并注册按钮：



# Django入门与实践-第16章：用户登录

首先，添加一个新的URL路径：

## myproject/urls.py

```
from django.conf.urls import url
from django.contrib import admin
from django.contrib.auth import views as auth_views

from accounts import views as accounts_views
from boards import views

urlpatterns = [
 url(r'^$', views.home, name='home'),
 url(r'^signup/$', accounts_views.signup, name='signup'),
 url(r'^login/$', auth_views.LoginView.as_view(template_name='login.html'), name='login'),
 url(r'^logout/$', auth_views.LogoutView.as_view(), name='logout'),
 url(r'^boards/(?P<pk>\d+)/$', views.board_topics, name='board_topics'),
 url(r'^boards/(?P<pk>\d+)/new/$', views.new_topic, name='new_topic'),
 url(r'^admin/', admin.site.urls),
]
```

在 `as_view()` 中，我们可以传递一些额外的参数，以覆盖默认值。在这种情况下，我们让`LoginView` 使用`login.html`模板。

编辑**settings.py**然后添加

## myproject/settings.py

```
LOGIN_REDIRECT_URL = 'home'
```

这个配置信息告诉Django在成功登录后将用户重定向到哪里。

最后，将登录URL添加到 **base.html** 模板中：

### templates/base.html

```
Log in
```

我们可以创建一个类似于注册页面的模板。创建一个名为 **login.html** 的新文件：

### templates/login.html

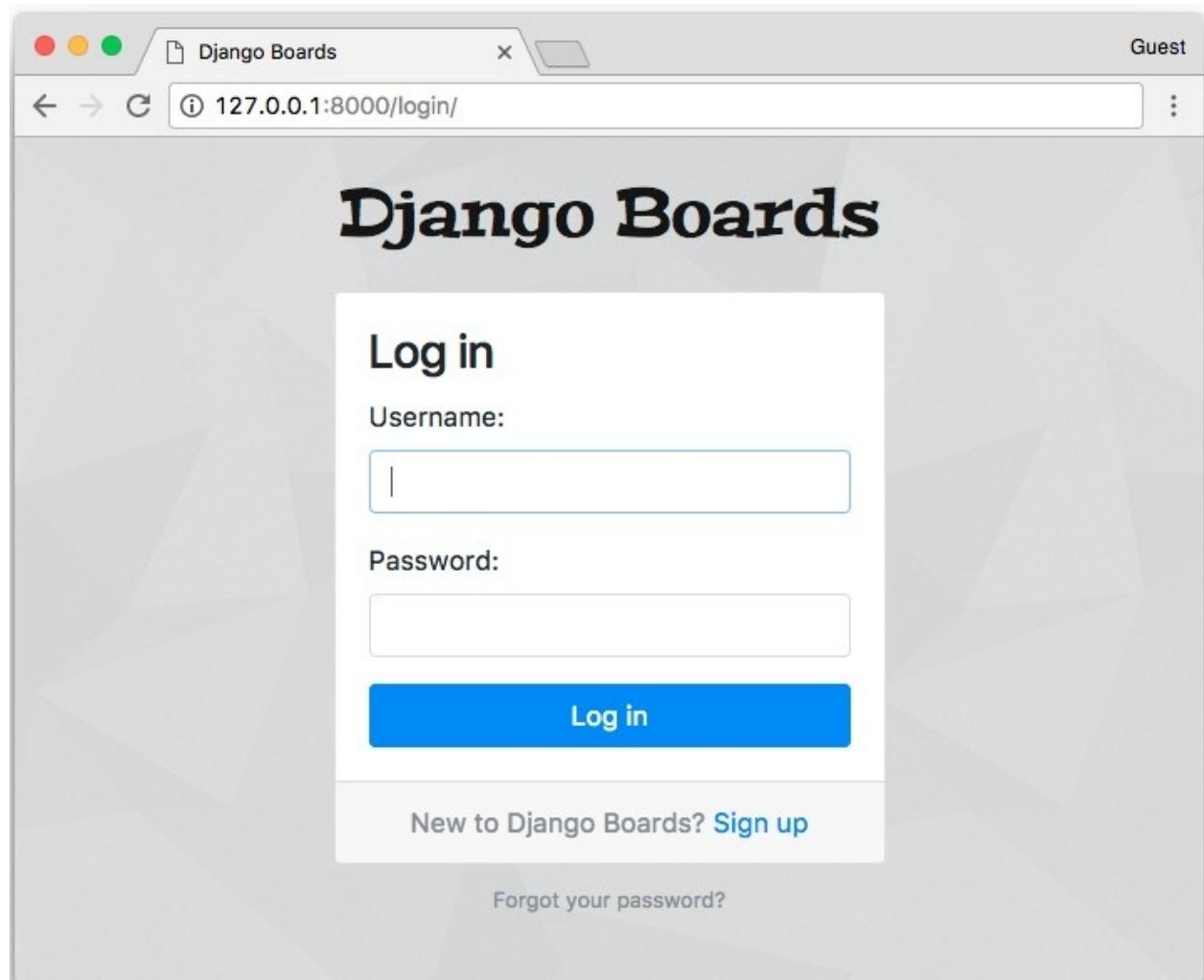
```
{% extends 'base.html' %}

{% load static %}

{% block stylesheet %}
 <link rel="stylesheet" href="{% static 'css/accounts.css' %}">
{% endblock %}

{% block body %}
 <div class="container">
 <h1 class="text-center logo my-4">
 Django Boards
 </h1>
 <div class="row justify-content-center">
 <div class="col-lg-4 col-md-6 col-sm-8">
 <div class="card">
 <div class="card-body">
 <h3 class="card-title">Log in</h3>
 <form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-primary bt
```

```
n-block">Log in</button>
 </form>
 </div>
 <div class="card-footer text-muted text-center">
 New to Django Boards? Sign up
 </div>
 <div class="text-center py-2">
 <small>
 Forgot your password?
 </small>
 </div>
 </div>
</div>
{% endblock %}
```



我们看到HTML模板中的内容重复了，现在来重构一下它。

创建一个名为**base\_accounts.html**的新模板：

**templates/base\_accounts.html**

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

{% load static %}

{% block stylesheet %}
 <link rel="stylesheet" href="{% static 'css/accounts.css' %}">
{% endblock %}

{% block body %}
 <div class="container">
 <h1 class="text-center logo my-4">
 Django Boards
 </h1>
 {% block content %}
 {% endblock %}
 </div>
{% endblock %}
```

现在在**signup.html**和**login.html**中使用它：

**templates/login.html**

```
{% extends 'base_accounts.html' %}

{% block title %}Log in to Django Boards{% endblock %}

{% block content %}
<div class="row justify-content-center">
<div class="col-lg-4 col-md-6 col-sm-8">
<div class="card">
<div class="card-body">
<h3 class="card-title">Log in</h3>
<form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-primary btn-block">Log in</button>
</form>
</div>
<div class="card-footer text-muted text-center">
 New to Django Boards? Sign up
</div>
</div>
<div class="text-center py-2">
<small>
 Forgot your password?
</small>
</div>
</div>
</div>
{% endblock %}
```

我们有密码重置的功能，因此现在让我们将其暂时保留为 #。

## templates/signup.html

```
{% extends 'base_accounts.html' %}

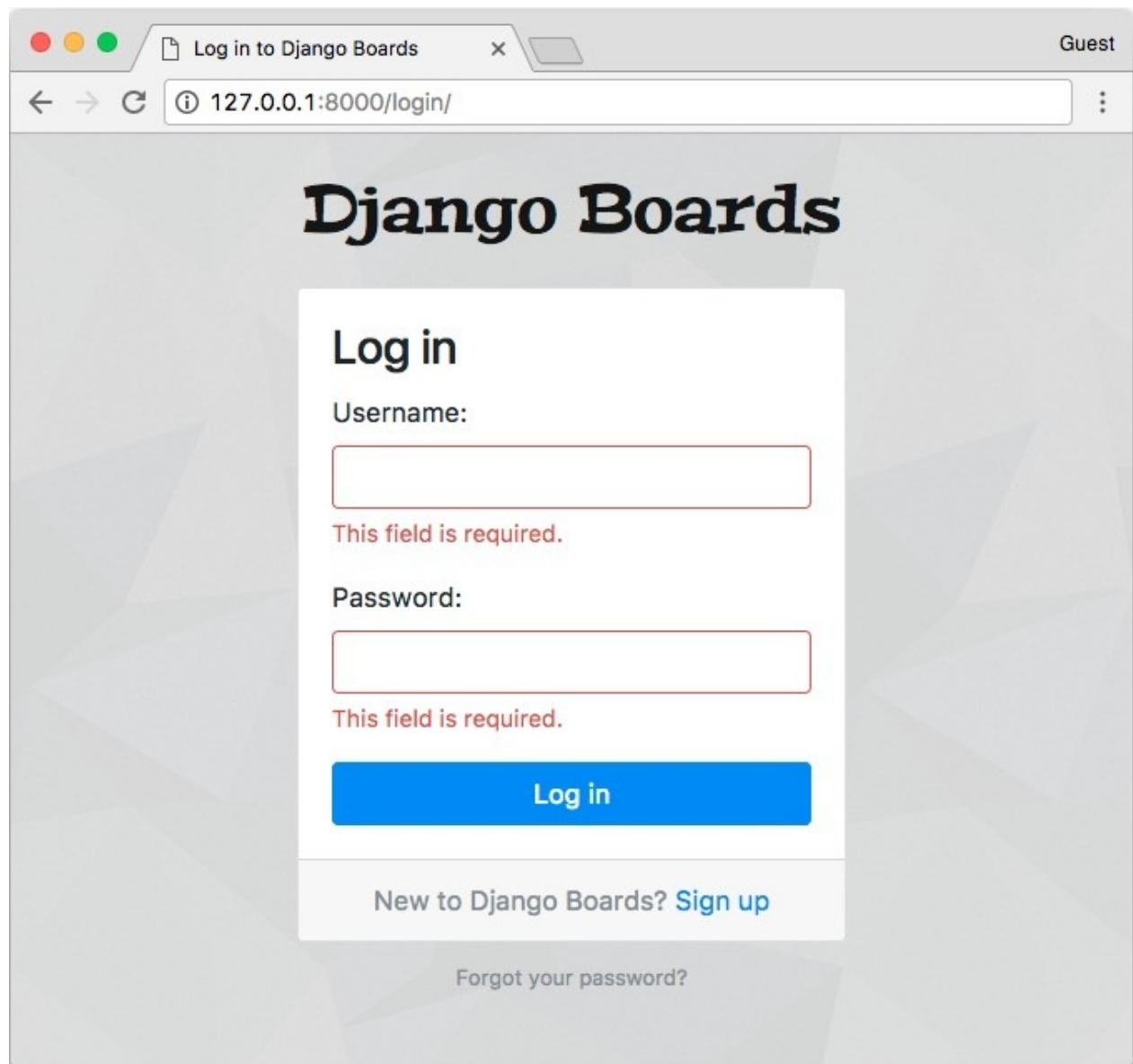
{% block title %}Sign up to Django Boards{% endblock %}

{% block content %}
<div class="row justify-content-center">
<div class="col-lg-8 col-md-10 col-sm-12">
<div class="card">
<div class="card-body">
<h3 class="card-title">Sign up</h3>
<form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-primary btn-block">Create an account</button>
</form>
</div>
<div class="card-footer text-muted text-center">
 Already have an account? Log in
</div>
</div>
</div>
</div>
{% endblock %}
```

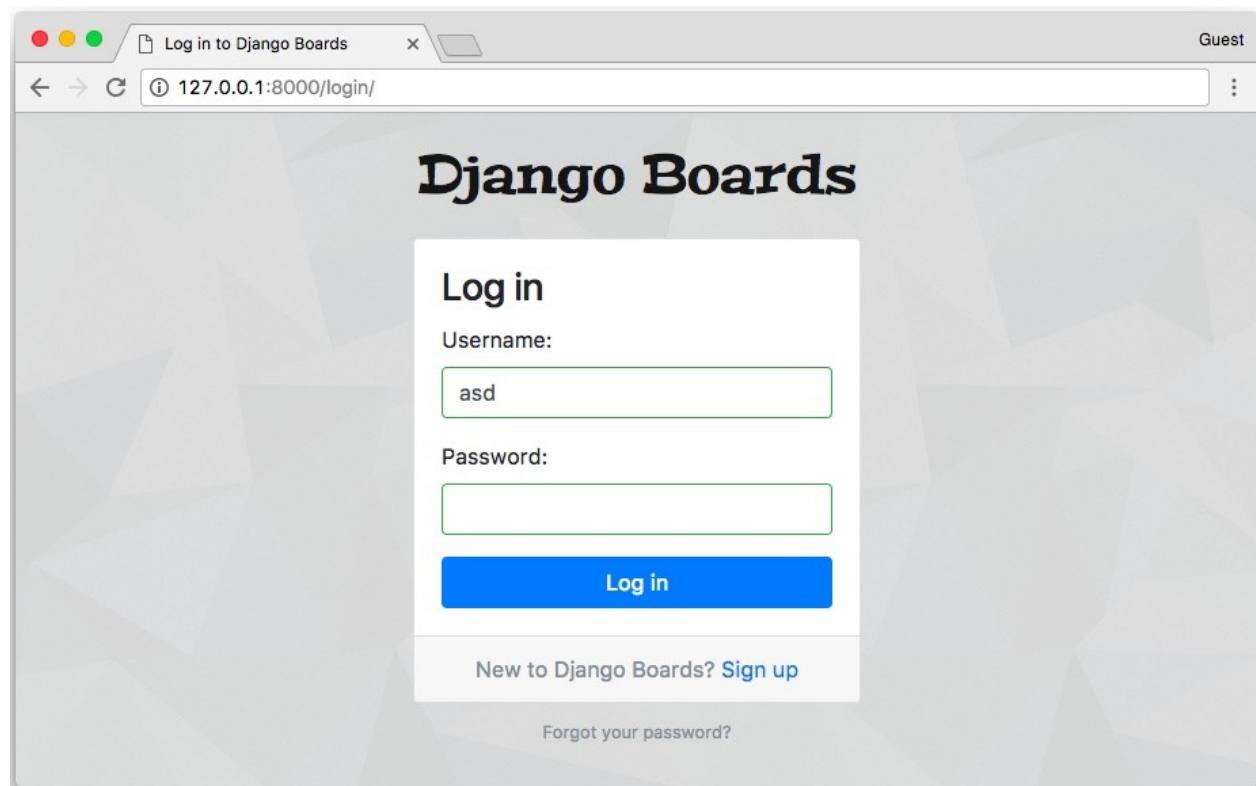
请注意，我们添加了登录链接：`<a href="{% url 'login' %}">Log in</a>`。

## 无登录信息错误

如果我们提交空白的登录信息，我们会得到一些友好的错误提示信息：



但是，如果我们提交一个不存在的用户名或一个无效的密码，现在就会发生这种情况：



有点误导，这个区域是绿色的，表明它们是良好运行的，此外，没有其他额外的信息。

这是因为表单有一种特殊类型的错误，叫做 **non-field errors**。这是一组与特定字段无关的错误。让我们重构**form.html**部分模板以显示这些错误：

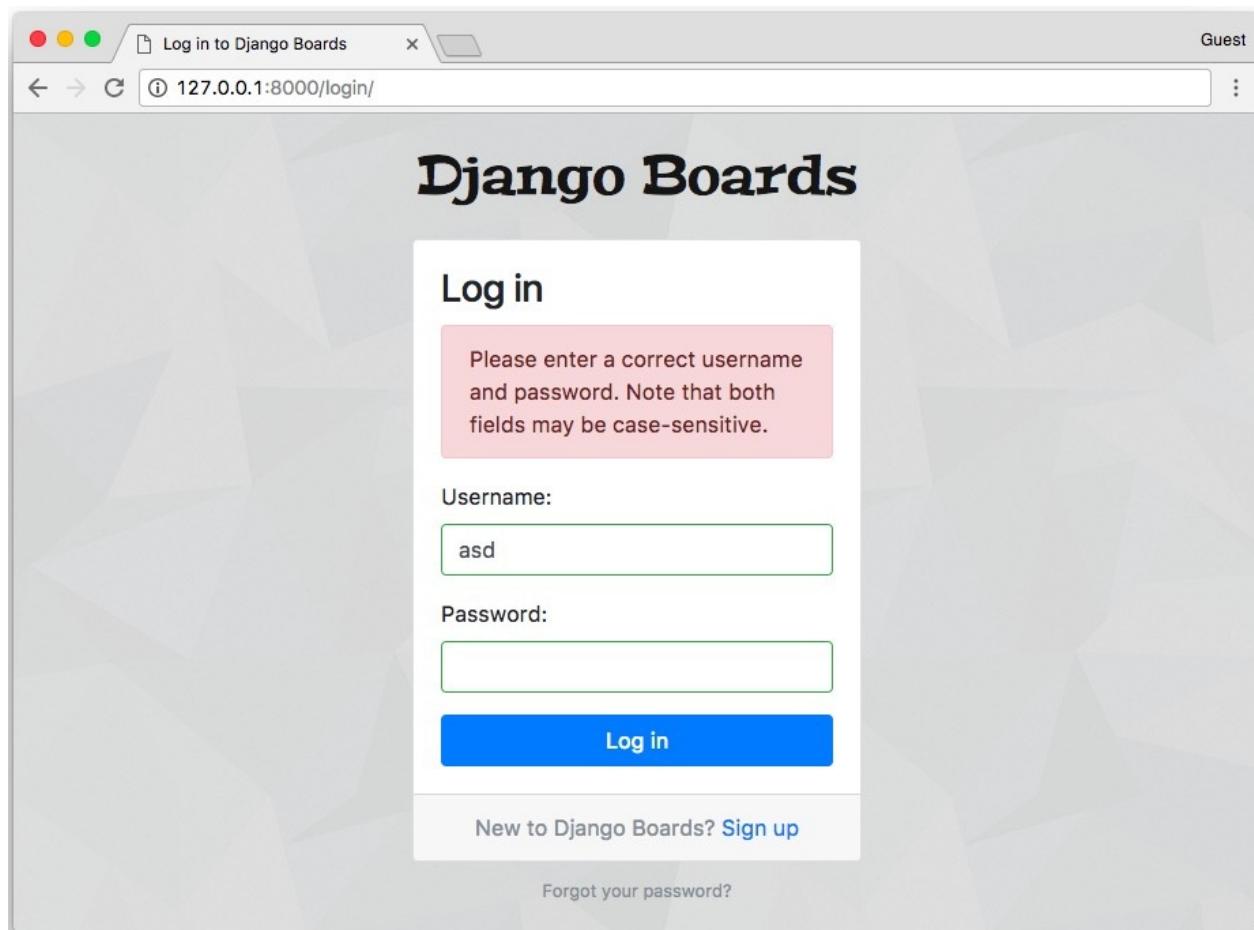
**templates/includes/form.html**

```
{% load widget_tweaks %}

{% if form.non_field_errors %}
 <div class="alert alert-danger" role="alert">
 {% for error in form.non_field_errors %}
 <p{% if forloop.last %} class="mb-0"{% endif %}>{{ error }}
 {% endfor %}
 </div>
{% endif %}

{% for field in form %}
 <!-- code suppressed -->
{% endfor %}
```

{% if forloop.last %} 只是一个小事情，因为 p 标签有一个空白的 margin-bottom。一个表单可能有几个**non-field error**，我们呈现了一个带有错误的 p 标签。然后我要检查它是否是最后一次渲染的错误。如果是这样的，我们就添加一个 Bootstrap 4 CSS类 mb-0，它的作用是代表了“margin bottom = 0”（底部边缘为0）。这样的话警告看起来就不那么奇怪了并且多了一些额外的空间。这只是一个非常小的细节。我这么做的原因只是为了保持间距的一致性。



尽管如此，我们仍然需要处理密码字段。问题在于，Django从不将密码字段的数据返回给客户端。因此，在某些情况下，不要试图做一次自作聪明的事情，我们可以直接忽略 `is-valid` 和 `is-invalid` 的CSS类。但是我们的表单模板看起来十分的复杂，我们可以将一些代码移动到模板标记中去。

## 创建自定义模板标签

在**boards**应用中，创建一个名为**templatetags**的新文件夹。然后在该文件夹内创建两个名为 `init.py` 和 `form_tags.py`的空文件。

文件结构应该如下：

```
myproject/
| -- myproject/
| | -- accounts/
| | -- boards/
| | | -- migrations/
| | | -- templatetags/ <-- here
| | | | -- __init__.py
| | | +-- form_tags.py
| | | | -- __init__.py
| | | | -- admin.py
| | | | -- apps.py
| | | | -- models.py
| | | | -- tests.py
| | | +-- views.py
| | -- myproject/
| | -- static/
| | -- templates/
| | -- db.sqlite3
| +-- manage.py
+-- venv/
```

在 **form\_tags.py** 文件中，我们创建两个模板标签：

**boards/templatetags/form\_tags.py**

```
from django import template

register = template.Library()

@register.filter
def field_type(bound_field):
 return bound_field.field.widget.__class__.__name__

@register.filter
def input_class(bound_field):
 css_class = ''
 if bound_field.form.is_bound:
 if bound_field.errors:
 css_class = 'is-invalid'
 elif field_type(bound_field) != 'PasswordInput':
 css_class = 'is-valid'
 return 'form-control {}'.format(css_class)
```

这些都是模板过滤器，他们的工作方式是这样的：

首先，我们将它加载到模板中，就像我们使用 `widget_tweaks` 或 `static` 模板标签一样。请注意，在创建这个文件后，你将不得不手动停止开发服务器并重启它，以便Django可以识别新的模板标签。

```
{% load form_tags %}
```

之后，我们就可以在模板中使用它们了。

```
{{ form.username|field_type }}
```

返回：

```
'TextInput'
```

或者在 **input\_class** 的情况下：

```
{% form.username|input_class %}

<!-- if the form is not bound, it will simply return: -->
'form-control'

<!-- if the form is bound and valid: -->
'form-control is-valid'

<!-- if the form is bound and invalid: -->
'form-control is-invalid'
```

现在更新 **form.html** 以使用新的模板标签：

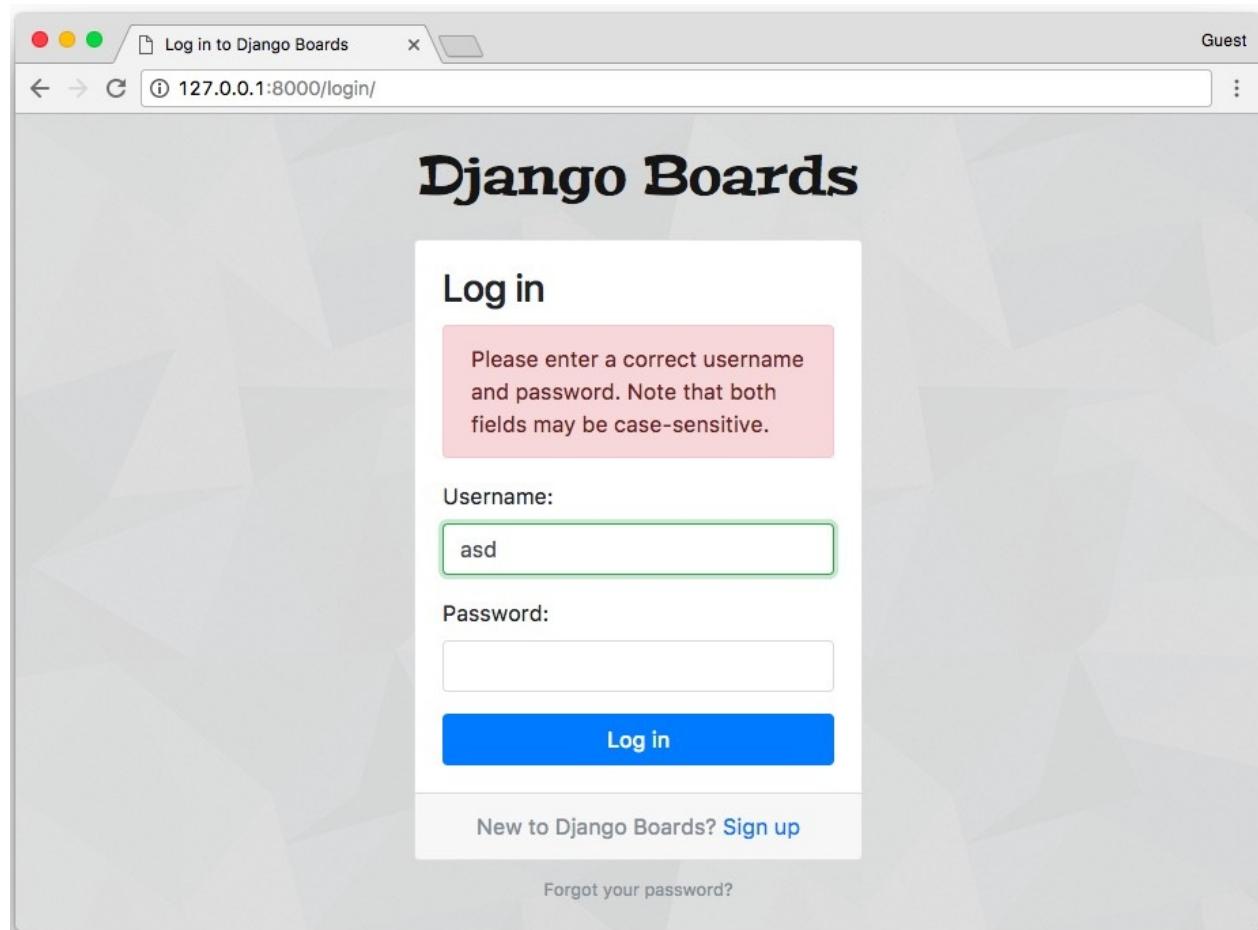
**templates/includes/form.html**

```
{% load form_tags widget_tweaks %}

{% if form.non_field_errors %}
 <div class="alert alert-danger" role="alert">
 {% for error in form.non_field_errors %}
 <p{% if forloop.last %} class="mb-0"{% endif %}>{{ erro
r }}</p>
 {% endfor %}
 </div>
{% endif %}

{% for field in form %}
 <div class="form-group">
 {{ field.label_tag }}
 {% render_field field class=field|input_class %}
 {% for error in field.errors %}
 <div class="invalid-feedback">
 {{ error }}
 </div>
 {% endfor %}
 {% if field.help_text %}
 <small class="form-text text-muted">
 {{ field.help_text|safe }}
 </small>
 {% endif %}
 </div>
{% endfor %}
```

这样的话就好多了是吧？这样做降低了模板的复杂性，它现在看起来更加整洁。并且它还解决了密码字段显示绿色边框的问题：



## 测试模板标签

首先，让我们稍微组织一下**boards**的测试。就像我们对account app 所做的那样。创建一个新的文件夹名为**tests**，添加一个**init.py**，复制**test.py**并且将其重命名为**test\_views.py**。

添加一个名为 **test\_templatetags.py**的新空文件。

```

myproject/
| -- myproject/
| | -- accounts/
| | -- boards/
| | | -- migrations/
| | | -- templatetags/
| | | -- tests/
| | | | -- __init__.py
| | | | -- test_templatetags.py <-- new file, empty
for now
| | | +-- test_views.py <-- our old file with all
the tests
| | | | -- __init__.py
| | | | -- admin.py
| | | | -- apps.py
| | | | -- models.py
| | | +-- views.py
| | | -- myproject/
| | | -- static/
| | | -- templates/
| | | -- db.sqlite3
| | +-- manage.py
+-- venv/

```

修复**test\_views.py**的导入问题：

### **boards/tests/test\_views.py**

```

from ..views import home, board_topics, new_topic
from ..models import Board, Topic, Post
from ..forms import NewTopicForm

```

执行测试来确保一切都正常。

### **boards/tests/test\_templatetags.py**

```
from django import forms
from django.test import TestCase
from ..templatetags.form_tags import field_type, input_class

class ExampleForm(forms.Form):
 name = forms.CharField()
 password = forms.CharField(widget=forms.PasswordInput())
 class Meta:
 fields = ('name', 'password')

class FieldTypeTests(TestCase):
 def test_field_widget_type(self):
 form = ExampleForm()
 self.assertEqual('TextInput', field_type(form['name']))
 self.assertEqual('PasswordInput', field_type(form['password']))

class InputClassTests(TestCase):
 def test_unbound_field_initial_state(self):
 form = ExampleForm() # unbound form
 self.assertEqual('form-control', input_class(form['name']))

 def test_valid_bound_field(self):
 form = ExampleForm({'name': 'john', 'password': '123'})
 # bound form (field + data)
 self.assertEqual('form-control is-valid', input_class(form['name']))
 self.assertEqual('form-control', input_class(form['password']))

 def test_invalid_bound_field(self):
 form = ExampleForm({'name': '', 'password': '123'})
 # bound form (field + data)
 self.assertEqual('form-control is-invalid', input_class(form['name']))
```

我们创建了一个用于测试的表单类，然后添加了覆盖两个模板标记中可能出现的场景的测试用例。

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.
.
.
Ran 32 tests in 0.846s

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

## 密码重置

密码重置过程中涉及一些不友好的 URL 模式。但正如我们在前面的教程中讨论的那样，我们并不需要成为正则表达式专家。我们只需要了解常见问题和它们的解决办法。

在我们开始之前另一件重要的事情是，对于密码重置过程，我们需要发送电子邮件。一开始有点复杂，因为我们需要外部服务。目前，我们不会配置生产环境使用的电子邮件服务。实际上，在开发阶段，我们可以使用Django的调试工具检查电子邮件是否正确发送。



## 控制台收发Email

这个主意来自于项目开发过程中，而不是发送真实的电子邮件，我们只需要记录它们。我们有两种选择：将所有电子邮件写入文本文件或仅将其显示在控制台中。我发现第二个方式更加方便，因为我们已经在使用控制台来运行开发服务器，并且设置更容易一些。

编辑 `settings.py` 模块并将 `EMAIL_BACKEND` 变量添加到文件的末尾。

### myproject/settings.py

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

## 配置路由

密码重置过程需要四个视图：

- 带有表单的页面，用于启动重置过程；
- 一个成功的页面，表示该过程已启动，指示用户检查其邮件文件夹等；
- 检查通过电子邮件发送token的页面
- 一个告诉用户重置是否成功的页面

这些视图是内置的，我们不需要执行任何操作，我们所需要做的就是将路径添加到 `urls.py` 并且创建模板。

### myproject/urls.py ([完整代码](#))

```
url(r'^reset/$',
 auth_views.PasswordResetView.as_view(
 template_name='password_reset.html',
 email_template_name='password_reset_email.html',
 subject_template_name='password_reset_subject.txt'
),
 name='password_reset'),
url(r'^reset/done/$',
 auth_views.PasswordResetDoneView.as_view(template_name='password_reset_done.html'),
 name='password_reset_done'),
url(r'^reset/(?P<uidb64>[0-9A-Za-z_\-]+)/(?P<token>[0-9A-Za-z]{1,13}-[0-9A-Za-z]{1,20})/$',
 auth_views.PasswordResetConfirmView.as_view(template_name='password_reset_confirm.html'),
 name='password_reset_confirm'),
url(r'^reset/complete/$',
 auth_views.PasswordResetCompleteView.as_view(template_name='password_reset_complete.html'),
 name='password_reset_complete'),
]
```

在密码重置视图中，`template_name` 参数是可选的。但我认为重新定义它是个好主意，因此视图和模板之间的链接比仅使用默认值更加明显。

在 **templates** 文件夹中，新增如下模板文件

- **password\_reset.html**
- **password\_reset\_email.html**:这个模板是发送给用户的电子邮件正文
- **password\_reset\_subject.txt**:这个模板是电子邮件的主题行，它应该是单行文件
- **password\_reset\_done.html**
- **password\_reset\_confirm.html**
- **password\_reset\_complete.html**

在我们开始实现模板之前，让我们准备一个新的测试文件。

我们可以添加一些基本的测试，因为这些视图和表单已经在Django代码中进行了测试。我们将只测试我们应用程序的细节。

在**accounts/tests** 文件夹中创建一个名为 **test\_view\_password\_reset.py** 的新测试文件。

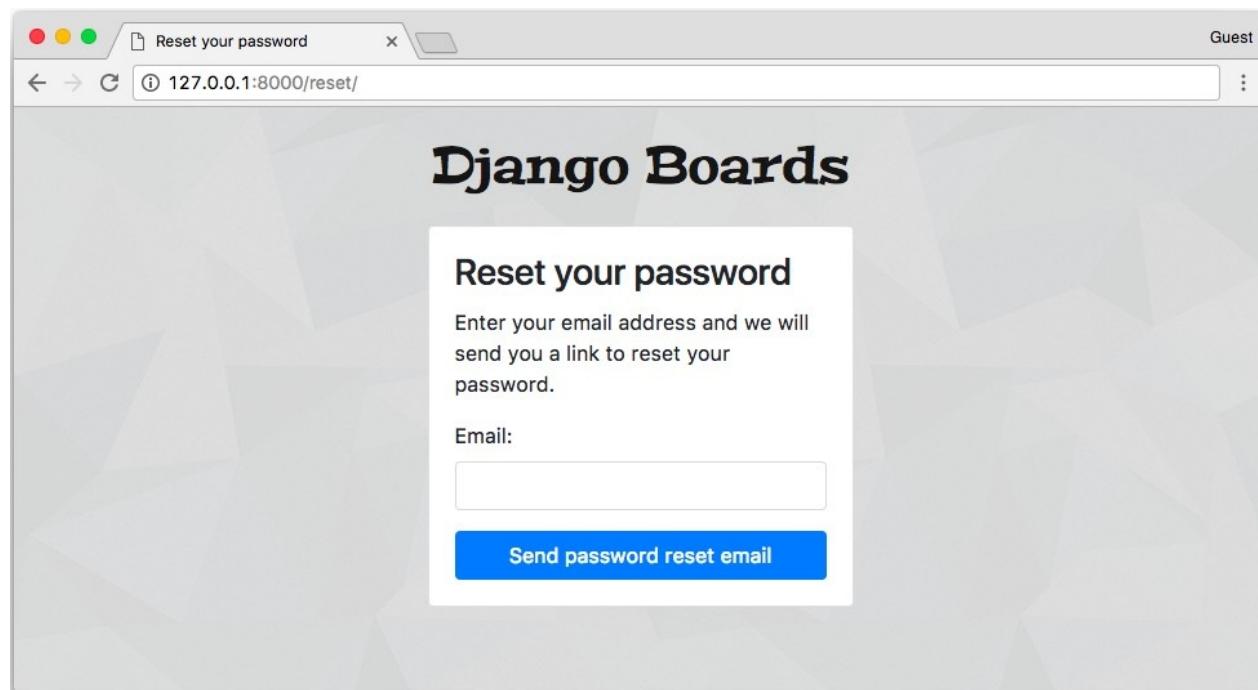
## 密码重置视图

### templates/password\_reset.html

```
{% extends 'base_accounts.html' %}

{% block title %}Reset your password{% endblock %}

{% block content %}
 <div class="row justify-content-center">
 <div class="col-lg-4 col-md-6 col-sm-8">
 <div class="card">
 <div class="card-body">
 <h3 class="card-title">Reset your password</h3>
 <p>Enter your email address and we will send you a link to reset your password.</p>
 <form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-primary btn-block">Send password reset email</button>
 </form>
 </div>
 </div>
 </div>
 </div>
{% endblock %}
```



### accounts/tests/test\_view\_password\_reset.py

```
from django.contrib.auth import views as auth_views
from django.contrib.auth.forms import PasswordResetForm
from django.contrib.auth.models import User
from django.core import mail
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase

class PasswordResetTests(TestCase):
 def setUp(self):
 url = reverse('password_reset')
 self.response = self.client.get(url)

 def test_status_code(self):
 self.assertEquals(self.response.status_code, 200)

 def test_view_function(self):
 view = resolve('/reset/')
 self.assertEquals(view.func.view_class, auth_views.PasswordResetView)
```

```
def test_csrf(self):
 self.assertContains(self.response, 'csrfmiddlewaretoken')

def test_contains_form(self):
 form = self.response.context.get('form')
 self.assertIsInstance(form, PasswordResetForm)

def test_form_inputs(self):
 """
 The view must contain two inputs: csrf and email
 """
 self.assertContains(self.response, '<input', 2)
 self.assertContains(self.response, 'type="email"', 1)

class SuccessfulPasswordResetTests(TestCase):
 def setUp(self):
 email = 'john@doe.com'
 User.objects.create_user(username='john', email=email,
 password='123abcdef')
 url = reverse('password_reset')
 self.response = self.client.post(url, {'email': email})

 def test_redirection(self):
 """
 A valid form submission should redirect the user to `password_reset_done` view
 """
 url = reverse('password_reset_done')
 self.assertRedirects(self.response, url)

 def test_send_password_reset_email(self):
 self.assertEqual(1, len(mail.outbox))
```

```
class InvalidPasswordResetTests(TestCase):
 def setUp(self):
 url = reverse('password_reset')
 self.response = self.client.post(url, {'email': 'donotexist@email.com'})

 def test_redirection(self):
 """
 Even invalid emails in the database should
 redirect the user to `password_reset_done` view
 """
 url = reverse('password_reset_done')
 self.assertRedirects(self.response, url)

 def test_no_reset_email_sent(self):
 self.assertEqual(0, len(mail.outbox))
```

### templates/password\_reset\_subject.txt

```
[Django Boards] Please reset your password
```

### templates/password\_reset\_email.html

Hi there,

Someone asked for a password reset for the email address {{ email }}.

Follow the link below:

[{{ protocol }}://{{ domain }}{% url 'password\\_reset\\_confirm' uidb64=uid token=token %}]({{ protocol }}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid token=token %})

In case you forgot your Django Boards username: {{ user.username }}

If clicking the link above doesn't work, please copy and paste the URL

in a new browser window instead.

If you've received this mail in error, it's likely that another user entered

your email address by mistake while trying to reset a password. If you didn't

initiate the request, you don't need to take any further action and can safely

disregard this email.

Thanks,

The Django Boards Team

```
django-beginners-guide — IPython: myproject/django-beginners-guide — python -m runpy manage.py runs...
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: [django Boards] Please reset your password
From: webmaster@localhost
To: vitor@simpleisbetterthancomplex.com
Date: Fri, 22 Sep 2017 18:53:39 -0000
Message-ID: <20170922185339.55713.86973@vitor-macbookair.local>

Hi there,

Someone asked for a password reset for the email address vitor@simpleisbetterthancomplex.com. Follow the link below:
http://127.0.1:8000/reset/Mw/4po-2b5f2d47c19966e294a1/

In case you forgot your Django Boards username: vitorfs

If clicking the link above doesn't work, please copy and paste the URL in a new browser window instead.

If you've received this mail in error, it's likely that another user entered your email address by mistake while trying to reset a password. If you didn't initiate the request, you don't need to take any further action and can safely disregard this email.

Thanks,
The Django Boards Team
```

我们可以创建一个特定的文件来测试电子邮件。在**accounts/tests** 文件夹中创建一个名为**test\_mail\_password\_reset.py**的新文件：

**accounts/tests/test\_mail\_password\_reset.py**

```
from django.core import mail
from django.contrib.auth.models import User
from django.urls import reverse
from django.test import TestCase

class PasswordResetMailTests(TestCase):
 def setUp(self):
 User.objects.create_user(username='john', email='john@doe.com', password='123')
 self.response = self.client.post(reverse('password_reset'),
 { 'email': 'john@doe.com' })
 self.email = mail.outbox[0]

 def test_email_subject(self):
 self.assertEqual('[Django Boards] Please reset your password', self.email.subject)

 def test_email_body(self):
 context = self.response.context
 token = context.get('token')
 uid = context.get('uid')
 password_reset_token_url = reverse('password_reset_confirm',
 kwargs={
 'uidb64': uid,
 'token': token
 })
 self.assertIn(password_reset_token_url, self.email.body)
 self.assertIn('john', self.email.body)
 self.assertIn('john@doe.com', self.email.body)

 def test_email_to(self):
 self.assertEqual(['john@doe.com'], self.email.to)
```

此测试用例抓取应用程序发送的电子邮件，并检查主题行，正文内容以及发给谁。

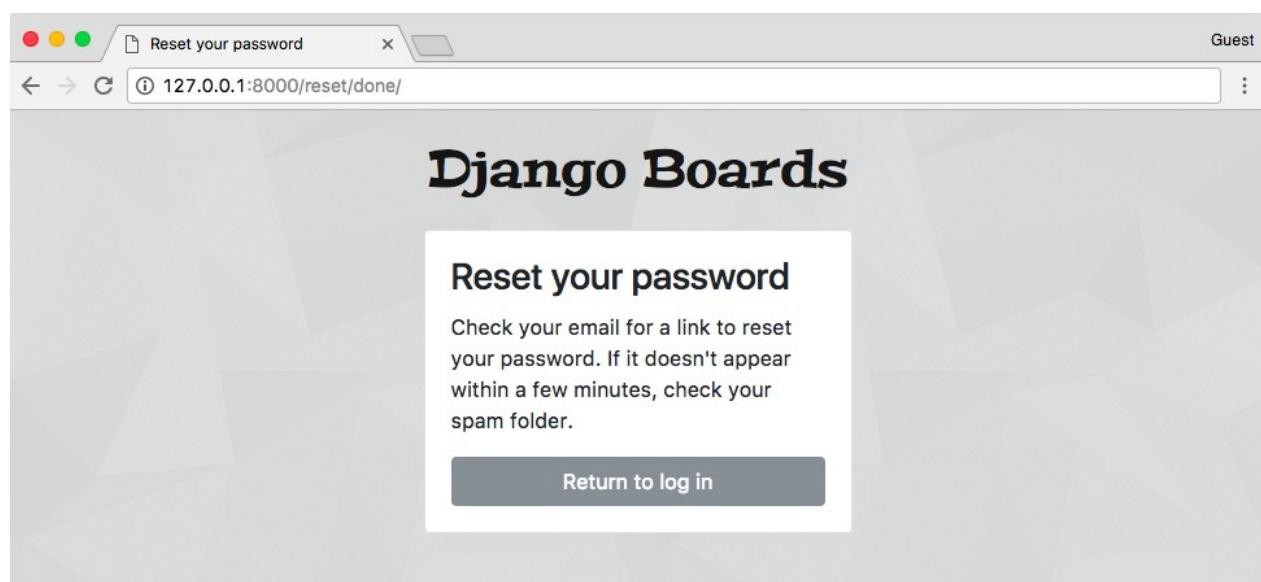
## 密码重置完成视图

### templates/password\_reset\_done.html

```
{% extends 'base_accounts.html' %}

{% block title %}Reset your password{% endblock %}

{% block content %}
<div class="row justify-content-center">
<div class="col-lg-4 col-md-6 col-sm-8">
<div class="card">
<div class="card-body">
<h3 class="card-title">Reset your password</h3>
<p>Check your email for a link to reset your password. If it doesn't appear within a few minutes, check your spam folder.</p>
Return to log in
</div>
</div>
</div>
</div>
{% endblock %}
```



**accounts/tests/test\_view\_password\_reset.py**

```
from django.contrib.auth import views as auth_views
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase

class PasswordResetDoneTests(TestCase):
 def setUp(self):
 url = reverse('password_reset_done')
 self.response = self.client.get(url)

 def test_status_code(self):
 self.assertEqual(self.response.status_code, 200)

 def test_view_function(self):
 view = resolve('/reset/done/')
 self.assertEqual(view.func.view_class, auth_views.PasswordResetDoneView)
```

## 密码重置确认视图

**templates/password\_reset\_confirm.html**

```
{% extends 'base_accounts.html' %}

{% block title %}
 {% if validlink %}
 Change password for {{ form.user.username }}
 {% else %}
 Reset your password
 {% endif %}
{% endblock %}

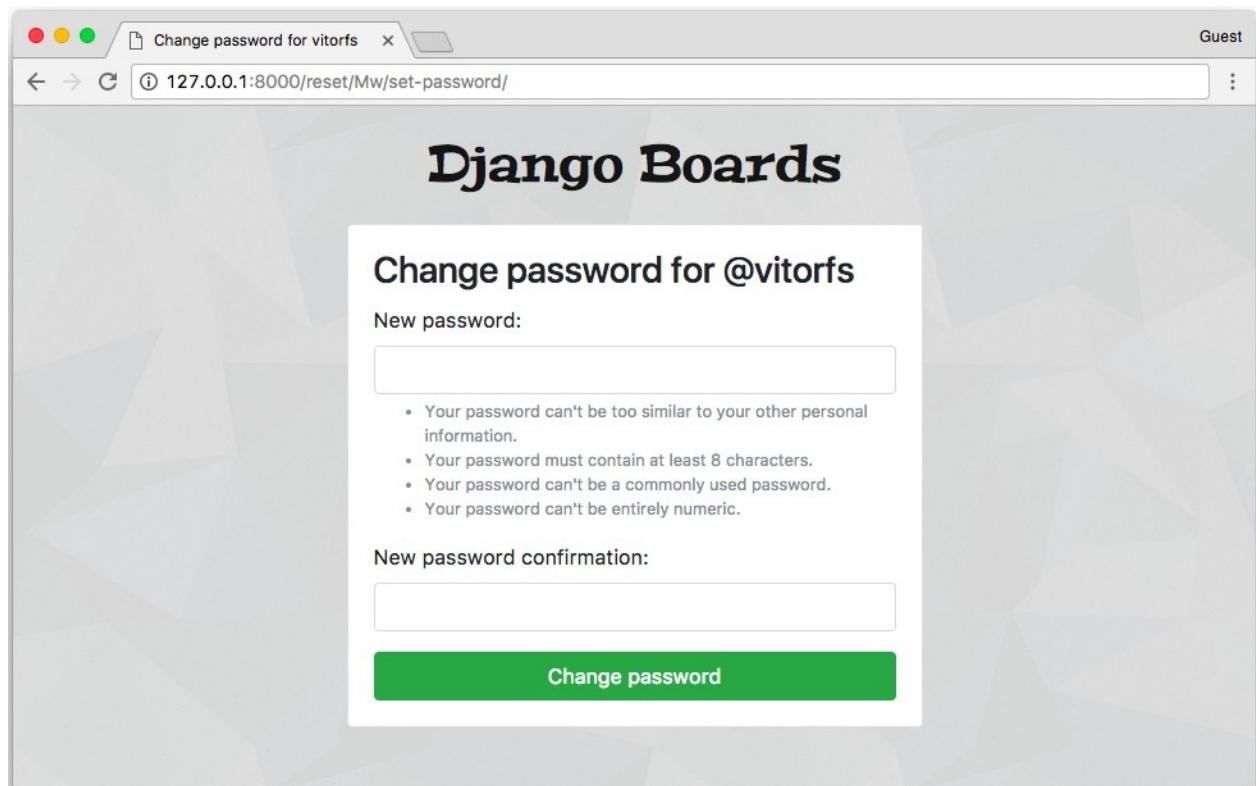
{% block content %}
 <div class="row justify-content-center">
```

```
<div class="col-lg-6 col-md-8 col-sm-10">
 <div class="card">
 <div class="card-body">
 {% if validlink %}
 <h3 class="card-title">Change password for @{{ form.user.username }}</h3>
 <form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-success btn-block">Change password</button>
 </form>
 {% else %}
 <h3 class="card-title">Reset your password</h3>
 <div class="alert alert-danger" role="alert">
 It looks like you clicked on an invalid password reset link. Please try again.
 </div>
 Request a new password reset link
 {% endif %}
 </div>
 </div>
</div>
{% endblock %}
```

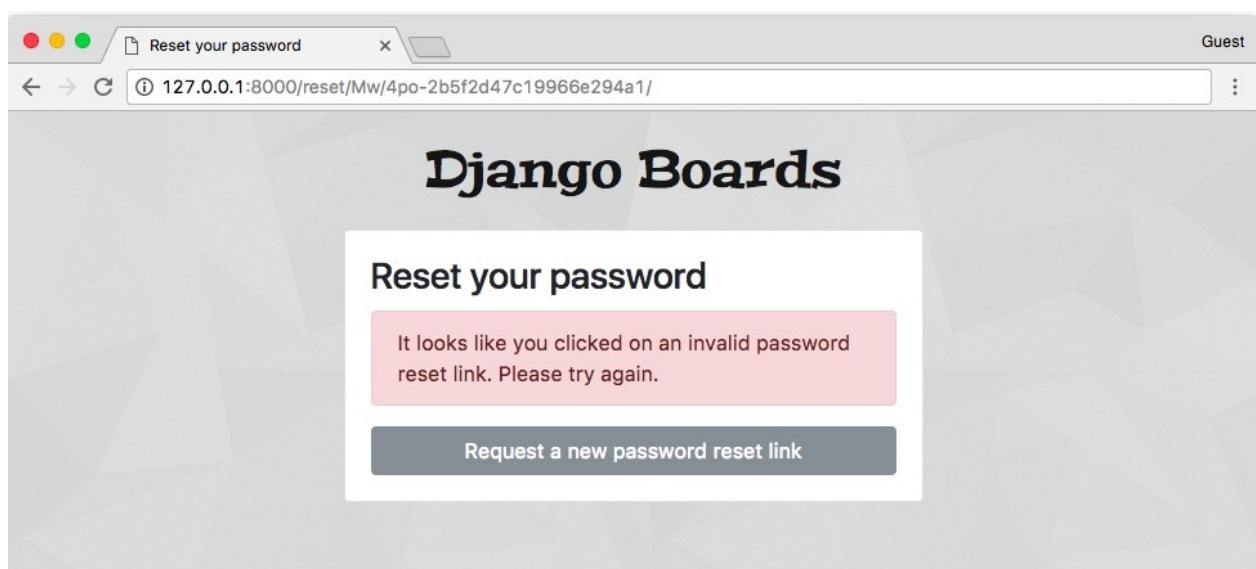
这个页面只能通过电子邮件访问，它看起来像这样：<http://127.0.0.1:8000/reset/Mw/4po-2b5f2d47c19966e294a1/>

在开发阶段，从控制台中的电子邮件获取此链接。

如果链接是有效的：



倘若链接已经被使用：



### accounts/tests/test\_view\_password\_reset.py

```
from django.contrib.auth.tokens import default_token_generator
from django.utils.encoding import force_bytes
from django.utils.http import urlsafe_base64_encode
from django.contrib.auth import views as auth_views
from django.contrib.auth.forms import SetPasswordForm
```

```
from django.contrib.auth.models import User
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase

class PasswordResetConfirmTests(TestCase):
 def setUp(self):
 user = User.objects.create_user(username='john', email='john@doe.com', password='123abcdef')
 ...
 create a valid password reset token
 based on how django creates the token internally:
 https://github.com/django/django/blob/1.11.5/django/contrib/auth/forms.py#L280
 ...
 self.uid = urlsafe_base64_encode(force_bytes(user.pk)).decode()
 self.token = default_token_generator.make_token(user)

 url = reverse('password_reset_confirm', kwargs={'uidb64': self.uid, 'token': self.token})
 self.response = self.client.get(url, follow=True)

 def test_status_code(self):
 self.assertEquals(self.response.status_code, 200)

 def test_view_function(self):
 view = resolve('/reset/{uidb64}/{token}/'.format(uidb64=self.uid, token=self.token))
 self.assertEquals(view.func.view_class, auth_views.PasswordResetConfirmView)

 def test_csrf(self):
 self.assertContains(self.response, 'csrfmiddlewaretoken')
```

```
def test_contains_form(self):
 form = self.response.context.get('form')
 self.assertIsInstance(form, SetPasswordForm)

def test_form_inputs(self):
 """
 The view must contain two inputs: csrf and two password fields
 """
 self.assertContains(self.response, '<input', 3)
 self.assertContains(self.response, 'type="password"', 2)

class InvalidPasswordResetConfirmTests(TestCase):
 def setUp(self):
 user = User.objects.create_user(username='john', email='john@doe.com', password='123abcdef')
 uid = urlsafe_base64_encode(force_bytes(user.pk)).decode()
 token = default_token_generator.make_token(user)

 """
 invalidate the token by changing the password
 """
 user.set_password('abcdef123')
 user.save()

 url = reverse('password_reset_confirm', kwargs={'uidb64': uid, 'token': token})
 self.response = self.client.get(url)

 def test_status_code(self):
 self.assertEqual(self.response.status_code, 200)

 def test_html(self):
 password_reset_url = reverse('password_reset')
 self.assertContains(self.response, 'invalid password')
```

```
 reset_link')
 self.assertContains(self.response, 'href="{0}"'.format(password_reset_url))
```

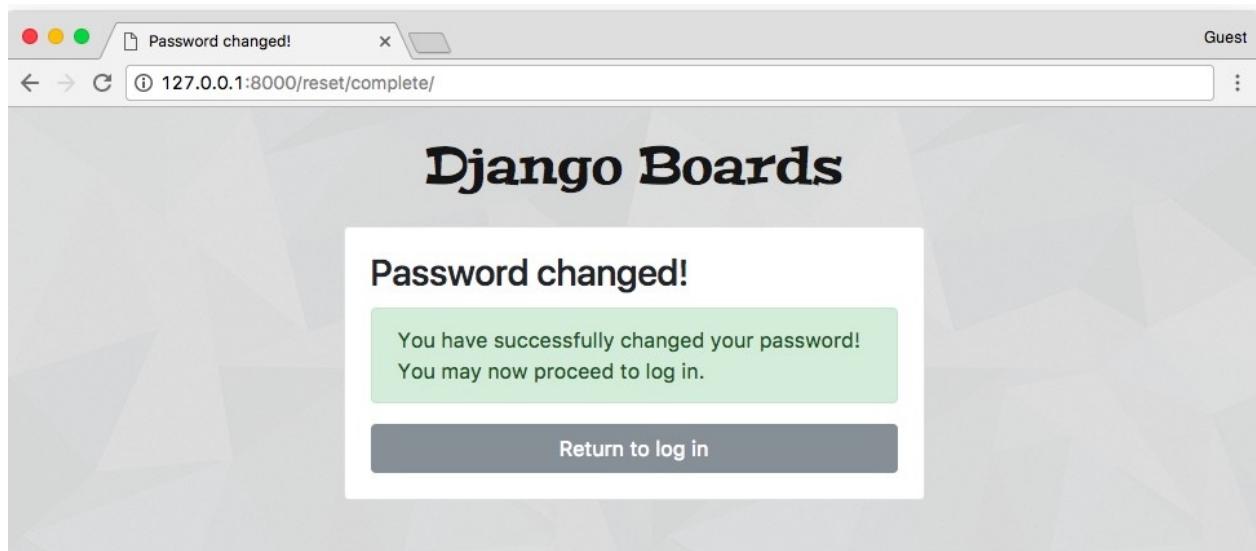
## 密码重置完成视图

### templates/password\_reset\_complete.html

```
{% extends 'base_accounts.html' %}

{% block title %}Password changed!{% endblock %}

{% block content %}
<div class="row justify-content-center">
 <div class="col-lg-6 col-md-8 col-sm-10">
 <div class="card">
 <div class="card-body">
 <h3 class="card-title">Password changed!</h3>
 <div class="alert alert-success" role="alert">
 You have successfully changed your password! You
 may now proceed to log in.
 </div>
 Return to log in
 </div>
 </div>
 </div>
</div>
{% endblock %}
```



**accounts/tests/test\_view\_password\_reset.py** ([view complete file contents](#))

```
from django.contrib.auth import views as auth_views
from django.core.urlresolvers import reverse
from django.urls import resolve
from django.test import TestCase

class PasswordResetCompleteTests(TestCase):
 def setUp(self):
 url = reverse('password_reset_complete')
 self.response = self.client.get(url)

 def test_status_code(self):
 self.assertEqual(self.response.status_code, 200)

 def test_view_function(self):
 view = resolve('/reset/complete/')
 self.assertEqual(view.func.view_class, auth_views.PasswordResetCompleteView)
```

## 密码更改视图

此视图旨在提供给希望更改其密码的登录用户使用。通常，这些表单由三个字段组成：旧密码、新密码、新密码确认。

### myproject/urls.py ([view complete file contents](#))

```
url(r'^settings/password/$', auth_views.PasswordChangeView.as_view(template_name='password_change.html'),
 name='password_change'),
url(r'^settings/password/done/$', auth_views.PasswordChangeDoneView.as_view(template_name='password_change_done.html'),
 name='password_change_done'),
```

这些视图仅适合登录用户，他们使用名为 `@login_required` 的装饰器，此装饰器可防止非授权用户访问此页面。如果用户没有登录，Django会将他们重定向到登录页面。

现在我们必须在**settings.py**中定义我们应用程序的登录URL：

### myproject/settings.py ([view complete file contents](#))

```
LOGIN_URL = 'login'
```

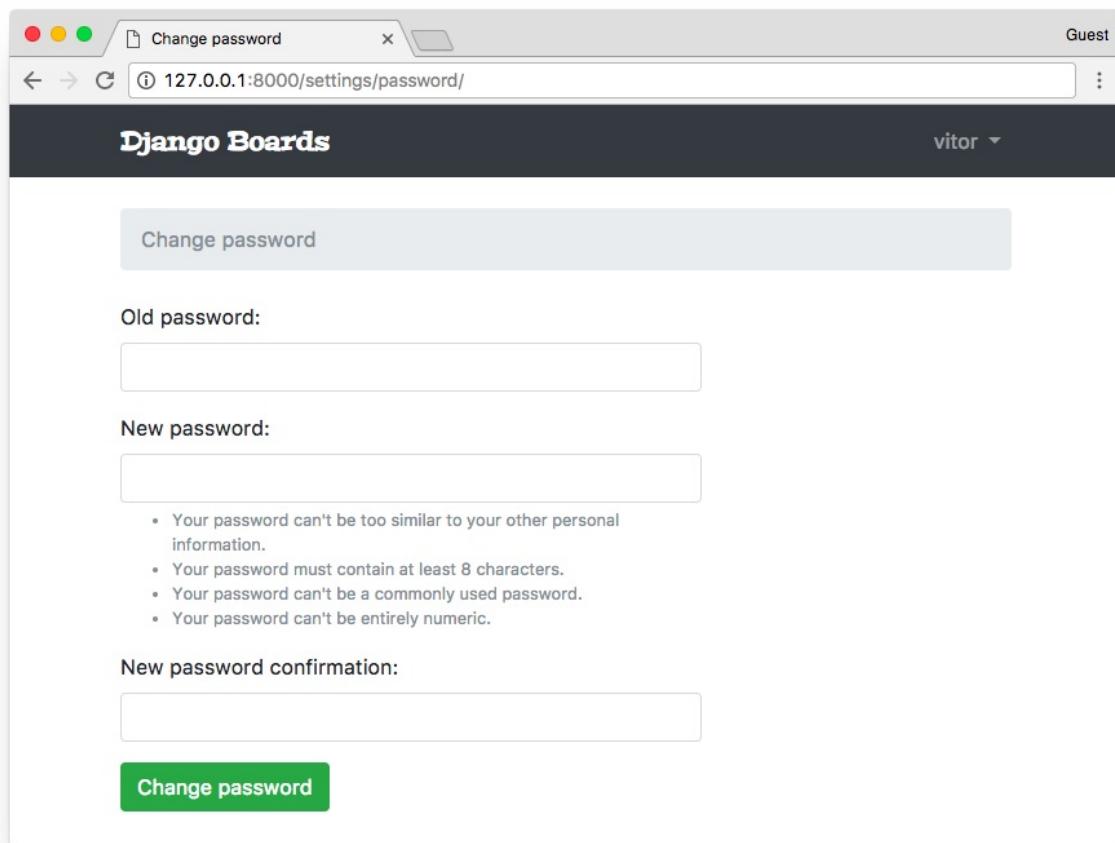
### templates/password\_change.html

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

{% block title %}Change password{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item active">Change password
{% endblock %}

{% block content %}
 <div class="row">
 <div class="col-lg-6 col-md-8 col-sm-10">
 <form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-success">Change
password</button>
 </form>
 </div>
 </div>
{% endblock %}
```



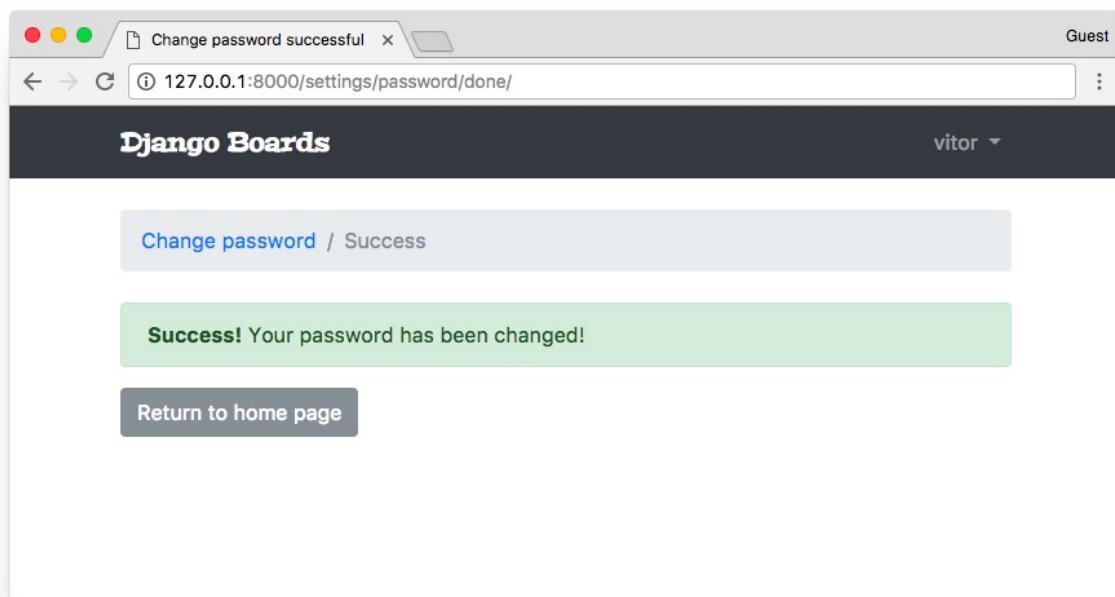
## templates/password\_change\_done.html

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

{% block title %}Change password successful{% endblock %}

{% block breadcrumb %}
<li class="breadcrumb-item">Change password
<li class="breadcrumb-item active">Success
{% endblock %}

{% block content %}
<div class="alert alert-success" role="alert">
Success! Your password has been changed!
</div>
Return to home page
{% endblock %}
```



关于密码更改视图，我们可以执行类似的测试用例，就像我们迄今为止所做的那样。创建一个名为**test\_view\_password\_change.py**的新测试文件。

我将在下面列出新的测试类型。你可以检查我为密码更改视图编写的所有测试，然后单击代码段旁边的查看文正文件内容链接。大部分测试与我们迄今为止所做的相似。我转移到一个外部文件以避免太过于复杂。

**accounts/tests/test\_view\_password\_change.py** ([view complete file contents](#))

```
class LoginRequiredPasswordChangeTests(TestCase):
 def test_redirection(self):
 url = reverse('password_change')
 login_url = reverse('login')
 response = self.client.get(url)
 self.assertRedirects(response, f'{login_url}?next={url}'')
```

上面的测试尝试访问**password\_change**视图而不登录。预期的行为是将用户重定向到登录页面。

**accounts/tests/test\_view\_password\_change.py** ([view complete file contents](#))

```
class PasswordChangeTestCase(TestCase):
 def setUp(self, data={}):
 self.user = User.objects.create_user(username='john',
email='john@doe.com', password='old_password')
 self.url = reverse('password_change')
 self.client.login(username='john', password='old_password')
 self.response = self.client.post(self.url, data)
```

在这里我们定义了一个名为**PasswordChangeTestCase** 的新类。它将进行基本的设置，创建用户并向 **password\_change**视图发送一个**POST** 请求。在下一组测试用例中，我们将使用这个类而不是 **TestCase**类来测试成功请求和无效请求：

**accounts/tests/test\_view\_password\_change.py** ([view complete file contents](#))

```
class SuccessfulPasswordChangeTests(PasswordChangeTestCase):
 def setUp(self):
 super().setUp({
 'old_password': 'old_password',
 'new_password1': 'new_password',
 'new_password2': 'new_password',
 })

 def test_redirection(self):
 """
 A valid form submission should redirect the user
 """
 self.assertRedirects(self.response, reverse('password_change_done'))

 def test_password_changed(self):
 """
 refresh the user instance from database to get the new password
 hash updated by the change password view.
 """
 self.user.refresh_from_db()
 self.assertTrue(self.user.check_password('new_password'))

 def test_user_authentication(self):
 """
 Create a new request to an arbitrary page.
 The resulting response should now have an `user` to its context, after a successful sign up.
 """
 response = self.client.get(reverse('home'))
 user = response.context.get('user')
 self.assertTrue(user.is_authenticated)
```

```
class InvalidPasswordChangeTests(PasswordChangeTestCase):
 def test_status_code(self):
 ...
 An invalid form submission should return to the same
 page
 ...
 self.assertEquals(self.response.status_code, 200)

 def test_form_errors(self):
 form = self.response.context.get('form')
 self.assertTrue(form.errors)

 def test_didnt_change_password(self):
 ...
 refresh the user instance from the database to make
 sure we have the latest data.
 ...
 self.user.refresh_from_db()
 self.assertTrue(self.user.check_password('old_passwor
d'))
```

`refresh_from_db()` 方法确保我们拥有最新的数据状态。它强制Django再次查询数据库以更新数据。考虑到**change\_password**视图会更新数据库中的密码，我们必须这样做。为了查看测试密码是否真的改变了，我们必须从数据库中获取最新的数据。

## 总结

对于大多数Django应用程序，身份验证是一种非常常见的用例。在本教程中，我们实现了所有重要视图：注册、登录、注销、密码重置和更改密码。现在我们有了一种方法来创建用户并进行身份验证，我们将能够继续开发应用程序和其他视图。

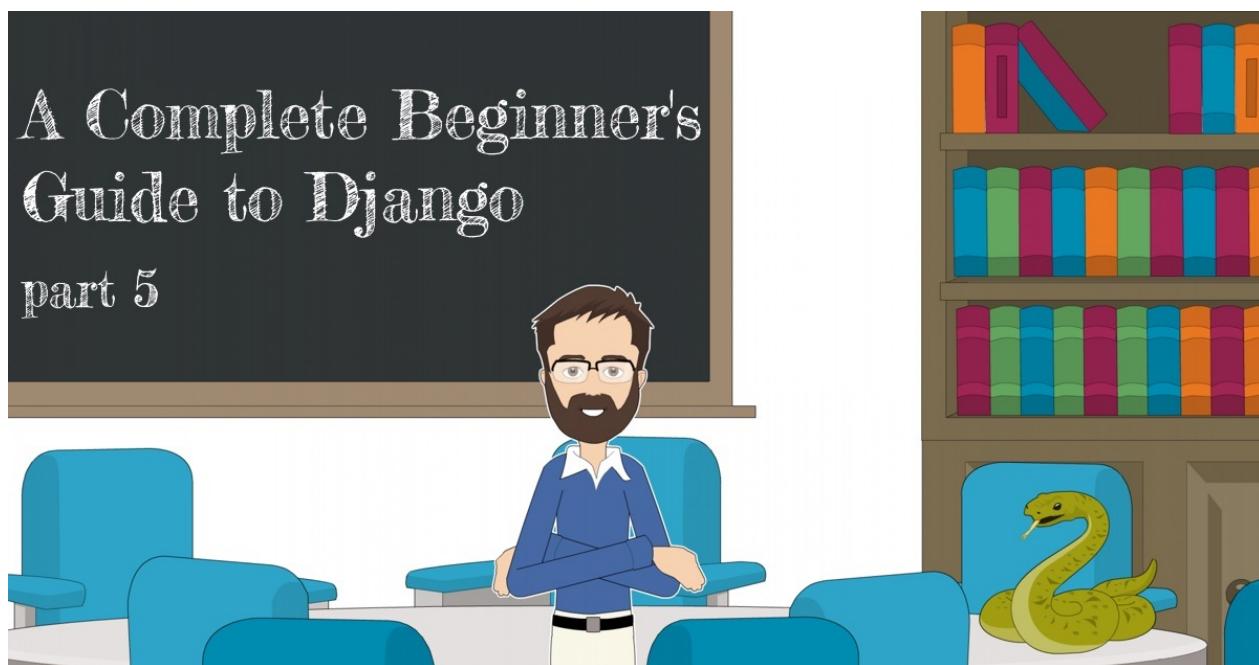
我们仍然需要改进很多关于代码设计的问题：模板文件夹开始变得乱七八糟。**boards** 应用测试仍然是混乱的。此外，我们必须开始重构新的主题视图，因为现在我们可以检索登录的用户。我们很快就将做到这一点。

我希望你喜欢本教程系列的第四部分！第五部分将于2017年10月2日下周发布，如果您希望在第五部分结束的时候收到通知，请您订阅我们的邮件列表。

该项目的源代码在GitHub上面可用，项目的当前状态在发布标签**v0.4-lw**下可以找到。链接如下：

<https://github.com/sibtc/django-beginners-guide/tree/v0.4-lw>

# Django入门与实践-第17章：保护视图

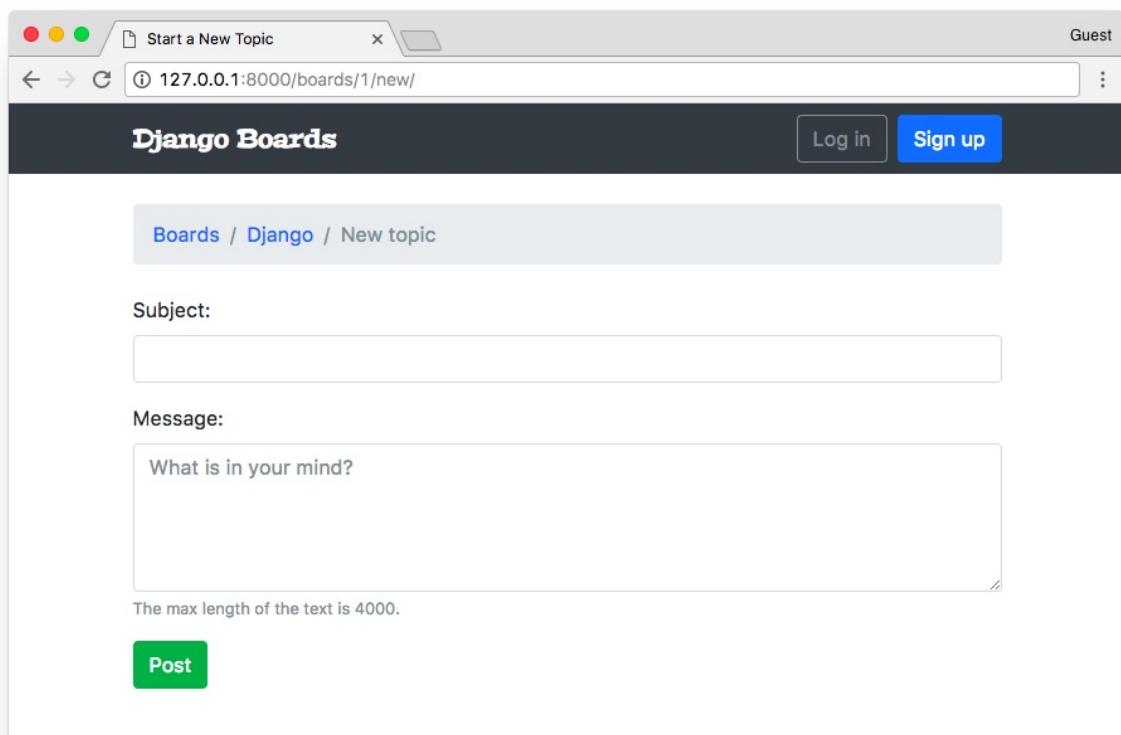


## 前言

欢迎来到本系列教程的第5部分，在这节课，我们将学习如何保护视图防止未登录的用户访问，以及在视图和表单中访问已经登录的用户，我们还将实现主题列表和回复列表视图，最后，将探索 Django ORM 的一些特性和数据迁移的简单介绍。

## 保护视图

我们必须保护视图防止那些未认证（登录）的用户访问，下面是发起一个新话题的页面



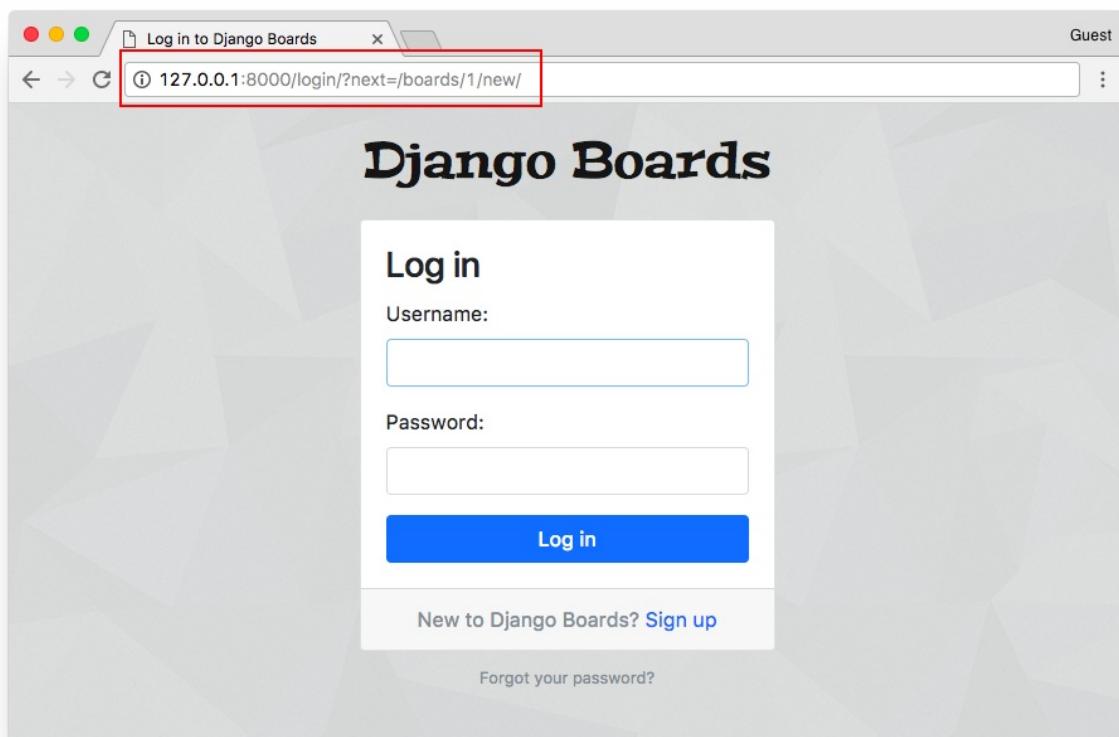
在上图中，用户还没有登录，尽管他们可以看到页面和表单。Django 有一个内置的 视图装饰器 来避免它被未登录的用户访问：

## boards/views.py (完整代码)

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

@login_required
def new_topic(request, pk):
 # ...
```

现在如果用户没有登录，将被重定向到登录页面：



注意查询字符串 **?next=/boards/1/new/**，我们可以改进登录模板以便利用 **next** 变量来改进我们的用户体验，（译注：实际上这步操作不加也没问题）

## 配置登录后的重定向地址

**templates/login.html** ([查看完整内容](#))

```
<form method="post" novalidate>
 {% csrf_token %}
 <input type="hidden" name="next" value="{{ next }}>
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-primary btn-block">Log
 in</button>
</form>
```

现在尝试登录，登录成功后，应用程序会跳转到原来所在的位置。



**next** 参数是内置功能的一部分（译注：详情请参考[Django官方文档](#)）

## 测试

现在添加一个测试用例确保主题发布视图被 `@login_required` 装饰器保护了，不过，我们还是先来重构一下 `boards/tests/test_views.py` 文件。

把 `test_views.py` 拆分成 3 个文件：

- `test_view_home.py` 包含 `HomeTests` 类 ([完整代码](#))
- `test_view_board_topics.py` 包含 `BoardTopicsTests` 类 ([完整代码](#))
- `test_view_new_topic.py` 包含 `NewTopicTests` 类 ([完整代码](#))

```
myproject/
| -- myproject/
| | -- accounts/
| | -- boards/
| | | -- migrations/
| | | -- templatetags/
| | | -- tests/
| | | | -- __init__.py
| | | | -- test_templatetags.py
| | | | -- test_view_home.py <-- here
| | | | -- test_view_board_topics.py <-- here
| | | +-- test_view_new_topic.py <-- and here
| | | | -- __init__.py
| | | -- admin.py
| | | -- apps.py
| | | -- models.py
| | | -- views.py
| | -- myproject/
| | -- static/
| | -- templates/
| | -- db.sqlite3
| +-- manage.py
+-- venv/
```

重新运行测试，确保一切正常。

现在在 **test\_view\_new\_topic.py** 中添加一个新测试用例，用来检查试图是否被 `@login_required` 保护：

**boards/tests/test\_view\_new\_topic.py** ([完成代码](#))

```
from django.test import TestCase
from django.urls import reverse
from ..models import Board

class LoginRequiredNewTopicTests(TestCase):
 def setUp(self):
 Board.objects.create(name='Django', description='Django board.')
 self.url = reverse('new_topic', kwargs={'pk': 1})
 self.response = self.client.get(self.url)

 def test_redirection(self):
 login_url = reverse('login')
 self.assertRedirects(self.response, '{login_url}?next={url}'.format(login_url=login_url, url=self.url))
```

在测试用例中，我们尝试在没有登录的情况下发送请求给 **new topic** 视图，期待的结果是请求重定向到登录页面。

# Django入门与实践-第18章：访问已登录用户

现在我可以改进 `new_topic` 视图，将发布主题的用户设置当前登录的用户，取代之前直接从数据库查询出来的第一个用户，之前这份代码是临时的，因为那时候还没有方法去获取登录用户，但是现在可以了：

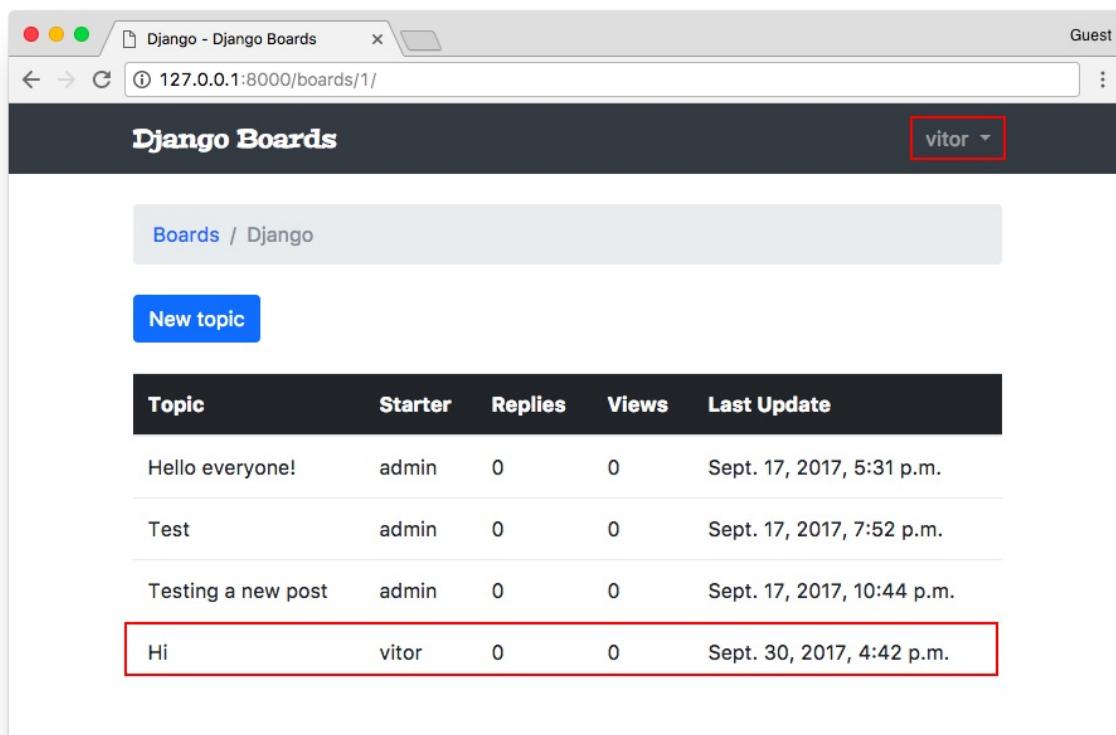
**boards/views.py** ([完整代码](#))

```
from django.contrib.auth.decorators import login_required
from django.shortcuts import get_object_or_404, redirect, render

from .forms import NewTopicForm
from .models import Board, Post

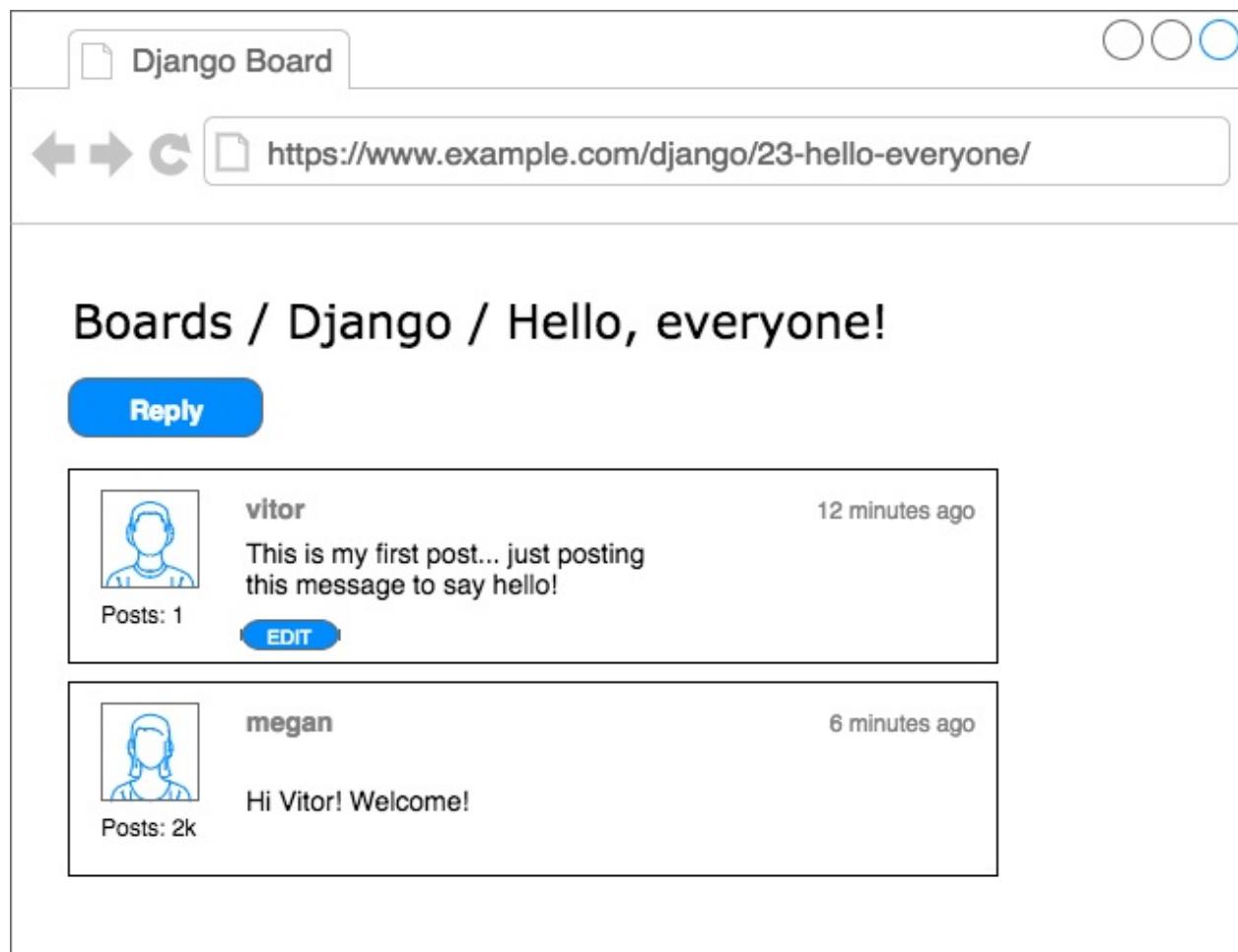
@login_required
def new_topic(request, pk):
 board = get_object_or_404(Board, pk=pk)
 if request.method == 'POST':
 form = NewTopicForm(request.POST)
 if form.is_valid():
 topic = form.save(commit=False)
 topic.board = board
 topic.starter = request.user # <- here
 topic.save()
 Post.objects.create(
 message=form.cleaned_data.get('message'),
 topic=topic,
 created_by=request.user # <- and here
)
 return redirect('board_topics', pk=board.pk) # T
 # TODO: redirect to the created topic page
 else:
 form = NewTopicForm()
 return render(request, 'new_topic.html', {'board': board, 'form': form})
```

我们可以添加一个新的主题快速验证一下：



## 主题回复列表

现在我们花点时间来实现主题的回复列表页面，先来看一下下面的线框图：



首先我们需要写URL路由：

**myproject/urls.py**([完成代码](#))

```
url(r'^boards/(\?P<pk>\d+)/topics/(\?P<topic_pk>\d+)/$', views.topic_posts, name='topic_posts'),
```

有两个关键字参数， `pk` 用于唯一标识版块（Board）， `topic_pk` 用于唯一标识该回复来自哪个主题。

**boards/views.py** ([完整代码](#))

```

from django.shortcuts import get_object_or_404, render
from .models import Topic

def topic_posts(request, pk, topic_pk):
 topic = get_object_or_404(Topic, board__pk=pk, pk=topic_pk)
 return render(request, 'topic_posts.html', {'topic': topic})

```

注意我们正在间接地获取当前的版块，记住，主题（topic）模型是关联版块（Board）模型的，所以我们可以访问当前的版块，你将在下一个代码段中看到：

**templates/topic\_posts.html**([完整代码](#))

```

{% extends 'base.html' %}

{% block title %}{{ topic.subject }}{% endblock %}

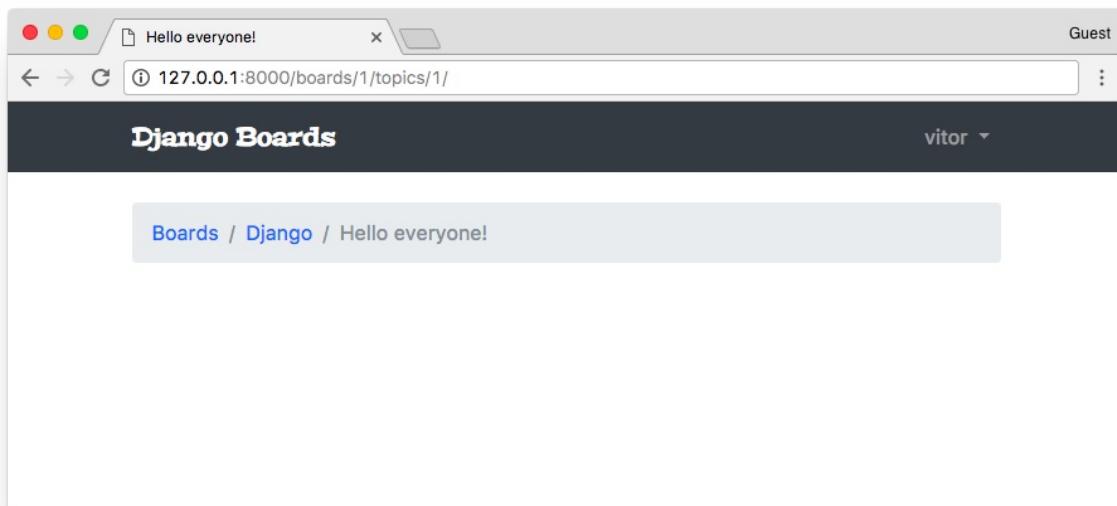
{% block breadcrumb %}
 <li class="breadcrumb-item">Boards
 <li class="breadcrumb-item">{{ topic.board.name }}
 <li class="breadcrumb-item active">{{ topic.subject }}
{% endblock %}

{% block content %}

{% endblock %}

```

现在你会看到我们在模板中 `board.name` 被替换掉了，在导航条，是使用的 `topic` 的属性：`topic.board.name`。



现在我们给**topic\_posts**添加一个新的测试文件：

**boards/tests/test\_view\_topic\_posts.py**

```

from django.contrib.auth.models import User
from django.test import TestCase
from django.urls import resolve, reverse

from ..models import Board, Post, Topic
from ..views import topic_posts

class TopicPostsTests(TestCase):
 def setUp(self):
 board = Board.objects.create(name='Django', description='Django board.')
 user = User.objects.create_user(username='john', email='john@doe.com', password='123')
 topic = Topic.objects.create(subject='Hello, world', board=board, starter=user)
 Post.objects.create(message='Lorem ipsum dolor sit amet', topic=topic, created_by=user)
 url = reverse('topic_posts', kwargs={'pk': board.pk, 'topic_pk': topic.pk})
 self.response = self.client.get(url)

 def test_status_code(self):
 self.assertEqual(self.response.status_code, 200)

 def test_view_function(self):
 view = resolve('/boards/1/topics/1/')
 self.assertEqual(view.func, topic_posts)

```

注意到，`setup`函数变得越来越复杂，我们可以创建一个 `minxin` 或者抽象类来重用这些代码，我们也可以使用第三方库来初始化设置一些测试数据，来减少这些样板代码。

同时，我们已经有了大量的测试用例，运行速度开始逐渐变得慢起来，我们可以通过用测试套件的方式测试指定的app。

```
python manage.py test boards
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
```

```
.....
```

```

```

```
Ran 23 tests in 1.246s
```

```
OK
```

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

我们还可以只运行指定的测试文件

```
python manage.py test boards.tests.test_view_topic_posts
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
```

```
..
```

```

```

```
Ran 2 tests in 0.129s
```

```
OK
```

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

抑或是指定单个测试用例

```
python manage.py test boards.tests.test_view_topic_posts.Topi
cPostsTests.test_status_code
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.

.
Ran 1 test in 0.100s

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

很酷，是不是？

继续前行！

在 `topic_posts.html` 页面中，我们可以创建一个for循环迭代主题下的回复

### **templates/topic\_posts.html**

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

{% load static %}

{% block title %}{{ topic.subject }}{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boar
ds
 <li class="breadcrumb-item"><a href="{% url 'board_topics'
topic.board.pk %}">{{ topic.board.name }}
 <li class="breadcrumb-item active">{{ topic.subject }}
{% endblock %}

{% block content %}

<div class="mb-4">
 Reply
```

```
>
</div>

{% for post in topic.posts.all %}
<div class="card mb-2">
 <div class="card-body p-3">
 <div class="row">
 <div class="col-2">

 <small>Posts: {{ post.created_by.posts.count }}</small>
 </div>
 <div class="col-10">
 <div class="row mb-3">
 <div class="col-6">
 <strong class="text-muted">{{ post.created_by.username }}
 </div>
 <div class="col-6 text-right">
 <small class="text-muted">{{ post.created_at }}</small>
 </div>
 </div>
 {{ post.message }}
 {% if post.created_by == user %}
 <div class="mt-3">
 Edit
 </div>
 {% endif %}
 </div>
 </div>
 </div>
 </div>
 <% endfor %>
 <% endblock %>
```

因为我们现在还没有任何方法去上传用户图片，所以先放一张空的图片，我从 [IconFinder](#) 下载了一张免费图片，然后保存在项目的 static/img 目录。

我们还没有真正探索过 Django 的 ORM，但代码 `post.created_by.posts.count` 在数据库中会执行一个 `select count` 查询。尽管结果是正确的，但不是一个好方法。因为它在数据库中造成了多次不必要的查询。不过现在不用担心，先专注于如何与应用程序进行交互。稍后，我们将改进此代码，以及如何改进那些复杂笨重的查询。（译注：过早优化是万恶之源）

另一个有意思的地方是我们正在测试当前帖子是否属于当前登录用户：`if post.created_by == user`，我们只给帖子的拥有者显示编辑按钮。

因为我们现在要在主题页面添加一个 URL 路由到主题的帖子列表，更新 `topic.html` 模版，加上一个链接：

### templates/topics.html ([完整代码](#))

```
{% for topic in board.topics.all %}
 <tr>
 <td>{
 { topic.subject }}</td>
 <td>{{ topic.starter.username }}</td>
 <td>0</td>
 <td>0</td>
 <td>{{ topic.last_updated }}</td>
 </tr>
{% endfor %}
```

# Django入门与实践-第19章：主题回复

现在让我们来实现回复帖子的功能，以便我们可以添加更多的数据和改进功能实现与单元测试。

The screenshot shows a web browser window titled "Django Board". The URL in the address bar is <https://www.example.com/django/23-hello-everyone/reply/>. The main content area is titled "Boards / Django / Hello, everyone! / Reply". It contains a "Message" input field with the placeholder "Weeeeeelcome!" and a blue "Post a reply" button. Below this, there are two messages in a list:

- megan** 6 minutes ago  
Hi Vitor! Welcome!
- vitor** 12 minutes ago  
This is my first post... just posting this message to say hello!

添加新的URL路由：

**myproject/urls.py**([完整代码](#))

```
url(r'^boards/(\?P<pk>\d+)/topics/(\?P<topic_pk>\d+)/reply/$',
 views.reply_topic, name='reply_topic'),
```

给回帖创建一个新的表单：

**boards/forms.py** ([完整代码](#))

```

from django import forms
from .models import Post

class PostForm(forms.ModelForm):
 class Meta:
 model = Post
 fields = ['message',]

```

一个新的受 `@login_required` 保护的视图，以及简单的表单处理逻辑

### boards/views.py(完整代码)

```

from django.contrib.auth.decorators import login_required
from django.shortcuts import get_object_or_404, redirect, render
from .forms import PostForm
from .models import Topic

@login_required
def reply_topic(request, pk, topic_pk):
 topic = get_object_or_404(Topic, board__pk=pk, pk=topic_pk)
 if request.method == 'POST':
 form = PostForm(request.POST)
 if form.is_valid():
 post = form.save(commit=False)
 post.topic = topic
 post.created_by = request.user
 post.save()
 return redirect('topic_posts', pk=pk, topic_pk=topic_pk)
 else:
 form = PostForm()
 return render(request, 'reply_topic.html', {'topic': topic, 'form': form})

```

现在我们再会到**new\_topic**视图函数，更新重定向地址（标记为 **#TODO** 的地方）

```
@login_required
def new_topic(request, pk):
 board = get_object_or_404(Board, pk=pk)
 if request.method == 'POST':
 form = NewTopicForm(request.POST)
 if form.is_valid():
 topic = form.save(commit=False)
 # code suppressed ...
 return redirect('topic_posts', pk=pk, topic_pk=topic.pk) # <- here
 # code suppressed ...
```

值得注意的是：在视图函数**replay\_topic**中，我们使用 `topic_pk`，因为我们引用的是函数的关键字参数，而在**new\_topic**视图中，我们使用的是 `topic.pk`，因为 `topic` 是一个对象（Topic模型的实例对象），`.pk` 是这个实例对象的一个属性，这两种细微的差别，其实区别很大，别搞混了。

回复页面模版的一个版本：

### **templates/reply\_topic.html**

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

{% load static %}

{% block title %}Post a reply{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boards
 <li class="breadcrumb-item">
```

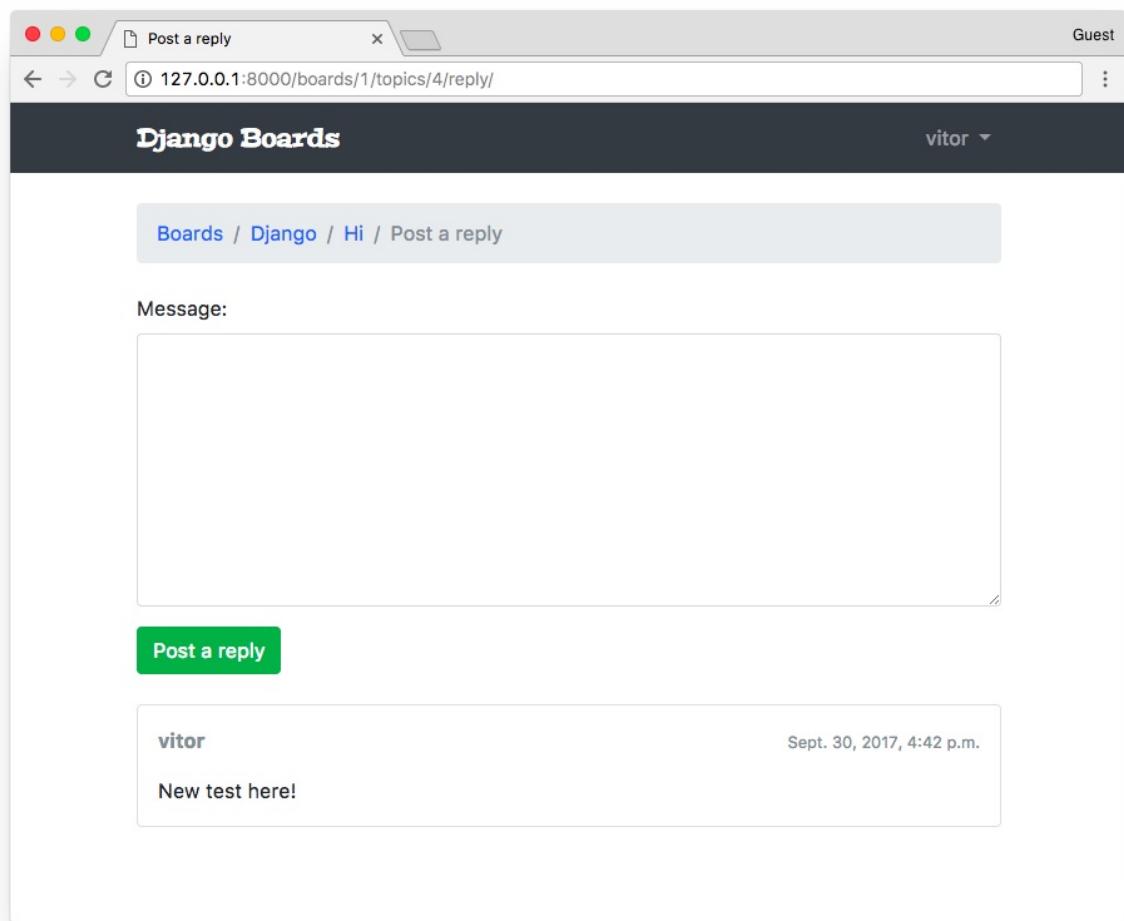
```
topic.board.pk %}">{{ topic.board.name }}
 <li class="breadcrumb-item"><a href="{% url 'topic_posts' t
opic.board.pk topic.pk %}">{{ topic.subject }}
 <li class="breadcrumb-item active">Post a reply
 {% endblock %}

{% block content %}

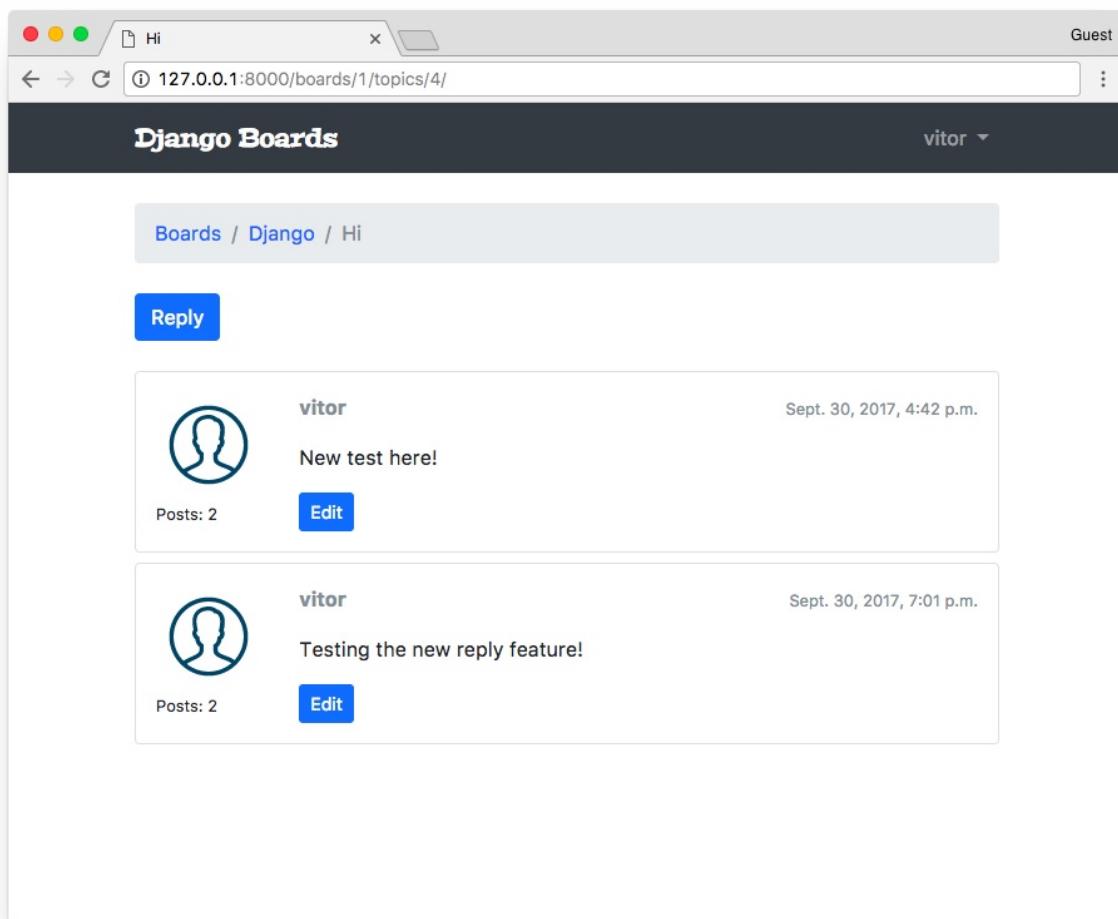
<form method="post" class="mb-4">
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-success">Post a reply
</button>
</form>

{% for post in topic.posts.all %}
 <div class="card mb-2">
 <div class="card-body p-3">
 <div class="row mb-3">
 <div class="col-6">
 <strong class="text-muted">{{ post.created_by.us
ername }}
 </div>
 <div class="col-6 text-right">
 <small class="text-muted">{{ post.created_at }}</
small>
 </div>
 </div>
 {{ post.message }}
 </div>
 </div>
 {% endfor %}

 {% endblock %}
```



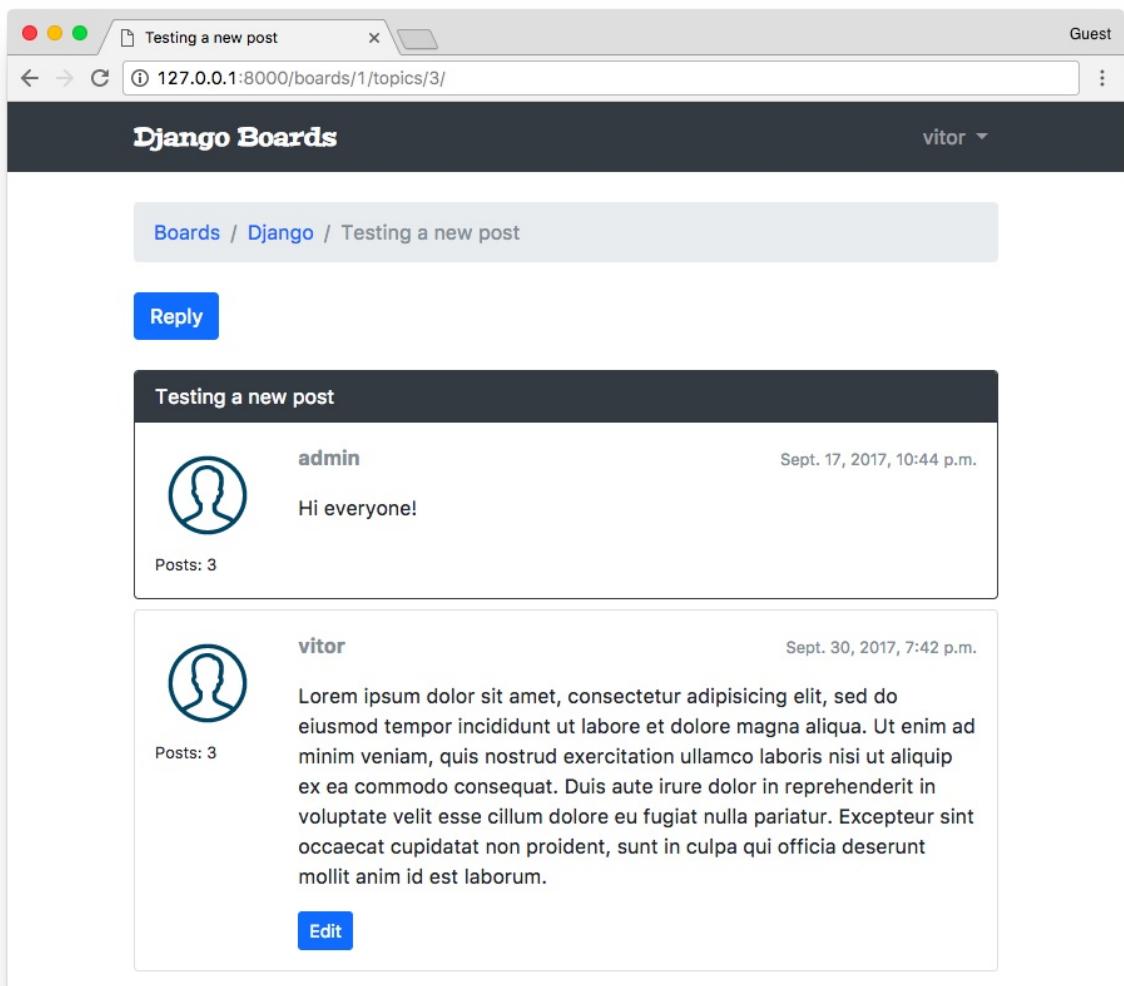
提交回复之后，用户会跳回主题的回复列表：



我们可以改变第一条帖子的样式，使得它在页面上更突出：

**templates/topic\_posts.html**([完整代码](#))

```
{% for post in topic.posts.all %}
 <div class="card mb-2 {% if forloop.first %}border-dark{% endif %}">
 {% if forloop.first %}
 <div class="card-header text-white bg-dark py-2 px-3">{%
 topic.subject %}</div>
 {% endif %}
 <div class="card-body p-3">
 <!-- code suppressed -->
 </div>
 </div>
{% endfor %}
```



现在对于测试，已经实现标准化流程了，就像我们迄今为止所做的一样。在 boards/tests 目录中创建一个新文件 `test_view_reply_topic.py`:

**boards/tests/test\_view\_reply\_topic.py** ([完整代码](#))

```
from django.contrib.auth.models import User
from django.test import TestCase
from django.urls import reverse
from ..models import Board, Post, Topic
from ..views import reply_topic

class ReplyTopicTestCase(TestCase):
 """
 Base test case to be used in all `reply_topic` view tests
 """

 def setUp(self):
 self.board = Board.objects.create(name='Django', description='Django board.')
 self.username = 'john'
 self.password = '123'
 user = User.objects.create_user(username=self.username, email='john@doe.com', password=self.password)
 self.topic = Topic.objects.create(subject='Hello, world', board=self.board, starter=user)
 Post.objects.create(message='Lorem ipsum dolor sit amet', topic=self.topic, created_by=user)
 self.url = reverse('reply_topic', kwargs={'pk': self.board.pk, 'topic_pk': self.topic.pk})

 class LoginRequiredReplyTopicTests(ReplyTopicTestCase):
 # ...

 class ReplyTopicTests(ReplyTopicTestCase):
 # ...

 class SuccessfulReplyTopicTests(ReplyTopicTestCase):
 # ...

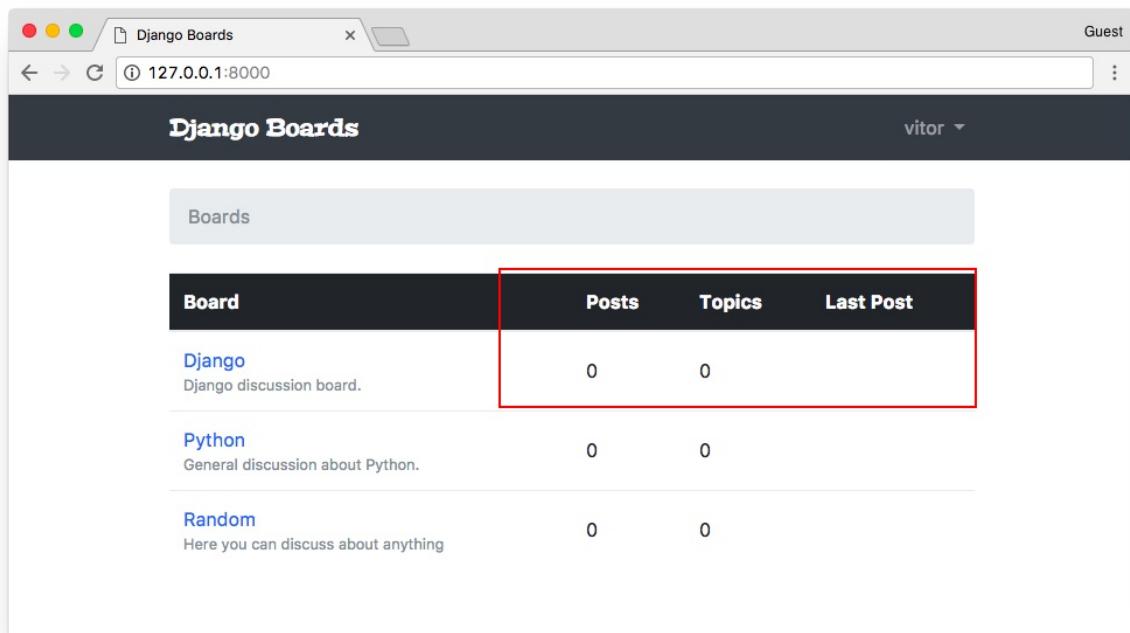
 class InvalidReplyTopicTests(ReplyTopicTestCase):
 # ...
```

这里的精髓在于自定义了测试用例基类**ReplyTopicTestCase**。然后所有四个类将继承这个测试用例。

首先，我们测试视图是否受 `@login_required` 装饰器保护，然后检查HTML输入，状态码。最后，我们测试一个有效和无效的表单提交。

# Django入门与实践-第20章： QuerySets（查询结果集）

现在我们花点时间来探索关于模型的 API。首先，我们来改进主页：



The screenshot shows a web browser window titled "Django Boards" at the URL "127.0.0.1:8000". The page displays a list of three boards: "Django", "Python", and "Random". Each board entry includes a brief description. The columns in the table are "Board", "Posts", "Topics", and "Last Post". The "Posts" column is highlighted with a red border.

Board	Posts	Topics	Last Post
Django Django discussion board.	0	0	
Python General discussion about Python.	0	0	
Random Here you can discuss about anything	0	0	

有3个任务：

- 显示每个板块的总主题数
- 显示每个板块的总回复数
- 显示每个板块的最后发布者和日期

在实现这些功能前，我们先使用Python终端

因为我们要在Python终端尝试，所以，把所有的 models 定义一个  
`__str__` 方法是个好主意

**boards/models.py(完整代码)**

```
from django.db import models
from django.utils.text import Truncator

class Board(models.Model):
 # ...
 def __str__(self):
 return self.name

class Topic(models.Model):
 # ...
 def __str__(self):
 return self.subject

class Post(models.Model):
 # ...
 def __str__(self):
 truncated_message = Truncator(self.message)
 return truncated_message.chars(30)
```

在 Post 模型中，使用了 `Truncator` 工具类，这是将一个长字符串截取为任意长度字符的简便方法（这里我们使用30个字符）

现在打开 Python shell

```
python manage.py shell
from boards.models import Board

First get a board instance from the database
board = Board.objects.get(name='Django')
```

这三个任务中最简单的一个就是获取当前版块的总主题数，因为 Topic 和 Board 是直接关联的。

```
board.topics.all()
<QuerySet [<Topic: Hello everyone!>, <Topic: Test>, <Topic: T
esting a new post>, <Topic: Hi>]>

board.topics.count()
4
```

就这样子。

现在统计一个版块下面的回复数量有点麻烦，因为回复并没有和 Board 直接关联

```
from boards.models import Post

Post.objects.all()
<QuerySet [<Post: This is my first topic.. :-)>, <Post: test.
>, <Post: Hi everyone!>,
 <Post: New test here!>, <Post: Testing the new reply featur
e!>, <Post: Lorem ipsum dolor sit amet,...>,
 <Post: hi there>, <Post: test>, <Post: Testing..>, <Post: s
ome reply>, <Post: Random random.>
]>

Post.objects.count()
11
```

这里一共11个回复，但是它并不全部属于 "Django" 这个版块的。

我们可以这样来过滤

```
from boards.models import Board, Post

board = Board.objects.get(name='Django')

Post.objects.filter(topic__board=board)
<QuerySet [<Post: This is my first topic.. :-)>, <Post: test.
>, <Post: hi there>,
<Post: Hi everyone!>, <Post: Lorem ipsum dolor sit amet, ...
>, <Post: New test here!>,
<Post: Testing the new reply feature!>
]>

Post.objects.filter(topic__board=board).count()
7
```

双下划线的 `topic__board` 用于通过模型关系来定位，在内部，Django 在 Board-Topic-Post之间构建了桥梁，构建SQL查询来获取属于指定版块下面的帖子回复。

最后一个任务是标识版块下面的最后一条回复

```
order by the `created_at` field, getting the most recent first
Post.objects.filter(topic__board=board).order_by('-created_at')
<QuerySet [<Post: testing>, <Post: new post>, <Post: hi there
>, <Post: Lorem ipsum dolor sit amet,...>,
<Post: Testing the new reply feature!>, <Post: New test here!>,
<Post: Hi everyone!>,
<Post: test.>, <Post: This is my first topic.. :-)>
]>

we can use the `first()` method to just grab the result that interest us
Post.objects.filter(topic__board=board).order_by('-created_at').first()
<Post: testing>
```

太棒了，现在我们来实现它

### boards/models.py ([完整代码](#))

```
from django.db import models

class Board(models.Model):
 name = models.CharField(max_length=30, unique=True)
 description = models.CharField(max_length=100)

 def __str__(self):
 return self.name

 def get_posts_count(self):
 return Post.objects.filter(topic__board=self).count()

 def get_last_post(self):
 return Post.objects.filter(topic__board=self).order_by(
 '-created_at').first()
```

注意，我们使用的是 `self`，因为这是Board的一个实例方法，所以我们就用这个Board实例来过滤这个 QuerySet

现在我们可以改进主页的HTML模板来显示这些新的信息

### templates/home.html

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

{% block breadcrumb %}
 <li class="breadcrumb-item active">Boards
{% endblock %}

{% block content %}
 <table class="table">
 <thead class="thead-inverse">
 <tr>
 <th>Board</th>
 <th>Posts</th>
 <th>Topics</th>
 <th>Last Post</th>
 </tr>
 </thead>
 <tbody>
 {% for board in boards %}
 <tr>
 <td>
 {{ board.name }}
 <small class="text-muted d-block">{{ board.description }}</small>
 </td>
 <td class="align-middle">
 {{ board.get_posts_count }}
 </td>
 <td class="align-middle">
 {{ board.topics.count }}
 </td>
 </tr>
 {% endfor %}
 </tbody>
 </table>
{% endblock %}
```

```

</td>
<td class="align-middle">
 {% with post=board.get_last_post %}
 <small>

 By {{ post.created_by.username }} at {{ post.created_at }}

 </small>
 {% endwith %}
</td>
</tr>
{% endfor %}
</tbody>
</table>
{% endblock %}

```

现在是这样的效果

Board	Posts	Topics	Last Post
Django Django discussion board.	9	4	By vitor at Oct. 1, 2017, 11:52 a.m.
Python General discussion about Python.	3	2	By vitor at Oct. 1, 2017, 11:47 a.m.
Random Here you can discuss about anything	1	1	By vitor at Oct. 1, 2017, 11:47 a.m.

运行测试：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
=====
=====
ERROR: test_home_url_resolves_home_view (boards.tests.test_view_home.HomeTests)

django.urls.exceptions.NoReverseMatch: Reverse for 'topic_posts' with arguments '(1, '')' not found. 1 pattern(s) tried: ['boards/(?P<pk>\d+)/topics/(?P<topic_pk>\d+)/$']
=====
=====
ERROR: test_home_view_contains_link_to_topics_page (boards.tests.test_view_home.HomeTests)

django.urls.exceptions.NoReverseMatch: Reverse for 'topic_posts' with arguments '(1, '')' not found. 1 pattern(s) tried: ['boards/(?P<pk>\d+)/topics/(?P<topic_pk>\d+)/$']
=====
=====
ERROR: test_home_view_status_code (boards.tests.test_view_home.HomeTests)

django.urls.exceptions.NoReverseMatch: Reverse for 'topic_posts' with arguments '(1, '')' not found. 1 pattern(s) tried: ['boards/(?P<pk>\d+)/topics/(?P<topic_pk>\d+)/$']
```

```


Ran 80 tests in 5.663s

FAILED (errors=3)
Destroying test database for alias 'default'...
```

看起来好像有问题，如果没有回复的时候程序会崩溃

### templates/home.html

```
{% with post=board.get_last_post %}
 {% if post %}
 <small>

 By {{ post.created_by.username }} at {{ post.created_at }}

 </small>
 {% else %}
 <small class="text-muted">
 No posts yet.
 </small>
 {% endif %}
{% endwith %}
```

再次运行测试：

```
python manage.py test
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
```

```
.....
.....

Ran 80 tests in 5.630s
```

```
OK
```

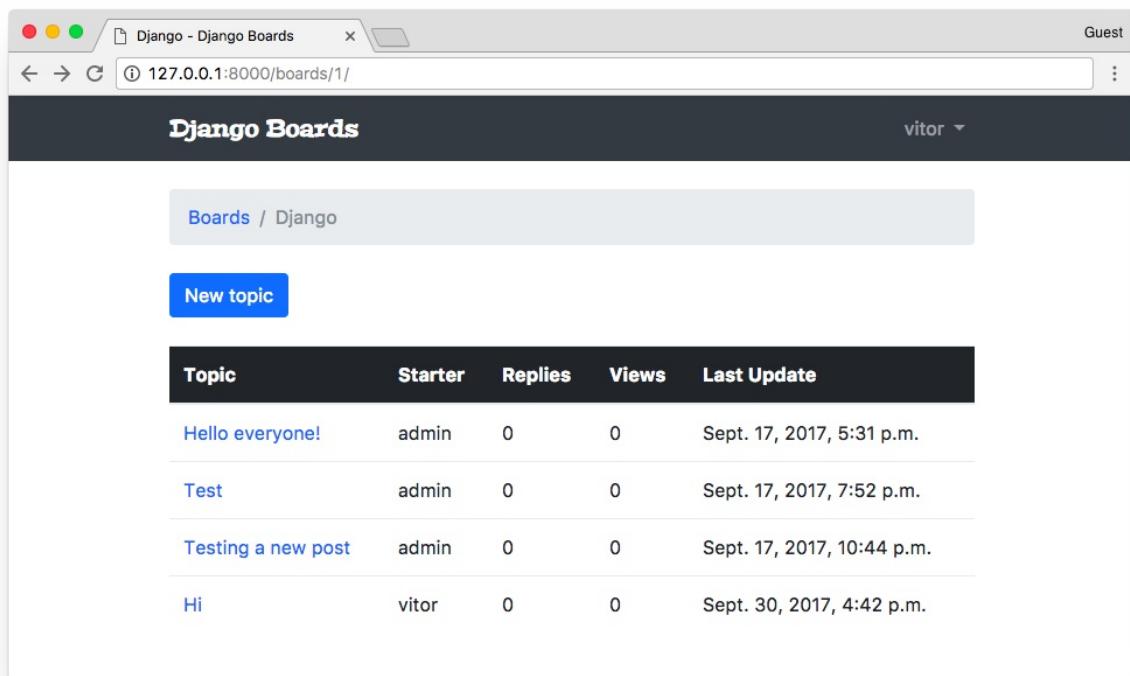
```
Destroying test database for alias 'default'...
```

我添加一个没有任何消息的版块，用于检查这个"空消息"

The screenshot shows a web browser window titled "Django Boards". The URL in the address bar is "127.0.0.1:8000". The top navigation bar includes "Guest", "admin ▾", and a "Boards" link. The main content area displays a table of boards:

Board	Posts	Topics	Last Post
Django Django discussion board.	9	4	By vitor at Oct. 1, 2017, 11:52 a.m.
Python General discussion about Python.	3	2	By vitor at Oct. 1, 2017, 11:47 a.m.
Random Here you can discuss about anything	1	1	By vitor at Oct. 1, 2017, 11:47 a.m.
New test board Empty board just for testing.	0	0	No posts yet.

现在是时候来改进回复列表页面了。



现在，我将告诉你另外一种方法来统计回复的数量，用一种更高效的方式和之前一样，首先在Python shell 中尝试

```
python manage.py shell
```

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

board = Board.objects.get(name='Django')

topics = board.topics.order_by('-last_updated').annotate(replies=Count('posts'))

for topic in topics:
 print(topic.replies)

2
4
2
1
```

这里我们使用 `annotate`，`QuerySet` 将即时生成一个新的列，这个新的列，将被翻译成一个属性，可通过 `topic.replies` 来访问，它包含了指定主题下的回复数。

我们来做一个小小的修复，因为回复里面不应该包括发起者的帖子

```
topics = board.topics.order_by('-last_updated').annotate(replies=Count('posts') - 1)

for topic in topics:
 print(topic.replies)

1
3
1
0
```

很酷，对不对？

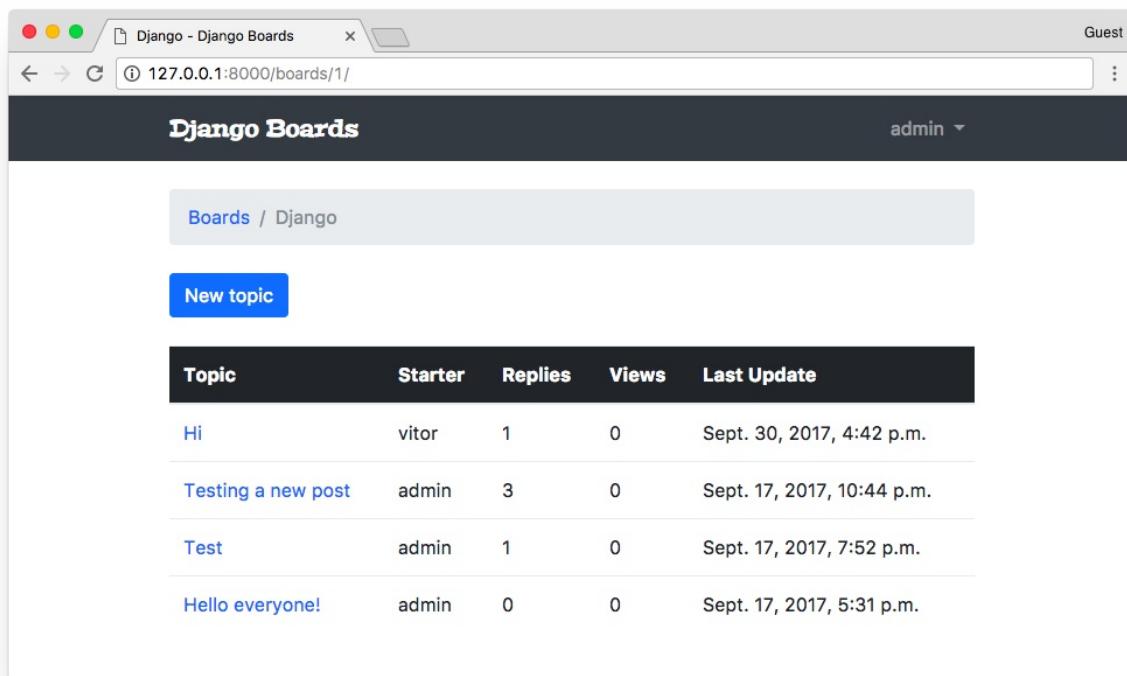
**boards/views.py** ([完整代码](#))

```
from django.db.models import Count
from django.shortcuts import get_object_or_404, render
from .models import Board

def board_topics(request, pk):
 board = get_object_or_404(Board, pk=pk)
 topics = board.topics.order_by('-last_updated').annotate(
 replies=Count('posts') - 1)
 return render(request, 'topics.html', {'board': board, 'topics': topics})
```

### templates/topics.html(完整代码)

```
{% for topic in topics %}
 <tr>
 <td>{%
 topic.subject %}</td>
 <td>{{ topic.starter.username }}</td>
 <td>{{ topic.replies }}</td>
 <td>0</td>
 <td>{{ topic.last_updated }}</td>
 </tr>
{% endfor %}
```



下一步是修复主题的查看次数，但是，现在我们需要添加一个新的字段

# Django入门与实践-第21章：迁移

迁移（Migration）是Django做Web开发的基本组成部分，它使得我们在演进应用的models时，它能使得models文件与数据库保持同步

当我们第一次运行命令 `python manage.py migrate` 的时候，Django会抓取所有迁移文件然后生成数据库 schema。

当Django应用了迁移之后，有一个特殊的表叫做`django_migrations`，在这个表中，Django注册了所有已经的迁移记录。

所以，如果我们重新运行命令：

```
python manage.py migrate
```

```
Operations to perform:
 Apply all migrations: admin, auth, boards, contenttypes, sessions
Running migrations:
 No migrations to apply.
```

Django 知道没什么事可做了。

现在我们添加在 Topic 模型中添加一个新的字段：

**boards/models.py**([完整代码](#))

```

class Topic(models.Model):
 subject = models.CharField(max_length=255)
 last_updated = models.DateTimeField(auto_now_add=True)
 board = models.ForeignKey(Board, related_name='topics')
 starter = models.ForeignKey(User, related_name='topics')
 views = models.PositiveIntegerField(default=0) # <- here

 def __str__(self):
 return self.subject

```

我们添加了一个 `PositiveIntegerField`，因为这个字段将要存储的是页面的浏览量，不可能是一个负数

在我们可以使用这个新字段前，我们必须更新数据库schema，执行命令 `makemigrations`

```

python manage.py makemigrations

Migrations for 'boards':
 boards/migrations/0003_topic_views.py
 - Add field views to topic

```

`makemigrations` 会自动生成**0003\_topic\_views.py**文件，将用于修改数据库（添加一个views字段）

现在运行命令 `migrate` 来应用迁移

```

python manage.py migrate

Operations to perform:
 Apply all migrations: admin, auth, boards, contenttypes, sessions
Running migrations:
 Applying boards.0003_topic_views... OK

```

现在我们可以用它来追踪指定主题被阅读了多少次

### boards/views.py (完整代码)

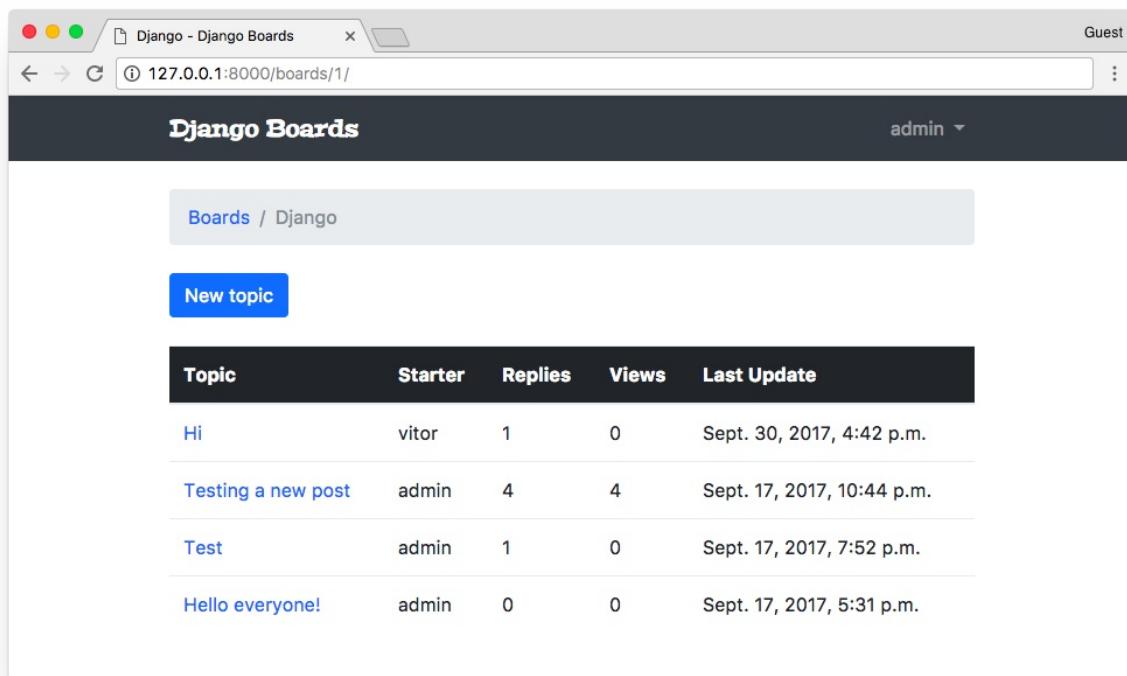
```
from django.shortcuts import get_object_or_404, render
from .models import Topic

def topic_posts(request, pk, topic_pk):
 topic = get_object_or_404(Topic, board__pk=pk, pk=topic_pk)
 topic.views += 1
 topic.save()
 return render(request, 'topic_posts.html', {'topic': topic})
```

### templates/topics.html(完整代码)

```
{% for topic in topics %}
<tr>
 <td>{ topic.subject }</td>
 <td>{{ topic.starter.username }}</td>
 <td>{{ topic.replies }}</td>
 <td>{{ topic.views }}</td> <!-- here -->
 <td>{{ topic.last_updated }}</td>
</tr>
{% endfor %}
```

现在打开一个主题，刷新页面几次，然后你会看到有页面阅读次数统计了。



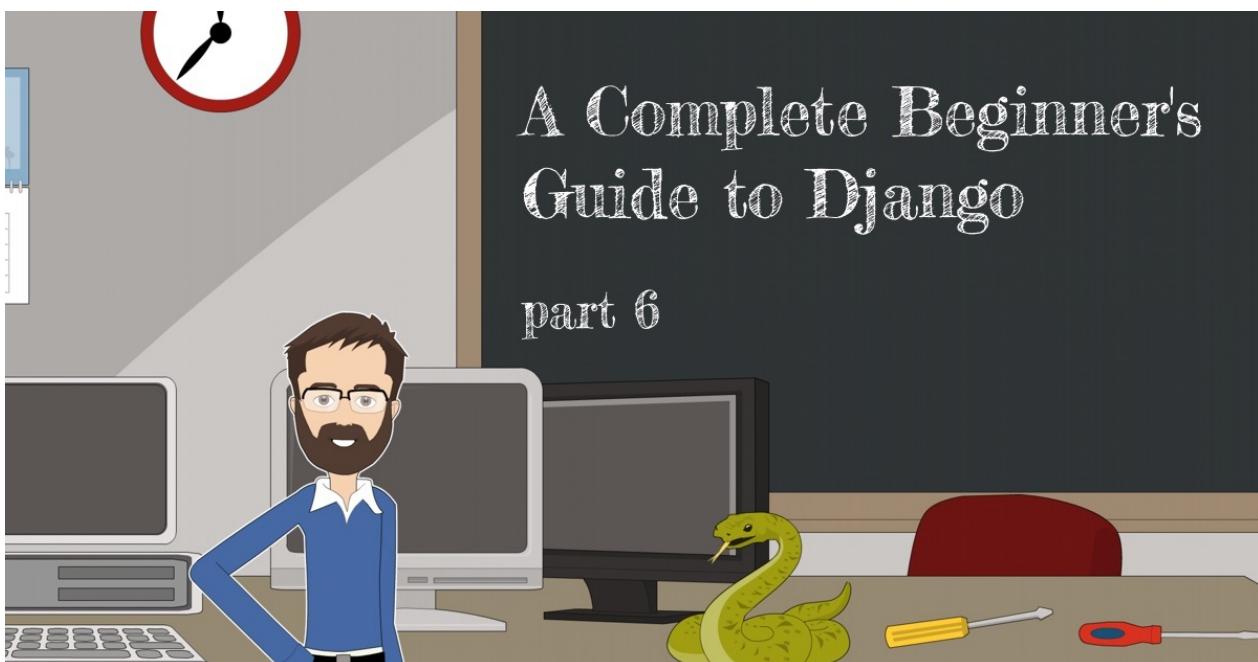
## 总结

在这节课中，我们在留言板的基础功能上取得了一些进步，还剩下一些东西等待去实现，比如：编辑帖子、我的账户（更改个人信息）等等。之后我们将提供markdown语法和列表的分页功能。

下一节主要使用基于类的视图来解决这些问题，在之后，我们将学习到如何部署应用程序到Web服务器中去。

这部分的完整代码可以访问：<https://github.com/sibtc/django-beginners-guide/tree/v0.5-lw>

# Django入门与实践-第22章：基于类的视图



## 前言

欢迎来到系列教程的第六部分！在这篇教程中，我们将详细探讨基于类的视图（简称CBV）。我们也将重构一些现有的视图，以便利用内置的**基于类的通用视图**（Generic Class-Based Views）。

这篇教程我们还将讨论许多其他主题，例如如何使用分页，如何使用markdown以及如何添加简单的编辑器。我们还将探索一个名为 **Humanize** 的内置软件包，用于对数据进行“人为操作”。

好了，伙计们！让我们来实现一些代码。今天我们还有很多工作要做！

## 视图策略

到头来，所有的Django视图其实都是函数。即便是CBV。在类的底层，它完成了所有的功能并最终返回一个视图函数。

引入了基于类的视图，使开发人员可以更轻松地重新使用和扩展视图。使用它们有很多好处，例如可扩展性，多重继承之类的面向对象技术的运用，HTTP 方法的处理是在单独的方法中完成的，而不是使用条件分支，并且还有通用的基于类的视图（简称GCBV）。

在我们继续教程之前，让我们清楚这三个术语的含义：

- 基于函数的视图 (FBV)
- 基于类的视图 (CBV)
- 基于类的通用视图 (GCBV)

FBV是Django视图中最简单的表示：它就是一个接收 **HttpRequest** 对象并返回一个 **HttpResponse** 的函数。

CBV是每个Django视图都被定义为一个扩展 `django.views.generic.view` 抽象类后的 Python 类。

GCBV是可以解决一些具体问题的内置的CBV集合，比如列表视图，创建，更新和删除视图等。

下面我们将探讨一些不同实现方式的例子。

## 基于函数的视图 (FBV)

### views.py

```
def new_post(request):
 if request.method == 'POST':
 form = PostForm(request.POST)
 if form.is_valid():
 form.save()
 return redirect('post_list')
 else:
 form = PostForm()
 return render(request, 'new_post.html', {'form': form})
```

### urls.py

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

## 基于类的视图 (CBV)

CBV是**View**类的集成类。这里的主要区别在于请求是以HTTP方法命名的类方法内处理的，例如**GET**，**POST**，**PUT**，**HEAD**等。

所以，在这里，我们不需要做一个条件来判断请求是一个**POST**还是它是一个**GET**。代码会直接跳转到正确的方法中。在**View**类中内部处理了这个逻辑。

### views.py

```
from django.views.generic import View

class NewPostView(View):
 def post(self, request):
 form = PostForm(request.POST)
 if form.is_valid():
 form.save()
 return redirect('post_list')
 return render(request, 'new_post.html', {'form': form})

 def get(self, request):
 form = PostForm()
 return render(request, 'new_post.html', {'form': form})
```

我们在**urls.py**模块中引用CBV的方式也会有点不同：

### urls.py

```
urlpatterns = [
 url(r'^new_post/$', views.NewPostView.as_view(), name='new_post'),
]
```

在这里，我们需要调用 `as_view()` 这个类方法，它将返回一个符合url模式视图函数。在某些情况下，我们也可以将一些关键字参数传递给 `as_view()`，以便自定义CBV的行为，就像我们用一些身份验证视图来定制模板一样。

总之，关于CBV的好处是我们可以在类里面添加更多的方法，也许可以这样做：

```
from django.views.generic import View

class NewPostView(View):
 def render(self, request):
 return render(request, 'new_post.html', {'form': self.form})

 def post(self, request):
 self.form = PostForm(request.POST)
 if self.form.is_valid():
 self.form.save()
 return redirect('post_list')
 return self.render(request)

 def get(self, request):
 self.form = PostForm()
 return self.render(request)
```

还可以创建一些通用视图来完成一些任务，以便我们可以在整个项目中重复利用它。

你需要知道的关于CBV的基本就是这些。就这么简单。

## 基于类的通过视图（GCBV）

接下来关于GCBV。这是一个不同的情况。正如我前面提到的，GCBV是内置CBV的常见用例。它们的实现往往使用多重继承（混合继承）和其他面向对象的策略。

它们非常灵活，可以节省很多的工作量。但是一开始可能相对比较难上手。

当我第一次开始使用Django时，我发现GCBV很难使用。起初，很难说出发牛了什么，因为代码执行流程看起来并不明显，因为在父类中隐藏了大量代码。官方文档也有一定的难度，主要是因为属性和方法有时分布在八个父类中。使用GCBV时，最好打开 [ccbv.co.uk](http://ccbv.co.uk) (需科学上网) 以便快速参考。不用担心，我们将一起探索它。

现在我们来看一个GCBV的例子。

### views.py

```
from django.views.generic import CreateView

class NewPostView(CreateView):
 model = Post
 form_class = PostForm
 success_url = reverse_lazy('post_list')
 template_name = 'new_post.html'
```

这里我们使用了一个用于创建模型对象的通用视图。它会处理所有表单处理并在表单有效时保存对象。

因为它是一个CBV，所以我们在 **urls.py** 中以与其他CBV相同的方式来引用它：

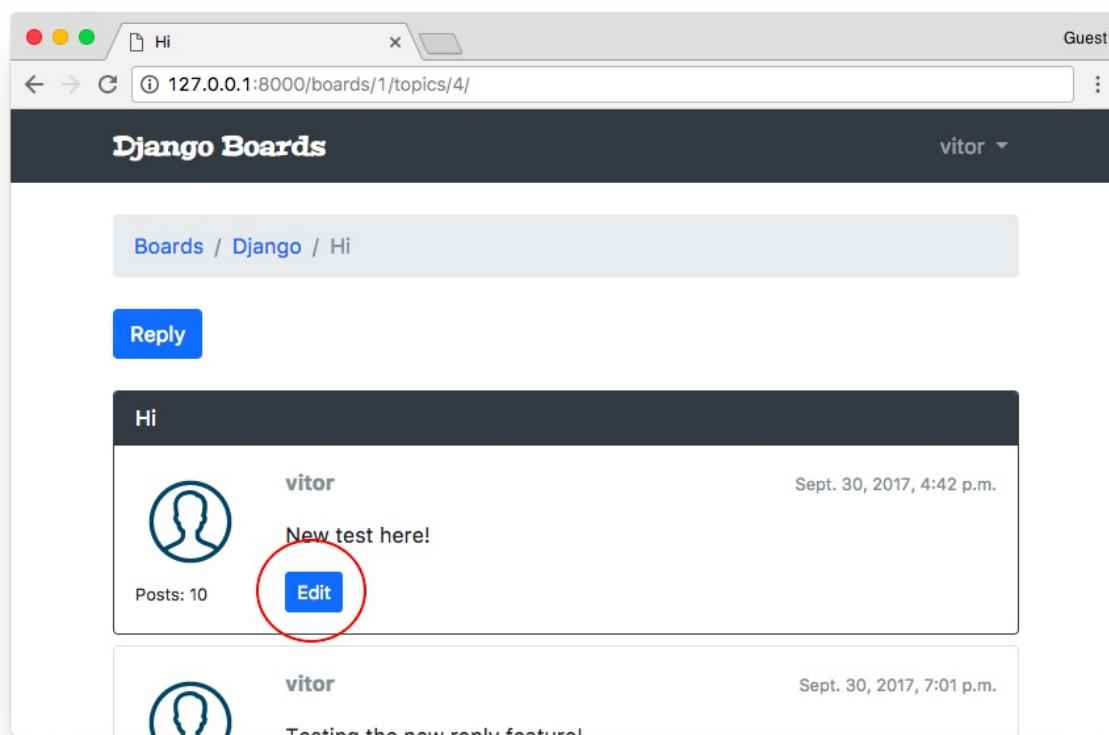
### urls.py

```
urlpatterns = [
 url(r'^new_post/$', views.NewPostView.as_view(), name='new_post'),
]
```

GCBV 中的其他例子还有：**detailview**, **deleteview**, **formview**, **updateview**, **listview**。

## 更新视图

让我们回到我们项目的实现。这次我们将使用 GCBV 来实现编辑帖子的视图：



[boards/views.py](#) (查看文件全部内容)

```

from django.shortcuts import redirect
from django.views.generic import UpdateView
from django.utils import timezone

class PostUpdateView(UpdateView):
 model = Post
 fields = ('message',)
 template_name = 'edit_post.html'
 pk_url_kwarg = 'post_pk'
 context_object_name = 'post'

 def form_valid(self, form):
 post = form.save(commit=False)
 post.updated_by = self.request.user
 post.updated_at = timezone.now()
 post.save()
 return redirect('topic_posts', pk=post.topic.board.pk,
, topic_pk=post.topic.pk)

```

使用 **UPDATEVIEW** 和 **CREATEVIEW**，我们可以选择定义 **form\_class** 或 **fields** 属性。在上面的例子中，我们使用 **fields** 属性来即时创建模型表单。在内部，Django 将使用模型表单工厂函数来组成 **POST** 模型的一种形式。因为它只是一个非常简单的表单，只有 **message** 字段，所以在这里我们可以这么做。但对于复杂的表单定义，最好从外部定义模型表单然后在这里引用它。

系统将使用 **pk\_url\_kwarg** 来标识用于检索 **Post** 对象的关键字参数的名称。就像我们在 **urls.py** 中定义一样。

如果我们没有设置 **context\_object\_name** 属性，**Post** 对象将作为“Object”在模板中可用。所以，在这里我们使用 **context\_object\_name** 来重命名它来发布。你会看到我们在下面的模板中如何使用它。

在这个特定的例子中，我们不得不重写 **form\_valid()** 方法来设置一些额外的字段，例如**updated\_by** 和 **updated\_at**。你可以在这里看到 **form\_valid()** 方法原本的样子：[updateview # form\\_valid](#)。

**myproject/urls.py** (查看文件全部内容)

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

urlpatterns = [
 # ...
 url(r'^boards/(?P<pk>\d+)/topics/(?P<topic_pk>\d+)/posts/
 (?P<post_pk>\d+)/edit/$',
 views.PostUpdateView.as_view(), name='edit_post'),
]
```

现在我们可以将链接添加到编辑页面：

**templates/topic\_posts.html** (查看文件全部内容)

```
{% if post.created_by == user %}
<div class="mt-3">
 <a href="{% url 'edit_post' post.topic.board.pk post.topi
c.pk post.pk %}"
 class="btn btn-primary btn-sm"
 role="button">Edit
 </div>
{% endif %}
```

**templates/edit\_post.html** (查看文件全部内容)

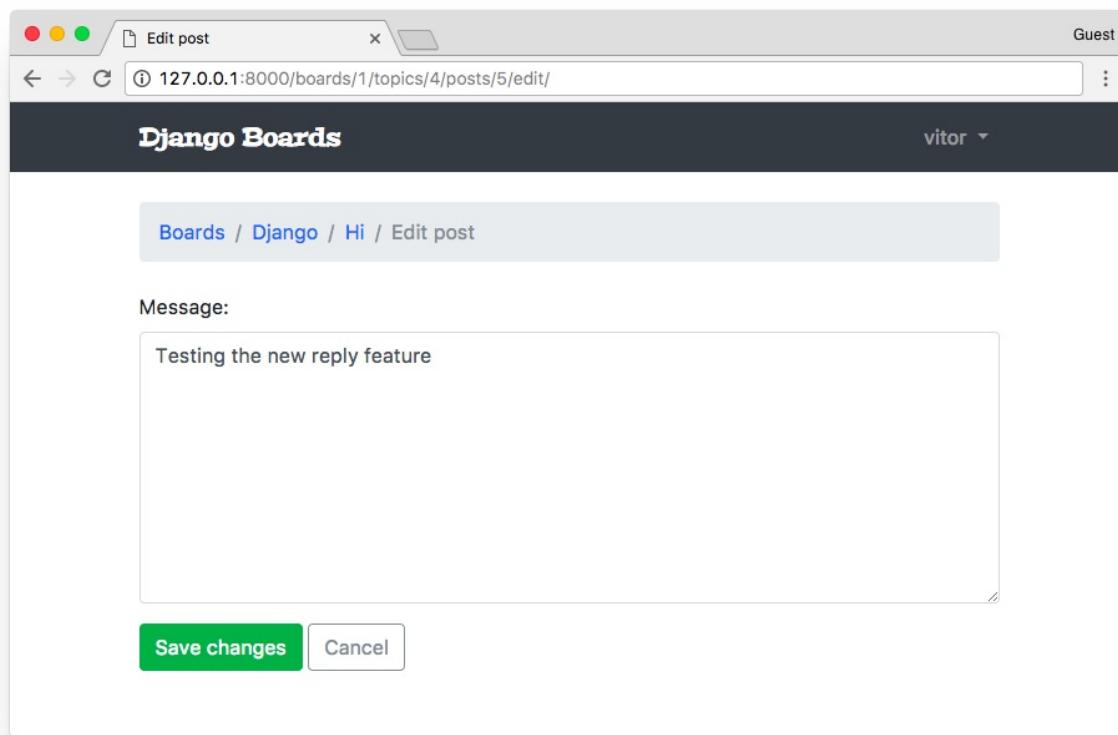
```
{% extends 'base.html' %}

{% block title %}Edit post{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item">Boar
ds
 <li class="breadcrumb-item"><a href="{% url 'board_topics'
post.topic.board.pk %}">{{ post.topic.board.name }}
 <li class="breadcrumb-item"><a href="{% url 'topic_posts'
post.topic.board.pk post.topic.pk %}">{{ post.topic.subject }}<
/a>
 <li class="breadcrumb-item active">Edit post
{% endblock %}

{% block content %}
 <form method="post" class="mb-4" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-success">Save changes
 </button>
 <a href="{% url 'topic_posts' post.topic.board.pk post.to
pic.pk %}" class="btn btn-outline-secondary" role="button">Ca
ncel
 </form>
{% endblock %}
```

现在观察我们如何引导到 post 对象：post.topic.board.pk。如果我们没有设置**context\_object\_name** 来发布，它将可以被用作：object.topic.board.pk。明白了吗？



## 测试更新视图

在 `boards/tests` 文件夹内创建一个名为 `test_view_edit_post.py` 的新测试文件。点击下面的链接，你会看到很多常用测试，就像我们在本教程中做的一样。我会在这里重点介绍一下新的内容：

**boards/tests/test\_view\_edit\_post.py** ([查看完整文件](#))

```
from django.contrib.auth.models import User
from django.test import TestCase
from django.urls import reverse
from ..models import Board, Post, Topic
from ..views import PostUpdateView

class PostUpdateViewTestCase(TestCase):
 ...
 # Base test case to be used in all `PostUpdateView` view tests
 ...

 def test_update_post(self):
 """Test updating a post via POST request to the PostUpdateView"""
 # Create a user and log them in
 user = User.objects.create_user(username='testuser', password='123')
 self.client.login(username='testuser', password='123')

 # Create a board, topic, and post
 board = Board.objects.create(name='Test Board', description='Test Description')
 topic = Topic.objects.create(board=board, subject='Test Topic')
 post = Post.objects.create(topic=topic, message='Initial message')

 # Create a new message for the post
 new_message = 'Updated message'

 # Make a POST request to update the post
 response = self.client.post(reverse('post_update', args=[post.id]), {'message': new_message})
```

```

def setUp(self):
 self.board = Board.objects.create(name='Django', description='Django board.')
 self.username = 'john'
 self.password = '123'
 user = User.objects.create_user(username=self.username, email='john@doe.com', password=self.password)
 self.topic = Topic.objects.create(subject='Hello, world', board=self.board, starter=user)
 self.post = Post.objects.create(message='Lorem ipsum dolor sit amet', topic=self.topic, created_by=user)
 self.url = reverse('edit_post', kwargs={
 'pk': self.board.pk,
 'topic_pk': self.topic.pk,
 'post_pk': self.post.pk
 })

class LoginRequiredPostUpdateViewTests(PostUpdateViewTestCase):
 def test_redirection(self):
 """
 Test if only logged in users can edit the posts
 """
 login_url = reverse('login')
 response = self.client.get(self.url)
 self.assertRedirects(response, '{login_url}?next={url}'.format(login_url=login_url, url=self.url))

class UnauthorizedPostUpdateViewTests(PostUpdateViewTestCase):
 def setUp(self):
 """
 Create a new user different from the one who posted
 """
 super().setUp()
 username = 'jane'
 password = '321'
 user = User.objects.create_user(username=username, em

```

```
 ail='jane@doe.com', password=password)
 self.client.login(username=username, password=password)
 self.response = self.client.get(self.url)

 def test_status_code(self):
 """
 A topic should be edited only by the owner.
 Unauthorized users should get a 404 response (Page Not Found)
 """
 self.assertEqual(self.response.status_code, 404)

class PostUpdateViewTests(PostUpdateViewTestCase):
 # ...

class SuccessfulPostUpdateViewTests(PostUpdateViewTestCase):
 # ...

class InvalidPostUpdateViewTests(PostUpdateViewTestCase):
 # ...
```

这里，重要的部分是：**PostUpdateViewTestCase**是我们定义的类，它可以在其他测试用例中复用。它只包含基本的设置，创建user, topic, boards等等。

**LoginRequiredPostUpdateViewTests** 这个类将测试检查该视图是否使用了 `@login_required` 装饰器。即只有经过身份验证的用户才能访问编辑页面。

**UnauthorizedPostUpdateViewTests** 这个类是创建一个新用户，与发布并尝试访问编辑页面的用户不同。本应用程序应该只能授权该文章的所有者进行编辑。

我们来运行一下这些测试：

```
python manage.py test boards.tests.test_view_edit_post
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...F.....F
=====
=====
FAIL: test_redirection (boards.tests.test_view_edit_post.LoginRequiredPostUpdateViewTests)

...
AssertionError: 200 != 302 : Response didn't redirect as expected: Response code was 200 (expected 302)

=====
=====
FAIL: test_status_code (boards.tests.test_view_edit_post.UnauthorizedPostUpdateViewTests)

...
AssertionError: 200 != 404

Ran 11 tests in 1.360s

FAILED (failures=2)
Destroying test database for alias 'default'...
```

首先，我们修复 `@login_required` 装饰器的问题。在 CBV 上使用视图装饰器的方式有一些不同。我们需要额外的导入：

**boards/views.py** ([查看文章文件](#))

```
from django.contrib.auth.decorators import login_required
from django.shortcuts import redirect
from django.views.generic import UpdateView
from django.utils import timezone
from django.utils.decorators import method_decorator
from .models import Post

@method_decorator(login_required, name='dispatch')
class PostUpdateView(UpdateView):
 model = Post
 fields = ('message',)
 template_name = 'edit_post.html'
 pk_url_kwarg = 'post_pk'
 context_object_name = 'post'

 def form_valid(self, form):
 post = form.save(commit=False)
 post.updated_by = self.request.user
 post.updated_at = timezone.now()
 post.save()
 return redirect('topic_posts', pk=post.topic.board.pk,
 topic_pk=post.topic.pk)
```

我们不能用 `@login_required` 装饰器直接装饰类。我们必须使用一个工具 `@method_decorator`，并传递一个装饰器（或一个装饰器列表）并告诉应该装饰哪个类。在 CBV 中，**装饰调度类**是很常见的。它是一个 Django 内部使用的方法（在 **View** 类中定义）。所有的请求都会经过这个类，所以装饰它会相对安全。

再次运行一下测试：

```
python manage.py test boards.tests.test_view_edit_post
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....F
=====
=====
FAIL: test_status_code (boards.tests.test_view_edit_post.UnauthorizedPostUpdateViewTests)

...
AssertionError: 200 != 404

Ran 11 tests in 1.353s

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

好的，我们解决了 `@login_required` 的问题，现在我们必须处理其他用户可以编辑所有帖子的问题。

解决这个问题最简单的方法是重写**UpdateView**的 `get_queryset` 方法。你可以在这里看到原始方法的源码--- [UpdateView#get\\_queryset.](#)。

**boards/views.py** [查看完整文件](#)

```
@method_decorator(login_required, name='dispatch')
class PostUpdateView(UpdateView):
 model = Post
 fields = ('message',)
 template_name = 'edit_post.html'
 pk_url_kwarg = 'post_pk'
 context_object_name = 'post'

 def get_queryset(self):
 queryset = super().get_queryset()
 return queryset.filter(created_by=self.request.user)

 def form_valid(self, form):
 post = form.save(commit=False)
 post.updated_by = self.request.user
 post.updated_at = timezone.now()
 post.save()
 return redirect('topic_posts', pk=post.topic.board.pk
, topic_pk=post.topic.pk)
```

通过这一行 `queryset = super().get_queryset()`，我们实现了重用父类，即，**UpdateView** 类的 `get_queryset` 方法。然后，我们通过给 `queryset` 添加一个额外的过滤条件，该过滤条件是通过请求中获取登录的用户来过滤内容。

再次测试：

```
python manage.py test boards.tests.test_view_edit_post
```

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.....

Ran 11 tests in 1.321s

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

一切顺利！

## 列表视图

我们现在可以利用 **CBV** 的功能来重构一些现有的视图。以主页为例，我们就先从数据库中抓取所有的 **boards** 并将其罗列在HTML中：

### boards/views.py

```
from django.shortcuts import render
from .models import Board

def home(request):
 boards = Board.objects.all()
 return render(request, 'home.html', {'boards': boards})
```

下面是我们如何使用 GCBV 为模型列表来重写它：

### boards/views.py [查看完整文件](#)

```
from django.views.generic import ListView
from .models import Board

class BoardListView(ListView):
 model = Board
 context_object_name = 'boards'
 template_name = 'home.html'
```

那么我们得修改一下 `urls.py` 模块中的引用：

### **myproject/urls.py** [查看完整文件](#)

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

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

如果检查一下主页，我们会看到没有什么变化，一切都按预期的运行。但是我们必须稍微调整我们的测试，因为现在我们现在的视图是 CBV 类型了。

### **boards/tests/test\_view\_home.py** [查看完整文件](#)

```
from django.test import TestCase
from django.urls import resolve
from ..views import BoardListView

class HomeTests(TestCase):
 # ...
 def test_home_url_resolves_home_view(self):
 view = resolve('/')
 self.assertEquals(view.func.view_class, BoardListView)
```

# Django入门与实践-第23章：分页实现

我们可以非常容易地使用 CBV 来实现分页功能。但首先我想手工分页，这样就更有助于我们理解背后的机制，这样它就不那么神秘了。

实际上对 boards 列表视图分页并没有意义，因为我们不期望有很多 boards。但无疑对于主题列表和帖子列表来说是需要一些分页的。

从现在起，我们将在 **board\_topics** 这个视图中来操作。

首先，我们添加一些帖子。我们可以直接使用应用程序的用户界面来添加几个帖子，或者打开 python shell 编写一个小脚本来为我们完成：

```
python manage.py shell
```

```
from django.contrib.auth.models import User
from boards.models import Board, Topic, Post

user = User.objects.first()

board = Board.objects.get(name='Django')

for i in range(100):
 subject = 'Topic test #{}'.format(i)
 topic = Topic.objects.create(subject=subject, board=board,
 starter=user)
 Post.objects.create(message='Lorem ipsum...', topic=topic
 , created_by=user)
```

Topic	Starter	Replies	Views	Last Update
Topic test #99	admin	0	0	Oct. 8, 2017, 1:41 p.m.
Topic test #98	admin	0	0	Oct. 8, 2017, 1:41 p.m.
Topic test #97	admin	0	0	Oct. 8, 2017, 1:41 p.m.
Topic test #96	admin	0	0	Oct. 8, 2017, 1:41 p.m.
Topic test #95	admin	0	0	Oct. 8, 2017, 1:41 p.m.
Topic test #94	admin	0	0	Oct. 8, 2017, 1:41 p.m.
Topic test #93	admin	0	0	Oct. 8, 2017, 1:41 p.m.
Topic test #92	admin	0	0	Oct. 8, 2017, 1:41 p.m.
Topic test #91	admin	0	0	Oct. 8, 2017, 1:41 p.m.

很好，现在我们有一些数据可以玩了。

在我们返回去写代码之前，让我们用 `python shell` 来做一些更多的实验：

```
python manage.py shell
```

```
from boards.models import Topic

All the topics in the app
Topic.objects.count()
107

Just the topics in the Django board
Topic.objects.filter(board__name='Django').count()
104

Let's save this queryset into a variable to paginate it
queryset = Topic.objects.filter(board__name='Django').order_b
y('-last_updated')
```

定义一个你要分页的查询集(**QuerySet**)的排序是很重要的。否则，会返回给你错误的结果。

现在让我们导入 **Paginator** 工具：

```
from django.core.paginator import Paginator

paginator = Paginator(queryset, 20)
```

这里我们告诉Django将查询集按照每页20个元素分页。现在让我们来研究一些 `paginator` 的属性：

```
count the number of elements in the paginator
paginator.count
104

total number of pages
104 elements, paginating 20 per page gives you 6 pages
where the last page will have only 4 elements
paginator.num_pages
6

range of pages that can be used to iterate and create the
links to the pages in the template
paginator.page_range
range(1, 7)

returns a Page instance
paginator.page(2)
<Page 2 of 6>

page = paginator.page(2)

type(page)
django.core.paginator.Page

type(paginator)
django.core.paginator.Paginator
```

这里我们必须注意，因为如果我们试图找到一个不存在的页面，分页器会抛出一个异常：

```
paginator.page(7)
EmptyPage: That page contains no results
```

或者如果我们随意传递进去一个不是页码数字的参数，也会报错：

```
paginator.page('abc')
PageNotAnInteger: That page number is not an integer
```

我们必须在设计用户界面时牢记这些细节。

我们来简单看一下 **Page** 类提供的属性和方法：

```
page = paginator.page(1)

Check if there is another page after this one
page.has_next()
True

If there is no previous page, that means this one is the first page
page.has_previous()
False

page.has_other_pages()
True

page.next_page_number()
2

Take care here, since there is no previous page,
if we call the method `previous_page_number()` we will get an exception:
page.previous_page_number()
EmptyPage: That page number is less than 1
```

## FBV 分页

这里是我们如何使用 FBV 来实现分页：

**boards/views.py** [查看完整文件](#)

```

from django.db.models import Count
from django.core.paginator import Paginator, EmptyPage, PageNotAnInteger
from django.shortcuts import get_object_or_404, render
from django.views.generic import ListView
from .models import Board

def board_topics(request, pk):
 board = get_object_or_404(Board, pk=pk)
 queryset = board.topics.order_by('-last_updated').annotate(replies=Count('posts') - 1)
 page = request.GET.get('page', 1)

 paginator = Paginator(queryset, 20)

 try:
 topics = paginator.page(page)
 except PageNotAnInteger:
 # fallback to the first page
 topics = paginator.page(1)
 except EmptyPage:
 # probably the user tried to add a page number
 # in the url, so we fallback to the last page
 topics = paginator.page(paginator.num_pages)

 return render(request, 'topics.html', {'board': board, 'topics': topics})

```

这部分的实现是使用了 Bootstrap 的四个分页组件来正确的渲染页面。你需要花时间阅读代码，看看它是否适合你。我们在这里使用的是我们之前用过的方法。在这种情况下，`topics` 不再是一个查询集（`QuerySet`），而是一个 `paginator.page` 的实例。

在 `topics` HTML列表的基础上，我们可以渲染分页组件：

[templates/topics.html 查看完整文件](#)

```
{% if topics.has_other_pages %}
<nav aria-label="Topics pagination" class="mb-4">
 <ul class="pagination">
 {% if topics.has_previous %}
 <li class="page-item">
 Previous

 {% else %}
 <li class="page-item disabled">
 Previous

 {% endif %}

 {% for page_num in topics.paginator.page_range %}
 {% if topics.number == page_num %}
 <li class="page-item active">
 {{ page_num }}
 (current)

 {% else %}
 <li class="page-item">
 {{ page_num }}

 {% endif %}
 {% endfor %}

 {% if topics.has_next %}
 <li class="page-item">
 Next

 {% else %}
```

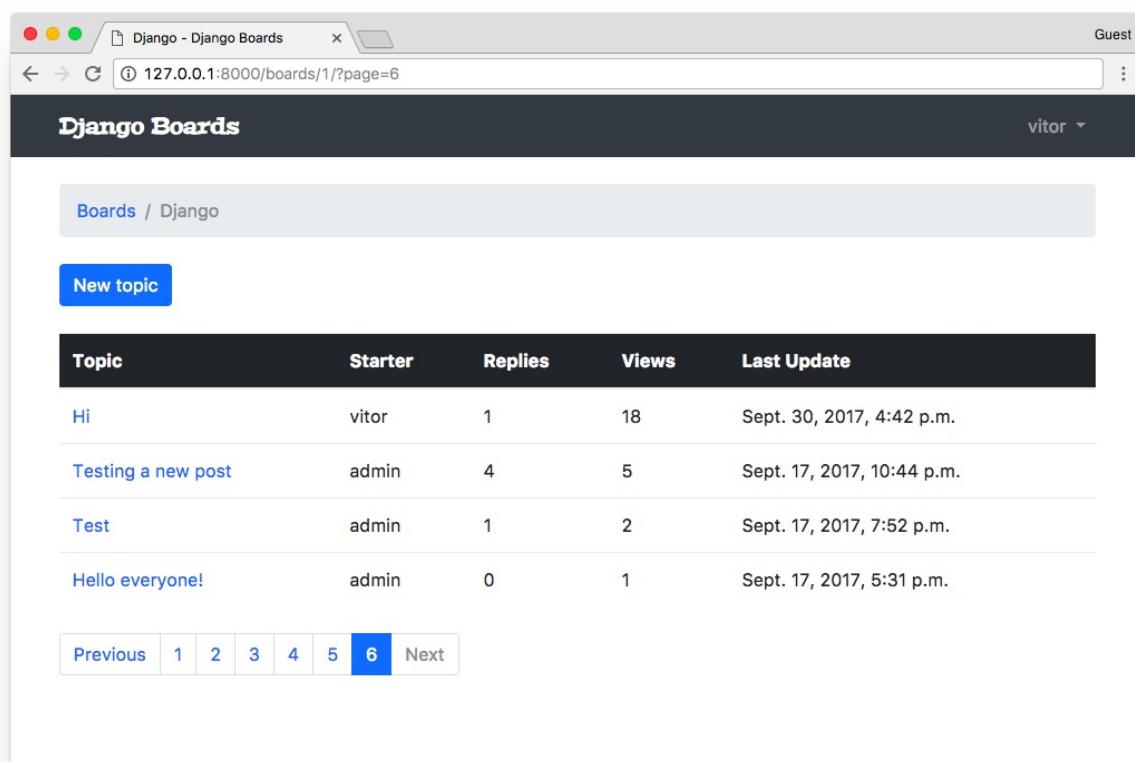
```

<li class="page-item disabled">
 Next

{% endif %}

</nav>
{% endif %}

```



## GCBV 分页

下面，相同的实现，但这次使用**ListView**。

**boards/views.py** [查看完整文件](#)

```
class TopicListView(ListView):
 model = Topic
 context_object_name = 'topics'
 template_name = 'topics.html'
 paginate_by = 20

 def get_context_data(self, **kwargs):
 kwargs['board'] = self.board
 return super().get_context_data(**kwargs)

 def get_queryset(self):
 self.board = get_object_or_404(Board, pk=self.kwargs.get('pk'))
 queryset = self.board.topics.order_by('-last_updated').annotate(replies=Count('posts') - 1)
 return queryset
```

在使用基于类的视图分页时，我们与模板中paginator进行交互的方式有点不同。它会在模板中提供以下变量。

:**paginator,page\_obj,is\_paginated,object\_list**,还有一个我们在**context\_object\_name** 中定义名字的变量。在我们的例子中，这个额外的变量将被命名为 **topics**，并且它将等同于 **object\_list**。

关于这个 **get\_context\_data**，其实，它就是我们在扩展 **GCBV** 时向请求上下文添加内容的方式。

但这里的主要是 **paginate\_by** 属性。一般情况下，只需添加它就足够了。

要记得更新 **urls.py** 哦：

**myproject/urls.py** [查看完整文件](#)

```

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

urlpatterns = [
 # ...
 url(r'^boards/(?P<pk>\d+)/$', views.TopicListView.as_view(),
 name='board_topics'),
]

```

现在我们来修改一下模板：

### templates/topics.html [查看完整文件](#)

```

{% block content %}
 <div class="mb-4">
 New topic
 </div>

 <table class="table mb-4">
 <!-- table content suppressed -->
 </table>

 {% if is_paginated %}
 <nav aria-label="Topics pagination" class="mb-4">
 <ul class="pagination">
 {% if page_obj.has_previous %}
 <li class="page-item">
 Previous

 {% else %}
 <li class="page-item disabled">
 Previous

 {% endif %}

 </nav>
 {% endif %}

```

```
{% for page_num in paginator.page_range %}
 {% if page_obj.number == page_num %}
 <li class="page-item active">

 {{ page_num }}
 (current)

 {% else %}
 <li class="page-item">
 <a class="page-link" href="?page={{ page_num }}"
>{{ page_num }}

 {% endif %}
{% endfor %}

 {% if page_obj.has_next %}
 <li class="page-item">
 <a class="page-link" href="?page={{ page_obj.next
_page_number }}">Next

 {% else %}
 <li class="page-item disabled">
 Next

 {% endif %}

 </nav>
{% endif %}

{% endblock %}
```

现在花点时间运行一下测试代码，如果有需要调整的地方就修一下。

**boards/tests/test\_view\_board\_topics.py**

```
from django.test import TestCase
from django.urls import resolve
from ..views import TopicListView

class BoardTopicsTests(TestCase):
 # ...
 def test_board_topics_url_resolves_board_topics_view(self):
 :
 view = resolve('/boards/1/')
 self.assertEquals(view.func.view_class, TopicListView
)
```

## 可复用的分页模板

就像我们在 `form.html` 中封装模板时做的一样，我们也可以为分页的HTML代码片创建类似的东西。

我们来对主题帖子页面进行分页，进而找到一种复用分页组件的方法。

**boards/views.py** [查看完整文件](#)

```

class PostListView(ListView):
 model = Post
 context_object_name = 'posts'
 template_name = 'topic_posts.html'
 paginate_by = 2

 def get_context_data(self, **kwargs):
 self.topic.views += 1
 self.topic.save()
 kwargs['topic'] = self.topic
 return super().get_context_data(**kwargs)

 def get_queryset(self):
 self.topic = get_object_or_404(Topic, board__pk=self.
 kwargs.get('pk'), pk=self.kwargs.get('topic_pk'))
 queryset = self.topic.posts.order_by('created_at')
 return queryset

```

更新一下 `url.py` [查看完整文件]

```

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

urlpatterns = [
 # ...
 url(r'^boards/(?P<pk>\d+)/topics/(?P<topic_pk>\d+)/$', vi
 ews.PostListView.as_view(), name='topic_posts'),
]

```

现在，我们从`topics.html`模板中获取分页部分的html代码片，并在`templates/includes`文件夹下面创建一个名为 `pagination.html` 的新文件，和 `forms.html` 同级目录：

```

myproject/
| -- myproject/
| | -- accounts/
| | -- boards/
| | -- myproject/
| | -- static/
| | -- templates/
| | | -- includes/
| | | | -- form.html
| | | +-- pagination.html <-- here!
| | +-- ...
| | -- db.sqlite3
| +-- manage.py
+-- venv/

```

### templates/includes/pagination.html

```

{% if is_paginated %}
<nav aria-label="Topics pagination" class="mb-4">
<ul class="pagination">
 {% if page_obj.has_previous %}
 <li class="page-item">
 Previous

 {% else %}
 <li class="page-item disabled">
 Previous

 {% endif %}

 {% for page_num in paginator.page_range %}
 {% if page_obj.number == page_num %}
 <li class="page-item active">
 {{ page_num }}

```

```
 (current)

{% else %}
 <li class="page-item">

{{ page_num }}

 {% endif %}
 {% endfor %}

 {% if page_obj.has_next %}
 <li class="page-item">
 Next

 {% else %}
 <li class="page-item disabled">
 Next

 {% endif %}

</nav>
{% endif %}
```

现在，我们在 `topic_posts.html` 文件中来使用它：

[templates/topic\\_posts.html 查看完整文件](#)

```
{% block content %}

<div class="mb-4">

 <button class="btn btn-primary" role="button">Reply</button>

</div>
```

```
{% for post in posts %}
 <div class="card {% if forloop.last %}mb-4{% else %}mb-2{
% endif %} {% if forloop.first %}border-dark{% endif %}">
 {% if forloop.first %}
 <div class="card-header text-white bg-dark py-2 px-3">
 {{ topic.subject }}</div>
 {% endif %}
 <div class="card-body p-3">
 <div class="row">
 <div class="col-2">

 <small>Posts: {{ post.created_by.posts.count }}</small>
 </div>
 <div class="col-10">
 <div class="row mb-3">
 <div class="col-6">
 <strong class="text-muted">{{ post.created_by.username }}
 </div>
 <div class="col-6 text-right">
 <small class="text-muted">{{ post.created_at }}</small>
 </div>
 </div>
 {{ post.message }}
 {% if post.created_by == user %}
 <div class="mt-3">
 <a href="{% url 'edit_post' post.topic.board.pk post.topic.pk post.pk %}"
 class="btn btn-primary btn-sm"
 role="button">Edit
 </div>
 {% endif %}
 </div>
 </div>
 </div>
 </div>
```

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

{% include 'includes/pagination.html' %}

{% endblock %}
```

别忘了修改主循环为 `{% for post in posts %}`。

我们同样也可以更新一下先前的模板，`topics.html` 模板同样也可以这个封装的分页模板。

[templates/topics.html](#) 查看完整文件

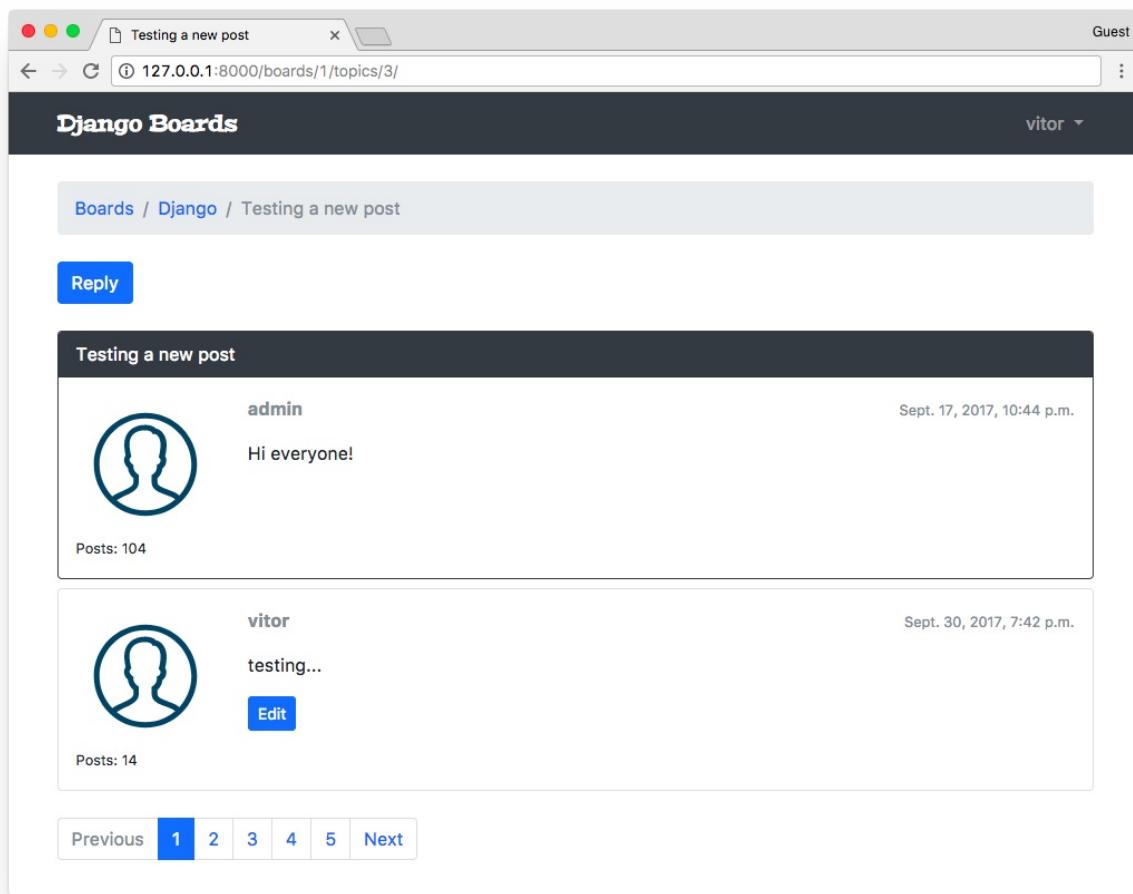
```
{% block content %}
<div class="mb-4">
 New topic
</div>

<table class="table mb-4">
 <!-- table code suppressed -->
</table>

{% include 'includes/pagination.html' %}

{% endblock %}
```

为了测试目的，你需要添加一些帖子（或者通过 `python shell` 去创建），然后修改代码中的 `paginate_by` 到一个较小的数字，比如 `2`，然后看看页面会有什么变化。



(查看完整文件)

更新一下测试用例：

## boards/tests/test\_view\_topic\_posts.py

```
from django.test import TestCase
from django.urls import resolve
from ..views import PostListView

class TopicPostsTests(TestCase):
 # ...
 def test_view_function(self):
 view = resolve('/boards/1/topics/1/')
 self.assertEquals(view.func.view_class, PostListView)
```

# Django入门与实践-第24章：我的账户视图

好的，那么，这部分将是我们最后的一个视图。之后，我们将专心来改进现有功能。

## accounts/views.py [查看完整文件](#)

```
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from django.urls import reverse_lazy
from django.utils.decorators import method_decorator
from django.views.generic import UpdateView

@method_decorator(login_required, name='dispatch')
class UserUpdateView(UpdateView):
 model = User
 fields = ('first_name', 'last_name', 'email',)
 template_name = 'my_account.html'
 success_url = reverse_lazy('my_account')

 def get_object(self):
 return self.request.user
```

## myproject/urls.py [查看完整文件](#)

```
from django.conf.urls import url
from accounts import views as accounts_views

urlpatterns = [
 # ...
 url(r'^settings/account/$', accounts_views.UserUpdateView
 .as_view(), name='my_account'),
]
```

**templates/my\_account.html**

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

{% block title %}My account{% endblock %}

{% block breadcrumb %}
 <li class="breadcrumb-item active">My account
{% endblock %}

{% block content %}
 <div class="row">
 <div class="col-lg-6 col-md-8 col-sm-10">
 <form method="post" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-success">Save changes</button>
 </form>
 </div>
 </div>
{% endblock %}
```

The screenshot shows a web browser window with the title bar "My account" and the URL "127.0.0.1:8000/settings/account/". The page itself is titled "Django Boards" and has a "Guest" status in the top right. A "vitor" dropdown menu is also present. The main content area is titled "My account" and contains three form fields: "First name:" with the value "Vitor", "Last name:" with the value "Freitas", and "Email address:" with the value "vitor@simpleisbetterthancomplex.com". Below these fields is a green "Save changes" button.

My account

Guest

vitor

Django Boards

My account

First name:

Vitor

Last name:

Freitas

Email address:

vitor@simpleisbetterthancomplex.com

Save changes

# Django入门与实践-第25章：Markdown 支持

让我们在文本区域添加 Markdown 支持来改善用户体验。你会看到要实现这个功能非常简单。

首先，我们安装一个名为 **Python-Markdown** 的库：

```
pip install markdown
```

我们可以在 **Post** 视图的 model 中添加一个新的方法：

[boards/models.py 查看完整文件](#)

```
from django.db import models
from django.utils.html import mark_safe
from markdown import markdown

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

 def get_message_as_markdown(self):
 return mark_safe(markdown(self.message, safe_mode='escape'))
```

这里我们正在处理用户的输入，所以我们需要小心一点。当使用 Markdown 功能时，我们需要先让它转义一下特殊字符，然后再解析出 Markdown 标签。这样做之后，输出字符串可以安全的在模板中使用。

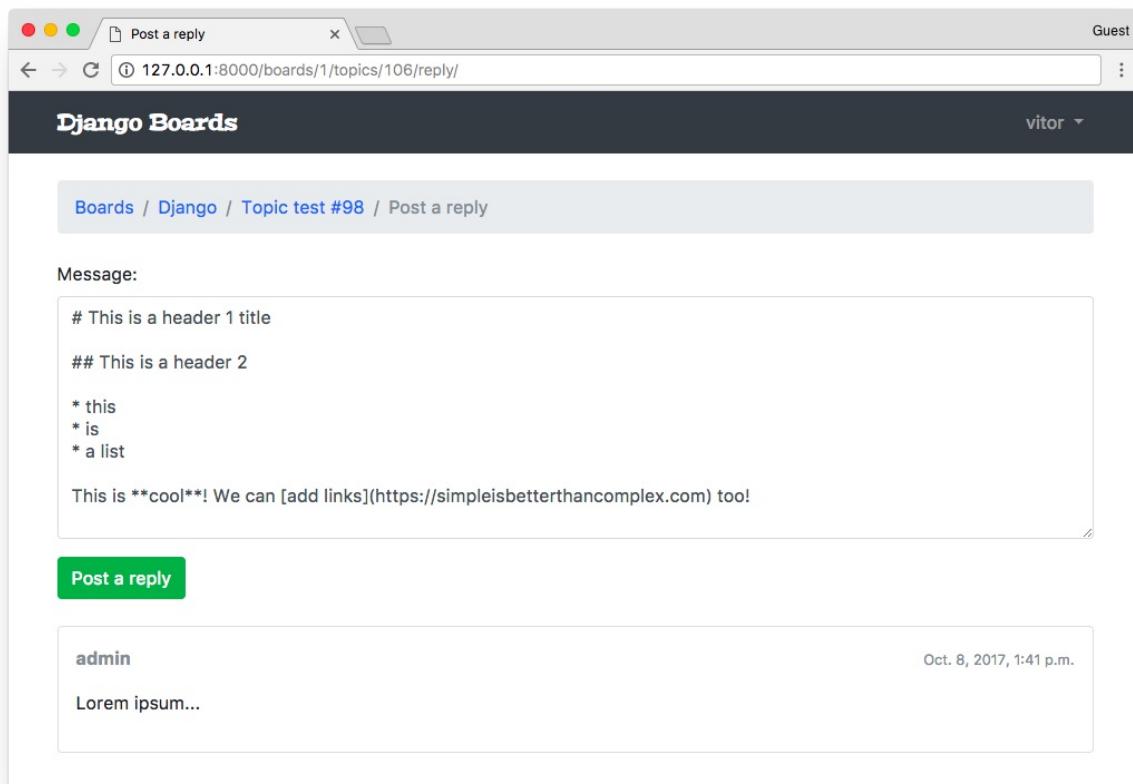
现在，我们只需要在模板 **topic\_posts.html** 和 **reply\_topic.html** 中修改一下 form。

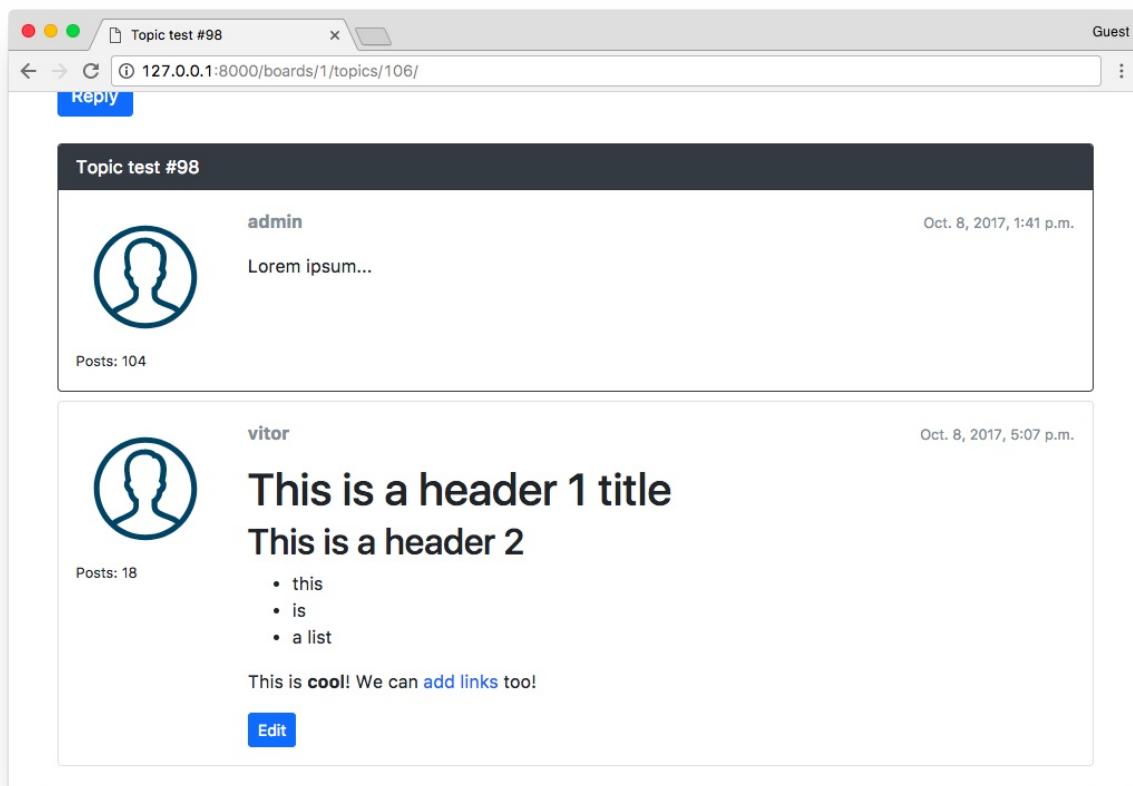
```
{{ post.message }}
```

修改为：

```
{% post.get_message_as_markdown %}
```

从现在起，用户就可以在帖子中使用 Markdown 语法来编辑了。





## Markdown 编辑器

我们还可以添加一个名为 **[SimpleMD](\*\*)\*\*** 的非常酷的 Markdown 编辑器。

可以下载 JavaScript 库，后者使用他们的CDN：

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/simplemde/latest/simplemde.min.css">
<script src="https://cdn.jsdelivr.net/simplemde/latest/simplemde.min.js"></script>
```

现在来编辑一下 **base.html**，为这些额外的Javascripts声明一个block (译者注：方便其他模板继承):

**templates/base.html** [查看完整文件](#)

```
<script src="{% static 'js/jquery-3.2.1.min.js' %}"></script>
<script src="{% static 'js/popper.min.js' %}"></script>
<script src="{% static 'js/bootstrap.min.js' %}"></script>

{% block javascript %}{% endblock %} <!-- Add this empty
block here! -->
```

首先来编辑 **reply\_topic.html** 模板：

[templates/reply\\_topic.html](#) 查看完整文件

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

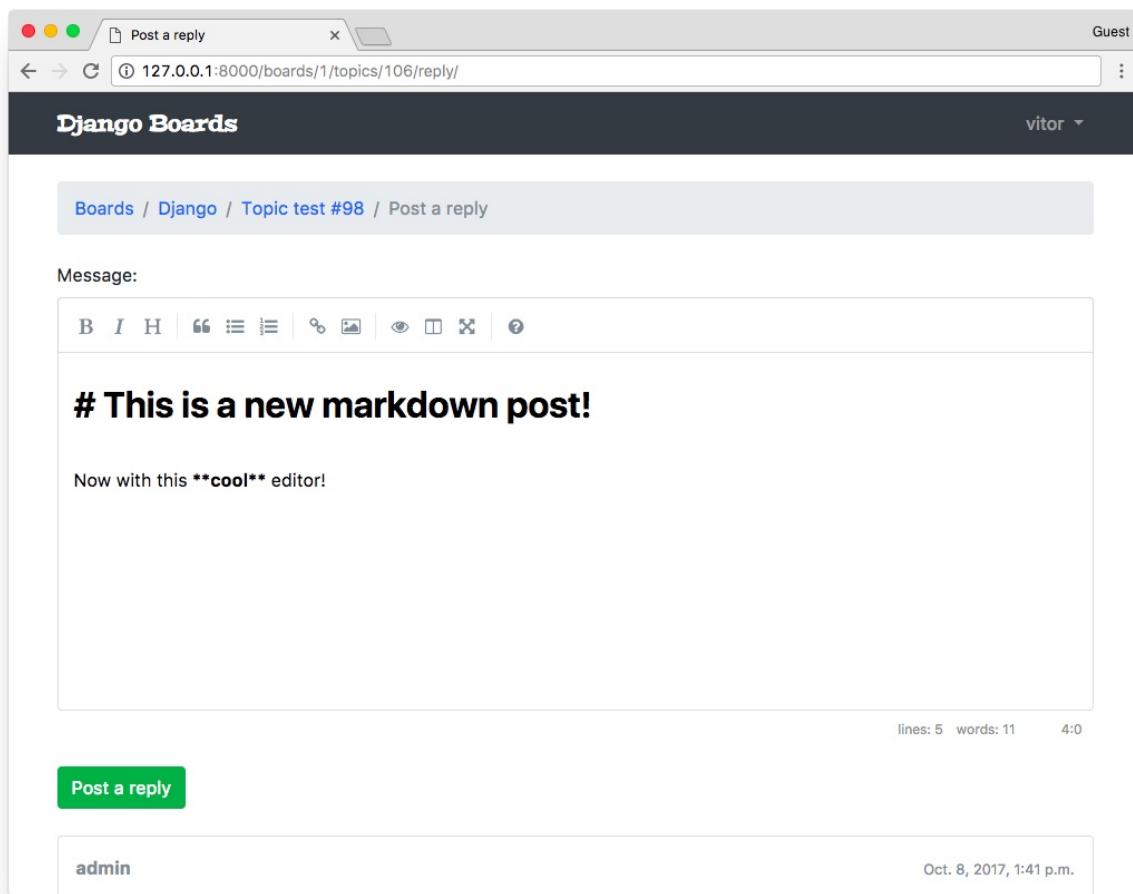
{% load static %}

{% block title %}Post a reply{% endblock %}

{% block stylesheet %}
 <link rel="stylesheet" href="{% static 'css/simplemde.min.css' %}">
{% endblock %}

{% block javascript %}
 <script src="{% static 'js/simplemde.min.js' %}"></script>
 <script>
 var simplemde = new SimpleMDE();
 </script>
{% endblock %}
```

默认情况下，这个插件会将它找到的第一个文本区域转换为 markdown 编辑器。所以这点代码应该就足够了：



接下来在 `edit_post.html` 模板中做同样的操作：

[templates/edit\\_post.html](#) 查看完整文件

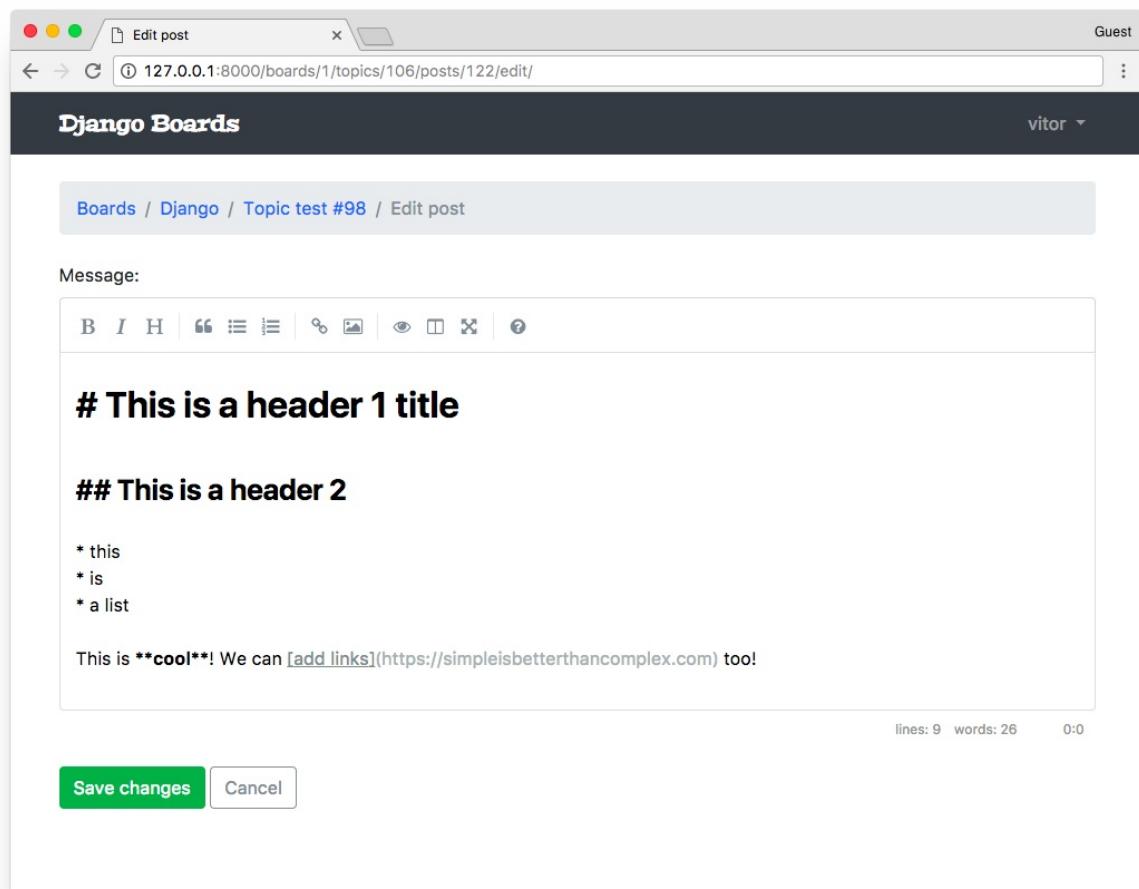
```
{% extends 'base.html' %}

{% load static %}

{% block title %}Edit post{% endblock %}

{% block stylesheet %}
 <link rel="stylesheet" href="{% static 'css/simplemde.min.css' %}">
{% endblock %}

{% block javascript %}
 <script src="{% static 'js/simplemde.min.js' %}"></script>
 <script>
 var simplemde = new SimpleMDE();
 </script>
{% endblock %}
```



# Django入门与实践-第26章：个性化工具

我觉得只添加内置的人性化(**humanize**)包就会很不错。它包含一组为数据添加“人性化 (human touch) ”的工具集。

例如，我们可以使用它来更自然地显示日期和时间字段。我们可以简单地显示：“2分钟前”，而不是显示整个日期。

我们来实践一下！首先，添加 `django.contrib.humanize` 到配置文件的 `INSTALLED_APPS` 中。

## myproject/settings.py

```
INSTALLED_APPS = [
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'django.contrib.humanize', # <- 这里

 'widget_tweaks',

 'accounts',
 'boards',
]
```

现在我们就可以在模板中使用它了。首先来编辑 `topics.html` 模板：

[templates/topics.html](#) 查看完整文件

```

{% extends 'base.html' %}

{% load humanize %}

{% block content %}
<!-- 代码被压缩 -->

<td>{{ topic.last_updated|naturaltime }}</td>

<!-- 代码被压缩 -->
{% endblock %}

```

我们所要做的就是在模板中加载 `{%load humanize%}` 这个模板标签，然后在模板中使用过滤器：`{{ topic.last_updated|naturaltime }}`

Topic	Starter	Replies	Views	Last Update
Testing humanize	vitor	0	1	3 seconds ago
Topic test #99	admin	3	8	4 hours ago
Topic test #98	admin	3	7	4 hours ago
Topic test #97	admin	0	0	4 hours ago
Topic test #96	admin	0	0	4 hours ago
Topic test #95	admin	0	0	4 hours ago

你当然可以将它添加到其他你需要的地方。

## Gravatar(添加头像用的库)

给用户个人信息添加图片的一种非常简单的方法就是使用 [Gravatar](#)。

在 **boards/templatetags** 文件夹内，创建一个名为 **gravatar.py** 的新文件：

### **boards/templatetags/gravatar.py**

```
import hashlib
from urllib.parse import urlencode

from django import template
from django.conf import settings

register = template.Library()

@register.filter
def gravatar(user):
 email = user.email.lower().encode('utf-8')
 default = 'mm'
 size = 256
 url = 'https://www.gravatar.com/avatar/{md5}?{params}'.format(
 md5=hashlib.md5(email).hexdigest(),
 params=urlencode({'d': default, 's': str(size)}))
 return url
```

基本上我们可以使用[官方提供的代码片段](#)。我只是做了一下适配，使得它可以在python 3环境中运行。

很好，现在我们可以将它加载到我们的模板中，就像之前我们使用人性化模板过滤器一样：

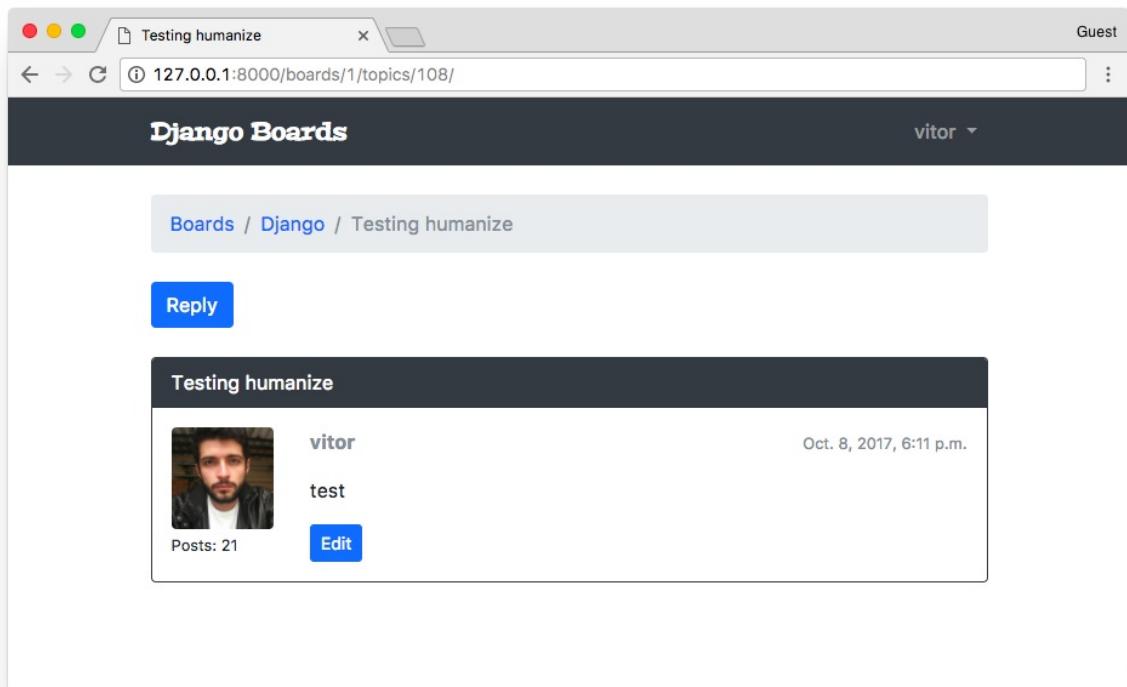
**templates/topic\_posts.html** [查看完整文件](#)

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

{% load gravatar %}

{% block content %}
<!-- code suppressed -->

<!-- code suppressed -->
{% endblock %}
```



## 最后调整

也许你已经注意到了，如果有人回复帖子时有一个小问题。我们没有更新 `last_update` 字段，因此主题的排序被打乱顺序了。

我们来修一下：

### boards/views.py

```
@login_required
def reply_topic(request, pk, topic_pk):
 topic = get_object_or_404(Topic, board__pk=pk, pk=topic_pk)
 if request.method == 'POST':
 form = PostForm(request.POST)
 if form.is_valid():
 post = form.save(commit=False)
 post.topic = topic
 post.created_by = request.user
 post.save()

 topic.last_updated = timezone.now() # <- 这里
 topic.save() # <- 这里

 return redirect('topic_posts', pk=pk, topic_pk=topic_pk)
else:
 form = PostForm()
return render(request, 'reply_topic.html', {'topic': topic, 'form': form})
```

接下来我们要做的事是需要控制一下页面访问统计系统。我们不希望相同的用户再次刷新页面的时候被统计为多次访问。为此，我们可以使用会话(sessions)：

### boards/views.py

```
class PostListView(ListView):
 model = Post
 context_object_name = 'posts'
 template_name = 'topic_posts.html'
 paginate_by = 20

 def get_context_data(self, **kwargs):
 session_key = 'viewed_topic_{}'.format(self.topic.pk)
 # <--这里
 if not self.request.session.get(session_key, False):
 self.topic.views += 1
 self.topic.save()
 self.request.session[session_key] = True
 # <--直到这里

 kwargs['topic'] = self.topic
 return super().get_context_data(**kwargs)

 def get_queryset(self):
 self.topic = get_object_or_404(Topic, board__pk=self.kwargs.get('pk'), pk=self.kwargs.get('topic_pk'))
 queryset = self.topic.posts.order_by('created_at')
 return queryset
```

现在我们可以在主题列表中提供一个更好一点的导航。目前唯一的选择是用户点击主题标题并转到第一页。我们可以实践一下这么做：

## boards/models.py

```

import math
from django.db import models

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

 def __str__(self):
 return self.subject

 def get_page_count(self):
 count = self.posts.count()
 pages = count / 20
 return math.ceil(pages)

 def has_many_pages(self, count=None):
 if count is None:
 count = self.get_page_count()
 return count > 6

 def get_page_range(self):
 count = self.get_page_count()
 if self.has_many_pages(count):
 return range(1, 5)
 return range(1, count + 1)

```

然后，在 `topics.html` 模板中，我们可以这样实现：

### `templates/topics.html`

```


| Topic | Starter | Replies | Views |
|-------|---------|---------|-------|
|-------|---------|---------|-------|


```

```
<th>Last Update</th>
</tr>
</thead>
<tbody>
{% for topic in topics %}
 {% url 'topic_posts' board.pk topic.pk as topic_url %}
}
<tr>
<td>
<p class="mb-0">
 {{ topic.subject }}
</p>
<small class="text-muted">
 Pages:
 {% for i in topic.get_page_range %}
 {{ i }}
 {% endfor %}
 {% if topic.has_many_pages %}
 ... Last Page
 {% endif %}
 </small>
</td>
<td class="align-middle">{{ topic.starter.username }}</td>
 <td class="align-middle">{{ topic.replies }}</td>
 <td class="align-middle">{{ topic.views }}</td>
 <td class="align-middle">{{ topic.last_updated|naturaltime }}</td>
</tr>
{% endfor %}
</tbody>
</table>
```

就像每个主题的小分页一样。请注意，我在 `table` 标签里还添加了 `table-striped` 类，使得表格有一个更好的样式。

Topic	Starter	Replies	Views	Last Update
<a href="#">Topic test #95</a> Pages: 1 2 3 4 ... Last Page	admin	1001	5	43 minutes ago
<a href="#">Topic test #94</a> Pages: 1	admin	1	2	44 minutes ago
<a href="#">Testing humanize</a> Pages: 1	vitor	0	22	an hour ago
<a href="#">Topic test #99</a> Pages: 1	admin	3	9	5 hours ago
<a href="#">Topic test #98</a> Pages: 1	admin	3	7	5 hours ago
<a href="#">Topic test #97</a> Pages: 1	admin	0	0	5 hours ago

在回复页面中，我们现在是列出了所有的回复。我们可以将它限制在最近的十个回复。

## boards/models.py

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

 def get_last_ten_posts(self):
 return self.posts.order_by('-created_at')[:10]
```

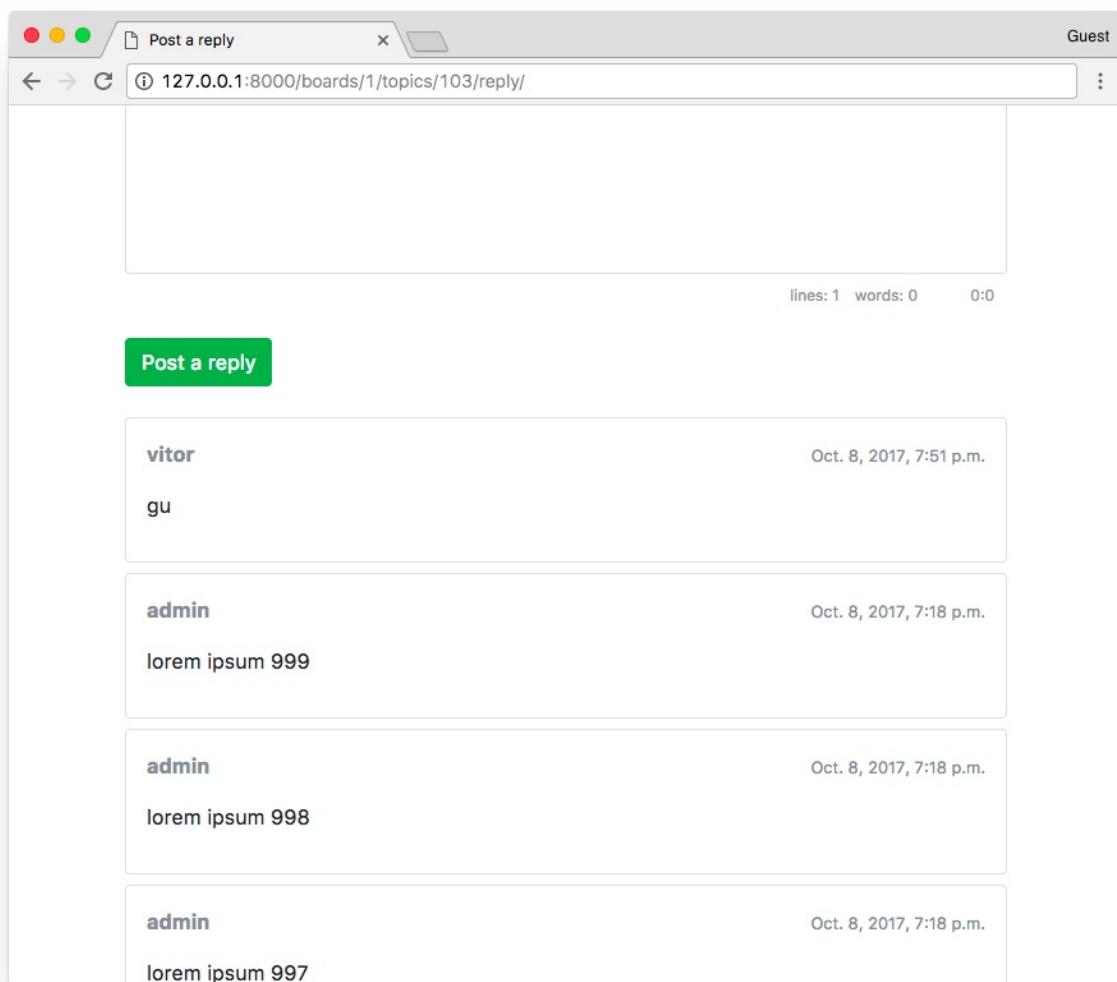
## templates/reply\_topic.html

```
{% block content %}

<form method="post" class="mb-4" novalidate>
 {% csrf_token %}
 {% include 'includes/form.html' %}
 <button type="submit" class="btn btn-success">Post a reply
</button>
</form>

{% for post in topic.get_last_ten_posts %} <!-- here! -->
 <div class="card mb-2">
 <!-- code suppressed -->
 </div>
{% endfor %}

{% endblock %}
```



另一件事是，当用户回复帖子时，我们现在是会再次将用户重定向到第一页。我们可以通过将用户送回到最后一页来改善这个问题。

我们可以在帖子上添加一个ID：

**templates/topic\_posts.html**

```
{% block content %}

<div class="mb-4">
 <a href="{% url 'reply_topic' topic.board.pk topic.pk %}"
 class="btn btn-primary" role="button">Reply
</div>

{% for post in posts %}
 <div id="{{ post.pk }}" class="card {% if forloop.last %}mb-4{% else %}mb-2{% endif %} {% if forloop.first %}border-dark{% endif %}">
 <!-- code suppressed -->
 </div>
{% endfor %}

{% include 'includes/pagination.html' %}

{% endblock %}
```

这里的重要点是 `<div id="{{ post.pk }}" ...>`。

然后我们可以在视图中像这样使用它：

### boards/views.py

```
@login_required
def reply_topic(request, pk, topic_pk):
 topic = get_object_or_404(Topic, board_pk=pk, pk=topic_pk)
 if request.method == 'POST':
 form = PostForm(request.POST)
 if form.is_valid():
 post = form.save(commit=False)
 post.topic = topic
 post.created_by = request.user
 post.save()

 topic.last_updated = timezone.now()
 topic.save()

 topic_url = reverse('topic_posts', kwargs={'pk': pk, 'topic_pk': topic_pk})
 topic_post_url = '{url}?page={page}#{id}'.format(
 url=topic_url,
 id=post.pk,
 page=topic.get_page_count()
)

 return redirect(topic_post_url)
 else:
 form = PostForm()
 return render(request, 'reply_topic.html', {'topic': topic, 'form': form})
```

在 `topic_post_url` 中，我们使用最后一页来构建一个url，添加一个锚点id等于帖子id的元素。

有了这个，这要求我们需要更新下面的这些测试用例：

**boards/tests/test\_view\_reply\_topic.py**

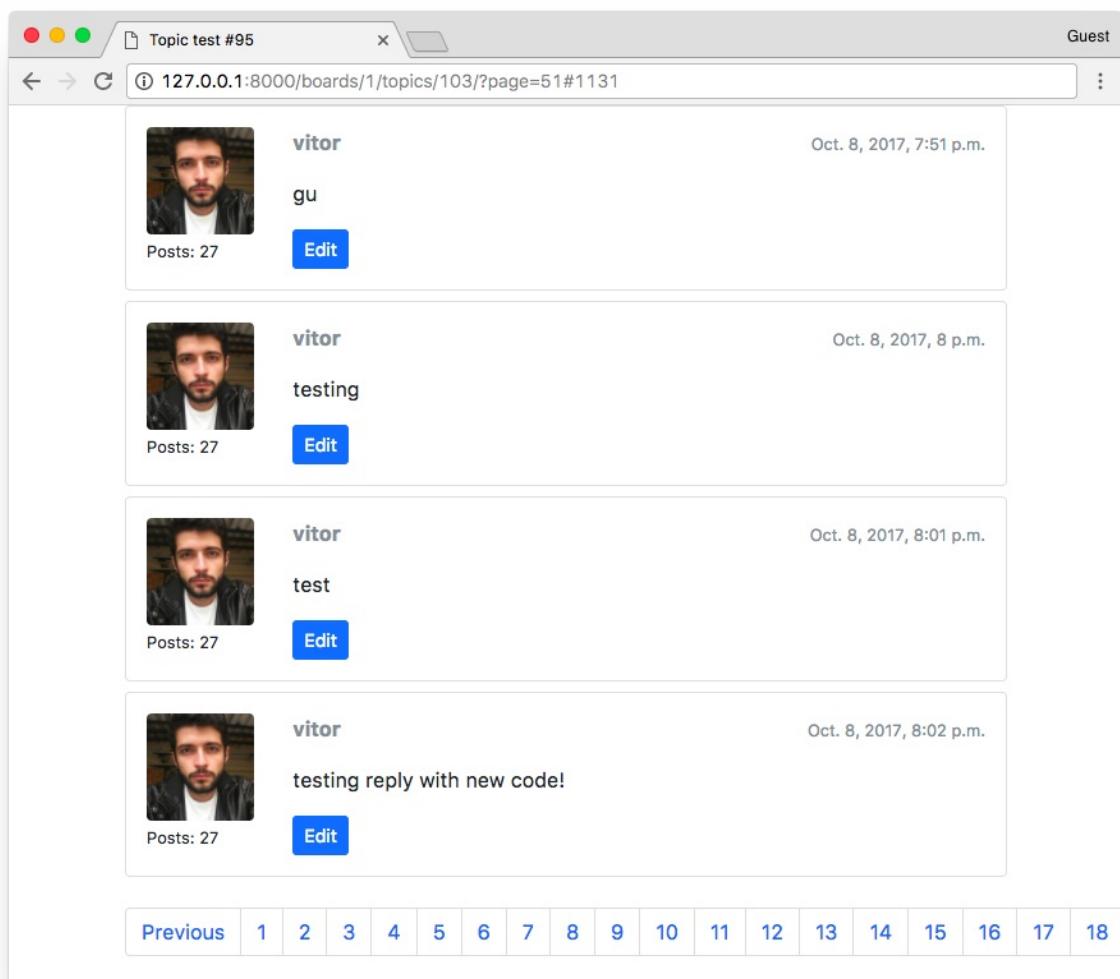
```

class SuccessfulReplyTopicTests(ReplyTopicTestCase):
 # ...

 def test_redirection(self):
 """
 A valid form submission should redirect the user
 """

 url = reverse('topic_posts', kwargs={'pk': self.board.pk, 'topic_pk': self.topic.pk})
 topic_posts_url = '{url}?page=1#2'.format(url=url)
 self.assertRedirects(self.response, topic_posts_url)

```



下一个问题，正如你在前面的截图中看到的，要解决分页时页数太多的问题。

最简单的方法是调整 **pagination.html** 模板：

### templates/includes/pagination.html

```
{% if is_paginated %}

<nav aria-label="Topics pagination" class="mb-4">
 <ul class="pagination">

 {% if page_obj.number > 1 %}
 <li class="page-item">
 First

 {% else %}
 <li class="page-item disabled">
 First

 {% endif %}

 {% if page_obj.has_previous %}
 <li class="page-item">
 Previous

 {% else %}
 <li class="page-item disabled">
 Previous

 {% endif %}

 {% for page_num in paginator.page_range %}
 {% if page_obj.number == page_num %}
 <li class="page-item active">

 {{ page_num }}
 (current)

 {% elif page_num > page_obj.number|add:'-3' and page_
```

```
num < page_obj.number|add:'3' %}
 <li class="page-item">

{{ page_num }}

 {% endif %}
 {% endfor %}

 {% if page_obj.has_next %}
 <li class="page-item">
 Next

 {% else %}
 <li class="page-item disabled">
 Next

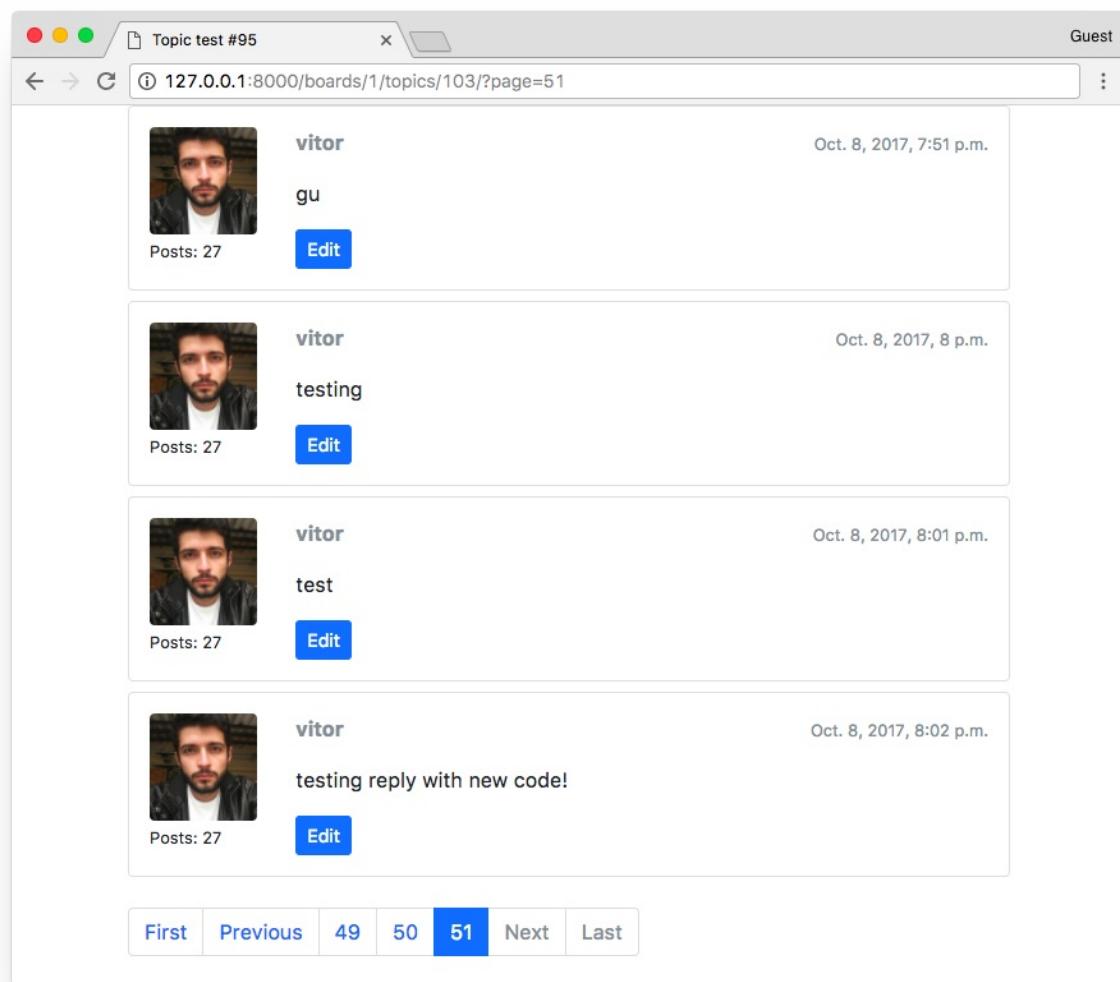
 {% endif %}

 {% if page_obj.number != paginator.num_pages %}
 <li class="page-item">
 Last

 {% else %}
 <li class="page-item disabled">
 Last

 {% endif %}

</nav>
{% endif %}
```



## 总结

在本教程中，我们完成了**Django board**项目应用的实现。我可能会发布一个后续的实现教程来改进代码。我们可以一起研究很多事情。例如数据库优化，改进用户界面，文件上传操作，创建审核系统等等。

下一篇教程将着重于部署。它将是关于如何将你的代码投入到生产中以及需要关注的一些重要细节的完整指南。

我希望你会喜欢本系列教程的第六部分！最后一部分将于下周2017年10月16日发布。如果你希望在最后一部分发布时收到通知，可以订阅我们的[邮件列表](#)。

该项目的源代码在github上找到。当前状态的该项目可以在发布标签 **v0.6-lw** 下找到。或者直接点击下面的链接：

<https://github.com/sibtc/django-beginners-guide/tree/v0.6-lw>

## Deployment.md

<https://simpleisbetterthancomplex.com/series/2017/10/16/a-complete-beginners-guide-to-django-part-7.html>