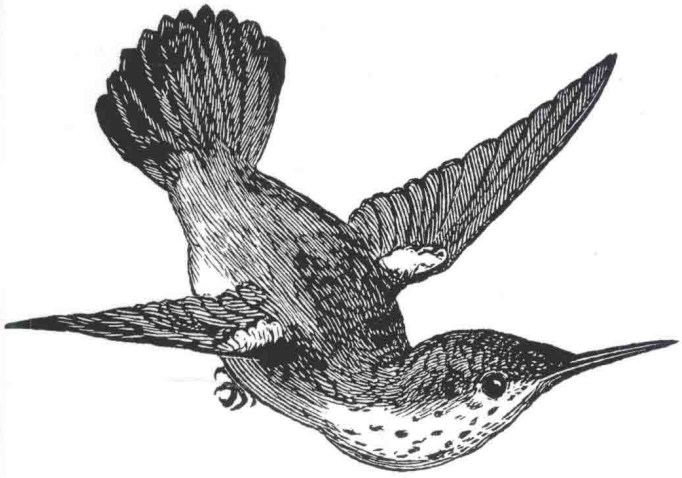
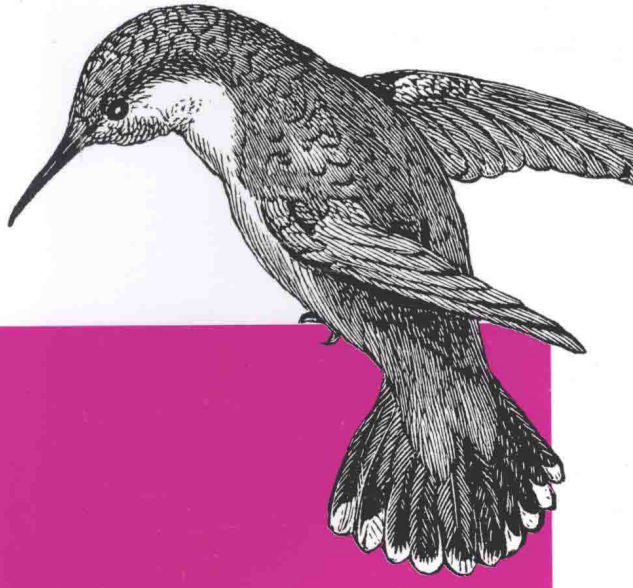
O'REILLY®



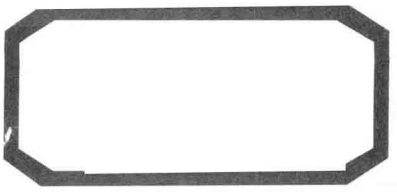


轻量级 Django

Lightweight Django

Julia Elman & Mark Lavin 著
侯荣涛 吴磊 译

中国电力出版社



轻量级 Django

Julia Elman & Mark Lavin 著

侯荣涛 吴磊 译

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

轻量级 Django /（美）茱莉亚·埃尔曼（Julia Elman），（美）马克·拉温（Mark Lavin）著；侯荣涛，吴磊译 . —北京：中国电力出版社，2016.10

书名原文：Lightweight Django

ISBN 978-7-5123-9396-7

I. ①轻… II. ①茱… ②马… ③侯… ④吴… III. ①软件工具－程序设计 IV. ① TP311.56

中国版本图书馆 CIP 数据核字（2016）第 116037 号

北京市版权局著作权合同登记

图字：01-2016-2777 号

封面设计 / Ellie Volckhausen，张健

出版发行 / 中国电力出版社（http://www.cepp.sgcc.com.cn）

地　　址 / 北京市东城区北京站西街 19 号（邮政编码 100005）

经　　销 / 全国新华书店

印　　刷 / 北京天宇星印刷厂

开　　本 / 787 毫米 ×980 毫米　16 开本　14 印张　268 千字

版　　次 / 2016 年 10 月第一版　2016 年 10 月第一次印刷

印　　数 / 0001 － 3000 册

定　　价 / 39.00 元（册）

O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

前言 1

绪论 7

第1章 世界上最小的Django项目 11

你好Django 11

改进 15

第2章 无状态的Web应用 22

什么是无状态？ 22

可复用应用与可组合服务 23

占位图片服务器 23

占位视图 25

创建主页面视图 31

第3章 创建静态站点生成器 38

使用Django创建静态站点 38

什么是快速原型？ 39

最初的项目结构 39

修饰页面 41

生成静态内容 52

处理和压缩静态文件 56

生成动态内容 60

第4章 构建REST API 67
Django和REST 67
Scrum板数据图 68
设计API 74
测试API 82
下一步 96
第5章 使用Backbone.js的客户端Django 97
Backbone简述 98
设置项目文件 99
连接Backbone到Django 104
客户端Backbone路由 106
构建用户认证 110
第6章 单页面Web应用 132
什么是单页面Web应用? 132
发现API 133
构建主页 136
sprint详情页面 141
CRUD任务 155
第7章 实时Django 165
HTML5实时API 165
在Tornado下使用websocket 167
客户端通信 175
第8章 Django与Tornado通信 195
从Tornado接收更新 195
改善服务器 200
最终的websocket服务器 212

前言

自 Django 创建以来，各种各样的开源社区已经构建了很多 Web 框架，比如 JavaScript 社区创建的 Angular.js、Ember.js 和 Backbone.js 之类面向前端的 Web 框架，它们是现代 Web 开发中的先驱。Django 从哪里入手来适应这些框架呢？我们如何将客户端 MVC 框架整合成为当前的 Django 基础架构？

本书讲述如何利用 Django 强大的“自支持”功能体系。本书的目标是引导读者跨越认为 Django 太繁重而无法进行快速应用开发的误区。从创建世界上最小的 Django 应用到构建 RESTful API，本书将引导读者学习如何利用这种流行的 Python Web 框架。

写这本书的目的

我们编写本书主要是出于对 Django 的热爱。我们的社区是令人惊奇的，它有很多资源可以用来学习 Django 和开发应用。尽管我们也喜欢包括官方的 Django 文档在内的一些资源，但这仅仅是对 Django 强大功能的过于关注，而非它的解耦设计。Django 是一个令人满意的框架，它带有很多用于构建 Web 应用的通用程序。在本书中，我们要突出说明如何将这些组件进行分离和间接替换，并从中选出最适合创建应用的一些组件来。同样地，我们想要将 Django 项目和应用的典型框架进行分解。我们的目标是使读者不再提出“我怎样用 Django 完成我的 X 工作”这样的问题，而是提出“Django 能帮我完成 X 工作吗？如果不能，社区有可用的工具吗”这样的问题。

另外，我们想要回答的问题是利用大量客户端实时交互，以及与本地可移动应用匹配的组件构建更多应用时，将 Django 放在网络的什么地方适用。作为框架，用户对 Django 一无所知，这使得一些喜欢 Django 的用户无法得到构建这种应用的指导。我们希望本书能够帮助解决这些问题，想看到 Django 及其社区日益壮大，并在多年后大家也成为它的一员。

本书读者对象

如果你喜欢阅读本书，你很可能是 Django 的中级用户。在学习完 Django 的注册教程以及编写了一些基本 Django 应用后，很想知道下一步该如何做。本书接下来会帮助你基本掌握如何利用 Django 的有效性和简单性。

或者，你可能正在进行一个 Django 项目，很想知道如何将 Backbone.js 整合到你的项目中。本书将引导你进行一些有关整合的最好练习，并为构建内容丰富的 Web 应用提供一个起点。

哪些人不适合阅读本书？

尽管本书适用许多不同背景的开发者，但并不能面面俱到。对于那些不具备编写 Python 和 JavaScript 程序能力的人来说，这本书可能很不适合。书中所有的概念和实例都是围绕这些语言编写的，并在每个章节中大量使用。对于那些初次接触 Django 的新手，也不适合。

关于本书示例

书中的每个项目示例都是在快速应用开发的主旨下精心制作的。在每一章，都将学到在项目管理、工具和团队合作支持下创建项目的方法。我们想要读者创建用于自己的项目，并能够根据自己的需要进行定制。一般而言，本书提供的实例代码，都可以在你的程序和文档中进行使用，无需得到我们的许可，除非是更新代码的重要内容。例如，使用本书中的几个代码块编写程序是不需要得到允许的。销售或分发 O'Reilly 书中的实例光盘就需要得到许可。引用本书或本书中的实例代码解答问题不需要得到允许。将本书中的大量实例代码集成到自己产品的文档中需要得到许可。

书中的代码样例可以从下面网站找到：*https://github.com/lightweightdjango/examples*

我们很赞赏对本书的引用，但不要求。通常引用包括标题、作者、出版商和 ISBN。例如：“《*Lightweight Django*》Julia Elman and Mark Lavin （O'Reilly），Copyright 2015 Julia Elman and Mark Lavin，978-1-491-94594-0”。

如果发现所使用的代码示例超出了上面的许可范围，可以随时通过邮件与我们联系：

permissions@oreilly.com。

本书结构

第 1 章，世界上最小的 Django 项目。

创建轻便简单的网络应用是本书的核心思想。在这一章，要创建一个可运行的单行文件“Hello World”的 Django 应用。

第 2 章，无状态的 Web 应用。

讲述占位符图像服务的创建方法。第 2 章将引导读者通过创建无状态网络应用来生成占位符图像的 URL。

第 3 章，创建静态站点生成器。

快速构建原型是一门有用的创建和辅助网络应用的技术。我们要通过创建静态网站生成器协助维护团队项目的方式来检验该技术的效果。

第 4 章，构建 REST API。

REST API 是创建丰富的、具有内容相关性的网络应用的重要部分。在这一章我们利用 Django 静态框架开始构建一个大比例的 Scrum 板应用。

第 5 章，使用 Backbone.js 的客户端 Django。

本章继续讲述在第 4 章中用新构建的 RESTful API 创建的 Backbone.js 应用。介绍创建新 Backbone 应用的每个组件以及如何用 Django 同步该客户端框架。

第 6 章，单页面 Web 应用。

单页面网络应用是一种能够创建丰富客户端网络应用的途径。在本章，我们将回到简单的 Backbone 应用中，不断使这个单页面应用更加健壮。

第 7 章，实时 Django。

这一章讲述创建响应实时交互的网络应用，为用户提供及时满意的服务。为了继续完成先前两章的项目，我们要将一个实时组件添加到使用网络插件和 Tornado 的 Scrum 板上，这是一个用 Python 编写的异步网络库。

第 8 章，Django 和 Tornado 通信。

将强大的 Django 与 Tornado 的健壮特性相结合是创建可变、实时的 Django 应用的重要标志。在本章中，我们将通过集成 Django 的通信能力扩展 Tornado 服务器的功能，来建立一种既安全又可交互的联系。

排版约定

本书使用如下排版约定：

斜体

表示新术语、URL、email 地址、文件名和文件扩展名。

等宽字体（`Constant width`）

用于程序列表，以及引用如变量、函数名、数据库、数据类型、环境变量、语句和关键词之类的程序元素。

加粗等宽字体（**`Constant width bold`**）

表示应由用户手动输入的命令或其他文本。

斜体等宽字体（*`Constant width italic`*）

表示要用用户提供的值或上下文确定的值来替换的文本。

所有代码实例运用省略号（…）表示先前现实的一些代码已经被跳过，来缩短实例代码或跳到最相关的代码段。

这个符号表示提示或建议。

这个符号表示一般注释。

这个符号表示警告或警示。

致谢

有许多人需要感谢，没有他们的努力这本书是不可能完成的。本书的编辑 Meghan 给了我们很大的支持。

感谢技术评论员 Aymeric Augustin、Jon Banafato、Barbara Shaurette 和 Marie Selvanadin 给我们的评价，无论是表扬还是批评，对我们尽力完成这本书都给予了帮助。也要感谢 Heather Scherer 对本书给予的技术指导。

对所有开源的开发人员和提供者表示感谢，他们无穷无尽的努力为本书提供了使用和写作所需的各种工具。

感谢早期版本的读者，他们给了我们完成工作的机会，使我们能够解决敲字和格式缺陷等问题，并提供了反馈，从而使错误得到纠正。

Julia

我非常感谢我极其可爱的家庭和亲密的朋友，他们对我写作本书给予了一贯的支持。感谢我的丈夫 Andrew，对我能力的信任和在漫长与崎岖的写作过程中的一贯鼓舞和坚定的支持。感谢我的母亲 Katherine，她使我超越了自己的能力。感谢我的继父 Tom，他教我如何使用无线电钻为我的汽车更换机油，为我灌输了艰苦工作的价值。感谢我的哥哥 Alex 和姐姐 Elizabeth，他们一直作为旁观者为我鼓劲。感谢我最好的朋友 Jenny，她给了我永恒的爱和终生的友谊。

还要对我的极好的合作者 Mark 的卓越才能和友谊表示感谢。他是我曾经合作过的最有才能的开发者之一。我们并肩完成了这本书，我不能想象与其他人合作撰写这本书会是怎样一个过程。

我还要非常感谢 Python 社区和在我的事业上给我灵感、鼓励和指导的特殊成员：James Bennett、Sean Bleier、Nathan Borror、Colin Copeland、Matt Croydon、Katie Cunningham、Selena Deckelmann、Jacob Kaplan-Moss、Jessica McKellar、Jesse Noller、Christian Metts、Lynn Root、Caleb Smith、Paul Smith、Karen Tracey、Malcolm Tredinnick、Ben Turner 和 Simon Willison。

Mark

首先，这本书离不开来自我的家庭的爱和支持。我妻子 Beth 和女儿 Kennedy 忍受我长时间工作，以及由于负担的加重而导致的脾气暴躁。还要感谢我的哥哥 Matt，他给了我

很深的见解和早期的反馈。感谢我的父母和我的哥哥 James，他们给了我无尽的支持。

感谢我的合作者 Julia。我们之间是在友谊和互相尊重中进行合作的。我将永远珍惜比我们合作成果建立的更加伟大的东西。

最后，感谢我在 Caktus 小组的同事们对我及时的支持、反馈和鼓励。

绪论

在开始之前，本章将对完成本书中的例子所需的知识和软件做一个概要介绍。

Python

本书面向的是具有 Python 经验的开发者，使用的是 Python 3。特别提醒，书中所有的示例都在 Python 3.3 和 Python 3.4 中得到了测试。尽管我们并不推荐，但对 Python 足够熟悉的读者也可以在需要的时候进行代码转换，然后在 Python 2.7 下完成这些例子。如果想要查看有关 Python 新版本中的特性以及使用于系统的安装指南，请访问：*https://www.python.org/downloads/*。

假定已经在本地开发设备上安装了 Python，并知道如何编辑 Python 文件和运行这些文件。尽管有些系统或者安装可能需要使用 `Python3` 或 `Python` 的完整版，如 `Python 3.3`、`Python 3.4`，但本书在命令行中调用 Python 时，将使用 `python` 命令。同样的，在安装新的程序包时，因为有些安装版本需要使用 `pip3` 命令，因此本书的示例将使用 `pip` 命令。针对本书和一般的 Python 开发，我们推荐使用 `virtualenv` 为每个项目创建一个隔离的 Python 环境。没有这样的隔离环境而通过 `pip` 安装程序包时，可能会需要访问根目录或计算机系统管理员的权限。假设没有创建隔离环境，就需要在 `pip` 命令前加上 `sudo` 前缀或一些用来获取这些权限的其他命令，但这些前缀都未在示例中写出。

Python 程序包

在开始本书之前，唯一需要安装的 Python 程序包是 Django。书中所有示例都是在 Django 1.8.3 下编写和测试的。建议使用 pip 进行安装：

```
hostname $ pip install Django==1.8.3
```

直到2015年8月，Django 1.8.3仍然处于发布的候选阶段。如果早先的版本无法使用，可以用 pip https://github.com/django/django/archive/stable/1.8.x.zip 命令从开发分支中安装 1.8 的先行版。

要了解更多关于 Django 新版本中的特性，请访问 *https://docs.djangoproject.com/en/dev/releases/1.8/*。要查看其他安装指令，也可以参阅 Django 安装指南。

在讲解各章的过程中，还要安装其他额外的软件包。第 1、2、3 章中的项目是相互独立的，可以在单独的虚拟环境中处理，只需部署 Django 即可。第 4 章到第 8 章包含一个大的项目，在这些章节中需要使用同一个虚拟环境。

Web 开发

由于 Django 是一个 Web 框架，本书假设读者具备了基本的 HTML 和 CSS 知识，因此 JavaScript 的例子更加深些，所需的相关知识会在接下来的段落中详细说明。如果对 HTTP 协议，尤其是各种 HTTP 动词（`GET`、`POST`、`PUT`、`DELETE` 等）的用法和作用有一定理解，则会很有帮助。

JavaScript

本书后面的章节会大量使用 JavaScript。需要熟练编写 JavaScript/jQuery。有 DOM 操作和使用 jQuery 调用 Ajax 经验的开发者，可以学做 Backbone.js 的示例。如果熟悉如 Angular.js、Ember.js 或 Knockoout.js 等其他的客户端架构，那就更超前了。本书并非是 Backbone.js 的权威指南。如果对 JavaScript 开发，尤其是 Backbone.js 的 MVC 结构不熟悉，推荐阅读一些 O'Reilly 的书：

- JavaScript: The Definitive Guide, by David Flanagan
- JavaScript: The Good Parts, by Douglas Crockford
- JavaScript Patterns, by Stoyan Stefanov
- Speaking JavaScript, by Axel Rauschmayer
- Developing Backbone.js Applications, by Addy Osmani

浏览器支持

本书中的示例使用了相对新的 HTML5 和 CSS3 的 API，因此希望使用现代的浏览器。在以下浏览器之前的版本并未经过全面的测试，可能不支持示例中用到的技术：

- IE 10+
- Firefox 28.0+
- Chrome 33.0+

应当熟悉如何在喜欢的浏览器中使用开发工具来修改潜在的错误，查看网络请求，以及使用 JavaScript 控制台。

附加软件

我们会在后面的章节中用到两种流行的数据库：PostgreSQL 和 Redis。在需要的章节中会给出简要的安装指导，但有关读者所使用系统的更完整的指南请查看官方文档。

PostgreSQL 是一个开源的关系型数据库系统，在 Django 社区中有着广泛的支持。所有 Django 支持的 PostgreSQL 版本在本书中都可以使用。Django 1.8 支持 PostgreSQL 9.0 及以上版本。

Redis 是一个开源的键值对缓存。本书中使用了 Redis 的 pub/sub 特性，因此需要 2.0 或更高版。

第 1 章

世界上最小的 Django 项目

我们中有多少人是从官方的注册教程中开启 Django 使用之旅的？对许多人来说这似乎只是一个过程，但作为一个介绍 Django 的入门就显得很困难了。我们需要运行各种命令、生成多个文件，甚至很难分辨出什么是项目，什么是应用。对于想要使用 Django 来创建应用的新手来说，他们会觉得选择一个 Web 框架有点太“重量级”了。要怎样才能让我们轻便、简单地开始，消除恐惧呢？

让我们来花点时间探询一下启动 Django 项目的推荐任务吧。通常创建新的项目要使用 `startproject` 命令。这个命令没有什么神奇的地方，只是创建一些文件和目录而已。

尽管 `startproject` 命令是一个很有用的工具，但却不是启动 Django 项目所必须的。可以根据需求，自由地设计自己的项目。对于大型项目，开发者会得益于 `startproject` 命令所提供的代码组织。然而，不应该因为这一命令的方便使用而忽略了它的工作和作用。

在本章中，我们将利用 Django 的基本生成模块设计一个创建简单项目的示例。这个轻量级的“Hello World”项目将使用单个文件的方法来创建一个简单的 Django 应用。

你好 Django

用一种新语言或者新框架创建一个“Hello World”示例是第一个通常项目。我们在 Flask 社区中看到这个简单的开始示例，以此来说明这是怎样一个轻量级的微型框架。

在本章中，我们将从一个 *hello.py* 文件开始。这一文件将包含运行 Django 项目所需的所有代码。为了得到能完整运行的项目，我们需要创建一个为根目录 URL 提供服务的视图，以及配置 Django 环境所需的设置。

创建视图

Django 是一个模型—模板—视图（*model-template-view*，MTV）框架。视图部分通常查看 HTTP 给出的请求和查询或者结构，这些信息是发送到表示层的数据。

让我们在 *hello.py* 文件中创建一个执行“Hello World”响应的简单方法。

```
from django.http import HttpResponse


def index(request):
    return HttpResponse('Hello World')
```

在大型项目中，这段代码通常会放在应用程序内的 *views.py* 文件中。不过，对于视图要放在应用内部什么位置，并没有规定。视图也不一定要放在 *views.py* 文件中。这更多是一种约定，但基于我们的项目结构，这并不是必需的。

URL 模式

要把视图绑定到网站结构上，需要将它与一个 URL 模式相关联。对于这个例子，服务器根目录会自己处理这个视图。Django 通过将一个正则表达式和可调用的参数相匹配与视图的 URL 进行关联。下面是一个在 hello.py 中创建关联的例子。

```
from django.conf.urls import url
from django.http import HttpResponse


def index(request):
    return HttpResponse('Hello World')


urlpatterns = (
    url(r'^$', index),
)
```

现在这个文件就把典型的 *views.py* 文件和根目录的 *urls.py* 文件合并起来了。同样的，需要注意 URL 模式不一定非要放在 *urls.py* 文件中。它们可以存在于任何可引用的 Python 模块中。

让我们进入到 Django 的设置阶段，并给出运行我们项目的几行简单代码。

设置

Django 设置详细说明了从数据库和缓存连接到国际化特性、静态和上传资源等方方面面。

对于许多刚上手的开发者，主要困惑来源于 Django 的设置。尽管近来的版本中已经在缩减默认设置文件的长度，但它还是有点过长。

这个例子会在调试模式下运行 Django。除此之外，Django 仅仅需要配置根目录 URL 的位置即可，它会使用该模块中 `urlpatterns` 文件定义的值。在 *hello.py* 的例子中，根目录的 URL 在当前模块中，它将会用到前面一节中定义的 `urlpatterns`。

```
from django.conf import settings


settings.configure(
    DEBUG=True,
    SECRET_KEY='thisisthesecretkey',
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)
...
```

这个例子中包括了一个固定的 `SECRET_KEY` 设置，在实际生产环境中不要使用这个设置。必须要生成用于默认会话和跨站点请求防伪（CSRF）保护密钥。对于所有产品站点来说，拥有一个保持私密的随机密钥 `SECRET_KEY` 非常重要。学习更多内容，请访问 *https://docs.djangoproject.com/en/1.7/topics/signing/* 中的文档。

在我们从 Django 导入更多内容之前，需要进行配置，因为框架的某些部分在引用之前需要进行配置。通常来说，这不会造成什么问题，因为这些设置会包含在它们自己的 *settings.py* 文件中。默认的 `startproject` 命令生成的文件也会包含这个例子中未使用的设置，例如国际化设置和静态资源。

运行示例

让我们看看在 `runserver` 中我们的例子运行的情况。典型的 Django 项目带有一个 *manage.py* 文件，这个文件用于运行如创建数据库表、启动开发服务器等多个命令。这个文件共有 10 行代码。我们会把这个文件中的相关部分加到 `hello.py` 中，来实现与 *manage.py* 相同的功能：

```
import sys


from django.conf import settings
```

```
settings.configure(
    DEBUG=True,
    SECRET_KEY='thisisthesecretkey',
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)


from django.conf.urls import url
from django.http import HttpResponse


def index(request):
    return HttpResponse('Hello World')


urlpatterns = (
    url(r'^$', index),
)


if __name__ == __main__'':
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

接下来可以在命令行中启动这个例子：

```
hostname $ python hello.py runserver
Performing system checks...

System check identified no issues (0 silenced).
August 06, 2014 - 19:15:36
Django version 1.7c2, using settings None
Starting development server at http://7.0.0.1:8000/
Quit the server with CONTROL-C.
```

并在选定的浏览器中访问 *http://localhost:8000/*，看到“Hello World”，如图 1-1 所示。

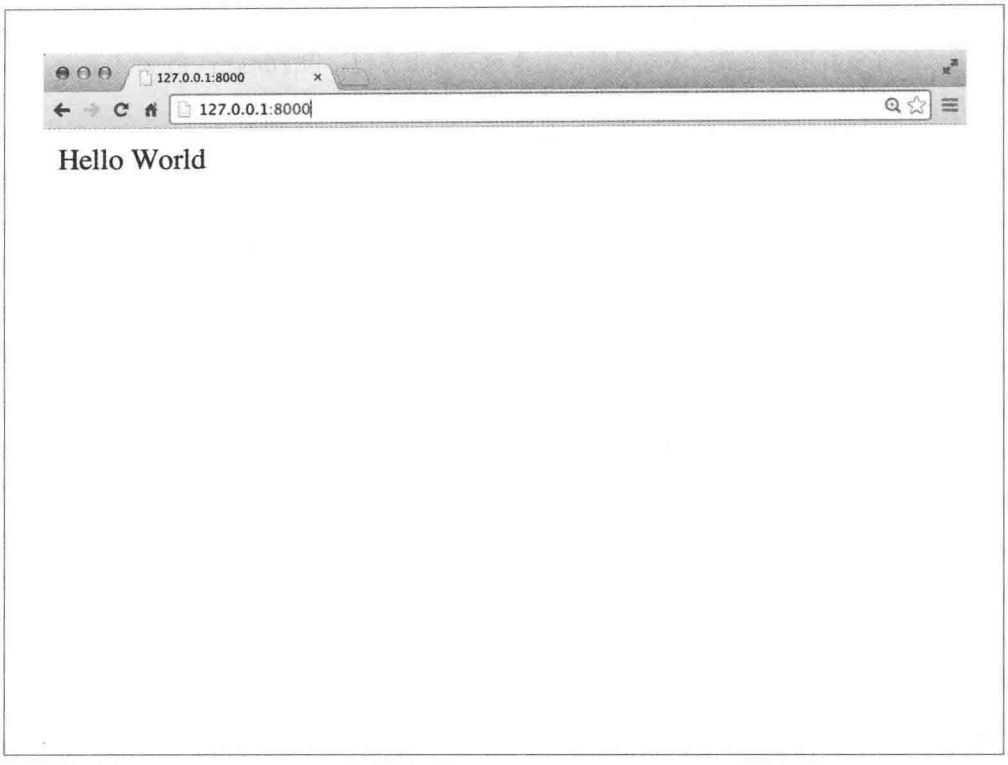


图 1-1：Hello World

现在，我们已经有了一个非常基础的文件结构，接下来继续向我们的文件中增加更多的元素。

改进

在这个例子中介绍了 Django 框架的一些基础知识，如编写视图、创建设置、运行管理命令等。Django 的核心是 Python 框架，用于处理 HTTP 请求并返回 HTTP 输出。至于过程中要干些什么，完全取决于你。

Django 还提供其他一些用于处理 HTTP 请求中所涉及的常见任务的工具，例如渲染 HTML、分析表单数据和持久保持会话状态等。尽管这并非是必要的，但重要的是要理解这些特性如何以一种轻量级的方式用在应用中。完成这些工作后，便可对 Django 的完整架构和它的实际能力有更好的理解。

WSGI 应用

到目前为止，我们的“Hello World”项目已经可以在 `runserver` 命令下运行了。这是一

个基于标准库中的 socket 服务器下的简单服务器。针对本地开发，它提供了诸如自动代码重载等许多有用的工具。不过虽然它对本地开发很方便，但 runserver 却不适合产品部署的安全性。

Web 服务器网关接口（WSGI）是一份有关 Web 服务器如何与 Django 一类的应用框架通信的规范，由 PEP 333 制定，并在 PEP 3333 中改进。有许多使用 WSGI 规范的 Web 服务器，包括 Apache 下的 mod_wsgi、Gunicorn、uWSGI、CherryPy、Tornado 和 Chaussette。

其中的每一个服务器都需要使用一个正确定义的 WSGI 应用。Django 通过 get_wsgi_application 提供了一个用于创建这个应用的简单接口。

```
...
from django.conf.urls import url
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponse
...
application = get_wsgi_application()

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

这个接口通常会包含在由 startproject 命令所创建的 *wsgi.py* 文件中。application 只是大多数 WSGI 服务器所用的一个约定俗成的名字，在需要的情况下也都可以在各自的配置选项中指定一个别的名称。

现在我们这个简单的 Django 项目已经准备好接入 WSGI 服务器了。Gunicorn 是纯 Python 的 WSGI 应用服务器中的热门选择，它拥有优良的性能，安装简便，并运行在 Python 3 下。Gunicorn 可以通过 Python 包索引（pip）来安装。

```
hostname $ pip install gunicorn==19.3.0
```

当 Gunicorn 安装完毕后，可以很简单地通过 gunicorn 命令来运行它：

```
hostname $ gunicorn hello --log-file=-
[2015-07-06 19:17:26 -0400] [37043] [INFO] Starting gunicorn 19.3.0
[2015-07-06 19:17:26 -0400] [37043] [INFO]
    Listening at: http://127.0.0.1:8000 (37043)
[2015-07-06 19:17:26 -0400] [37043] [INFO] Using worker: sync
[2015-07-06 19:17:26 -0400] [37046] [INFO] Booting worker with pid: 37046
```

正如输出所见，这个实例是在 Gunicorn 的 19.1.1 版本下运行的。显示的时间包含了开发者所在时区的时差，这个时差根据所处的位置会有所不同。评判器和工作机的进程 ID 也会有所区别。

自 19 版本起，Gunicorn 默认不再向控制台中输出日志。需要加上 **--log-file=-** 选项来开启向控制台输出日志的功能。可以在 *http://docs.gunicorn.org/en/19.1/* 中查询更多 Gunicorn 的设置。

如同 Django 中 runserver 命令一样，服务器会监听 *http://127.0.0.1:8000/*。这为我们进行更简便的配置带来了方便。

附加配置

尽管 Gunicorn 是为 Web 服务器准备的一款产品，但应用本身却还没有做好产品的准备，因为 DEBUG 永远不会在产品中启用。正如前面我们提到的，由于 SECRET_KEY 是非随机的，所以为了增加安全性，也应该随机生成。

想要了解更多关于 DEBUG 和 SECRET_KEY 设置的安全含义，请参看 Django 的官方文档。

这就引出了 Django 社区中一个常见的问题：一个项目该如何管理开发、阶段运行和形成产品环境下的不同设置呢？ Django 的维基（wiki）给出了一长串解决方案，其中有许多针对解决这一问题的可复用的应用。有关对这些应用的比较，可以在 Django Package 中找到。尽管这些选择在某些情况下可能是理想的，例如将 *settings.py* 文件转换成一个包并为每个环境创建模块，但它们并不适合我们例子中的单文件设置。

Twelve Factor App 是一个搭建、部署 HTTP 服务应用的方法。这一方法推荐使用单独的配置和代码，同时也要把这些配置储存到环境变量中。这样就简化了部署时对配置的更改，也让配置与操作系统无关。

下面让我们把这一方法应用到我们的 *hello.py* 例子中。这里只有两个设置可能会在不同环境下改变：DEBUG 和 SECRET_KEY。

```
import os
import sys

from django.conf import settings

DEBUG = os.environ.get('DEBUG', 'on') == 'on'

SECRET_KEY = os.environ.get('SECRET_KEY', os.urandom(32))
```

```
settings.configure(
    DEBUG=DEBUG,
    SECRET_KEY=SECRET_KEY,
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)
```

可能会注意到，DEBUG 的默认值是 True，如果没有设置 SECRET_KEY，每次应用载入时会随机生成一个值。对于这样一个虚构的示例是没问题的，不过如果应用中用到 Django 的某一部分需要 SECRET_KEY 保持不变，例如在设置 cookies 时，这样的设置经常会导致会话失效。

让我们来看看如何将其转换为启动应用的。要禁用 DEBUG 设置，我们需要将 DEBUG 这个环境变量设置为开启之外的其他值。在类 UNIX 系统（如 Linux、OS X、FreeBSD）中，环境变量是通过命令行中的 export 命令设置的。在 Windows 下，则是用 set 进行设置的。

```
hostname $ export DEBUG=off
hostname $ python hello.py runserver
CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.
```

正如在错误中看到的，我们的应用没有配置 ALLOWED_HOSTS 设置。ALLOWED_HOSTS 被用于验证 HTTP HOST 的标头值，应该被设置为一个可接受的 HOST 值的列表。如果应用是为了用于 example.com，ALLOWED_HOSTS 应该只允许请求 example.com 的客户。如果 ALLOWED_HOSTS 环境变量没有设置，那么它就会只允许来自 localhost 的请求。正如下面这段来自 *hello.py* 的代码片段所示。

```
import os
import sys


from django.conf import settings
DEBUG = os.environ.get('DEBUG', 'on') == 'on'

SECRET_KEY = os.environ.get('SECRET_KEY', os.urandom(32))

ALLOWED_HOSTS = os.environ.get('ALLOWED_HOSTS', 'localhost').split(',')

settings.configure(
    DEBUG=DEBUG,
    SECRET_KEY=SECRET_KEY,
    ALLOWED_HOSTS=ALLOWED_HOSTS,
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
```

```
    ),
)
```

借助完成的 ALLOWED_HOSTS 变量的设置，可以对引入的 HTTP HOST 标头的值进行验证。

要更加详细地了解 ALLOWED_HOSTS 设置，请参看 Django 官方文档。

在开发环境之外，应用可能需要为多个主机提供服务，如 localhost 和 example.com，所以要用逗号分隔配置我们指定的多个主机名。

```
hostname $ export DEBUG=off
hostname $ export ALLOWED_HOSTS=localhost,example.com
hostname $ python hello.py runserver
...
[06/Jul/2015 19:45:53] "GET / HTTP/1.1" 200 11
```

这里为我们提供了一个跨环境配置的灵活方法。虽然在改动如 INSTALL_APPS 或 MIDDLEWARE_CLASSES 等更加复杂的设置时会略微麻烦些，但这跟总的方法是一致的。这种方法提倡在不同环境下做最小的改动。

如果要在不同环境下做一些复杂的改动，要花些时间对应用的测试性以及部署会造成的影响加以考虑。

我们可以通过在 shell 中删除环境变量或者开启一个新的 shell 来重置 DEBUG。

```
hostname $ unset DEBUG
```

可复用模板

到目前为止，这个例子一直着重的是对由 Django 的 startproject 命令创建的布局进行重新设置。然而，该命令同时也允许使用模板来提供布局。将文件转换为可复用的模板，用相同的基本布局设计未来的项目是很容易的。

startproject 的模板是一个目录或 zip 文件，当命令运行时形成 Django 模板。默认情况下，所有 Python 源文件都会被制成模板。制作过程中会把 project_name、project_directory、secret_key 和 docs_version 作为上下文传递。文件名同样会被制作到这个

上下文中。要把 *hello.py* 转换到项目模板中（*project_name/project_name.py*），文件的相关部分需要用这些变量来替换。

```
import os
import sys

from django.conf import settings

DEBUG = os.environ.get('DEBUG', 'on') == 'on'

SECRET_KEY = os.environ.get('SECRET_KEY', '{{ secret_key }}')

ALLOWED_HOSTS = os.environ.get('ALLOWED_HOSTS', 'localhost').split(',')

settings.configure(
    DEBUG=DEBUG,
    SECRET_KEY=SECRET_KEY,
    ALLOWED_HOSTS=ALLOWED_HOSTS,
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)

from django.conf.urls import url
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponse

def index(request):
    return HttpResponse('Hello World')

urlpatterns = (
    url(r'^$', index),
)


application = get_wsgi_application()

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

接下来让我们把这个文件以*project_name.py*的名字保存在一个叫*project_name*文件夹下。此外，代替使用默认的 `os.urandom` 来创建 `SECRET_KEY`，这个代码会在每次新建项目时创建一个随机密钥的默认值。这将保证 `SECRET_KEY` 在项目层面上是固定的，也保证在项目间充分随机。

要通过 `startproject` 使用模板，可以使用 `--template` 参数。

```
hostname $ django-admin.py startproject foo --template=project_name
```

这条命令会在 foo 文件夹下创建一个 *foo.py* 文件，可以像原先的 *hello.py* 文件一样执行。

正如在示例中简要说明的，我们完全可以不用 `startproject` 命令来编写、运行 Django 项目。Django 的默认设置和布局并不适用所有项目。`startproject` 中的 `--template` 选项可以用于扩展或压缩这些默认设置，正如本章所介绍的。

同 Python 的所有项目一样，把代码分割成多个模块是过程中很重要的一部分。对于这样一个充分关注细节并只有几个 URL 的站点，我们的“Hello World”示例也是一个可行的方案。

这一方案另外一个有趣的地方是，Django 内置一个模板引擎或对象关系映射（ORM）不能直接显示。很明显可以自由选择 Python 库，只要它能很好地解决问题，不一定要按照官方教程说明的样子使用 Django 的 ORM，相反，可以依照自己的喜好使用 ORM。在下一章的项目中会将这个单文件示例扩展为提供简单的 HTTP 服务，并使用 Django 中更多的工具。

第 2 章

无状态的 Web 应用

大多数 Django 应用或教程都是围绕各类用户生成的内容来进行的，例如待办事项、博客和内容管理系统。因为 Django 起源于传媒业，所示这并不奇怪。

2005 年，Django 最初由位于堪萨斯州劳伦斯的世界在线所开发，作为记者们快速创建 Web 内容的一种方式。自此之后，它被用于多个出版组织，例如华盛顿邮报、卫报、PolitiFact 和 Onion。由于 Django 在这方面的应用，可能会让人们认为它主要用于内容的发布，或 Django 只是一个内容管理系统。但随着诸如 NASA 这样的大型组织采用 Django 作为它们的框架，Django 显然已经超出它原有的应用。

在前一章我们创建的一个很小的项目，只用到了 Django 核心的 HTTP 操作和 URL 路由技术。在本章中我们将扩展该示例，创建一个无状态的 Web 应用，这个应用会用到包括输入验证、缓存和模板在内的更多 Django 功能。

什么是无状态？

HTTP 本身是一个无状态的协议，意思是每一个到达服务器的请求都独立于之前的请求。如果需要某个特定的状态，则需把它加到应用层上。像 Django 这样的框架使用 Cookie 以及其他机制将同一客户端发送的请求绑定在一起。

在存储于服务器上的持久会话的帮助下，应用程序可以处理许多任务，如在多个请求间保持用户权限验证。但这也带来许多挑战，例如，像在分布式服务器架构中每种请求在这种一致状态下的读取和可能的写入问题。

正如你所能想象的，无状态的应用并不在服务器上维持一致的状态。假如需要进行权限或其他的用户资格验证，就必须经由客户端传递每个请求。通常这会使无状态的应用十

分易于扩展、缓存，以及负载均衡。把无状态的应用链接化也得简单，因为 URL 可以传输大多数状态。我们可以采用可复用应用和可组合服务两种方式，下面一节将这些内容加以介绍。

可复用应用与可组合服务

Django 社区的大部分关注点集中在构建可复用的应用上，这种应用可安装配置到任何 Django 项目。然而，拥有许多不同组件的大型应用，通常具有相当复杂的体系结构。

抵御复杂架构的一种方法就是将大型网站分解成可组合的服务，即一个个相互关联的细小服务。但这并不意味着它们在某些地方不能或不会共享代码，而是意味着每个服务都可以被单独地配置和构建。

无状态组件（如 REST API）都是非常适合的备用选项，它们都可用于创建能独立部署和调试的 Django 项目。让我们利用 Django 刚刚描述的两种技术，通过创建占位图片服务器来构建第一章的项目。

占位图片服务器

占位图片经常被用于应用程序原型、实例项目或者测试环境中。一个典型的占位图片服务器会接收一个指定图片大小的 URL 并生成该图片。URL 可能会携带其他信息，如图片的颜色或图片中显示的文本。由于所有创建请求图片所需的信息都包含在 URL 里，不需要进行权限验证，所以是一个很好的无状态应用的候选。

我们首先借助第一章创建的 `project_name` 模板，利用 `startproject` 创建一个称为 `placeholder` 的新项目。

```
hostname $ django-admin.py startproject placeholder --template=project_name
```

这里会生成一个 *placeholder.py* 文件供我们开始搭建服务。如果正确地使用了项目模板，*placeholder.py* 应该像下面这样：

```
import os
import sys

from django.conf import settings

DEBUG = os.environ.get('DEBUG', 'on') == 'on'

SECRET_KEY = os.environ.get('SECRET_KEY',
    '%jv_4#hoaqwig2gu!eg#^ozptd*a@88u(aasv7z!7xt^5(*i&k')
```

```
ALLOWED_HOSTS = os.environ.get('ALLOWED_HOSTS', 'localhost').split(',')

settings.configure(
    DEBUG=DEBUG,
    SECRET_KEY=SECRET_KEY,
    ALLOWED_HOSTS=ALLOWED_HOSTS,
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)

from django.conf.urls import url
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponse

def index(request):
    return HttpResponse('Hello World')

urlpatterns = (
    url(r'^$', index),
)

application = get_wsgi_application()

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

SECRET_KEY 设置会不同于这个已发布的版本，因为它是由 startproject 命令随机生成的。

在完成初始设置后，就可以开始编写视图和制作响应这些视图的页面了。

视图

因为这个应用很简单，我们只使用两个视图来生成响应。第一个视图用来按照请求的宽度和高度渲染占位图像。另一个视图用来渲染主页面的内容，解释项目运作以及生成一些样例图片的方法。由于我们在运行 startproject 时使用了 Django 的 --template 标记，已经创建完成（见 *placeholder.py* 的代码片段）了 index，用于之后的调整。

```
...
def placeholder(request, width, height):
    # TODO: Rest of the view will go here
    return HttpResponse('Ok')

def index(request):
    return HttpResponse('Hello World')
...
```

完成这些简单的视图后，下面应该思考显示占位图片的 URL 结构了。

URL 模式

当打开生成的 *placeholder.py* 文件时，会注意到那里有个关于服务器根目录的 URL 模式。另外，我们还需要一个指向刚刚创建的 `placeholder` 视图的路由。

`placeholder` 视图还会用到两个参数：`width` 和 `height`。正如前面所提到的，这些参数会由 URL 捕获并传递给视图。由于它们只能是整数类型，要通过 URL 对它们进行强制转换。因为 Django 的 URL 模式使用正则表达式来匹配输入的 URL，可以很轻易地传递这些参数。

捕获的模式组会以位置参数的形式传递给视图，命名的组会以关键字参数的形式传递。通过 `?P` 语法来捕获被命名的组，并用 `[0-9]` 来匹配任意数字字符。

下面的代码片段展示了如何布置 URL 式样生成这些数值的方法：

```
...
urlpatterns = (
    url(r'^image/(?P<width>[0-9]+)x(?P<height>[0-9]+)/$',placeholder,
        name='placeholder'),
    url(r'^$', index, name='homepage'),
)
...
```

完成这些模式的设置后，像 */image/30x25/* 这样的 URL 的请求会被路由到 `placeholder` 视图中，并传递这些数值（如 `width=30`、`height=25`）。同时以 `homepage` 为名称与 `placeholder` 视图新的路由一起被加入到 index 路由上。通常，给 URL 模式命名是个很好的习惯，我们将看到在后面开始创建模板时这种命名会格外有用。

占位视图

除了最初的 HTTP 请求，占位视图还要接收两个整数参数来设置图片的高度和宽度。尽管正则表达式会确保高度和宽度只包含整数，但它们还是会以字符串的形式传递给视图。

视图需要对它们进行转换并确保它们在一个可以管理的尺寸内。我们可以使用 Django 表单来轻松实现用户输入验证。

典型的表单用于验证 POST 和 GET 内容，但它们也可用于验证 URL 或 cookie 的特定值。下面是一个来自于 *placeholder.py* 文件的简单表单，它用来验证图片的高度和宽度：

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

class ImageForm(forms.Form):
    """Form to validate requested placeholder image."""

    height = forms.IntegerField(min_value=1, max_value=2000)
    width = forms.IntegerField(min_value=1, max_value=2000)

def placeholder(request, width, height):
...
```

如你所见，视图所做的第一件事就是验证请求图片的尺寸。如果表单有效，那么高度和宽度就可以通过表单的 `clean_data` 属性来得到。在这里，高度和宽度会被转换成整数型，视图会保证这些值介于 `1` 到 `2000` 之间。我们还需要在表单上加入验证，以便在数值不正确时发送错误信息，参见下面这段来自 *placeholder.py* 文件的代码：

```
...
from django import forms
from django.conf.urls import url
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponse, HttpResponseBadRequest

class ImageForm(forms.Form):
    """Form to validate requested placeholder image."""

    height = forms.IntegerField(min_value=1, max_value=2000)
    width = forms.IntegerField(min_value=1, max_value=2000)


def placeholder(request, width, height):
    form = ImageForm({'height': height, 'width': width})
    if form.is_valid():
        height = form.cleaned_data['height']
        width = form.cleaned_data['width']
        # TODO: Generate image of requested size
        return HttpResponse('Ok')
    else:
        return HttpResponseBadRequest('Invalid Image Request')
...
```

如果表单无效，视图会发送一个错误信息返回给客户端。这里视图返回了一个 HttpResponseBadRequest 信息，它是一个 HttpResponse 的子类，发送了一个 400 Bad Request 的响应。

图片处理

尽管现在视图已经可以接收并处理来自客户端的高度和宽度的请求，但它还不能生成实际的图片。要在 Python 中处理图片，需要像下面这样安装 Pillow：

```
hostname $ pip install Pillow
```

默认情况下，安装会尝试从源代码编译 Pillow。如果环境中没有安装编译器，或是丢失了一些必要的头文件，安装会失败。有关不同平台上的安装说明，参见 *https://pillow.readthedocs.org/en/latest/installation.html*。

通过 Pillow 创建图片需要两个参数：一个以元组表示的颜色模式和尺寸。这个 *placeholde.py* 中的视图使用 RGB 模式和在表单中处理过的数据尺寸。这里还有第三个可选参数，用来设置图片的颜色。在 Pillow 中，图片上的每个像素都默认为黑色。

```
...
from io import BytesIO                                                    ❶
from PIL import Image                                                     ❷
...

class ImageForm(forms.Form):
    """Form to validate requested placeholder image."""

    height = forms.IntegerField(min_value=1, max_value=2000)
    width = forms.IntegerField(min_value=1, max_value=2000)

    def generate(self, image_format='PNG'):                               ❸
        """Generate an image of the given type and return as raw bytes."""
        height = self.cleaned_data['height']
        width = self.cleaned_data['width']
        image = Image.new('RGB', (width, height))                         ❹
        content = BytesIO()
        image.save(content, image_format)
        content.seek(0)
        return content


def placeholder(request, width, height):
    form = ImageForm({'height': height, 'width': width})
    if form.is_valid():
        image = form.generate()                                           ❺
```

```
            return HttpResponse(image, content_type='image/png')
        else:
            return HttpResponseBadRequest('Invalid Image Request')
    ...
```

❸ ImageForm 中加入了一个新的 generate 方法，用于封装创建图片的逻辑。它接收一个参数来设置图片格式，默认为 PNG，并以字节的形式返回图片内容。

❷❹ 使用 URL 给定并由表单确定的宽度和高度，通过 Pillow 的 Image 类创建一张新图片。

❶❺ 视图调用 form.generate 来获取创建的图片，图片的字节被用于之后创建响应体。

接着表单会验证尺寸以防止请求过大的图片消耗过多服务器资源。一旦图片被验证通过，视图会成功返回一张请求宽度和高度的 PNG 图片。图片的内容不写入磁盘，直接发送给客户端。

然而，一张全黑的图片，没有任何尺寸信息，不是一个很漂亮或实用的占位图片。通过 Pillow 我们可以使用 ImageDraw 模块在图片中加入文字，如下面这段 *placeholder.py* 中的代码所示：

```
...
from PIL import Image, ImageDraw                                    ❶
...

class ImageForm(forms.Form):
    ...
    def generate(self, image_format='PNG'):
        """Generate an image of the given type and return as raw bytes."""
        height = self.cleaned_data['height']
        width = self.cleaned_data['width']
        image = Image.new('RGB', (width, height))
        draw = ImageDraw.Draw(image)                                ❷
        text = '{} X {}'.format(width, height)
        textwidth, textheight = draw.textsize(text)
        if textwidth < width and textheight < height:
            texttop = (height - textheight) // 2
            textleft = (width - textwidth) // 2
            draw.text((textleft, texttop), text, fill=(255, 255, 255))
        content = BytesIO()
        image.save(content, image_format)
        content.seek(0)
        return content
...
```

❶❷ generate 使用 ImageDraw 向尺寸合适的地方加入覆盖文字。

通过 ImageDraw，表单使用当前图片来创建在图片上显示宽度和高度的文本。

既然我们已经得到了有效的占位图片，让我们加一些缓存来帮助最小化服务器请求。

加入缓存

每次请求视图时，占位图片视图都会重新生成图片。由于图片的宽度和高度是通过最初值来设置的，因此我们常常对服务器提出了不必要的请求。

缓存是避免这种重复的一种办法。当要确定如何为服务提供缓存时，有两种方案可供考虑：服务器端和客户端。对于服务器端缓存，可以方便地使用 Django 的缓存工具。这会把内存开销转换为缓存存储，同时会节约生成图片所需的 CPU 周期，如下面这段 *placeholder.py* 中的代码所示：

```
...
from django.conf.urls import url
from django.core.cache import cache                                    ❶
...
class ImageForm(forms.Form):
...
    def generate(self, image_format='PNG'):
        """Generate an image of the given type and return as raw bytes."""
        height = self.cleaned_data['height']
        width = self.cleaned_data['width']
        key = '{}.{}.{}'.format(width, height, image_format)           ❷
        content = cache.get(key)                                       ❸
        if content is None:
            image = Image.new('RGB', (width, height))
            draw = ImageDraw.Draw(image)
            text = '{} X {}'.format(width, height)
            textwidth, textheight = draw.textsize(text)
            if textwidth < width and textheight < height:
                texttop = (height - textheight) // 2
                textleft = (width - textwidth) // 2
                draw.text((textleft, texttop), text, fill=(255, 255, 255))
            content = BytesIO()
            image.save(content, image_format)
            content.seek(0)
            cache.set(key, content, 60 * 60)                           ❹
  return content
```

❷ 通过宽度、高度和图片格式生成一个缓存键值。

❶❸ 在新图片创建前，检查缓存是否已经存储了图片。

❹ 当未找到缓存图片且创建了新图片时，通过键值将图片在缓存中保存一小时。

Django 默认使用本地过程、内存缓存，但也可以使用不同的后端（像 Memcached 或文件系统），这要通过 `CACHES` 设置来配置。

还有一个补充方案，就是关注客户端性能并使用浏览器内建的缓存。Django 引入了一个创建并使用视图的 ETag 标头的 `etag` 修饰符。该修饰符接收一个参数，一个从请求和视

图参数中生成 ETag 标头的函数。下面是一个 *placeholder.py* 中的样例，展示了如何把它加入到视图中：

```
import hashlib                                                    ❶
import os
...
from django.http import HttpResponse, HttpResponseBadRequest
from django.views.decorators.http import etag                     ❷
...

def generate_etag(request, width, height):                        ❸
    content = 'Placeholder: {0} x {1}'.format(width, height)
    return hashlib.sha1(content.encode('utf-8')).hexdigest()


@etag(generate_etag)                                              ❹
def placeholder(request, width, height):
...
```

❶❸ generate_etag 是个新的函数，接收 placeholder 视图中的参数。它使用 hashlib 来返回一个基于 width 和 height 值变化的不透明的 ETag 值。

❷❹ generate_etag 函数会被发送到 placeholder 视图的 etag 修饰符中。

使用这个修饰符，服务器将在浏览器第一次请求时生成该图片。在后续的请求中，如果浏览器发送了一个匹配 ETag 的请求，它将会收到 304 Not Modified 作为对图片的响应。浏览器会使用缓存的图片，这可以节省带宽以及重新生成 HttpResponse 的时间。

这个视图仅仅基于宽度和高度创建图像。如果加入了背景色、图片文本等其他特性，生成 ETag 时也需要把这些纳入考虑。

django.middleware.common.CommonMiddleware 在 MIDDLEWARE_CLASSES 中被启用，如果 USE_ETAGS 设置被启用，同样也支持 ETag 的创建和使用。然而中间件和修饰符在工作上有些不同。中间件会基于响应内容的 md5 哈希值来计算 ETag。这就需要视图完成创建图片的所有工作以用于计算哈希值。结果是浏览器同样会收到 304 Not Modified 响应，并且也会节省带宽。使用 etag 修饰符具有在视图被访问之前进行 ETag 计算的优势，这样还可以节约计算时间和资源。

下面是 *placeholder.py* 中完整的 placeholder 视图，以及表单和修饰符函数：

```
...
class ImageForm(forms.Form):
    """Form to validate requested placeholder image."""
```

```
    height = forms.IntegerField(min_value=1, max_value=2000)
    width = forms.IntegerField(min_value=1, max_value=2000)

    def generate(self, image_format='PNG'):
        """Generate an image of the given type and return as raw bytes."""
        height = self.cleaned_data['height']
        width = self.cleaned_data['width']
        key = '{}.{}.{}'.format(width, height, image_format)
        content = cache.get(key)
        if content is None:
            image = Image.new('RGB', (width, height))
            draw = ImageDraw.Draw(image)
            text = '{} X {}'.format(width, height)
            textwidth, textheight = draw.textsize(text)
            if textwidth < width and textheight < height:
                texttop = (height - textheight) // 2
                textleft = (width - textwidth) // 2
                draw.text((textleft, texttop), text, fill=(255, 255, 255))
            content = BytesIO()
            image.save(content, image_format)
            content.seek(0)
            cache.set(key, content, 60 * 60)
        return content


def generate_etag(request, width, height):
    content = 'Placeholder: {0} x {1}'.format(width, height)
    return hashlib.sha1(content.encode('utf-8')).hexdigest()


@etag(generate_etag)
def placeholder(request, width, height):
    form = ImageForm({'height': height, 'width': width})
    if form.is_valid():
        image = form.generate()
        return HttpResponse(image, content_type='image/png')
    else:
        return HttpResponseBadRequest('Invalid Image Request')
...
```

在完成占位图片视图后，让我们回头来创建主页面视图完成我们的应用。

创建主页面视图

主页面会渲染一个基本的 HTML 模板，用于解释项目的运行情况并包含一些示例图片。到目前为止，还没有对 Django 渲染模板的功能进行配置。也没有对资源文件进行配置，例如 JavaScript、CSS 和模板等。让我们对这些静态资源进行一些必要的设置：TEMPLATE_DIRS 和 STATICFILES_DIRS。

在 Django 1.8 中增加 TEMPLATES 设置，以便在单个项目中配置多个模板引擎。这将替换旧的 TEMPLATE_* 设置。

加入 Static 和 Template 设置

Django 的模板加载器会自动寻找安装应用中的模板和静态资源。由于这个项目并没有引入任何 Django 应用，因此我们需要在 TEMPLATE_DIRS 和 STATICFILES_DIRS 中手动配置这些路径。django.contrib.staticfiles 同样需要被加入到 INSTALLED_APPS 中，这样 {% static %} 标记以及 collectstatic 命令才能有效。

下面给出了一个目录此时应该具备的结构：

```
placeholder/
    placeholder.py
```

模板和静态资源会准确地存放于目录中靠近 *placeholder.py* 文件的地方，像这样：

```
placeholder/
    placeholder.py
    templates/
        home.html
    static/
        site.css
```

为了避免对这些路径的硬性编码，我们将使用 Python 标准库里的 os 模块，创建相对于 *placeholder.py* 文件的路径。

```
import hashlib
import os
import sys

from django.conf import settings

DEBUG = os.environ.get('DEBUG', 'on') == 'on'

SECRET_KEY = os.environ.get('SECRET_KEY',
    '%jv_4#hoaqwig2gu!eg#^ozptd*a@88u(aasv7z!7xt^5(*i&k')

BASE_DIR = os.path.dirname(__file__)

settings.configure(
    DEBUG=DEBUG,
    SECRET_KEY=SECRET_KEY,
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
```

```
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
    INSTALLED_APPS=(
        'django.contrib.staticfiles',
    ),
    TEMPLATES=(
        {
           'BACKEND': 'django.template.backends.django.DjangoTemplates',
           'DIRS': (os.path.join(BASE_DIR, 'templates'),),
        },
    ),
    STATICFILES_DIRS=(
        os.path.join(BASE_DIR, 'static'),
    ),
    STATIC_URL='/static/',

...
```

现在，让我们加入一个简单的模板结构，以一个简洁的前端界面完成这个示样项目。

主页面模板和 CSS

这个项目首页的作用是利用一些简单的文档和示例，展示如何使用占位图片。它应当被存放于 *template/home.html* 目录中邻接的 *placeholder.py* 文件。我们还将把 `src` 引用加入到占位路由来请求这些图片。

```
{% load staticfiles %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Django Placeholder Images</title>
    <link rel="stylesheet" href="{% static 'site.css' %}" type="text/css">
</head>
<body>
    <h1>Django Placeholder Images</h1>
    <p>This server can be used for serving placeholder
    images for any web page.</p>
    <p>To request a placeholder image of a given width and height
    simply include an image with the source pointing to
    <b>/image/&lt;width&gt;x&lt;height&gt;/</b>
    on this server such as:</p>
    <pre>
        &lt;img src="{{ example }}" &gt;
    </pre>
    <h2>Examples</h2>
    <ul>
        <li><img src="{% url 'placeholder' width=50 height=50 %}"></li>
        <li><img src="{% url 'placeholder' width=100 height=50 %}"></li>
        <li><img src="{% url 'placeholder' width=50 height=100 %}"></li>
    <ul>
</body>
</html>
```

这里是一些加入到 *site.css* 中的简单样式，帮助创建一个简洁的布局。此外，不要忘记把这些文件存放到 *static/* 文件夹下，正如之前在 STATICFILES_DIRS 中设置的：

```
body {
text-align: center;
}

ul {
list-type: none;
}

li {
    display: inline-block;
}
```

像先前讲述的一样，需要在 *placeholder.py* 文件旁边把这个文件存为 *static/site.css*。最后，我们需要在 *placeholder.py* 中更新 index 视图来渲染这个模板：

```
...
from django.core.cache import cache
from django.core.urlresolvers import reverse                          ❶
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponse, HttpResponseBadRequest
from django.shortcuts import render                                   ❷
from django.views.decorators.http import etag
...
def index(request):
    example = reverse('placeholder', kwargs={'width': 50, 'height':50})   ❸
    context = {
        'example': request.build_absolute_uri(example)
    }
    return render(request, 'home.html', context)                      ❹
...
```

❶❸ 更新过的 index 视图通过翻转 placeholder 视图来创建一个 URL 样例，并将它传给模板上下文。

❷❹ 通过 render 快捷方式来渲染 *home.html* 模板。

完整的项目

到目前为止，应该得到了一个同下面的文件相类似的完整的 *placeholder.py* 文件。再加上之前创建的 *home.html* 和 *site.css* 文件，我们已经完成了一个简单的 Django 占位图片服务：

```
import hashlib
import os
import sys
```

```
from io import BytesIO
from PIL import Image, ImageDraw

from django.conf import settings

DEBUG = os.environ.get('DEBUG', 'on') == 'on'

SECRET_KEY = os.environ.get('SECRET_KEY',
    '%jv_4#hoaqwig2gu!eg#^ozptd*a@88u(aasv7z!7xt^5(*i&k')

ALLOWED_HOSTS = os.environ.get('ALLOWED_HOSTS', 'localhost').split(',')


BASE_DIR = os.path.dirname(__file__)

settings.configure(
    DEBUG=DEBUG,
    SECRET_KEY=SECRET_KEY,
    ALLOWED_HOSTS=ALLOWED_HOSTS,
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
    INSTALLED_APPS=(
        'django.contrib.staticfiles',
    ),
    TEMPLATES=(
        {
            'BACKEND': 'django.template.backends.django.DjangoTemplates',
            'DIRS': (os.path.join(BASE_DIR, 'templates'),)
        },
    ),
    STATICFILES_DIRS=(
        os.path.join(BASE_DIR, 'static'),
    ),
    STATIC_URL='/static/',
)

from django import forms
from django.conf.urls import url
from django.core.cache import cache
from django.core.urlresolvers import reverse
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponse, HttpResponseBadRequest
from django.shortcuts import render
from django.views.decorators.http import etag


class ImageForm(forms.Form):
    """Form to validate requested placeholder image."""

    height = forms.IntegerField(min_value=1, max_value=2000)
    width = forms.IntegerField(min_value=1, max_value=2000)
```

```
    def generate(self, image_format='PNG'):
        """Generate an image of the given type and return as raw bytes."""
        height = self.cleaned_data['height']
        width = self.cleaned_data['width']
        key = '{}.{}.{}'.format(width, height, image_format)
        content = cache.get(key)
        if content is None:
            image = Image.new('RGB', (width, height))
            draw = ImageDraw.Draw(image)
            text = '{} X {}'.format(width, height)
            textwidth, textheight = draw.textsize(text)
            if textwidth < width and textheight < height:
                texttop = (height - textheight) // 2
                textleft = (width - textwidth) // 2
                draw.text((textleft, texttop), text, fill=(255, 255, 255))
            content = BytesIO()
            image.save(content, image_format)
            content.seek(0)
            cache.set(key, content, 60 * 60)
        return content


def generate_etag(request, width, height):
    content = 'Placeholder: {0} x {1}'.format(width, height)
    return hashlib.sha1(content.encode('utf-8')).hexdigest()


@etag(generate_etag)
def placeholder(request, width, height):
    form = ImageForm({'height': height, 'width': width})
    if form.is_valid():
        image = form.generate()
        return HttpResponse(image, content_type='image/png')
    else:
        return HttpResponseBadRequest('Invalid Image Request')


def index(request):
    example = reverse('placeholder', kwargs={'width': 50, 'height':50})
    context = {
        'example': request.build_absolute_uri(example)
    }
    return render(request, 'home.html', context)


urlpatterns = (
    url(r'^image/(?P<width>[0-9]+)x(?P<height>[0-9]+)/$', placeholder,
        name='placeholder'),
    url(r'^$', index, name='homepage'),
)

application = get_wsgi_application()

if __name__ == "__main__":
    from django.core.management import execute_from_command_line
```

```
    execute_from_command_line(sys.argv)
```

现在让我们继续使用 runserver 来查看新创建的占位图片服务：

```
hostname $ python placeholder.py runserver
```

将会看到如图 2-1 所示的屏幕。

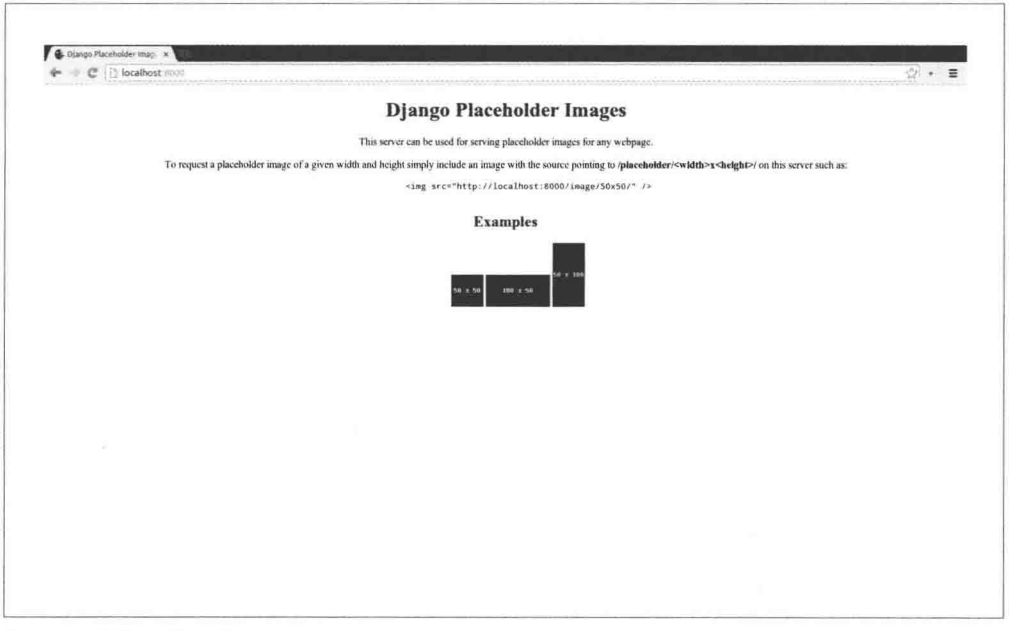


图 2-1：占位图片屏幕截图

如果 *site.css* 返回一个 404 错误，那么请检查是否启用了 DEBUG 设置。在 DEBUG 设置为 False 的情况下，Django 的开发服务器并不处理静态资源。或许还要取消 DEBUG 环境变量的设置，参见之前章节的介绍。

现在这个项目不再是个玩具“Hello World”示例了。它是一个 Django 创建的具有完整功能的 Web 应用。这个项目展示了如何通过 Django 以一种极轻量级的方式开发一些相当复杂的系统。它不仅使用到了 Django 提供的 HTTP 工具，还使用了内置的输入验证、缓存、设置模板，以及静态文件管理。下一章，我们将会重点讨论创建另外一个轻量级示例——运用 Django 构建一个静态站点生成器。

第 3 章

创建静态站点生成器

Django 作为框架被大家所称道的一个原因是“追赶工期的完美主义者”。正如我们在第 2 章里提到的，它最初是源于新闻编辑室，所以记者们可以在忙碌中快速地生产、编辑、发布新闻稿件。自从 Django 于 2005 年 7 月开源以来，已被大量 Web 开发者用于快速创建各种规模的网站和应用。作为当时的第一个这样的产品，现已成长为 Web 开发的主力。现在我们随处可见人们在使用它，无论是个人站点，还是像 Pinterest、Instagram 这样的大型应用。

虽然开始项目前，选择一个最好的架构是一项重要的内容，然而决定一个合适的工作流同样要优先考虑。想要得到好的最终产品，关键是要让设计师和开发者有效地合作。不过，当设计师和开发者合作办公时，Django 的效率如何呢？

我们发现有一种方式能让设计师和开发者很好地合作，就是让他们同时开始该项目。设计师和开发者两个小组都能够迅速行动起来开始利用框架中各自所需的部分。在我们的 Django 项目中创建一个静态站点应用，是对实现这一目标的一种有益的指导。

使用 Django 创建静态站点

大多数静态站点的例子都包括如何创建博客站点，如 Jekyll 或 Hyde。这通常包含一系列简易的基础模板、URL 模式和用于每个静态页面的文件结构。尽管 Django 最初目的是为了在快节奏环境下工作，但它并不像大多数开发者设想的那样是一款用于创建静态站点的框构。在本章中，我们将带你使用 Django 开发一个快速原型站点，以便快速地生成网站和应用。

什么是快速原型？

在一个大型团队里开发 Web 应用时，过程往往是非常重要的一项内容。然而为了达成最终产品，过程中很快会产生大量不必要的步骤。快速原型流程就是一种摆脱这些步骤，能更加有效地实现最终目标的方式。这一原则与设计 Django 的目的是一致的。

Bermon Painter 在他题为“线框图已逝，快速原型永存”的演讲中，曾列举了快速原型过程的 5 个步骤：

1. **观察与分析**。这一步是要明确最终用户的目标。在这个阶段团队的所有成员都在极尽脑力，思考所要做的一切（例如：谁是用户？每个用户想从产品中得到什么）。
2. **开发**。借助 HTML、CSS 和 JavaScript，团队合力只用到灰盒示例和简单的布局开发出一个最小可行产品（minimum viable product，MVP）。
3. **部署**。创建一种无缝的方式，让团队中每个能接触代码的成员均可配置和部署代码，以便查看或者测试。
4. **采纳和培训**。指导用户如何使用新功能，以及有关产品的设计，并听取他们的反馈。
5. **迭代和维护**。采纳用户的反馈，并把它们迭代回快速原型过程。这有助于对终端产品精益求精，得到更好的结果。此外，还要记得维护代码以跟进最新发布的程序包。

Django 在哪里，又是以怎样的方式来适应快速原型的工作流程呢？在本章接下来的篇幅中，我们会以创建静态内容的形式，深入讲解如何在 Django 项目中实现快速原型过程中的开发和部署。我们将要开发一个通用的网站来帮助理解创建一个作为快速原型工具的静态站点所需的过程。

最初的项目结构

正如在前两章中所做的，我们将使用单个文件方法来生成设置和其他运行项目所需的组件。我们还需要创建一些其他典型的 Django 项目文件，通常这些都是使用 `startproject` 命令生成的。让我们从生成最初的架构开始。

文件（目录）架构

我们的快速原型站点只需要很少一些基础元素就可以运行，如视图、URL 和模板。我们从创建站点的基础文件结构开始。创建一个名为 *sitebuilder* 的文件夹，并将 *init.py*、views.*py*、*urls.py* 和 *templates* 文件夹放入该文件夹。

我们还需要在*sitebuilder*的根目录下加入一个名为*static*的文件夹，来存放所需要的图片、CSS 文件和 JavaScript 文件。同时还要创建空的*js*文件夹和*css*文件夹，并把它们放到 templates 文件夹中。

基于在前两章中已经学习到的内容，我们将使用单文件方法来存储应用所需的设置和其他配置。设置会被存放在一个名为*prototypes.py*的文件中。到现在为止，应该创建了像下面的文件结构这样的一系列文件夹：

```
prototypes.py
sitebuilder
    __init__.py
    static/
        js/
        css/
    templates/
urls.py
views.py
```

接着我们要开始添加一些简单的设置，让项目运行起来。

基本设置

现在我们已经创建了文件布局，只需要再进行一点设置就可以让我们的项目运行起来。在*prototypes.py*文件中加入下面的基本设置，同时把我们的 sitebuilder 应用加到 INSTALLED_APPS 设置里。

```
import sys

from django.conf import settings


settings.configure(
    DEBUG=True,
    SECRET_KEY='bOmqvak1p2sqm6p#+8o8fyxf+ox(le)8&jh_5^sxa!=7!+wxj0',
    ROOT_URLCONF='sitebuilder.urls',
    MIDDLEWARE_CLASSES=(),
    INSTALLED_APPS=(
        'django.contrib.staticfiles',
        'sitebuilder',
    ),
    TEMPLATES=(
        {
            'BACKEND': 'django.template.backends.django.DjangoTemplates',
            'DIRS': [],
            'APP_DIRS': True,
        },
    ),
```

```
    STATIC_URL='/static/',
)

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

execute_from_command_line(sys.argv)
```

在 Django 1.8 中，`django.contrib.webdesign` 会被弃用，`{% lorem %}` 则成为内建模板标签。这意味着 `django.contrib.webdesign` 会被移出 Django2.0 发行版。

我们还需要在 *urls.py* 中加入 URL 设置来完成我们最初的项目结构。打开文件（*/sitebuilder/urls.py*），加入以下一行代码创建基础设置：

```
urlpatterns = ()
```

现在让我们尝试启动项目来进行一次快速的完整性检查，确保一切都正常运行。

```
hostname $ python prototype.py runserver
Performing system checks...

System check identified no issues (0 silenced).
July 06, 2015 - 22:01:47
Django version 1.8.3, using settings None
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

当在浏览器中访问 *http://127.0.0.1:8000*，将会看到如图 3-1 所示的界面。

有了基础设置并将项目运行起来后，就可以开始创建模板结构和修饰我们的原型页面了。

修饰页面

在绝大多数 Django 项目中，创建一个基础模板是搭建前端结构的重要一环。完成基础创建后，就可以继续构建余下的模板了。

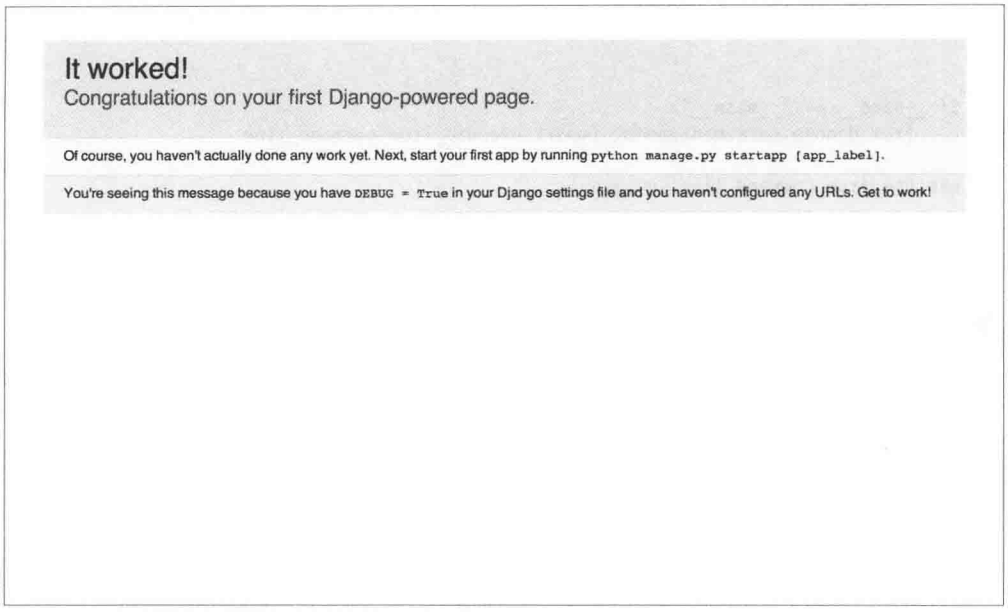


图 3-1：最初的项目界面

创建基础模板

和其他静态站点生成器一样，通常整个站点是从局部基础模板搭建起来的。这些模板会包含一些通用布局，在整个项目中都将被用到，这些布局还要为每个页面创建基础结构。对于这个应用，我们将创建两个基础模板：*base.html* 和 *page.html*。

我们从创建 *base.html* 开始，把它放到 *sitebuilder/templates* 目录下，并添加以下布局。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>{% block title %}Rapid Prototypes{% endblock %}</title>
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
```

由于我们正在创建的是一个静态站点，因此需要将 Django 的静态文件标记加入到模板中。我们还要创建一个名为 *site.css* 的空白 CSS 文件，并把它放入 *static/css* 文件夹，以便后期使用。

```
hostname $ touch sitebuilder/static/css/site.css
```

现在将下面的新结构添加到 *base.html* 模板中。

```
{% load staticfiles %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>{% block title %}Rapid Prototypes{% endblock %}</title>
    <link rel="stylesheet" href="{% static 'css/site.css' %}">
</head>
<body>
    {% block content %}{% endblock %}
</body>
</html>
```

由于我们还没有添加任何特定 URL 模式或视图，当运行 `python prototypes.py runserver` 时，我们还会看到“It worked！”模板（见图 3-1）。

接下来一节，我们要开始创建视图以及 URL，并添加 *page.html* 来搭建我们的原型。

静态页面生成器

就像其他静态站点生成器一样，希望用我们的生成器可以方便快捷地创建新页面。要实现这一目的，我们需要添加引用文件路径所需的设置变量、修饰页面所需的视图，以及指向动态页面所需的轻量级 URL 结构。

先来添加一个 pages 目录存放所有原型页面：

```
hostname $ mkdir pages
```

我们所有的原型模板都会放在这个目录下。现在让我们使用 *prototypes.py* 文件中的设置变量来添加一种引用这个文件夹的方式。

```
import os
import sys

from django.conf import settings


BASE_DIR = os.path.dirname(__file__)
...
    STATIC_URL='/static/',
    SITE_PAGES_DIRECTORY=os.path.join(BASE_DIR, 'pages'),
...
```

现在可以很方便地访问存放原型文件的 *pages* 文件夹。我们最终想要搭建的效果是创建一个具有 URL 结构的站点，并通过每个页面的内容来匹配 *pages* 文件夹下的文件。页面的布局将由定义在 *templates* 文件夹中的模板来确定。这可以帮助我们从页面布局中分离页面内容。

在制作静态输出之前，我们先要创建 *views.py* 文件（在 *sitebuilder* 文件夹下），来动态修饰页面以供我们在本地使用。先来添加一个视图修饰 *pages* 文件夹下的每个模板。

```
import os

from django.conf import settings
from django.http import Http404
from django.shortcuts import render
from django.template import Template
from django.utils._os import safe_join


def get_page_or_404(name):
    """Return page content as a Django template or raise 404 error."""
    try:
        file_path = safe_join(settings.SITE_PAGES_DIRECTORY, name)          ❶
    except ValueError:
        raise Http404('Page Not Found')
    else:
        if not os.path.exists(file_path):
            raise Http404('Page Not Found')

    with open(file_path, 'r') as f:                                         ❷
        page = Template(f.read())

    return page


def page(request, slug='index'):
    """Render the requested page if found."""
    file_name = '{}.html'.format(slug)
    page = get_page_or_404(file_name)
    context = {
        'slug': slug,
        'page': page,
    }
    return render(request, 'page.html', context)                            ❸
```

❶ 使用 `safe_join` 来将页面文件路径和模板文件名连接起来，并返回规范化的最终的绝对路径。

❷ 打开每个文件并使用文件内容创建新的 Django 模板。

❸ 将要修饰的 `page` 和 `slug` 上下文传递给 *page.html* 布局模板。

本项目中，把这个目录下的文件视为 Django 模板，不过还有种方案是把页面内容定义成 Markdown、RestructuredText 或其他自行选择的标记语言。更灵活的方法是基于页面扩展名来修饰页面的内容。我们把有关这一部分内容的改进留给读者作为练习。

要完成这个视图，我们需要创建 *page.html* 模板（在 *sitebuilder/template* 下）对原型页面进行修饰，记住修饰所包含的环境是通过一个名为 page 的上下文变量传递的。

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

{% block title %}{{ block.super}} - {{ slug|capfirst }}{% endblock %}

{% block content %}
    {% include page %}
{% endblock %}
```

所有的视图环境放置妥当后，现在来创建 *urls.py* 文件（在 *sitebuilder* 文件夹下），文件中包含发送对列表和详细页面请求的地点。

```
from django.conf.urls import url

from .views import page

urlpatterns = (
    url(r'^(?P<slug>[\w./-]+)/$', page, name='page'),
    url(r'^$', page, name='homepage'),
)
```

完成基本模板后，现在就可以添加创建静态站点所需的任意内容了。服务器根目录 / 将在不传递 slug 参数的形式下调用 page 视图，这意味着它会使用默认的 index 作为 slug 值。为了修饰主页面，我们在 *pages* 文件夹下添加一个基础的 *index.html* 模板来测试应用的运行情况。

```
<h1>Welcome To the Site</h1>
<p>Insert marketing copy here.</p>
```

```
hostname $ python prototype.py runserver
Performing system checks...

System check identified no issues (0 silenced).
July 06, 2015 - 22:03:47
Django version 1.8.3, using settings None
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

当在浏览器中访问 *http://127.0.0.1:8000* 网址时，将能看到图 3-2 所示的页面。

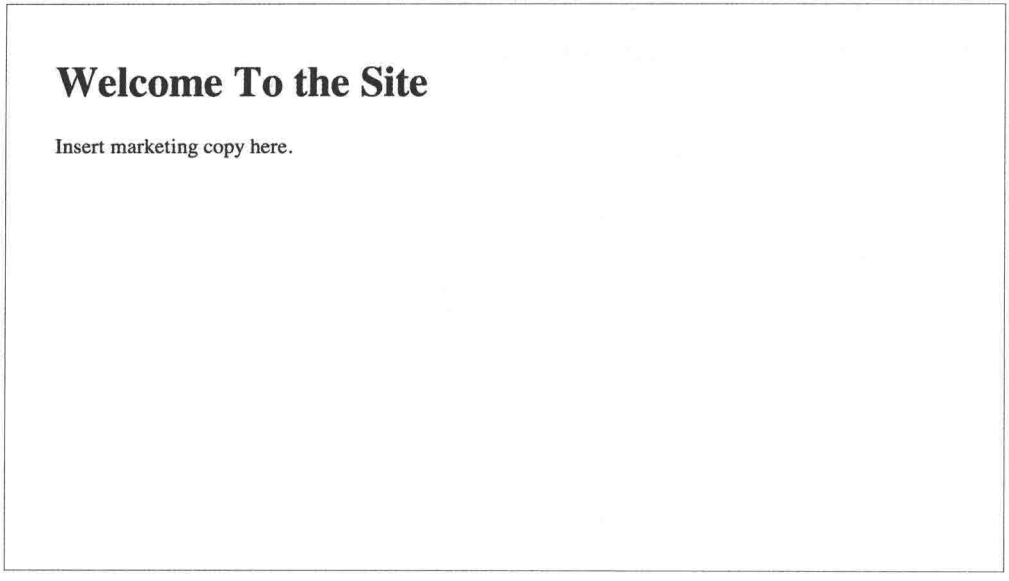


图 3-2 ：欢迎页面

现在可以随心所欲地往 pages 文件夹下添加页面来帮助制作原型。在下一节中，我们将继续搭建我们的工程，为它加入一些基本样式和布局。

基本样式

我们在之前提到过，这一过程放在项目的开始。Twitter Bootstrap 是一个帮助创建初始样式的很好的选择。所以我们接着要将所需的文件下载到 static 目录下，将 Twitter Bootstrap 结合到我们的样式中。

使用下面的 URL 将静态依赖文件下载到桌面：*https:// github.com/twbs/bootstrap/releases/download/v3.2.0/bootstrap-3.2.0-dist.zip*。释放文件并把它们分别放到 static 目录下合适的位置。完成后的目录结构应该像这样：

```
prototypes.py
pages/
sitebuilder
    __init__.py
    static/
        js/
            bootstrap.min.js
        css/
            bootstrap-theme.css.map
            bootstrap-theme.min.css
            bootstrap.css.map
            bootstrap.min.css
            site.css
```

```
        fonts/
            glyphicons-halflings-regular.eot
            glyphicons-halflings-regular.svg
            glyphicons-halflings-regular.ttf
            glyphicons-halflings-regular.woff
    templates/
    urls.py
    views.py
```

由于项目中即将创建的内容还依赖 jQuery，因此也要下载。访问 *http://cdnjs.cloudflare.com/ajax/libs/ jquery/2.1.1/jquery.min.js*，并把该版本存到 *static/js* 目录下。

现在我们要将对这些文件的引用添加到 *base.html* 模板（在 *sitebuilder/templates*）中。

```
{% load staticfiles %}
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width,initial-scale=1">
        <title>{% block title %}Rapid Prototypes{% endblock %}</title>
        <link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
        <link rel="stylesheet" href="{% static 'css/site.css' %}">
    </head>
    <body>
        {% block content %}{% endblock %}
        <script src="{% static 'js/jquery.min.js' %}"></script>
        <script src="{% static 'js/bootstrap.min.js' %}"></script>
    </body>
</html>
```

现在，我们可以开始使用 Twitter Bootstrap 中的 CSS 和 JavaScript 所提供的最基本的样式和交互来添加原型了。

原型布局和导航

一般情况下，在开始创建一个网站时都会首先思考首页应该是什么样子。在这一节中我们就来创建一个简单的首页布局，接着使用一个简单的导航来制作其他页面。

我们还将使用 Twitter Bootstrap 中基于列的布局样式，并使用框架提供的 12 列网格。我们来改进 *index.html* 模板（在 *pages* 文件夹下），并使用这些样式创建一个简单的销售页面。

```
<div class="jumbotron">
    <div class="container">
        <h1>Welcome To the Site</h1>
        <p>Insert marketing copy here.</p>
    </div>
</div>
```

```
<div class="container">
    <div class="row">
        <div class="col-md-6">
            <h2>About</h2>
            <p>{% lorem %}</p>
        </div>
        <div class="col-md-6">
            <h2>Contact</h2>
            <p>{% lorem %}</p>
            <p>
                <a class="btn btn-default"
                    href="{% url 'page' 'contact' %}" role="button">
                    Contact us >>
                </a>
            </p>
        </div>
    </div>
</div>

<hr>

<footer>
    <div class="container">
        <p>&copy; Your Company {% now 'Y' %}</p>
    </div>
</footer>
```

这里需要注意几件事情。首先，我们使用了 page 的 URL 来创建到其他原型页面的 URL。现在我们可以创建具有指向其他静态页面功能的动态链接的静态站点。本节稍后将讲述这一部分内容。如果现在尝试访问这些功能，将会出现 404 Not Found 错误。

同样还会注意到，我们这里正在使用 Django 的 {% lorem %} 标签来生成主页的占位文字。这些标签帮助我们更快地创建页面，甚至在还没有可用副本的情况下也有效。

我们还需要在 *site.css* 文件（位于 */sitebuilder/static/css* 下）里添加一些 CSS 来为内容加上内边距。这是使用 Twitter Bootstrap 过程中通常的习惯，由于它的样式结构，通常让设计者自己来决定内边距的大小。

```
body {
    padding: 50px 0 30px;
}
```

由于还要添加更多的页面，我们在 *base.html* 模板（在 *sitebuilder/templates* 下）里添加一些基础的导航来作为站点级别的导航。

```
...
<body id="{% block body-id %}body{% endblock %}">
    {% block top-nav-wrapper %}
    <div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
        <div class="container">
```

```
                <div class="navbar-header">
                    <button type="button" class="navbar-toggle"
                        data-toggle="collapse" data-target=".navbar-collapse">
                        <span class="sr-only">Toggle navigation</span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </button>
                    <a class="navbar-brand" href="/">Rapid Prototypes</a>
                </div>
                <div class="collapse navbar-collapse">
                    <ul class="nav navbar-nav">
                        <li {% if slug == 'index' %}class="active"{% endif %}>
                            <a href="/">Home</a>
                        </li>
                        <li {% if slug == 'contact' %}class="active"{% endif %}>
                            <a href="{% url 'page' 'contact' %}">Contact</a>
                        </li>
                    </ul>
                    <ul class="nav navbar-nav navbar-right">
                        <li {% if slug == 'login' %}class="active"{% endif %}>
                            <a href="{% url 'page' 'login' %}">Login</a>
                        </li>
                    </ul>
                </div>
            </div>
        </div>
        {% endblock %}
        {% block content %}{% endblock %}
...
```

你将会看到，在 <body> 标签内已经添加了 {% block body-id %} 模板标签。这是一个很实用的技巧，可以帮助我们为页面的每个部分指定 CSS 样式。我们还需要把这些内容加到 */sitebuilder/templates* 下的 *page.html* 布局中。我们将使用页面的 {{ slug }} 值来为每个页面创建动态的 ID 值。

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

{% block title %}{{ block.super }} - {{ slug|capfirst }}{% endblock %}

{% block body-id %}{{ slug|slugify }}{% endblock %}    ❶

{% block content %}
    {% include page %}
{% endblock %}
```

❶ 这里我们还添加了一个 slugify 筛选器来将所有页面生成的片段转换为小写值，并生成一致的 ID 值。

让我们再次运行 `python prototypes.py runserver` 命令，作一个快速检查。当在浏览器中访问 *http://127.0.0.1:8000/* 时，会看到如图 3-3 所示的界面。

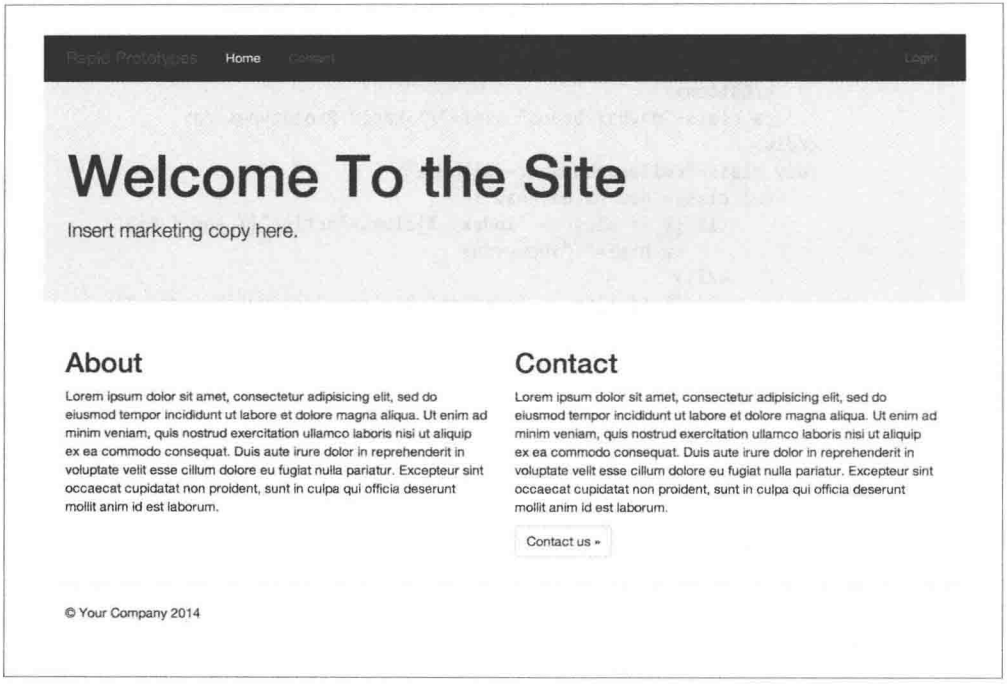


图 3-3：加入样式后的欢迎页面

到目前为止，我们已经完成了基础结构的创建。让我们为 *index.html* 中引用的联系方式和登录页面继续添加一些模板。先从登录页面开始，在 *pages* 文件夹下添加 *login.html* 模板，如图 3-4 所示。

```
<div class="container">
    <div class="row">
        <form class="form-signup col-md-6 col-md-offset-3" role="form">
            <h2 class="form-signin-heading">Login to Your Account</h2>
            <div class="form-group">
                <input type="email" class="form-control"
                    placeholder="Email address" required=""
                    autofocus="" autocomplete="off" >
            </div>
            <div class="form-group">
                <input type="password" class="form-control"
                    placeholder="Password" required="" autocomplete="off" >
            </div>
            <div class="form-group">
                <button class="btn btn-lg btn-primary btn-block"
                    type="submit">Login</button>
            </div>
        </form>
```

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

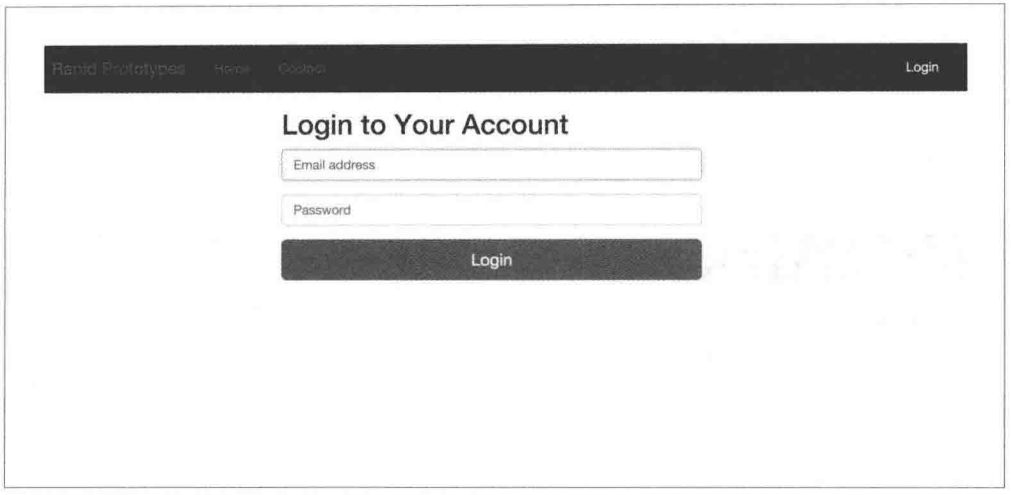


图 3-4：登录页面

尽管我们已经得到了一个样式美观的登录页面，但它还没有什么功能。这里就是一个展示设计师和开发者如何并行工作的好例子，他们各自仍然在按自己的计划来完成各自的任务。当登录表单即将同步到后端时，已经包含了一些上下文和样式的内容。让我们以同样的方法在 pages 文件夹下加入 *contact.html* 模板，并制作一个基本的联系方式页面，如图 3-5 所示。

```
<div class="container">
    <div class="row">
        <form class="form-contact col-md-6" role="form">
            <h2>Contact Us</h2>
            <div class="form-group">
                <input type="email" class="form-control"
                    placeholder="Email address" required="" >
            </div>
            <div class="form-group">
                <input type="text" class="form-control"
                    placeholder="Title" required="" >
            </div>
            <div class="form-group">
                <textarea class="form-control" required=""
                    placeholder="Your message..." ></textarea>
            </div>
            <div class="form-group">
                <button class="btn btn-lg btn-primary btn-block"
                    type="submit">Submit</button>
            </div>
        </form>
        <div class="col-md-4 col-md-offset-2">
```

```
            <h2>Our Office</h2>
            <address>
                <strong>Company Name</strong><br>
                123 Something St<br>
                New York, NY 00000<br>
                (212) 555 - 1234
            </address>
        </div>
    </div>
</div>
```

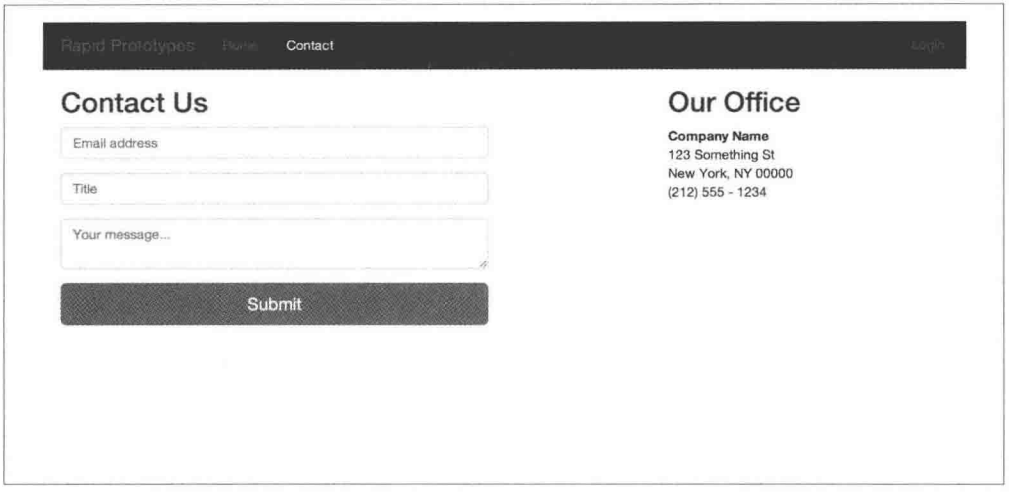


图 3-5：（“Contact Us”）页面

同样的，尽管这个表单还不能动态使用，我们已经生成了简单的原型样式并可以轻松地同步。这是在项目中使用快速原型的核心概念中的一个，它描绘了设计师和开发者如何无缝并行工作的方法。

在下节中，我们将通过一个管理命令，把这一动态内容变或真正的静态。

生成静态内容

尽管我们的确已经有了一个可以创建简单静态页面的原型应用，但从技术上来说我们的应用还不算一个生成静态内容（如 HTML）的静态站点。让我们来创建一个自定义管理命令，用于将内容输出到可部署的静态文件夹下。

配置设置

我们需要在 *sitebuilder* 目录下添加一些文件夹来创建自定义管理命令。还需要为命令起一个唯一的名字并把它加入到这个文件的结构中。我们的文件夹应该组织成下面的样子：

```
prototypes.py
pages/
sitebuilder
    __init__.py
    management/
        __init__.py
        commands/
            __init__.py
            build.py
    static/
        js/
        css/
        fonts/
    templates/
    urls.py
    views.py
```

在开始创建自定义命令前，需要几步设置来让静态输出目录正确工作。在 *prototypes.py* 文件中加入以下内容。

```
settings.configure(
...
    STATIC_URL='/static/',
    SITE_PAGES_DIRECTORY=os.path.join(BASE_DIR, 'pages'),
    SITE_OUTPUT_DIRECTORY=os.path.join(BASE_DIR, '_build'),     ❶
    STATIC_ROOT=os.path.join(BASE_DIR, '_build', 'static'),     ❷
)
...
```

❶ 这一设置配置了一旦命令完成后生成的静态文件所存放的输出目录。

❷ 这里启用了存放在 _build 目录下的静态内容。

如同 STATIC_ROOT 那样，SITE_OUTPUT_DIRECTORY 会存放生成的内容并忽略版本管理的检查。完成这些设置并搭建好文件夹结构后，就可以动手编写管理命令了。

自定义管理命令

现在我们可以来自定义管理命令去生成和输出静态站点了。我们将用到之前创建的 *build.py* 文件（在 */sitebuilder/management/commands* 下）以及下面的代码。

```
import os
import shutil

from django.conf import settings
from django.core.management import call_command
from django.core.management.base import BaseCommand
from django.core.urlresolvers import reverse
from django.test.client import Client
```

```
def get_pages():                                                    ❶
    for name in os.listdir(settings.SITE_PAGES_DIRECTORY):
        if name.endswith('.html'):
            yield name[:-5]


class Command(BaseCommand):
    help = 'Build static site output.'
    leave_locale_alone=True
    def handle(self, *args, **options):
        """Request pages and build output."""
        if os.path.exists(settings.SITE_OUTPUT_DIRECTORY):          ❷
            shutil.rmtree(settings.SITE_OUTPUT_DIRECTORY)
        os.mkdir(settings.SITE_OUTPUT_DIRECTORY)
        os.makedirs(settings.STATIC_ROOT, exists_ok=True)
        call_command('collectstatic', interactive=False,            ❸
            clear=True, verbosity=0)
        client = Client()
        for page in get_pages():                                    ❹
            url = reverse('page', kwargs={'slug': page})
            response = client.get(url)
            if page == 'index':
                output_dir = settings.SITE_OUTPUT_DIRECTORY
            else:
                output_dir = os.path.join(settings.SITE_OUTPUT_DIRECTORY, page)
                os.makedirs(output_dir)
            with open(os.path.join(output_dir, 'index.html'), 'wb') as f:  ❺
            f.write(response.content)
```

❷ 检查 *output* 目录是否存在。如果已经存在，则将它删除并创建一个新的 output 目录。

❸ 该行使用 `call_command` 执行 `collectstatic` 命令，将站点中所有的静态文件复制到 `STATIC_ROOT` 里，这项操作已经被配置到了 `SITE_OUTPUT_DIRECTORY` 中。

❶❹ 这里遍历 *pages* 文件夹，收集文件夹中所有 *.html* 文件。

❺ 在这里将模板修饰为静态内容。使用 Django 测试客户端，模拟抓取网站页面并将修饰过的内容写入 `SITE_OUTPUT_DIRECTORY`。

现在，我们可以运行管理命令来创建 *build* 目录了，它会被放置在 *projects* 目录的根级目录下。

```
hostname $ python prototypes.py build
```

在项目目录的根级目录下，应该能看到一个包含所有静态文件的 *build* 目录。接着让我们使用一个简易的本地 Python 服务器来测试我们的静态文件。

```
hostname $ cd _build
```

```
hostname $ python -m http.server 9000
Serving HTTP on 0.0.0.0 port 9000 ...
```

到目前为止，当访问 *0.0.0.0:9000* 时，会看到和图 3-3 一样的索引页面。现在我们可以把这些文件部署到我们想要的任何工作平台上，并展示快速原型过程。

生成单个页面

在各种项目中，有时候我们只想修改站点中的一个文件。然而，对于我们现有的管理命令来说，每一次都需要重新生成整个站点文件目录。让我们来为它配置一个一次生成一个页面的功能。

开始之前，我们要在 *build.py*（位于 */sitebuilder/management/commands/* 下）中加入一个参数，用于控制生成单个文件或全部内容。

```
...
from django.core.management import call_command
from django.core.management.base import BaseCommand, CommandError          ❶
...
class Command(BaseCommand):
    help = 'Build static site output.'
    leave_locale_alone = True

    def add_arguments(self, parser):
        parser.add_argument('args', nargs='*')          ❷

    def handle(self, *args, **options):
        """Request pages and build output."""          ❸
        if args:
            pages = args
            available = list(get_pages())
            invalid = []
            for page in pages:
                if page not in available:
                    invalid.append(page)
            if invalid:
                msg = 'Invalid pages: {}'.format(', '.join(invalid))     ❹
                raise CommandError(msg)
        else:
            pages = get_pages()
            if os.path.exists(settings.SITE_OUTPUT_DIRECTORY):
                shutil.rmtree(settings.SITE_OUTPUT_DIRECTORY)
            os.mkdir(settings.SITE_OUTPUT_DIRECTORY)
        os.makedirs(settings.STATIC_ROOT, exists_ok = True)
        call_command('collectstatic', interactive=False,
            clear=True, verbosity=0)
        client = Client()
        for page in pages:
            url = reverse('page', kwargs={'slug': page})
            response = client.get(url)
```

```
if page == 'index':
    output_dir = settings.SITE_OUTPUT_DIRECTORY
else:
    output_dir = os.path.join(settings.SITE_OUTPUT_DIRECTORY, page)
    if not os.path.exists(output_dir):                          ❺
        os.makedirs(output_dir)
with open(os.path.join(output_dir, 'index.html'), 'wb') as f:
    f.write(response.content)
```

❷❸ 检查是否有参数传入命令。支持一次传入多个文件名。

❿❹ 如果文件名中的某一个不存在，则报错。

❺ 由于开始时并不总是希望新建整个目录，所以需要处理目录已经存在的情况。

现在可以运行下面的命令，将单个页面编译到静态内容中。

```
hostname $ python prototypes.py build index
```

在下一节中，我们将使用Django的静态文件和django-compressor来进一步配置静态文件。

处理和压缩静态文件

对于Web上每一个页面请求来说，修饰每一个页面都需要经历许多步骤。每个请求都会增加页面载入负担。为了减少消耗的时间，Django内建了缓存框架供我们应用到Web应用上。

加快页面载入的另一种方式是创建压缩的静态文件。在本节中，我们将探索使用哈希以及利用django-conpressor两种方式来压缩文件，为页面处理创建一个更加快捷有效的过程。

哈希 CSS 和 JavaScript 文件

Django有些很有用的设置帮助我们存储、处理图片、CSS，以及JavaScript文件。我们将使用这些设置中的一种来创建CSS和JavaScript文件名的哈希值。这是一个很有用的技术，可以在文件发生改变时清除浏览器的缓存。借助于这些基于文件内容的独一无二的文件名，Web服务器可以通过对截止时间标头进行配置，以便在很久以后还继续使用这些文件。

开始之前，我们先在*prototypes.py*文件中加入一些必要的设置：

```
...
settings.configure(
    DEBUG=True,
```

```
    SECRET_KEY='bOmqvak1p2sqm6p#+8o8fyxf+ox(le)8&jh_5^sxa!=7!+wxjO',
    ROOT_URLCONF='sitebuilder.urls',
    MIDDLEWARE_CLASSES=(),
    INSTALLED_APPS=(
        'django.contrib.staticfiles',
        'sitebuilder',
    ),
    TEMPLATES=(
        {
            'BACKEND': 'django.template.backends.django.DjangoTemplates',
            'DIRS': [],
            'APP_DIRS': True,
        },
    ),
    STATIC_URL='/static/',
    SITE_PAGES_DIRECTORY=os.path.join(BASE_DIR, 'pages'),
    SITE_OUTPUT_DIRECTORY=os.path.join(BASE_DIR, '_build'),
    STATIC_ROOT=os.path.join(BASE_DIR, '_build', 'static'),
    STATICFILES_STORAGE='django.contrib.staticfiles.storage.
CachedStaticFilesStorage',
)
...
```

设置好 STATICFILES_STORAGE 后，当 DEBUG 设置为 False 时，我们的文件会得到一个与其相关的唯一的哈希值。下面我们需要把 DEBUG 设置加入到 build 管理命令中（位于 */sitebuilder/management/commands/build.py*）。

```
...

def handle(self, *args, **options):
    """Request pages and build output."""
    settings.DEBUG = False
...
```

现在来运行管理命令，看看 build 目录下生成的哈希文件名。

```
hostname $ python prototypes.py build
```

打开 *build/static* 目录，可以看到 CSS 和 JavaScript 都有了唯一的与其相关的哈希文件名。生成的 HTML 文件也会使用哈希名而不是通过 {% static %} 标签引用的原始文件名。

除了这项有用的技术外，Django 社区还在其他程序包中为我们提供了更有效的方法。下一节，我们将探索如何利用 django-compressor 来压缩并最小化我们的 CSS 和 JavaScript 文件。

压缩静态文件

减少页面负载的另一种方法是压缩并最小化 CSS 和 JavaScript 文件。常用的 Python 包 django-compressor 帮助 Django 项目完成这个任务。这个包还有一些强大的功能，例如 LESS/Sass 编译。

在开始使用 django-compressor 之前，需要将它装入我们的项目，并添加到设置中。

```
hostname $ pip install django-compressor==1.5
```

现在删除本节之前添加的缓存设置，并在 *prototypes.py* 中加入一些 django-compressor 需要的设置，以便正常工作。

```
...
settings.configure(
    DEBUG=True,
    SECRET_KEY='bOmqvak1p2sqm6p#+8o8fyxf+ox(le)8&jh_5^sxa!=7!+wxjO',
    ROOT_URLCONF='sitebuilder.urls',
    MIDDLEWARE_CLASSES=(),
    INSTALLED_APPS=(
        'django.contrib.staticfiles',
        'sitebuilder',
        'compressor',
    ),
    TEMPLATES=(
        {
            'BACKEND': 'django.template.backends.django.DjangoTemplates',
            'DIRS': [],
            'APP_DIRS': True,
        },
    ),
    STATIC_URL='/static/',
    SITE_PAGES_DIRECTORY=os.path.join(BASE_DIR, 'pages'),
    SITE_OUTPUT_DIRECTORY=os.path.join(BASE_DIR, '_build'),
    STATIC_ROOT=os.path.join(BASE_DIR, '_build', 'static'),
    STATICFILES_FINDERS=(
        'django.contrib.staticfiles.finders.FileSystemFinder',
        'django.contrib.staticfiles.finders.AppDirectoriesFinder',
        'compressor.finders.CompressorFinder',
    )
)
...
```

正如在代码中所看到的，我们把 compressor 加进了 INSTALLED_APPS 列表。此外，还添加了 STATICFILES_FINDER 以及让 django-compressor 与 Django 的 staticfiles contrib 应用协同工作所需的其他必要设置。

要查看修改 django-compressor 的设置列表以及默认行为，请访问 *http://django-compressor.readthedocs.org/en/devel op/settings/*。

要在自定义管理命令中实现压缩，需要在 *build.py*（位于 */sitebuilder/management/commands/*）中开启压缩，并在收集完静态文件后运行 call_command 命令。

```
...

def handle(self, *args, **options):
    """Request pages and build output."""
    settings.DEBUG = False
    settings.COMPRESS_ENABLED = True
    ...
    call_command('collectstatic', interactive=False, clear=True, verbosity=0)
    call_command('compress', interactive=False, force=True)
...
```

现在可以向*base.html*模板(在*/sitebuilder/templates*下)的静态资源中添加{% compress %}块标记。

```
{% load staticfiles compress %}
<!DOCTYPE html>
...

{% compress css %}

<link rel="stylesheet" href="{% static 'css/bootstrap.min.css' %}">
<link rel="stylesheet" href="{% static 'css/site.css' %}">

{% endcompress %}

...

{% compress js %}

<script src="{% static 'js/jquery.min.js' %}"></script>

<script src="{% static 'js/bootstrap.min.js' %}"></script>

{% endcompress %}

...
```

让我们用新的设置测试一下管理命令中的这个新配置。

```
hostname $ python prototypes.py build
Found 'compress' tags in:
    /path/to/your/project/sitebuilder/templates/base.html
    /path/to/your/project/sitebuilder/templates/page.html
Compressing... done
Compressed 2 block(s) from 2 template(s).
```

当下，可以在*build*目录下看到一个*CACHE*文件夹，存放有压缩过的CSS和JavaScript文件。HTML输出中的CSS和JavaScript文件引用同样会被更新。不仅文件名会被更新，就连对两个样式表的引用也将变成一个，外加两个`<script>`标签，如下面这段*build/index.html*的代码片段所示。

```
...
<link rel="stylesheet" href="/static/CACHE/css/00c333162d8e.css" type="text/css" />
...
<script type="text/javascript" src="/static/CACHE/js/a115a2614bdb.js"></script>
...
```

在 django-compressor 的默认设置下，这些文件仅仅是这一代码块下的两个文件的组合。还可选加一些附加设置和配置，通过像 YUI 这样的压缩器对它们运行。

我们可以使用 django-compressor 在 LESS 或 Sass 这样的预处理器上编译 CSS，而不是直接使用 CSS。同样，JavaScript 还可以由 CoffeeScript 来构建。即使我们在以某种非正规的方式使用 Django，这些社区工具同样可以帮助我们完成工作。

在每次要生成新压缩和可调用的静态文件时，可以运行 `build` 命令。下一节，我们将在模板中创建动态内容。

生成动态内容

正如到目前所看到的，Django 中的快速原型模板即使在没有完整后端的情况下也可生成一个简单的网站。唯一缺失的功能是生成动态文件。其他静态站点实现这一功能的方式是使用 YAML 文件。我们并不希望在 Django 应用中添加任何不必要的文件基础设施，因为这并不是一个理想的方案。作为替代，我们将在 *views.py* 文件中添加一些元素在静态站点中支持动态内容。

更新模板

现在在当前的静态站点中只有一些基础模板。让我们来添加一个 *pricing.html* 模板（在 */pages* 下），用于在本节之后添加动态内容。

```
<div class="container">
    <div class="row">
        <div class="col-md-12">
            <h2>Available Pricing Plans</h2>
        </div>
    </div>
    <div class="row">
        <div class="col-md-4">
            <div class="panel panel-success">
                <div class="panel-heading">
                    <h4 class="text-center">Starter Plan - Free</h4>
                </div>
                <ul class="list-group list-group-flush text-center">
                    <li class="list-group-item">Feature #1</li>
                    <li class="list-group-item disabled">Feature #2</li>
```

```
                    <li class="list-group-item disabled">Feature #3</li>
                </ul>
                <div class="panel-footer">
                    <a class="btn btn-lg btn-block btn-success" href="#">
                        BUY NOW!
                    </a>
                </div>
            </div>
        </div>
        <div class="col-md-4">
            <div class="panel panel-danger">
                <div class="panel-heading">
                    <h4 class="text-center">Basic Plan - $10 / month</h4>
                </div>
                <ul class="list-group list-group-flush text-center">
                    <li class="list-group-item">Feature #1</li>
                    <li class="list-group-item">Feature #2</li>
                    <li class="list-group-item disabled">Feature #3</li>
                </ul>
                <div class="panel-footer">
                    <a class="btn btn-lg btn-block btn-danger" href="#">
                        BUY NOW!
                    </a>
                </div>
            </div>
        </div>
        <div class="col-md-4">
            <div class="panel panel-info">
                <div class="panel-heading">
                    <h4 class="text-center">Enterprise Plan - $20 / month</h4>
                </div>
                <ul class="list-group list-group-flush text-center">
                    <li class="list-group-item">Feature #1</li>
                    <li class="list-group-item">Feature #2</li>
                    <li class="list-group-item">Feature #3</li>
                </ul>
                <div class="panel-footer">
                    <a class="btn btn-lg btn-block btn-info" href="#">
                        BUY NOW!
                    </a>
                </div>
            </div>
        </div>
    </div>
    <div class="row">
        <div class="col-md-12">
            <p>{% lorem %}</p>
        </div>
    </div>
</div>
```

还要更新 *base.html* 模板（在 */sitebuilder/templates* 下），加入一个指向新页面的链接。

```
...
<ul class="nav navbar-nav">
    <li {% if slug == 'index' %}class="active"{% endif %}>
        <a href="/">Home</a>
    </li>
    <li {% if slug == 'pricing' %}class="active"{% endif %}>
        <a href="{% url 'page' 'pricing' %}">Pricing</a>
    </li>
    <li {% if slug == 'contact' %}class="active"{% endif %}>
        <a href="{% url 'page' 'contact' %}">Contact</a>
    </li>
</ul>
...
```

快速运行一遍 runserver 命令，看看我们新创建的定价页面，如图 3-6 所示。

尽管这个项目看似到此应该结束了，不过现在可以开始思考如何添加动态内容了，这对我们的工作流会很有帮助。

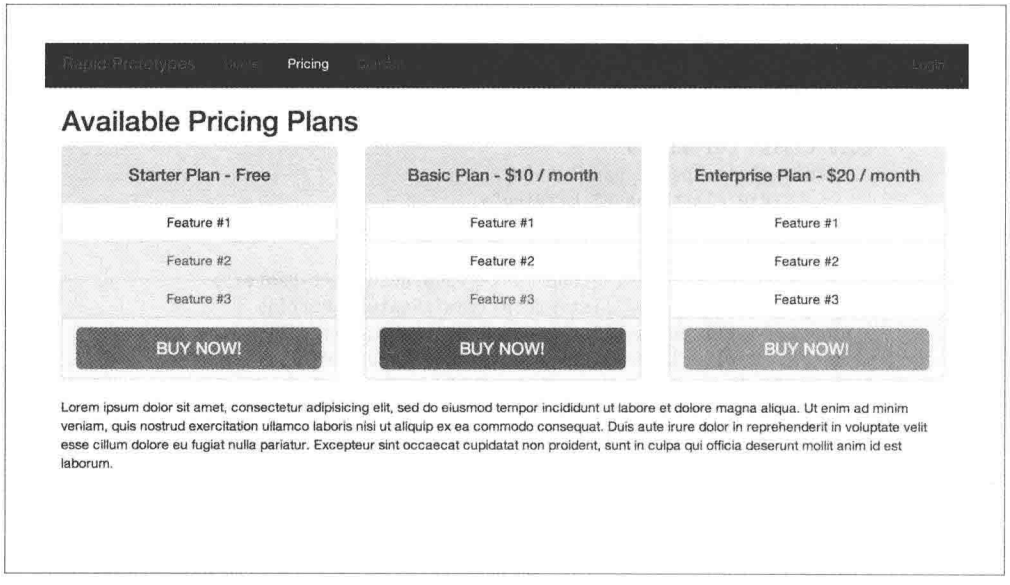


图 3-6：定价页面

添加元数据

开始之前，我们需要改进现有的 *views.py*（在 *sitebuilder* 文件夹下），来支持输入一种新的包含内容的 {% block %} 元素。

```
import json                                                    ❶
import os
```

```
...
from django.template import Template, Context                    ❷
from django.template.loader_tags import BlockNode                ❸
from django.utils._os import safe_join
...
def get_page_or_404(name):
    """Return page content as a Django template or raise 404 error."""
    try:
        file_path = safe_join(settings.SITE_PAGES_DIRECTORY, name)
    except ValueError:
        raise Http404('Page Not Found')
    else:
        if not os.path.exists(file_path):
            raise Http404('Page Not Found')

    with open(file_path, 'r') as f:
        page = Template(f.read())

    meta = None
    for i, node in enumerate(list(page.nodelist)):               ❹
        if isinstance(node, BlockNode) and node.name == 'context':
           meta = page.nodelist.pop(i)
           break
    page._meta = meta
    return page


def page(request, slug='index'):
    """Render the requested page if found."""
    file_name = '{}.html'.format(slug)
    page = get_page_or_404(file_name)
    context = {
        'slug': slug,
        'page': page,
    }
    if page._meta is not None:                                   ❺
        meta = page._meta.render(Context())
        extra_context = json.loads(meta)
        context.update(extra_context)
     return render(request, 'page.html', context)
```

❸❹ 代码遍历页面原始的节点列表，检查名称为 `context` 的 `BlockNode`。`BlockNode` 是 Django 模板中用于创建 `{% block %}` 元素的一个类。如果找到名为 `context` 的 `BlockNode`，它会定义一个元变量供我们包含内容。

❶❷ 元数据上下文通过 Python 的 json 模块来渲染，将 `{% block context %}` 转换成可理解
❺ 的 Python。

那么我们为什么要将 JSON（JavaScript Object Notation）作为上下文载入页面呢？很重要的一点是，我们不仅需要考虑到来源于数据库结构的数据，还要考虑一种对我们更方便引入的方式。举个例子，假如我们想要使用 RESTful API 来获取数据，最有可能的方

式就是用 JSON 来传递数据。在快速原型构建阶段，通过用这种方式思考，能更轻易地从简单的 {% block content %} 切换到动态载入的内容。

让我们继续在 *base.html* 模板（在 */sitebuilder/templates* 下）中加一些上下文引用，来测试所有内容。

```
{% load staticfiles compress %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <meta name="description"
        content="{{ description|default:'Default prototype description' }}">
    <meta name="keywords" content="{{ keywords|default:'prototype' }}">
    <title>{% block title %}Rapid Prototypes{% endblock %}</title>
...
```

到此可以看到，我们已经加入了一些新的 <meta> 标签引用和相应的默认值。我们来更新定价页面（位于 *pages* 文件夹下的 *pricing.html* 文件），使用新的上下文块来改变这些数值。

```
{% block context %}
{
    "description": "Product Pricing Plans",
    "keywords": "products,pricing,buy me"
}
{% endblock %}

<div class="container">
...
```

查看定价页面的源代码，会看到如图 3-7 所示的元描述和关键字。

```
<!DOCTYPE html>
▼<html lang="en">
  ▼<head>
     <meta charset="utf-8">
     <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width,initial-scale=1">
     <meta name="description" content="Product Pricing Plans">
     <meta name="keywords" content="products,pricing,buy me">
     <title>Rapid Prototypes - Pricing</title>
     <link rel="stylesheet" href="/static/css/bootstrap.min.css">
     <link rel="stylesheet" href="/static/css/site.css">
   ▶<style>…</style>
   ▶<style id="style-1-cropbar-clipper">…</style>
    </head>
```

图 3-7：定价页面的元数据

现在我们可以在模板内轻易地添加和编辑动态内容了。我们继续来清理定价模板（*/pages/pricing.html*），用这个代码块来载入定价方案上下文。

```
{% block context %}
{
    "description": "Product Pricing Plans",
    "keywords": "products,pricing,buy me",
    "plans": [
        {
            "name": "Starter",
            "price": "Free",
            "class": "success",
            "features": [1]
        },
        {
            "name": "Basic",
            "price": "$10 / month",
            "class": "danger",
            "features": [1, 2]
        },
        {
            "name": "Enterprise",
            "price": "$20 / month",
            "class": "info",
            "features": [1, 2, 3]
        }
    ],
    "features": [
        "Feature #1",
        "Feature #2",
        "Feature #3"
    ]
}
{% endblock %}

<div class="container">
    <div class="row">
        <div class="col-md-12">
            <h2>Available Pricing Plans</h2>
        </div>
    </div>
    <div class="row">
        {% for plan in plans %}
            <div class="col-md-4">
                <div class="panel panel-{{ plan.class }}">
                    <div class="panel-heading">
                        <h4 class="text-center">
                            {{ plan.name }} Plan - {{ plan.price }}
                        </h4>
                    </div>
                    <ul class="list-group list-group-flush text-center">
                        {% for feature in features %}
                            <li class="list-group-item
                                {% if forloop.counter not in plan.features %}
```

```
                                    disabled
                                {% endif %}">
                                {{ feature }}
                            </li>
                        {% endfor %}
                    </ul>
                    <div class="panel-footer">
                        <a class="btn btn-lg btn-block btn-{{ plan.class }}"
                            href="#">
                            BUY NOW!
                        </a>
                    </div>
                </div>
            </div>
        {% endfor %}
    </div>
    <div class="row">
        <div class="col-md-12">
            <p>{% lorem %}</p>
        </div>
    </div>
</div>
```

通过将数据以上下文块的形式添加，我们可以像对待外部数据一样来遍历数据。制作原型过程不仅仅考虑了布局，也开始思考了如何修饰数据的问题。

通过这个练习，会发现只需几步就可以使用 Django 架构创建一个快速原型工具和有效的工作流程。这为设计师和开发者能够并行工作创造一款成功的最终产品提供了一个高效有用的方法。这个项目的最终目标并非创建一个动态网站（正如 Django 通常所做的），除此之外我们还具备了使用灵活的 Django 模板技术、管理命令抽象，以及诸如 `django-compressor` 这样的社区扩展的能力。

其中一些相同的技术可以被应用到一些单页面的 Web 应用上，即 HTML 客户端用 Django 开发，但被输出和部署成静态 HTML。我们将从使用 `django-rest-framework` 以及其独有的可浏览界面构建 RESTful-API 服务器开始，在接下来的章节中对这方面做更多的探索。

第 4 章

构建 REST API

现在距美国计算机科学家 HTTP 规范的主要作者之一的 Roy Fielding 提出作为架构风格的可重现状态协议（REST）已有十多年。这么多年以来要感谢构建 Web 服务的热潮，因为它使 REST 得到了蓬勃的发展。

以无状态的形式，通过创建特殊标记来接收缓存、分层和扩展，REST API 使用已有的 HTTP 功能（GET、POST、PUT 和 DELETE）创建、更新，以及删除新的资源。REST 这一术语常常被误用于说明返回 JSON 而非 HTML 的 URL。绝大多数使用者没有意识到的一点是，要成为 RESTful 的架构，Web 服务必须满足一些规范约束。更进一步说，应用程序必须被分解成客户端 - 服务器端模型，服务器端必须保持完全无状态。服务器不存放任何客户端上下文，资源文件也需要被唯一且统一地标记。客户端还能够在资源响应中使用链接和元数据对 API 以及转移状态进行导航。除了如 API 根节点等少许固定节点之外，客户端不应该对资源的存在性以及功能作出假设。

在本章中，我们将探讨如何在 Django 中利用 RESTful 架构的威力。

Django 和 REST

在创建拥有大量数据的网站时，通常的做法是将 Django 与 REST 结合起来。Django 社区中有大量可复用的应用，帮助在创建 API 时遵从 REST 的严格原则。近些年来最流行的两个应用是 `django-tastypie` 和 `django-rest-framework`。这两个应用都支持从 ORM 或非 ORM 数据创建资源，支持可插拔的认证和许可，还支持包括 JSON、XML、YAML 和 HTML 在内的多种序列化方法。

有关进行比较的其他功能包可参阅：*https://www.djangopackages.com/grids/g/api/*.

在本章中我们将借助 `django-rest-framework` 的力量来构建我们的 API 架构。该可复用的 Django 应用最好的特性之一是支持创建自带文档以及可供 Web 浏览的 API。

`django-rest-framework` 并不像 `django-tastypie` 那样内建了从查询参数到 ORM 的过滤器转换。不过 `django-rest-framework` 有可插拔的过滤器后端，可以很简单地集成 `django-filter` 并提供这一功能。

我们来安装所有依赖包并开始项目。当用户查看 API 数据时，可浏览的 API 视图会利用 `Markdown` 将文档字符串转换为页面。

```
hostname $ pip install djangorestframework==3.1.3
hostname $ pip install django-filter==0.10.0
hostname $ pip install Markdown==2.6.2
```

既然已经安装好了所需的文件，让我们专注于构建一个用于任务板的 REST API 模型。

Scrum 板数据图

在任何项目中，对数据的建模都是关键的第一步。我们来花点时间列出在创建模型时需要考虑的要点：

- 任务板通常见于 Scrum 风格的开发中，用于在当前的 sprint 中管理任务。
- 任务从后台日志中被转移到待办任务集合中。
- 过程中可以通过多种状态来跟踪进度，例如“进行中”或者“测试中”。
- 一个任务要成为“已完成”状态，之前的任务必须是它的一部分。

对于视觉型学习者来说，图 4-1 所示的直观图是一种帮助开始理解数据间如何交互的好办法。

如你所见，我们在布局项目时需要考虑多个步骤。在有了上述的定义之后，接下来可以开始设计最初的项目架构了。

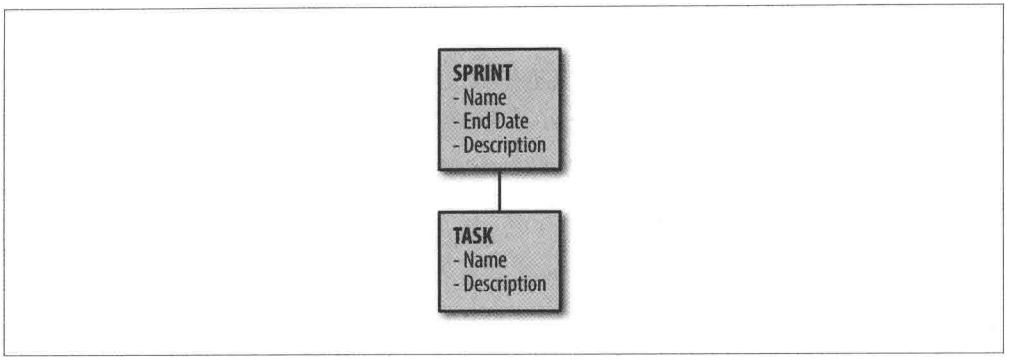


图 4-1：Scrum 板数据图

最初的项目布局

我们将通过运行基础的 `startproject` 命令来创建最初的项目布局。在运行命令的同时，我们还会在名为 `scrum` 的项目和名为 `board` 的应用之间传递数据。

```
hostname $ django-admin.py startproject scrum
hostname $ cd scrum
hostname $ python manage.py startapp board
```

现在，所创建的项目文件夹应如下所示：

```
scrum/
    manage.py
    scrum/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    board/
        migrations/
            __init__.py
        __init__.py
        admin.py
        models.py
        tests.py
        views.py
```

在最近版本的 Django 中，startapp 模板有所更新，如果使用不同版本的 Django，输出结果可能有些不同。Django 1.6 版中加入了 admin.py，1.7 版中加入了 migrations 文件夹。

让我们继续来配置 *settings.py* 文件，使其与 `django-rest-framework` 及其继承的设置一同工作。

项目设置

在创建这个新 Django 项目时，我们需要更新默认的项目设置（在 *scrum* 文件夹 *settings.py* 中）加入 `django-rest-framework`，并删减之前章节中的默认设置。同样，因为服务器不会保留客户端的状态，所以 `contrib.sessions` 引用也会被删除。这将破坏默认 Django 管理工具的使用，意味着那些引用也有可能被删除。

```
...
INSTALLED_APPS = (                                          ❶
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.staticfiles',
    # Third party apps
    'rest_framework',
    'rest_framework.authtoken',
    # Internal apps
    'board',
)
...
MIDDLEWARE_CLASSES = (                                      ❷
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.securityMiddleware;
)
...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'scrum',
    }
}
...
```

❶ 这些变动会把 `django.contrib.admin`、`django.contrib.seesions` 和 `django.contrib.messages` 从 `INSTALLED_APPS` 中移除。新的 `INSTALLED_APPS` 中包含 `rest_framework`、`rest_framework.authtoken` 和本章将要创建的 ***board*** 应用。

❷ `django.contrib.sessions.middleware.SessionMiddleware`，`django.contrib.auth.middleware.AuthenticationMiddleware`，`django.contrib.auth.middleware.SessionAuthenticationMiddleware` 和 `django.contrib.messages.middleware.MessageMiddleware` 都是 `startproject` 的默认设置，但由于不再安装这些应用，所以已经将它们从 `MIDDLEWARE_CLASSES` 中删除了。

本项目将使用 PostgresSQL 而非默认的 SQLite 数据库。这一变动需要安装 `psycopg2`，即 PostgreSQL 的 Python 驱动，还需要创建数据库。

```
hostname $ pip install psycopg2==2.6.1
hostname $ createdb -E UTF-8 scrum
```

数据库的设置会将假设 PostgreSQL 正在监听默认的 UNIX 套接字，并允许鉴别或信任对当前用户的认证。这些设置可能需要针对服务器配置进行调整。请查看 Django 文档获取这些设置的信息。

既然 Django 管理工具已经从 `INSTALLED_APPS` 中被移除，它在 *scrum/urls.py* 中的引用同样可以被移除。

如果是 OS X 用户，我们推荐通过专门用于 OS X 的包管理器（Homebrew）来安装 Postgre。要学习或查看更多有关如何安装 Homebrew 的指导，请访问：*http://brew.sh/*。

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

from rest_framework.authtoken.views import obtain_auth_token

urlpatterns = [
    url(r'^api/token/', obtain_auth_token, name='api-token'),
]
```

如你所见，所有已有的模式都被移除并替换为一条关于 `rest-framework.authtoken.views.obtain_auth_token` 的 URL，作为视图用于将用户名和密码的组合交换为 API 记号。我们在项目的下一步骤中还会使用到 `include` 引用。

没有 Django 管理工具？

想象管理一个不带 Django 管理工具的 Django 项目，也许某些 Django 开发者或用户会感到很痛苦。在这个应用中，因为我们不使用或不需要 Django 的会话管理，而这是使用管理工具所需要的，所以移除了管理工具。`django-rest-framework` 提供的可浏览的 API，可以作为管理工具的一个简易替代。

你可能会发现，仍然需要维护 Django 管理工具，因为其他应用依赖它。如果是这种情况，可以保留管理工具，并简单地为项目创建两套设置：一套用于 API，另一套用于管理工具。管理工具设置中会保留会话和消息应用以及认证的中间件。还可能需要包含管理工具 URL 的另一套根 URL。有了这些配置，一些服务器会处理站点的管理工具，另一些会处理 API。设想在一个更加大型的应用中，可能会存在大相径庭的扩展需求和考虑，这里提供了一种访问 Django 管理工具的洁净途径。

模型

完成项目的基本架构后，现在可以开始构建应用的数据模型了。首先，任务会被分解成一些 sprint，每个 sprint 会带有可选的名字和描述，以及唯一的结束日期。将这些模型加入 *board/models.py* 文件中：

```
from django.db import models
from django.utils.translation import ugettext_lazy as _

class Sprint(models.Model):
    """Development iteration period."""

    name = models.CharField(max_length=100, blank=True, default='')
    description = models.TextField(blank=True, default='')
    end = models.DateField(unique=True)

    def __str__(self):
       return self.name or _('Sprint ending %s') % self.end
```

尽管本书是针对 Python 3.3 及以上版本编写，但 Django 提供了一个 `python_2_unicode_compatible` 修饰符，可用于使这个模型兼容 Python 2.7 和 Python 3。更多有关 Python 3 接口以及兼容性的小技巧，请参阅 Django 文档。

我们还需要创建一个任务模型存放某 sprint 下的一系列任务。任务有名称、可选描述、关联一个 sprint（或存储它们的后台日志），以及分配给它们的用户，还包括开始日期、结束日期和截止日期。

我们还应该注意到任务存在下列状态：

- 未开始。
- 进行中。
- 测试中。
- 已完成。

让我们把这个 `STATUS_CHOICES` 列表加入到数据模型（*board/models.py*）：

```
from django.conf import settings
from django.db import models
from django.utils.translation import ugettext_lazy as _

...
class Task(models.Model):
    """Unit of work to be done for the sprint."""
```

```
    STATUS_TODO = 1
    STATUS_IN_PROGRESS = 2
    STATUS_TESTING = 3
    STATUS_DONE = 4

    STATUS_CHOICES = (
        (STATUS_TODO, _('Not Started')),
        (STATUS_IN_PROGRESS, _('In Progress')),
        (STATUS_TESTING, _('Testing')),
        (STATUS_DONE, _('Done')),
    )

    name = models.CharField(max_length=100)
    description = models.TextField(blank=True, default='')
    sprint = models.ForeignKey(Sprint, blank=True, null=True)
    status = models.SmallIntegerField(choices=STATUS_CHOICES, default=STATUS_TODO)
    order = models.SmallIntegerField(default=0)
    assigned = models.ForeignKey(settings.AUTH_USER_MODEL, null=True, blank=True)
    started = models.DateField(blank=True, null=True)
    due = models.DateField(blank=True, null=True)
    completed = models.DateField(blank=True, null=True)

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

用户引用通过 `settings.AUTH_USER_MODEL` 支持默认的 `User` 模型。尽管项目会使用默认的 `User` 模型，但 `board` 应用还将被设计得尽可能可重复使用。更多关于自定义以及引用 `User` 模型的信息，请参阅：*https://docs.djangoproject.com/en/1.8/topics/auth/customizing/#referencing-the-user-model*。

该项目需要用到两个模型，不过我们会发现在这个数据模型中还有一些明显的不足。一个不足是这个模型只跟踪单个项目的 sprint，并且假设所有系统用户都与这个项目相关。此外，任务模型中的任务状态也是固定的，这就使得这些状态不能用在 sprint 模型上，也不支持自定义任务流。

如果开发的应用只用于单个项目上，这些不足大多都是可接受的，不过显然当我们的目的是将这个任务面板做成一个服务（SaaS）产品时，就不能满足需求了。

既然已经写好了模型，只需要运行 `makemigrations` 和 `migrate` 指令，使 Django 正确更新数据库就够了。

```
hostname $ python manage.py makemigrations board
Migrations for 'board':
  0001_initial.py:
  - Create model Sprint
  - Create model Task
hostname $ python manage.py migrate
```

```
Operations to perform:
  Synchronize unmigrated apps: rest_framework, staticfiles
  Apply all migrations: authtoken board auth, contenttypes,
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
  Installing custom SQL...
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name...OK
  Applying auth.0001_initial... 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 authtoken.0001_initial... Ok
  Applying board.0001_initial... OK
```

现在让我们来运行 `createsuperuser` 命令，使用指定的用户名创建一个超级用户。对于以后的示例，我们假设创建的用户名是 *demo*，密码是 *test*。

```
hostname $ python manage.py createsuperuser
Username (leave blank to use 'username'): demo
Email address: demo@example.com
Password:
Password (again):
Superuser created successfully.
```

设计 API

有了模型之后，我们就可以把精力放在 API 本身上了。就 URL 结构而言，我们想要构造的 API 应当如下所示：

```
/api/
    /sprints/
        /<id>/
    /tasks/
        /<id>/
    /users/
        /<username>/
```

需要重点考虑的是客户端如何操作这些 API。客户端可以通过一个到 API 根节点 */api/* 的 GET 请求来查看 API 后面的区段。在这些区段中客户可以查看 sprint、任务和用户列表，也可以创建这些新内容。当用户更深一步查看某个具体的 sprint 时，它可以显示出与该 sprint 关联的任务。通过资源文件之间的超媒体链接可以使客户端对 API 进行导航操作。

我们选择不在 API 的 URL 中加入版本号。尽管许多热门 API 中加入了版本号，但我们感觉针对 RESTful 的习惯，管理版本的最好的方式是对于不同 API 版本使用不同内容形式。不过，如果确实希望在 API 中加入版本号，尽管添加就是。

Sprint 端点

通过 ViewSet 可以很轻易地使用 `django-rest-framework` 创建与 Django 模型关联的资源文件。要为 */api/sprints/* 创建 ViewSet，需要说明如何用 API 将模型序列化和串并转化。这一工作是由创建于 *board/serializers.py* 中的序列化器来处理的。

```
from rest_framework import serializers

from .models import Sprint


class SprintSerializer(serializers.ModelSerializer):

    class Meta:
        model = Sprint
        fields = ('id', 'name', 'description', 'end', )
```

在这个简单的例子中，所有字段都是通过 API 来显示的。

让我们在 *board/views.py* 中创建 ViewSet。

```
from rest_framework import viewsets

from .models import Sprint
from .serializers import SprintSerializer


class SprintViewSet(viewsets.ModelViewSet):
    """API endpoint for listing and creating sprints."""

    queryset = Sprint.objects.order_by('end')
    serializer_class = SprintSerializer
```

如你所见，`ModelViewSet` 提供了使用相应 HTTP 功能进行创建、读取、更新、删除（CRUD）操作所需的框架。如果没有在视图自身进行设置，有关认证、许可、分页和过滤器的默认设置则由 `REST_FRAMEWORK` 的设置词典进行管理。

Django REST 框架文档中记录了所有可用的设置以及它们的默认值。

要对这个视图进行明确设置。因为余下的视图也会用到这些默认设置，因此可以通过 *board/views.py* 中的混合类实现这一点。

```
from rest_framework import authentication, permissions, viewsets

from .models import Sprint
from .serializers import SprintSerializer


class DefaultsMixin(object):                                          ❶
    """Default settings for view authentication, permissions,
    filtering and pagination."""

    authentication_classes = (
        authentication.BasicAuthentication,
        authentication.TokenAuthentication,
    )
    permission_classes = (
        permissions.IsAuthenticated,
    )
    paginate_by = 25
    paginate_by_param = 'page_size'
    max_paginate_by = 100


class SprintViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating sprints."""

    queryset = Sprint.objects.order_by('end')
    serializer_class = SprintSerializer
```

❶ DefaultMixin 将作为通过 API 视图类定义这些选项的基类之一。

现在我们来为用户许可加入一些认证。认证会使用 HTTP 基础的验证或基于标记的认证。使用基础的验证方式会更便于通过 Web 浏览器来使用可浏览的 API。因为我们是基于所有系统用户都是项目一员的假设，所以这个例子的许可系统并不完善。唯一需要的许可是用户已经通过认证。

任务和用户端点

我们需要将任务显示在自身的端点上，要着手在类似 sprint 这样的端点上创建一个序列化器和视图集 ViewSet。首先让我们在 *board/serializers.py* 中创建这个序列化器。

```
from rest_framework import serializers

from .models import Sprint, Task

...
class TaskSerializer(serializers.ModelSerializer):
```

```
    class Meta:
        model = Task
        fields = ('id', 'name', 'description', 'sprint', 'status', 'order',
            'assigned', 'started', 'due', 'completed', )
```

乍一看感觉还不错，但这样写序列化器会带来一些问题。status 会显示数字而非状态相关的文本。通过在 *board/serializers.py* 中添加另一个 status_display 字段来显示状态文本，以便轻松地解决这个问题。

```
from rest_framework import serializers

from .models import Sprint, Task

...
class TaskSerializer(serializers.ModelSerializer):

    status_display = serializers.SerializerMethodField()    ❶

    class Meta:
        model = Task
        fields = ('id', 'name', 'description', 'sprint',
            'status', 'status_display', 'order',
            'assigned', 'started', 'due', 'completed', )

    def get_status_display(self, obj):
        return obj.get_status_display()
```

❶ status_display 是一个只读字段，返回序列化器中 get_status_display 中方法的值。

序列化器遇到的第二个问题是，assigned 是一个指向 User 模型的外键。这里显示的是用户的主键，然而我们的 URL 结构期望通过用户名来引用用户。我们可以在 *board/serializers.py* 中使用 SlugRelatedField 来解决这个问题。

```
...
class TaskSerializer(serializers.ModelSerializer):

    assigned = serializers.SlugRelatedField(
        slug_field=User.USERNAME_FIELD, required=False, allow_null = true,
        queryset=user.objects.all())
    status_display = serializers.SerializerMethodField()

    class Meta:
        model = Task
        fields = ('id', 'name', 'description', 'sprint',
            'status', 'status_display', 'order',
            'assigned', 'started', 'due', 'completed', )

    def get_status_display(self, obj):
        return obj.get_status_display()
```

最后，需要为 User 模型创建一个序列化器。让我们稍微花些时间回顾一下，我们的

User 模型可能取自于另外一个模型，我们应用的目的是让这个模型尽可能地被复用。所以我们需要在 *board/serializers.py* 中使用 get_user_model Django 工具来创建一种洁净的交换方式。

```
from django.contrib.auth import get_user_model

from rest_framework import serializers

from .models import Sprint, Task

User = get_user_model()

...
class UserSerializer(serializers.ModelSerializer):

    full_name = serializers.CharField(source='get_full_name', read_only=True)

    class Meta:
       model = User
       fields = ('id', User.USERNAME_FIELD, 'full_name', 'is_active', )
```

这个序列化器假设如果使用的是自定义的 User 模型，它是对 django.contrib.auth.models.CustomUser 的扩展，它始终会包含 USERNAME_FIELD 属性、get_full_name 方法和 is_active 属性。还要注意，由于 get_full_name 是一个方法，序列化器中的字段会被标记成只读。

创建完序列化器后，接着将在 *board/views.py* 中创建任务和用户的视图集 ViewSet。

```
from django.contrib.auth import get_user_model

from rest_framework import authentication, permissions, viewsets

from .models import Sprint, Task
from .serializers import SprintSerializer, TaskSerializer, UserSerializer


User = get_user_model()

...
class TaskViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating tasks."""

    queryset = Task.objects.all()
    serializer_class = TaskSerializer


class UserViewSet(DefaultsMixin, viewsets.ReadOnlyModelViewSet):
    """API endpoint for listing users."""

    lookup_field = User.USERNAME_FIELD
    lookup_url_kwarg = User.USERNAME_FIELD
```

```
    queryset = User.objects.order_by(User.USERNAME_FIELD)
    serializer_class = UserSerializer
```

这两个视图集都类似 SprintViewSet，但在 UserViewSet 中一个值得注意的区别是，它改为从 ReadOnlyModelViewSet 来继承。正如其名字所示，它不会通过 API 显示创建新用户或编辑已有用户的动作。UserViewSet 通过设置 lookup_field 将查看方式从使用用户 ID 改为使用用户名。注意考虑到一致性，lookup_url_kwargs 也一同被更改了。

连接路由

目前 board 应用已经有了基本的数据模型和视图逻辑，但它还没有连接到 URL 路由系统上。django-rest-framework 有自己的用于处理 ViewSet 的路由扩展，每个 ViewSet 都会以一个指定的 URL 前缀来注册路由。它们将被加入到名为 *board/urls.py* 的新文件中。

```
from rest_framework.routers import DefaultRouter

from . import views


router = DefaultRouter()
router.register(r'sprints', views.SprintViewSet)
router.register(r'tasks', views.TaskViewSet)
router.register(r'users', views.UserViewSet)
```

最后，路由需要被加入到位于 *scrum/urls.py* 的根节点的 URL 配置中。

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

from rest_framework.authtoken.views import obtain_auth_token

from board.urls import router


urlpatterns = [
    url(r'^api/token/', obtain_auth_token, name='api-token'),
    url(r'^api/', include(router.urls)),
]
```

现在 Scrum 板应用已经完成了基本模型、视图和 URL 结构的创建，接着该创建 RESTful 应用的其余部分了。

链接资源文件

RESTful 应用的一个很重要的限制就是作为应用状态（HATEOAS）引擎的超媒体文件。有了这些限制，RESTful 客户端应该可以通过服务器生成的超媒体响应来和应用进行交互了。也就是说，客户端只需要留意仅有的几个服务器端点。从这些端点中，客户端将

可以通过使用描述资源消息来判断资源在服务器上是否可用。客户端必须能够解释服务器响应，并且从链接等元数据中分离出相应的资源文件。

要怎样把这些信息转化成我们的资源文件呢？服务器会在这些资源文件中提供哪些有用的链接？首先，每个资源文件都应该知道自己的 URL。sprint 同样要为它们所相关的任务和后台日志提供 URL。分配给某个用户的任务也应该提供一个指向用户资源的链接。用户应该提供一种获取所有分配给该用户的任务方式。有了这些，API 客户端就可以回答绝大多数在操作时遇到的常见问题了。

为了为客户端提供一个查看这些链接的统一位置，每个资源文件在响应时都会包含一个 `links` 分量。作为起步，最简单的方式便是将所有资源文件链接回它们自身，如下面这段 *board/serializers.py* 的代码一样。

```
from django.contrib.auth import get_user_model

from rest_framework import serializers
from rest_framework.reverse import reverse                                    ❶

from .models import Sprint, Task


User = get_user_model()


class SprintSerializer(serializers.ModelSerializer):

    links = serializers.SerializerMethodField()                               ❷

    class Meta:
        model = Sprint
        fields = ('id', 'name', 'description', 'end', 'links', )

    def get_links(self, obj):                                                 ❸
        request = self.context['request']
        return {
            'self': reverse('sprint-detail',
                kwargs={'pk': obj.pk},request=request),
        }


class TaskSerializer(serializers.ModelSerializer):

    assigned = serializers.SlugRelatedField(
        slug_field=User.USERNAME_FIELD, required=False,queryset=user.objects.all))
    status_display = serializers.SerializerMethodField()
    links = serializers.SerializerMethodField()                               ❹

    class Meta:
        model = Task
        fields = ('id', 'name', 'description', 'sprint',
```

```
            'status', 'status_display', 'order',
            'assigned', 'started', 'due', 'completed', 'links', )

    def get_status_display(self, obj):
        return obj.get_status_display()

    def get_links(self, obj):                                          ❺
        request = self.context['request']
        return {
            'self': reverse('task-detail',
                kwargs={'pk': obj.pk}, request=request),
        }


class UserSerializer(serializers.ModelSerializer):

    full_name = serializers.CharField(source='get_full_name', read_only=True)
    links = serializers.SerializerMethodField()                        ❻

    class Meta:
        model = User
        fields = ('id', User.USERNAME_FIELD, 'full_name',
            'is_active', 'links', )

    def get_links(self, obj):                                          ❼
        request = self.context['request']
        username = obj.get_username()
        return {
            'self': reverse('user-detail',
                kwargs={User.USERNAME_FIELD: username}, request=request),
      }
```

❶ 这是对 `rest_framework.reverse.reverse` 的一个新引用。

❷❹ 每个序列化器都有一个新的只读字段 links 提供给响应主体。

❻

❸❺ 为了赋予 links 的值，每个序列化器都有一个 `get_links` 方法生成相关链接。

❼

现在每个资源文件都有了一个新的 `links` 字段，它会返回一个由 `get_links` 方法返回的字典数据。目前为止，字典中只有单独一个名为 `self` 的键，它与资源的详细信息相链接。`get_links` 并没有使用 Django 中标准的 `reverse` 方法，而是用到的 `django-rest-framework` 中一个稍加修改的版本，这一版本会返回完整的 URI，不仅包含路径，还包含主机名和协议名。要得到这些信息，`reverse` 需要当前这个请求，它将在我们使用标准 ViewSet 时默认通过上下文传递给序列化器。

分配给 sprint 的任务将反向链接这个 sprint。如果任务已经指派给某个用户，还可以通过反转 URL 链接回这个用户，如下面这段 *board/serializers.py* 的代码所示。

```
...
class TaskSerializer(serializers.ModelSerializer):
...
    def get_links(self, obj):
        request = self.context['request']
        links = {
            'self': reverse('task-detail',
                kwargs={'pk': obj.pk}, request=request),
            'sprint': None,
            'assigned': None
        }
        if obj.sprint_id:
            links['sprint'] = reverse('sprint-detail',
                kwargs={'pk': obj.sprint_id}, request=request)
        if obj.assigned:
            links['assigned'] = reverse('user-detail',
                kwargs={User.USERNAME_FIELD: obj.assigned}, request=request)
        return links
```

从 sprint 或用户链接到对应的任务需要涉及过滤器，我们将在下一节中添加这部分内容。

测试 API

有了 API 的模型和视图之后，就要测试我们的 API 了。我们先用浏览器对其进行查看，之后将用 Python shell 来测试。

使用可浏览的 API

`django-rest-framework` 最棒的特性之一就是提供了可视化工具并可以创建可浏览的 API。也许你会在产品系统中禁用这一功能，但它的确是个非常强大的浏览 API 的途径。尤其是，使用可浏览 API 能帮助你思考客户端如何随着响应中给出的链接在 API 间导航。

客户端应当能够通过一个固定端点进入 API，并通过服务器返回的资源表述对 API 进行研究。第一步是使用 `runserver` 让开发服务器运作起来：

```
hostname $ python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
July 07, 2015 - 02:04:48
Django version 1.8.3, using settings 'scrum.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

现在可以在喜爱的浏览器上通过访问 *http://127.0.0.1:8000/api/* 查看 API 了，如图 4-2 所示。

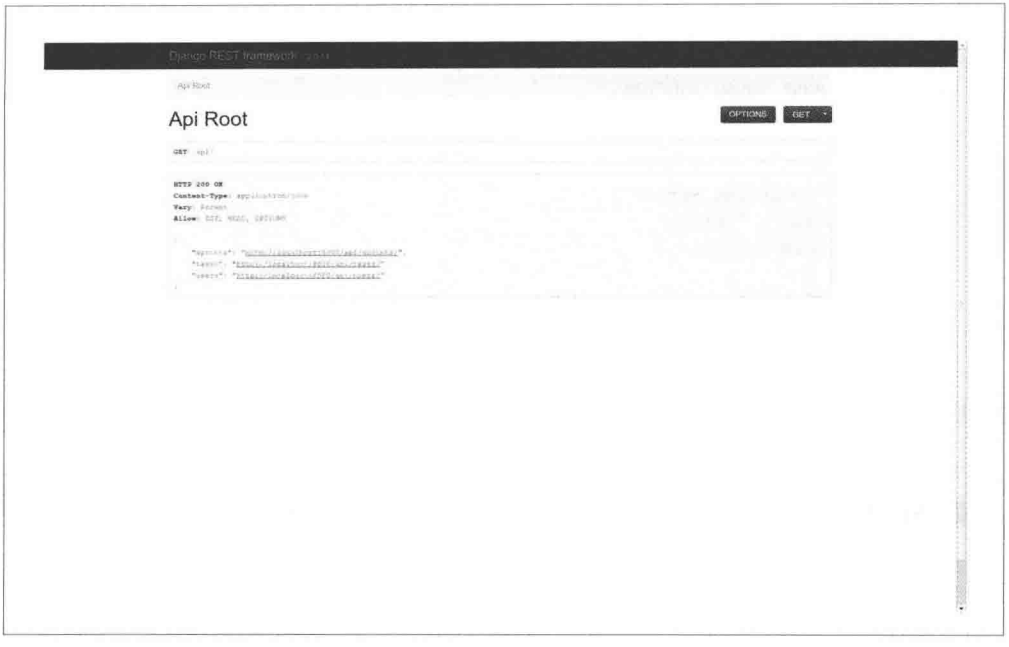


图 4-2：django-rest-framework 的 API 主页

如你所见，这里提供一个非常美观的可视化工具来查看我们新制作的 API。当对 */api/* 发出 GET 请求时显示该响应。它简单显示了顶层上所有由 API 定义的可用资源。尽管 API 的根节点不需要认证，但所有子资源都需要验证。当单击一个链接时，浏览器会提示输入用户名和密码来进行 HTTP 验证。可以使用之前创建的用户名和密码。

```
HTTP 200 OK
Vary: Accept
Content-Type: application/json
Allow: GET, HEAD, OPTIONS

{
     "sprints": "http://localhost:8000/api/sprints/",
     "tasks": "http://localhost:8000/api/tasks/",
     "users": "http://localhost:8000/api/users/"
}
```

单击 sprint 的链接会列出所有可用的 sprint 资源。目前我们还没有添加任何资源，所以返回是空的。不过页面底部有一个表单，可以来添加新的 sprint。我们来创建一个名叫“Something Sprint”的新 sprint，描述一栏填上“Test”，设置任意一个未来的日期作为结束日期。日期应当是 YYYY-MM-DD 这样的 ISO 格式。单击 POST 按钮会显示 201 状态码以及成功响应。

```
HTTP 201 CREATED
```

```
Vary: Accept
Content-Type: application/json
Location: http://localhost:8000/api/sprints/1/
Allow: GET, POST, HEAD, OPTIONS

{
    "id": 1,
    "name": "Something Sprint",
    "description": "Test",
    "end": "2020-12-31",
    "links": {
        "self": "http://localhost:8000/api/sprints/1/"
    }
}
```

刷新 sprint 列表就显示出该条 sprint，如图 4-3 所示。

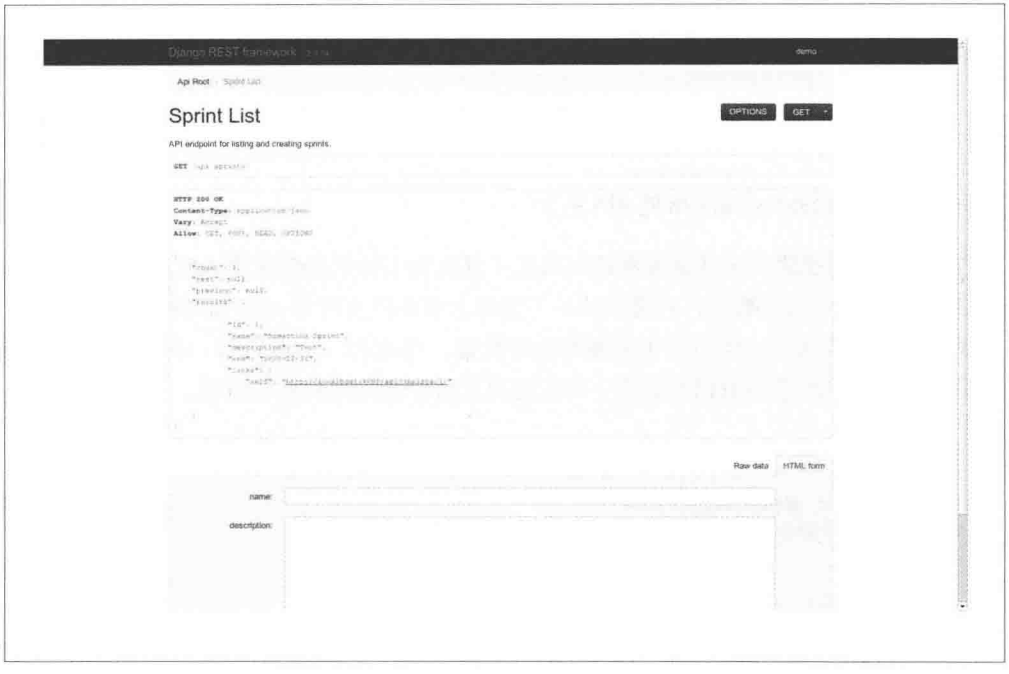


图 4-3：sprint API 截图

重新回退到位于 */api/* 的根节点，现在可以来浏览任务了。与 sprint 类似，从 API 根节点单击任务链接将显示出包含所有任务的列表，但现在里面还什么都没有。我们用底部的表单创建一个任务。给它起名为“First Task”，并使用之前创建的 sprint。同样，API 会返回 201 状态码以及新建任务的详细信息。

```
HTTP 201 CREATED
Vary: Accept
```

```
Content-Type: application/json
Location: http://localhost:8000/api/tasks/1/
Allow: GET, POST, HEAD, OPTIONS

{
    "id": 1,
    "name": "First Task",
    "description": "",
    "sprint": 1,
    "status": 1,
    "status_display": "Not Started"
    "order": 0,
    "assigned": null,
    "started": null,
    "due": null,
    "completed": null,
    "links": {
        "assigned": null,
        "self": "http://localhost:8000/api/tasks/1/",
        "sprint": "http://localhost:8000/api/sprints/1/"
    }
}
```

刷新任务列表将返回该任务，如图 4-4 所示。

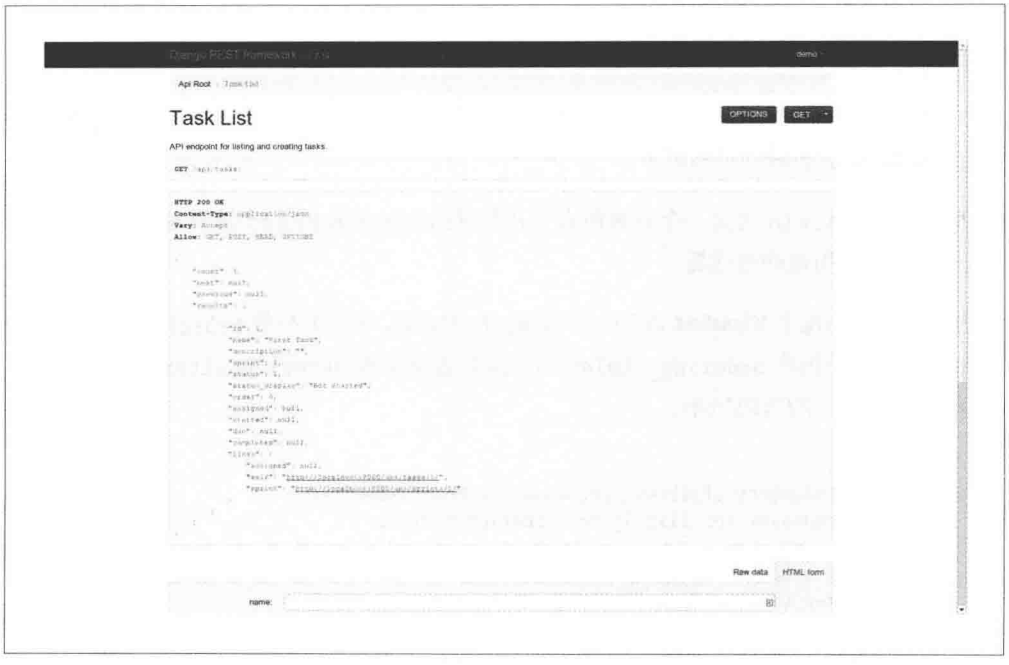


图 4-4：任务 API 截图

有了可浏览 API 的基本结构，让我们来为之后组织结构化页面数据添加一些过滤器。

添加过滤器

之前我们提到过，django-rest-framework 支持使用多种过滤器后端。可以把它们都添加到 *board/views.py* 中的 DefaultsMixin 来支持所有资源。

```
...
from rest_framework import authentication, permissions, viewsets, filters    ❶
...
class DefaultsMixin(object):
    """Default settings for view authentication, permissions,
    filtering and pagination."""

    authentication_classes = (
        authentication.BasicAuthentication,
        authentication.TokenAuthentication,
    )
    permission_classes = (
        permissions.IsAuthenticated,
    )
    paginate_by = 25
    paginate_by_param = 'page_size'
    max_paginate_by = 100
    filter_backends = (                                                      ❷
        filters.DjangoFilterBackend,
        filters.SearchFilter,
        filters.OrderingFilter,
    )
...
```

❶ 将 filters 添加到引用列表中。

❷ DefaultsMixin 定义一个存放所有可用的 filter_backends 的列表，通过已有的 ViewSet 启用这些过滤器。

我们可以通过为每个 **ViewSet** 添加一个 search_fields 字段来配置 SearchFilter；通过添加一个可用于排序 ordering_fields 的字段列表来配置 OrderingFilter。如下面这段 *board/views.py* 中的代码所示。

```
...
class SprintViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating sprints."""

    queryset = Sprint.objects.order_by('end')
    serializer_class = SprintSerializer
    search_fields = ('name', )                                               ❶
    ordering_fields = ('end', 'name', )                                      ❷


class TaskViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating tasks."""
```

```
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    search_fields = ('name', 'description', )                          ❸
    ordering_fields = ('name', 'order', 'started', 'due', 'completed', )  ❹


class UserViewSet(DefaultsMixin, viewsets.ReadOnlyModelViewSet):
    """API endpoint for listing users."""

    lookup_field = User.USERNAME_FIELD
    lookup_url_kwarg = User.USERNAME_FIELD
    queryset = User.objects.order_by(User.USERNAME_FIELD)
serializer_class = UserSerializer
search_fields = (User.USERNAME_FIELD, )                                ❺
```

❶❸ 将 search_fields 添加到所有视图中，允许搜索指定列表中的字段。

❺

❷❹ 在 AP 中使用 ordering_fields 对 sprint 和任务进行排序。在 API 响应中始终使用用户名对用户进行排序。

由于现在只有一个叫“First Task”的任务，通过 *http://localhost:8000/api/tasks/?search=foo* 搜索“foo”不会产生结果，但使用 *http://localhost:8000/api/tasks/?search=first* 搜索“first”将会得到。

要处理针对任务的额外筛选，我们可以使用 DjangoFilterBackend。这需要我们在 TaskViewSet 中定义一个 filter_class。filter_class 字段应当是 django_filters.FilterSet 的子类。在 *board/views.py* 中添加新的一行代码。

```
import django_filters

from .models import Task


class TaskFilter(django_filters.FilterSet):

    class Meta:
        model = Task
        fields = ('sprint', 'status', 'assigned', )
```

这里是对 django-filter 最基本的使用，基于模型定义来创建过滤器。每个 TaskFilter 中定义的字段都会被转化成一个查询参数，供客户端来过滤结果集。首先，必须要在 *board/views.py* 中把它和 TaskViewSet 关联起来。

```
...
from .forms import TaskFilter                                          ❶
from .models import Sprint, Task
from .serializers import SprintSerializer, TaskSerializer, UserSerializer
```

```
...
class TaskViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating tasks."""

    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    filter_class = TaskFilter                                          ❷
    search_fields = ('name', 'description', )
    ordering_fields = ('name', 'order', 'started', 'due', 'completed', )
...
```

❶❷ 这里引用了一个新的 TaskFilter 类，并分配给 **TaskViewSet.filter_class**。

有了这些，客户端就可以对 sprint、status 或 assigned 用户进行过滤。不过，对任务过滤不一定需要 sprint，但这个过滤器不支持在无 sprint 的情况下选择任务。在现有的数据模型中，未被分配给 sprint 的任务会被视作后台日志任务。要解决这个问题，可以在 *board/views.py* 的 TaskFilter 中添加一个新字段。

```
import django_filters

from .models import Task


class NullFilter(django_filters.BooleanFilter):
    """Filter on a field set as null or not."""

    def filter(self, qs, value):
        if value is not None:
            return qs.filter(**{'%s__isnull' % self.name: value})
        return qs


class TaskFilter(django_filters.FilterSet):

    backlog = NullFilter(name='sprint')

    class Meta:
        model = Task
        fields = ('sprint', 'status', 'assigned', 'backlog', )
```

这样访问 *http://localhost:8000/api/tasks/?backlog=True* 将会返回所有未分配给 sprint 的任务了。TaskFilter 还有个问题是，assigned 字段是通过 pk 来引用用户，但 API 的其余部分都是使用 username 作为唯一标识符。我们通过调整 *board/views.py* 下的 ModelChoiceField 来修正这个问题。

```
import django_filters

from django.contrib.auth import get_user_model                         ❶

from .models import Task

User = get_user_model()                                                ❷
```

```
class NullFilter(django_filters.BooleanFilter):
    """Filter on a field set as null or not."""

    def filter(self, qs, value):
        if value is not None:
            return qs.filter(**{'%s__isnull' % self.name: value})
        return qs


class TaskFilter(django_filters.FilterSet):

    backlog = NullFilter(name='sprint')

    class Meta:
        model = Task
        fields = ('sprint', 'status', 'assigned', 'backlog', )

    def __init__(self, *args, **kwargs):                                    ❸
        super().__init__(*args, **kwargs)
        self.filters['assigned'].extra.update(
            {'to_field_name': User.USERNAME_FIELD})
```

❶❷ 像其他完成的模块那样获取对已安装 User 模型的引用。

❸ 更新 assigned 字段，使用 User.USERNAME_FIELD 而不是默认的 pk 作为字段引用。

有了这些修改，我们就可以通过 *http://localhost:8000/api/tasks/?assigned=demo* 代替 *http://localhost:8000/api/tasks/?assigned=1* 来获取只分配给一个 *demo* 用户的任务了。

sprint 还可以使用一些复杂的过滤。客户端也许会对还没结束或者在某个区间内结束的 sprint 感兴趣。我们可以通过在 *board/views.py* 中创建 SprintFilter 来实现它。

```
...
from .models import Task, Sprint
...

class SprintFilter(django_filters.FilterSet):

    end_min = django_filters.DateFilter(name='end', lookup_type='gte')
    end_max = django_filters.DateFilter(name='end', lookup_type='lte')

    class Meta:
        model = Sprint
        fields = ('end_min', 'end_max', )
```

然后我们在 *board/views.py* 中以相似风格来把它与 SprintViewSet 联系起来。

```
...
from .forms import TaskFilter, SprintFilter
...
```

```
class SprintViewSet(DefaultsMixin, viewsets.ModelViewSet):
    """API endpoint for listing and creating sprints."""

    queryset = Sprint.objects.order_by('end')
    serializer_class = SprintSerializer
    filter_class = SprintFilter                                    ❶
    search_fields = ('name', )
    ordering_fields = ('end', 'name', )
```

❶ 与 TaskViewSet 类似，SprintViewSet 现在使用新的 SprintFilter 定义 filter_class 字段。

访问 *http://localhost:8000/api/sprints/?end_min=2014-07-01* 将显示所有 2014 年 7 月 1 日之后结束的 sprint，访问 *http://localhost:8000/api/sprints/?end_max=2014-08-01* 将显示所有在 2014 年 8 月 1 日之前结束的 sprint。还可以将这些组合起来，把 sprint 限制在某个给定日期范围内。

由于 sprint 和任务的视图支持过滤，可以通过修改 *board/serializers.py* 来添加链接，将 sprint 和用户与相关的任务相联系。

```
...

class SprintSerializer(serializers.ModelSerializer):
...
    def get_links(self, obj):
        request = self.context['request']
        return {
            'self': reverse('sprint-detail',
                kwargs={'pk': obj.pk}, request=request),
            'tasks': reverse('task-list',
                request=request) + '?sprint={}'.format(obj.pk),
        }
...
class UserSerializer(serializers.ModelSerializer):
...
    def get_links(self, obj):
        request = self.context['request']
        username = obj.get_username()
        return {
            'self': reverse('user-detail',
                kwargs={User.USERNAME_FIELD: username}, request=request),
            'tasks': '{}?assigned={}'.format(
                reverse('task-list', request=request), username)
         }
```

有了过滤器，为了继续利用可浏览界面的好处，让我们添加一些验证来保障数据状态的安全。

添加验证

尽管使用前端界面是很有用的，但对 API 不需要进行太多研究就会发现存在一些问题。特别是，它允许修改不应该被修改的东西。此外它还允许创建已经结束的 sprint。这都是来自于序列化器的问题。

到目前为止，我们的关注点在于如何把数据模型序列化成字典并在后来转化成客户端的 JSON、XML、YAML 等。关于客户端请求标记如何转化成创建模型实例这一部分，我们还没有做任何的修改工作。对于典型的 Django 视图，这部分可以由 Form 或 `ModelForm` 来完成。在 `django-rest-framework` 中，这部分内容由序列化器来处理。不出意外，这里的 API 和 Django 表单的 API 类似。事实上，序列化器字段使用了 Django 表单字段中现有的逻辑。

API 需要避免创建在当前日期和时间之前已经出现的 sprint。要解决这个问题，`SprintSerializer` 需要检查客户端提交的截止日期数值。每个序列化器字段都有个 `validate_<field>` 标记，用于对该字段进行额外的验证。再有，这项功能对应于 Django 表单中的 `clean_<field>`。这项功能要加入到 *board/serializers.py* 的 `SprintSerializer` 中。

```
from datetime import date                                                    ❶

from django.contrib.auth import get_user_model
from django.utils.translation import ugettext_lazy as _                      ❷

from rest_framework import serializers
from rest_framework.reverse import reverse

from .models import Sprint, Task


User = get_user_model()


class SprintSerializer(serializers.ModelSerializer):

    links = serializers.SerializerMethodField()

    class Meta:
        model = Sprint
        fields = ('id', 'name', 'description', 'end', 'links', )

    def get_links(self, obj):
        request = self.context['request']
        return {
            'self': reverse('sprint-detail',
                kwargs={'pk': obj.pk}, request=request),
            'tasks': reverse('task-list',
                request=request) + '?sprint={}'.format(obj.pk),
        }
```

```
    def validate_end(self, value):                                    ❸
        new = self.instance is None
        changed = self.instance and self.instance.end != value
        if (new or changed) and (value < date.today()):
            msg = _('End date cannot be in the past.')
            raise serializers.ValidationError(msg)
        return value
...
```

❶❷ 这些内容是对标准库中 datetime 以及能够转换错误信息的 ugettext_lazy 的新的引用。

❸ validate_end 对所有新创建或者更新的 sprint 进行检查，验证结束日期是否大于或等于当前日期。

让我们在 *http://localhost:8000/api/sprints/* 中尝试创建一个过往的 sprint，看看验证是否正常运行。现在可能出现 400 BAD REQUEST 的响应。

```
HTTP 400 BAD REQUEST
Vary: Accept
Content-Type: application/json
Allow: GET, POST, HEAD, OPTIONS

{
    "end": [
        "End date cannot be in the past."
    ]
}
```

由于只是对当前项目进行检查，保证了校验只被应用到新创建的对象或正在修改结束日期的对象。而且允许对过往 sprint 的名字而不是结束日期进行修改。

和这个问题类似，还没有对新建和编辑进行校验。这些任务既可以被添加到已完成的 sprint 中，也可以在没被标记为已完成时就设置了完成日期。此外，任务还可以在没开始时就设置开始日期。

涉及多个字段的验证条件要通过 validate 方法来处理，与表单中的 clean 方法一致，如下面这段 *board/serializers.py* 中的代码所示。

```
...
class TaskSerializer(serializers.ModelSerializer):
...
    def validate_sprint(self, value):                                 ❶
        if self.instance and self.instance.pk:
            if value != self.instance.sprint:
                if self.instance.status == Task.STATUS_DONE:
                    msg = _('Cannot change the sprint of a completed task.')
                    raise serializers.ValidationError(msg)
                if value and value.end < date.today():
```

```
                msg = _('Cannot assign tasks to past sprints.')
                raise serializers.ValidationError(msg)
        else:
            if value and value.end < date.today():
                msg = _('Cannot add tasks to past sprints.')
                raise serializers.ValidationError(msg)
        return value

    def validate(self, attrs):                                          ❷
        sprint = attrs.get('sprint')
        status = attrs.get('status',Task.STAUS_TODO)
        started = attrs.get('started')
        completed = attrs.get('completed')
        if not sprint and status != Task.STATUS_TODO:
            msg = _('Backlog tasks must have "Not Started" status.')
            raise serializers.ValidationError(msg)
        if started and status == Task.STATUS_TODO:
            msg = _('Started date cannot be set for not started tasks.')
            raise serializers.ValidationError(msg)
        if completed and status != Task.STATUS_DONE:
            msg = _('Completed date cannot be set for uncompleted tasks.')
            raise serializers.ValidationError(msg)
        return attrs
...
```

❶ `validate_sprint` 确保 sprint 在任务完成前不被修改，而且任务也不会分配给已完成的 sprint。

❷ `validate` 确保字段的组合对任务有意义。

由于这些验证只由序列化器而非模型来处理，如果项目中的其他部分正在修改或创建模型，则不会进行这些检查。如果启用了 Django 管理工具，那么过往的 sprint 依旧可以被添加。类似的，如果项目加入了一些引用代码，它们依然可以添加过往的 sprint。

验证完成后，来看看怎样使用 Python 客户端浏览我们的 RESTful API，看看如何使用 Python 捕获数据。

使用 Python 客户端

可浏览的 API 是一种非常简便的研究 API 的方式，它确保了资源链接是有意义的。然而，对于在编程客户端上使用 API，它还提供了同样的体验。要理解开发者如何使用 API，可以用 Python 写一个简单的客户端，这里我们使用流行的 `requests` 库。首先，使用 `pip` 来安装 `requests` 库：

```
hostname $ pip install requests==2.7.0
```

在一个终端中用 `runserver` 运行服务器，在另一个终端中打开 Python 的交互 shell。与浏览器相类似，我们从在 *http://localhost:8000/api/* 上获取 API 根节点开始：

```
hostname $ python
>>> import requests
>>> import pprint
>>> response = requests.get('http://localhost:8000/api/')
>>> response.status_code
200
>>> api = response.json()
>>> pprint.pprint(api)
{'sprints': 'http://localhost:8000/api/sprints/',
 'tasks': 'http://localhost:8000/api/tasks/',
 'users': 'http://localhost:8000/api/users/'}
```

API 根节点列出了所有在它之下的 sprint、任务和用户的子资源。在现有的配置下这一视图不需要任何认证，不过余下的资源还需要认证。我们在 shell 中继续，如果尝试在未授权的情况下获取一个资源会返回 401 错误：

```
>>> response = requests.get(api['sprints'])
>>> response.status_code
401
```

可以通过用 `auth` 参数传递用户名和密码对客户端进行认证：

```
>>> response = requests.get(api['sprints'], auth=('demo', 'test'))
>>> response.status_code
200
```

记住，这个例子假设了存在一个用户名为 *demo*，密码为 *test* 的用户。这些设置应该已经在本章前面的“模型”一节里创建数据表的过程中完成。

使用这个 *demo* 用户，要创建一个新 sprint 并添加一些任务。创建 sprint 需要发送 POST 请求给 sprint 端点，并提供 sprint 的名字和结束日期。

```
>>> import datetime
>>> today = datetime.date.today()
>>> two_weeks = datetime.timedelta(days=14)
>>> data = {'name': 'Current Sprint', 'end': today + two_weeks}
>>> response = requests.post(api['sprints'], data=data, auth=('demo', 'test'))
>>> response.status_code
201
>>> sprint = response.json()
>>> pprint.pprint(sprint)
{'description': '',
 'end': '2014-08-31',
```

```
 'id': 2,
 'links': {'self': 'http://localhost:8000/api/sprints/2/',
           'tasks': 'http://localhost:8000/api/tasks/?sprint=2'},
 'name': 'Current Sprint'}
```

创建好了 sprint，现在来添加一些相关的任务。sprint 的 URL 定义了对它的唯一引用，并将在创建任务时将其传递给请求。

```
>>> data = {'name': 'Something Task', 'sprint': sprint['id']}
>>> response = requests.post(api['tasks'], data=data, auth=('demo', 'test'))
>>> response.status_code
201
>>> task = response.json()
>>> pprint.pprint(task)
{'assigned': None,
 'completed': None,
 'description': '',
 'due': None,
 'id': 2,
 'links': {'assigned': None,
           'self':'http://localhost:8000/api/tasks/2/',
           'sprint': 'http://localhost:8000/api/sprints/2/'},
 'name': 'Something Task',
 'order': 0,
 'sprint': 2,
 'started': None,
 'status': 1,
 'status_display': 'Not Started'}
```

我们可以通过给 URL 发送附带新信息的 PUT 请求更新任务。让我们来更新它的状态和开始日期，并把它指派给 *demo* 用户。

```
>>> task['assigned'] = 'demo'
>>> task['status'] = 2
>>> task['started'] = today
>>> response = requests.put(task['links']['self'],
        ...data=task, auth=('demo', 'test'))
>>> response.status_code
200
>>> task = response.json()
>>> pprint.pprint(task)
{'assigned': 'demo',
 'completed': None,
 'description': '',
 'due': None,
 'id': 2,
 'links': {'assigned': 'http://localhost:8000/api/users/demo/',
          'self': 'http://localhost:8000/api/tasks/2/',
          'sprint': 'http://localhost:8000/api/sprints/2/'},
 'name': 'Something Task',
 'order': 0,
 'sprint': 2,
 'started': '2014-08-17',
```

```
    'status': 2,
    'status_display': 'In Progress'}
```

请注意在整个过程中 URL 并非由客户端创建，而是由 API 提供。与使用可浏览的 API 类似，客户端不需要知道 URL 是如何构建的。客户端浏览 API，发现资源和逻辑在于解析服务器响应的信息并管理它们。这就让客户端非常容易维护。

下一步

有了 REST API，以及对如何用 Python 客户端使用 API 的基本理解，现在我们可以继续在 JavaScript 编写的客户端中使用 API。下一章将会探索如何在这个 REST API 之上使用 Backbone.js 创建一个单页面的 Web 应用。

第 5 章

使用 Backbone.js 的客户端 Django

从 20 世纪 90 年代以来，最初使用的一些框架已经成为了创作网站、应用和服务的主流方式。因为框架可以帮助减轻创建这些内容所花费的一般开销，所以不难理解框架得以很快流行的缘故。

在 Django 最初发布时，它被视为一种非常独特且富有创新的 Web 开发方式。让我们来回顾一下这段时间的 Web 开发。表 5-1 列出了 2005 年 11 月末全球浏览器市场的份额。

表 5-1：2005 年 11 月全球浏览器使用统计

浏览器	百分比
Internet Explorer	84.45%
Firefox	11.51%
Safari	1.75%
Opera	0.77%

如你所见，在 Django 发布时移动浏览器甚至还没有达到表中所列这些项目的规模。直到第一代 iPhone 发布（2007 年）时移动浏览器才开始攀升。同时涌现出一些开发 Web 应用、创建快捷的交互网站更好的工具（例如 2006 年的 jQuery）。

从新的移动时代开始，创建有用的应用程序接口（API）实际上已经成为管理客户端数据的技术。相应地，人们发明了多种客户端 MVC 框架来帮助引导处理不同种类数据集的过程。

每一款客户端 MVC 框架都有不同等级的复杂度、学习曲线，以及创建应用结构的特性集合。有些时候很难决定哪款框架对应用最适合，或对项目最有帮助。图 5-1 是个简单的图表，列出了一些常见的 JavaScript MVC 框架的复杂程度、学习曲线以及特性集合。

图 5-1：以复杂程度排列的 JavaScript MVC 框架

对于你和你的团队来说，选择正确的框架是一个非常重要的决定，必须在涉及上述选项时考虑周全。如图 5-1 所示，这些客户端框架中的每一个都有与它自身等级一致的丰富的特性集合可供选择。从可以用于全栈工作的 Meteor，到简单却功能不减的 Angular.js，对于选择哪一款最适合你的 Django 项目是很难决定的。

因为本书重点关注的是更简单小巧模型的开发，很少涉及其他 JavaScript 框架提供的众多 UI 特性集合，所以我们决定使用 Backbone 作为示例项目。不过，其中许多有关代码组织、客户端服务器分离的相同方法同样也可以应用到其他客户端框架上。

Backbone 简述

Backbone 于 2010 年末由 Jeremy Ashekenas 发布。最初的目的是通过客户端架构来创建一个结构化的、简单管理项目的 JavaScript 文件的途径。Backbone 经常被视为最简单的客户端 MVC 方案，它只需要两个依赖程序：Underscore.js 和 jQuery。Underscore.js 是一个代码库，用于添加跨浏览器运行的功能性程序工具，如 `map`、`reduce`、`filter`、`pluck` 和 `partial` 等。Underscore.js 同样有个模板引擎，可以编译为 JavaScript 方法，将 JSON 数据源渲染为 HTML 或其他文本。jQuery 是一款流行的跨平台 DOM 管理代码库，带有用于创建 Backbone 所需的 AJAX 请求的工具。

和其他框架一样，理解创建项目时所需的语法、布局和功能是很重要的。我们来花点时间回顾一下，在使用 Backbone 时，模型、视图、控制器和路由是什么意思。

Models（模型）

模型是所有 Backbone 项目的主要信息来源。它们是定义项目所需数据和对象的地方，与 Django 的模型恰当对应。

Collections（集合）

集合是一种有序的方法，用于帮助组织多个模型以及模型内部的数据集合。MVC 框架的这些方法帮助我们以有组织的方式管理应用数据的不同集合和类型。Backbone 中的集合大体和 Django 中的 *queryset* 一致。

Views（视图）

Backbone 中的视图是创造应用可视化表示的一部分，与在 Django 内使用的视图和模板类似。视图与模型异步工作，创建与应用数据之间的动态交互。

Routers（路由）

路由是一种通过 History API 来为应用创建 URL 的方式。在路由之前，Backbone 应用使用哈希代码段来创建导航。将路由连接事件和动作，可以在单页面 Web 应用上制作出多页面的体验。

有了这些关于 Backbone 中模型、集合、视图以及路由的基础知识，就可以开始理解这款非常简单的框架是怎样在创建单页面 Web 应用时变得如此强大。现在将 Django 连接 Backbone，继续完成我们第 4 章的 REST API 项目。

有关 Backbone 更多的内容和教程，请访问官方网站。

设置项目文件

回顾第 4 章的工作，快速查看一下我们的 Scrum 板项目是在哪里结束的。在第 4 章中，我们完成了如下工作：

- 使用 `django-rest-framework` 创建了一个 REST API。
- 启动了一个叫 `scrum` 的 Django 项目。
- 启动了一个叫 `board` 的 Django 应用。
- 添加了带有序列器的 REST API 校验。

JavaScript 依赖

现在我们需要添加一个 *templates* 文件夹来存放单页面模板。同样还需要添加一个存放客户端依赖包的 *static* 文件夹。下面是我们的文件夹结构组织：

```
scrum/
board/
    migrations/
    static/
        board/
            css/
            js/
    templates/
    forms.py
    models.py
    serializers.py
    test.py
    urls.py
    views.py
```

访问以下网址下载 Backbone 以及它所需的所有相关文件：

- Backbone 1.1.2：*http://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.1.2/backbone.js*
- jQuery 2.1.1：*http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.js*
- Underscore 1.6.0：*http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.6.0/underscore.js*

尽管这些文件都可以通过 CDN 得到，但我们还是直接把它们下载到项目中，避免在本地工作时依赖网络连接。这里使用的也是未压缩的代码库版本，方便在本地调试。在部署产品时，可以选择直接使用 CDN 上的压缩版，或者使用 `django-compressor` 来自行压缩（类似第 3 章）。

客户端依赖管理

近些年，在解决客户端依赖管理的问题上，出现了许多竞争方案。各个方案间有许多割裂。某些客户端代码包，如 Backbone，支持通过使用 npm、bower、volo、jam 和 ringojs 在内的包管理器安装。如果你青睐使用包管理方案管理 JavaScript 和 CSS 依赖，bower 似乎处于领先地位，但只有时间能证明。也许最好的客户端包管理器还没写出来。

下载完所需文件后，把它们放到一个叫 *vendor* 的文件夹里，这个文件夹在我们的 *board/static/board* 文件夹中：

```
scrum/
board/
  migrations/
  static/
    board/
      css/
      js/
      vendor/
        backbone.js
        jquery.js
        underscore.js
  forms.py
  models.py
  serializers.py
  test.py
  urls.py
  views.py
```

这些文件和文件夹被嵌套在 *static* 文件夹下一个叫 *board* 的文件夹里，以此来满足 Django 有关可复用应用静态资源的命名空间的惯例。现在我们再向 *board/templates/board* 文件夹中加入一个简单的 *index.html* 文件，开始应用的布局：

```
scrum/
board/
  migrations/
  static/
    board/
    css/
    js/
    vendor/
  templates/
    board/
      index.html
  forms.py
  models.py
  serializers.py
  test.py
  urls.py
  views.py
```

相似地，*index.html* 文件位于 *board* 文件夹下，为模板创建了一个命名空间来做应用模板。由于我们只是在创建单页面 Web 应用，所以这个项目仅仅需要单个模板。通常在 Django 项目中会创建一个包含许多模板的 *templates* 文件夹。由于我们绝大多数的模板代码会使用 Underscore.js 的 `_.template` 工具编写，因此所有的交互都会发生在这一单个模板中。

打开 *index.html* 文件（在 *board/templates/board* 里），开始布置有关模板代码依赖的基础工作：

```
{% load staticfiles %}
<!DOCTYPE html>
```

```
<html class="no-js">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <title>Scrum Board</title>
        <meta name="description" content="">
        <meta name="viewport" content="width=device-width, initial-scale=1">
    </head>
    <body>
        <script src="{% static 'board/vendor/jquery.js' %}"></script>
        <script src="{% static 'board/vendor/underscore.js' %}"></script>
        <script src="{% static 'board/vendor/backbone.js' %}"></script>
    </body>
</html>
```

在页面加入依赖文件时，加载顺序很重要。当在页面上链接文件时，请确保遵循上述结构。

有了文件夹结构、依赖文件、模板之后，我们开始思考如何搭建实际的 Backbone 应用文件结构。

Backbone 应用文件的组织

在创建有良好架构的应用时，模块化是关键的一点。让我们花点时间稍微回顾一下之前有关简单 Backbone 模板的概况，我们至少需要用模型、视图、路由来创建一个完整的应用。当我们使用单个页面存储这些代码时，试图架构出一个洁净、模块化的代码基础并没有太多意义。

在 *static/board/js* 文件夹下，为应用的每个部分添加待编写的空白文件：

```
scrum/
board/
  migrations/
  static/
    board/
    css/
    js/
      models.js
      router.js
      views.js
      vendor/
    templates/
    forms.py
    models.py
    serializers.py
    test.py
```

```
urls.py
views.py
```

下面花费一些时间将新文件的不同部分与文件需要的内容进行对应，以便后面开始编写项目时进行参考。

models.js

Backbone 模型是定义数据模型的地方，也是状态管理的地方。在该文件中我们要处理 CSRF（跨站请求伪造）记号和用户会话管理，同时还要将 API 接收的数据显示在页面上。

views.js

每个 Backbone 视图都要包含各类需要的“页面”，使用户在应用中方便地操作。视图同样会包含 Scrum 板中我们用来更新 sprint 和任务信息的表单数据。

router.js

简单来说，这个地方是在应用中对每个 URL 路径初始化的地方。

创建完项目结构，接着就可以开始把 Django 连接到新布局的 Backbone 应用上了。

让我们把刚刚描述的文件加入到 *board/templates/board* 下的 *index.html* 网页中：

```
{% load staticfiles %}
<!DOCTYPE html>
<html class="no-js">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <title>Scrum Board  Project</title>
        <meta name="description" content="">
        <meta name="viewport" content="width=device-width, initial-scale=1">
    </head>
    <body>
        <script src="{% static 'board/vendor/jquery.js' %}"></script>
        <script src="{% static 'board/vendor/underscore.js' %}"></script>
        <script src="{% static 'board/vendor/backbone.js' %}"></script>
        <script src="{% static 'board/js/models.js' %}"></script>
        <script src="{% static 'board/js/views.js' %}"></script>
        <script src="{% static 'board/js/router.js' %}"></script>
    </body>
</html>
```

我们要直接把这些文件放在依赖中，因为即将要编写的代码需要这些文件。

连接 Backbone 到 Django

在开始讨论用什么方法将 Backbone 连接到 Django 最好时，要考虑以下几个问题：是否需要某种包来进行连接？在进行安全连接时有哪些特别的任务需要执行？

在这一部分，我们将开始对上下文进行布局，以便为我们的 Scrum 板应用创建洁净且可复用的基础设施。

在 Django 应用中，服务器端创建的动态代码生存在模板上下文变量中。由于我们的 Django 模板代码与应用所需的代码不可互换，只好通过创建一种易理解的方式来转移数据。通过将模板上下文变量用 JSON 这种 Backbone 应用可以读取的方式来传递，我们可以很轻易地实现这个过程。

由于我们在 Backbone.js 应用间传递变量时已经创建了这个数据结构，因此可以为我们的模型、集合、视图以及路由添加一些初始的空值。打开 *index.html* 文件（在 *board/templates/board* 下），在 Backbone 依赖后面添加下面的内容：

```
...
<script src="{% static 'board/vendor/backbone.js' %}"></script>
<script id="config" type="text/json">
  {
    "models": {},
    "collections": {},
    "views": {},
    "router": null
   }
</script>
<script src="{% static 'board/js/models.js' %}"></script>
...
```

通常，我们会在应用的各种文件中复用这些变量。与其每次都要对它们进行加载，不如加入一些模块，把它们添加到静态的 *js* 文件夹（在 *board/static/board* 下）下的一个叫 *app.js* 的初始化文件中，代码如下：

```
var app = (function ($) {
    var config = $('#config'),
        app = JSON.parse(config.text());

    return app;
})(jQuery);
```

如你所见，我们使用自调用函数来解析在模板中创建的 JSON，并将它转换成在应用中到处可以使用的变量。把解析过的配置值返回给全局的 App 变量，以便在整个应用中都使用这些值启动配置和状态。

完成这些设置后，我们就可以以洁净的模块方式开始搭建我们的客户端架构了。我们创建的 *index.html* 文件（位于 *board/templates/board*）应当类似如下代码：

```
{% load staticfiles %}
<!DOCTYPE html>
<html class="no-js">
    <head>
        <meta charset="utf-8">

        <meta http-equiv="X-UA-Compatible" content="IE=edge">

        <title>Scrum Board Project</title>

        <meta name="description" content="">

        <meta name="viewport" content="width=device-width, initial-scale=1">
    </head>
    <body>
        <script src="{% static 'board/vendor/jquery.js' %}"></script>
        <script src="{% static 'board/vendor/underscore.js' %}"></script>
        <script src="{% static 'board/vendor/backbone.js' %}"></script>
        <script id="config" type="text/json">
            {
                "models": {},
                "collections": {},
                "views": {},
                "router": null
            }
        </script>

        <script src="{% static 'board/js/app.js' %}"></script>                    ❶
        <script src="{% static 'board/js/models.js' %}"></script>
        <script src="{% static 'board/js/views.js' %}"></script>
        <script src="{% static 'board/js/router.js' %}"></script>
    </body>
</html>
```

❶ 在配置模块之后和项目文件之前引用 *app.js*。

JavaScript 模块

正如客户端包管理器一样，也有众多有关如何搭建模块化 JavaScript 应用的工具和选项。不过，我们觉得这个方法学起来既简单又容易理解。该方法思路清晰，步骤灵活。话虽如此，还有许多像 CommonJS 和 Asynchronous Module Definition（AMD）这样更加正式的模块加入到 JavaScript 社区，还有一些模块是 ECMAScript 6 的一部分。遵循这些规范，像 RequireJS、Browserify 和 ES6 Module Transpiler 这样的工具为我们在浏览器中使用模块定义提供了帮助。在许多其他 Backbone 教程和资源中也会用到这些模块，很值得对它们进行一番研究。

现在，我们已经可以在 Backbone 应用中使用多个变量了。下面继续搭建路由，看看它们是怎样与 REST API 交互的。

客户端 Backbone 路由

在开始创建 Backbone 应用的结构中，要使用的主要组件是客户端路由和视图。然而在 Django 中创建路由和在客户端应用中生成路由会有很大区别。

在 Django 中，我们要为绝大多数交互创建路由。例如，我们通常会为每个需要执行的 CRUD（创建、读取、更新、删除）任务创建路由。有了 Backbone 之后，我们可以利用它的状态特性在视图里封装这些任务，避免创建不必要的路由。我们将在本章中通过示例说明如何构建应用架构。

让我们从创建一个基础的主页视图和路由做起，开始创建应用架构的基础部分。

创建基础的主页视图

既然我们要制作一个单页面的 Web 应用，很容易决定要从主页开始。正如本章之前所提到的，我们的绝大多数模板代码写在 *index.html* 文件里，并渲染到浏览器上。我们在 *views.js* 文件（位于 *board/static/board/js*）中创建一个简单的视图，来渲染主页模板：

```
(function ($, Backbone, _, app) {                              ❶

    var HomepageView = Backbone.View.extend({

        templateName: '#home-template',                                ❷
        initialize: function () {
            this.template = _.template($(this.templateName).html());   ❸
        },
        render: function () {
            var context = this.getContext(),
                html = this.template(context);
             this.$el.html(html);
        },
       getContext: function () {
            return {};
        }
    });

    app.views.HomepageView= HomepageView;                              ❹

})(jQuery, Backbone, _, app);                                          ❺
```

❶❺ 你会注意到在这个立即调用的函数中，我们使用了一些工具：jQuery、Backbone、Underscore.js 和 App.js。这不仅使得代码更加简洁，同时还可以访问到由 *app.js* 定

义的全局变量。这个函数的表达式基于 JavaScript 的闭包，函数没有返回值，用来把对象隔离在全局范围之外。

❷ 需要告诉应用用什么 Underscore.js 模板来渲染主页。我们可以方便地借助即将用到的 ID 选择器来指派模板名称。请记住这一点，因为接下来我们就要开始搭建主页模板了。

❸ 使用 Underscore.js 的 `_.template` 工具将主页模板渲染到 HTML。这将在之后的段落中得以处理。

❹ 由于 `HomepageView` 被加入到 `app.views` 字典里，因此可以将其在应用的其他部分（也就是路由中）中使用。

现在，我们已经为设置路由连接我们简单的主页视图做好了准备。

设置最简单的路由

正如之前示例展示的那样，考虑如何通过架构代码创建一个稳定的基础供以后使用是很重要的。在 *router.js* 文件（位于 *board/static/board/js*）的上下文中，我们将使用 *view.js* 中相同的方法并将 jQuery、Backbone、underscore.js 和 app 的全局配置传递给一个匿名函数：

```
(function ($, Backbone, _, app) {
  var AppRouter= Backbone.Router.extend({
    routes: {
        '': 'home'                                                    ❶
    },
    initialize: function (options) {
        this.contentElement= '#content';                              ❷
        this.current= null;
        Backbone.history.start();
    },
    home: function () {
        varview = new app.views.HomepageView({el: this.contentElement});
        this.render(view);
    },
    render: function (view) {                                         ❸
        if (this.current) {
            this.current.undelegateEvents();
            this.current.$el= $();
            this.current.remove();
        }
        this.current= view;
        this.current.render();
     }
  });
```

```
    app.router= AppRouter;                                          ❹

})(jQuery, Backbone, _, app);
```

❶ 这是为应用定义路由的地方。由于我们目前只关注主页路由，因此只简单地加入这一单个引用。

❷ 类似我们之前在 *index.html* 中的结构，这一行是我们引用 ID 选择器的地方，它是 Underscore 模板将要加载到的地方。调用 `Backbone.history.start()` 会触发路由，去为用户调用第一个匹配好的路径。

❸ `render` 是一个帮助函数，用于从一个视图切换到另一个视图时帮助路由追踪。

❹ 在这里，将路由定义附加到 app 配置中，让它在整个项目范围内可用。

有了主页视图和路由，我们可以继续来搭建第一个 Underscore.js 模板。

使用 Underscore.js 中的 __.template

当为 Django 应用开发模板时，最好从考虑模板继承以及每个模板与其他模板如何关联开始。当创建 Backbone 应用时，我们通常会使用强大的 Underscore.js 库来将每个模板放置在各自包含的代码中，来开始这个过程。

Underscore.js 是一系列工具的集合，用于和 JavaScript 配合创建函数式的编程环境。有超过 60 种函数供我们使用，来实现我们用函数式编程语言时所期望的典型动作。在本章这一节中，我们将主要使用 `_.template` 工具，因为它与为 Backbone 编写的应用相关。

`_.template` 工具的主要目的是创建编译到函数中的 JavaScript 模板，该模板可以很便利地被渲染到页面上。正如在构建视图时简单描述的那样，我们需要在 *board/template/board/index.html* 中创建一个 `home-template`，作为对主页模板代码的引用。

```
{% load staticfiles %}
<!DOCTYPE html>
<html class="no-js">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <title>Scrum Board</title>
        <meta name="description" content="">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <script type="text/html" id="home-template">                    ❶
            <h1>Welcome to my Scrum Board Project!</h1>
        </script>
    </head>
    <body>
```

```
        <div id="content"></div>                                          ❷

        <script src="{% static 'board/vendor/jquery.js' %}">
        </script><script src="{% static 'board/vendor/underscore.js' %}"></script>
        <script src="{% static 'board/vendor/backbone.js' %}"></script>
        <script id="config" type="text/json">
            {
                "models": {},
                "collections": {},
                "views": {},
                "router": null
            }
        </script>

        <script src="{% static 'board/js/app.js' %}"></script>
        <script src="{% static 'board/js/models.js' %}"></script>
        <script src="{% static 'board/js/views.js' %}"></script>
        <script src="{% static 'board/js/router.js' %}"></script>
    </body>
</html>
```

❶ 这里我们定义了 home-template，目前它只是静态 HTML。

❷ 加入一个 ID 为 content 的 div，匹配路由用的选择器。

你会注意到，我们已经把主页模板加入到页面的 <head> 部分。当考虑页面加载顺序时，我们希望确保模板在需要被视图代码使用前已经定义了。由于文件从上到下依次加载，我们把模板代码放在这里确保当视图代码载入时已经被定义。

渲染主页的最后工作就是在页面载入时开启路由。可以通过在 *board/static/board/js/app.js* 中初始化路由来实现：

```
var app = (function ($) {
    var config = $('#config'),
        app = JSON.parse(config.text());

    $(document).ready(function () {                                       ❶
        var router = new app.router();
    });

    return app;
})(jQuery);
```

❶ 我们同样要确保路由在 DOM 已经完全为应用运行准备好后才被初始化。最初的 app.router 会是 null，当 *router.js* 载入时它会被加入。当使用 jQuery 的 $(document).ready 事件时，确保 app.router 会在 *router.js* 加载后初始化。

要查看这些综合在一起的效果，需要创建一个简单的 Django 视图 *scrum/urls.py* 来渲染 *index.html*：

```
from django.conf.urlsimport include, url
from django.views.generic import TemplateView                              ❶

from rest_framework.authtoken.views import obtain_auth_token

from board.urls import router

urlpatterns = [
    url(r'^api/token/', obtain_auth_token, name='api-token'),
    url(r'^api/', include(router.urls)),
    url(r'^$',TemplateView.as_view(template_name='board/index.html')),     ❷
]
```

❶ 我们使用了内建的通用模板 `TemplateView`。

❷ 在路由中加入了一个新的模式来将服务器根节点渲染成 *board/index.html* 模板。

当在浏览器中查看时，会见到如图 5-2 所示的输出。

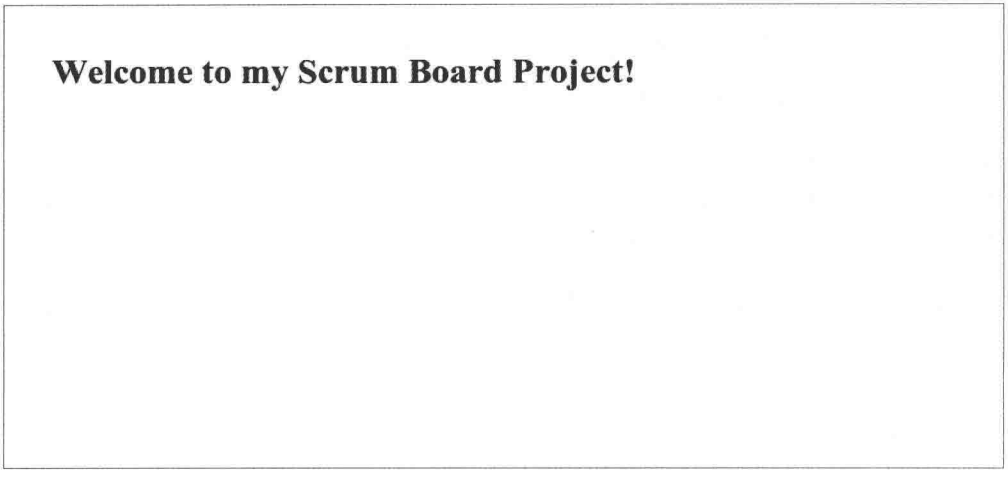


图 5-2：欢迎主页

恭喜！现在已经为我们的 Scrum 板应用创建好了最初的设置。

构建用户认证

在第 4 章中，我们启用了在 `django-rest-framework` 中内建的基于记号的验证。这一过程期望用户记号在每个请求的 `Authorization` 头部进行传递。回忆一下，这个应用不使用 Django 默认的基于会话 cookie 的验证。正如我们在构建 Python API 客户端时所见，客户端必须要维护这个状态并在每个 API 请求中传递验证信息。下一步我们要创建一个地方让客户端来维持登录状态。

创建会话模型

正如本章前面所提到的，Backbone 模型是应用数据定义的地方。我们通常从为数据设计组织结构开始构建应用。

会话是一种让我们存储 API 记号数据的方式，用来在应用内创建用户验证。我们从为用户会话创建一个基本模型开始。将下面这段代码加入到 *board/static/board/js* 下的 *models.js* 中：

```
(function ($, Backbone, _, app) {

    varSession = Backbone.Model.extend({
        defaults: {                                          ❶
            token: null
        },
        initialize: function (options) {
            this.options= options;
            this.load();
        },
        load: function () {
            var token = localStorage.apiToken;               ❷
            if (token) {
                this.set('token', token);
            }
        },
        save: function (token) {
            this.set('token', token);
            if (token === null) {                            ❸
                localStorage.removeItem('apiToken');
            } else {
                localStorage.apiToken= token;
            }
        },
        delete: function () {
            this.save(null);
        },
        authenticated: function () {                         ❹
            return this.get('token') !== null;
        }
    });

    app.session= new Session();                              ❺

})(jQuery, Backbone, _, app);
```

这段代码中包含了许多我们处理 API 记号时所需的内容：

❶ 这里将记号变量设置为默认的空值。希望确保变量已经可用，并设置它的初始值。

❷ 这里使用了基于 localStorage 中捕获的数值记号的初始设置。

❸ 这里的代码检查是否为真实的记号数值。如果不是，移除该数值并对用户取消授权。

❹ 方法查看当前模型实例下记号是否存在。

❺ 代码创建一个供应用使用的会话模型。

在这段代码最后，我们使用一个立即调用的函数表达式，只在有请求时执行，并将它设置在全局范围之外。代码中还少了一样东西，让我们实际处理发送过来的 HTTP 请求。让我们在 *board/static/board/js/model.js* 中添加一种方式，为 `XMLHttpRequest`（`xhr`）设置请求的头信息：

```
(function ($, Backbone, _, app) {

    var Session = Backbone.Model.extend({
        defaults: {
            token: null
        },
        initialize: function (options) {
            this.options= options;
            $.ajaxPrefilter($.proxy(this._setupAuth, this));            ❶
            this.load();
        },
        load: function () {
            var token = localStorage.apiToken;
            if (token) {
                this.set('token', token);
            }
        },
        save: function (token) {
            this.set('token', token);
            if (token === null) {
                localStorage.removeItem('apiToken');
            } else {
                localStorage.apiToken= token;
            }
        },
        delete: function () {
            this.save(null);
        },
        authenticated: function () {
          return this.get('token') !== null;
        },
        _setupAuth: function (settings, originalOptions, xhr) {        ❷
            if (this.authenticated()) {
                xhr.setRequestHeader(                                   ❸
                    'Authorization',
                    'Token ' + this.get('token')
                );
            }
        }
    });
```

```
    app.session= new Session();

})(jQuery, Backbone, _, app);
```

❷❸ 以 JavaScript 对象的形式返回 XMLHttpRequest，使用回调函数处理状态更新。因此我们可以连同头信息和参数一起发送 HTTP 请求给 Web 服务器。在我们的会话模型中，会注意到我们定义了一个 `_setupAuth` 方法。该方法会检查验证，并在通过验证后将记号传入我们 XMLHttpRequest 的头信息参数中。

❶ 我们希望在实例化会话模型之前，检查用户是否已经被验证过了。这里我们会使用 `$.ajaxPrefilter` 基于 `_setupAuth` 方法的结果来代理记号。

除了验证记号之外，还要在所有 POST/PUT/DELETE 请求中发送 CSRF 记号。好在 Django 已经提供了一段代码让我们从 cookie 中获取记号并将它随合适的头信息发送，如 *board/static/board/js/models.js* 所示：

```
(function ($, Backbone, _, app) {

    // CSRF helper functions taken directly from Django docs
    function csrfSafeMethod(method) {                                    ❶
        // these HTTP methods do not require CSRF protection
        return (/^(GET|HEAD|OPTIONS|TRACE)$/i.test(method));
}

function getCookie(name) {                                               ❷
    var cookieValue = null;
    if (document.cookie &&document.cookie != '') {
        var cookies = document.cookie.split(';');
        for (vari= 0; i<cookies.length; i++) {
            var cookie = $.trim(cookies[i]);
            // Does this cookie string begin with the name we want?
            if (cookie.substring(0, name.length+ 1) == (name + '=')) {
                cookieValue= decodeURIComponent(
                    cookie.substring(name.length+ 1));
                break;
            }
        }
    }
    return cookieValue;
}

// Setup jQuery ajax calls to handle CSRF
$.ajaxPrefilter(function (settings, originalOptions, xhr){           ❸, ❹
    var csrftoken;
    if (!csrfSafeMethod(settings.type) && !this.crossDomain) {
        // Send the token to same-origin, relative URLs only.
        // Send the token only if the method warrants CSRF protection
        // Using the CSRFToken value acquired earlier
        csrftoken= getCookie('csrftoken');
        xhr.setRequestHeader('X-CSRFToken', csrftoken);
```

```
        }
    });

    varSession = Backbone.Model.extend({
    ...
})(jQuery, Backbone, _, app);                                    ❺
```

❶❷❸这 3 个帮助方法直接摘自：*https://docs.djangoproject.com/en/1.8/ref/ contrib/csrf/#ajax*

❹ 由于方法是自调用的，这些 `$.ajaxPrefilter` 调用会在 *models.js* 载入后执行。

❺ 这些内容用于处理验证。如果当前会话模型存有 API 记号，记号会被加入到 `Authorization` 请求头信息中，如 `django-rest-framework` 预想的一样。

这里假设了项目使用的是默认的 `csrftoken` 作为 cookie 名称。如果需要，可以通过 *app.js* 解析的配置来配置记号。

这会劫持所有 AJAX 请求，并添加必要的验证和 CSRF 记号。尽管这很有用，但仍然没有一种方式能让代码从服务器重新得到记号。既然我们已经有了一个验证会话信息的基础结构，我们需要创建一个登录视图来提供一种验证用户的方式。

创建登录视图

创建表单是 Web 应用开发进程中最重要也是最复杂的一部分。这是用户为 Web 创建内容的方式，因此需要经过仔细考虑。

我们的登录视图将包含几个不同的状态：初始、错误以及成功状态。在开发视图的过程中，我们需要以这样的方式来架构，避免跑偏太远或是省略这些状态。让我们从在 *board/static/board/js/views.js* 中设置一个基本的登录视图开始：

```
...
var LoginView= Backbone.View.extend({
    id: 'login',
    templateName: '#login-template',
    initialize: function () {
        this.template= _.template($(this.templateName).html());
    },
    render: function () {
        var context = this.getContext(),
            html = this.template(context);
        this.$el.html(html);
    },
```

```
    getContext: function () {
        return {};
    }
});
...
```

应当把这个视图定义在 *views.js* 中定义的自调用函数中。从目前开始，假定在没有其他说明的情况下，所有的 JavaScript 文件中的代码片段都是写在自调用函数中的。

如你所见，这个视图非常类似我们之前创建的主页视图。因为我们知道将以同样的模式创建更多的视图来完成我们的应用，让我们来借助 Backbone 的扩展性，在 *board/static/board/js/views.js* 中创建一个通用的视图模板，以此来为所有视图建模。

```
...
var TemplateView = Backbone.View.extend({
    templateName: '',
    initialize: function () {
        this.template = _.template($(this.templateName).html());
    },
    render: function () {
        var context = this.getContext(),
            html = this.template(context);
        this.$el.html(html);
    },
    getContext: function () {
        return {};
    }
});
...
```

我们可以使用这个模板来创建每个视图，并将 DRY（don't repeat yourself，不要重复自己）方法应用到我们的代码上。我们的视图（*board/static/board/js/views.js*）应当看上去像这样：

```
...
var TemplateView = Backbone.View.extend({
    templateName: '',
    initialize: function () {
        this.template = _.template($(this.templateName).html());
    },
    render: function () {
        var context = this.getContext(),
            html = this.template(context);
        this.$el.html(html);
    },
    getContext: function () {
        return {};
    }
});

var HomepageView = TemplateView.extend({                    ❶
    templateName: '#home-template'
});
```

```
var LoginView = TemplateView.extend({                    ❷
    id: 'login',
    templateName: '#login-template'
});

app.views.HomepageView = HomepageView;                   ❸
app.views.LoginView = LoginView;
...
```

❶❷ 现在 HomepageView 和 LoginView 从基础的 TemplateView 中扩展，清除了重复的代码。

❸ 将 HomepageView 和 LoginView 加入到 app.views 字典中。TemplateView 是一个具体实现，不需要用于 *views.js* 之外，所以将其隐藏。

通过创建这个通用模板，将很轻易地看到它在开发 Backbone 应用时会多么有用。然而，LoginView 需要做的还很多，而非仅仅是渲染模板。它要渲染登录表单，处理提交并检查是否成功。

Backbone 社区有许多扩展用于像我们正在用 TemplateView 做的这样，致力于减少一些同样的样板代码。这些扩展中包括 MarionetteJS（即 Backbone.Marionette）、Thorax、Chaplin 和 Vertebrae。如果不想自己写这些基础类，也可以考虑使用这些扩展中的一个。

让我们从将 login-template 添加到 *index.html* 文件（位于 *board/templates/board*）开始吧。

```
...
<script type="text/html" id="login-template">
  <form action="" method="post">
      <label for="id_username">Username</label>
      <input id="id_username" type="text" name="username"
          maxlength="30" required />
      <label for="id_password">Password</label>
      <input id="id_password" type="password"
          name="password" required />
      <button type="submit">Login</button>
  </form>
</script>
</head>
...
```

这个表单包含两个输入：username 和 password，对应匹配 obtain_auth_token 视图所需的内容。定义好模板后，视图需要处理表单提交。Backbone 视图可以为其控制的元素绑定事件，这要用到 events 映射接口，如这里 *board/static/board/js/views.js* 中所示。

```
...
var LoginView = TemplateView.extend({
    id: 'login',
    templateName: '#login-template',                    ❶
    events: {
        'submit form': 'submit'
    },
    submit: function (event) {                                    ❷
        var data = {};
        event.preventDefault();
        this.form= $(event.currentTarget);
        data = {
            username: $(':input[name="username"]', this.form).val(),
            password: $(':input[name="password"]', this.form).val()
         };
        // TODO: Submit the login form
    }
});
...
```

❶ 时间属性，监听关于 LoginView 的元素内任何表单元素的 submit 事件。当一个事件被触发，它就会执行 submit 回调。

❷ submit 回调阻止了表单的提交，因此视图可以来提交并处理结果。

可以在官方文档中查看更多有关 Backbone 视图事件绑定的内容。

在执行 submit 方法的过程中，我们从表单数值中获取用户名和密码。视图还不知道将这些信息提交到哪个 URL 上。从第 4 章中定义的项目 URL 中得知，这些信息应当在 */api/token/* 上。我们可以使用 *app.js* 文件解析的应用配置在客户端上配置这些信息，像这里 *board/templates/board/index.html* 所示。

```
...
<script id="config" type="text/json">
    {
        "models": {},
        "collections": {},
        "views": {},
        "router": null,
        "apiRoot": "{% url 'api-root' %}",                    ❶
        "apiLogin": "{% url 'api-token' %}"
    }
</script>
...
```

❶ 这里添加了两个配置：一个是 API 的根节点，另一个是 API 的登录 URL。

在使用 url 模板标签时，可以通过 Django 后台定义的 URL 模式来控制客户端。这有助于避免今后在改变 api-root 或 api-token 时所带来的问题。

有了这个信息，客户端现在可以提交到连接了我们的 API 的正确的登录视图，像这里 *board/static/board/js/views.js* 中所示。

```
...
Var LoginView= TemplateView.extend({
    id: 'login',
    templateName: '#login-template',
    events: {
        'submit form': 'submit'
    },
    submit: function (event) {
        var data = {};
        event.preventDefault();
        this.form= $(event.currentTarget);
        data = {
            username: $(':input[name="username"]', this.form).val(),
            password: $(':input[name="password"]', this.form).val()
     };
     $.post(app.apiLogin, data)                                          ❶
         .done($.proxy(this.loginSuccess, this))
         .fail($.proxy(this.loginFailure, this));
    },
    loginSuccess: function (data) {                                      ❷
        app.session.save(data.token);
        this.trigger('login', data.token);
    },
    loginFailure: function (xhr, status, error) {                        ❸
        var errors = xhr.responseJSON;
        this.showErrors(errors);
    },
    showErrors: function (errors) {
        // TODO: Show the errors from the response
    }
 });
...
```

❶ 通过使用新的 app.apiLogin 配置，用户名和密码信息将提交到后台。

❷ 当登录成功时，会使用之前定义的 app.session 保存记号，并触发 login 事件。

❸ 当失败时，错误信息会显示给用户。

你会注意到，我们现在也将错误信息加入到了 LoginView 中。错误信息会以 JSON 的形式返回，并将错误列表的字段名作为匹配键，返回不属于某个特定字段的错误。

```
{"username": ["This field is required."], "password": ["This field is required."]}

{"non_field_errors": ["Unable to login with provided credentials."]}
```

现在，我们需要添加一种方式在表单某处显示错误信息。让我们在 *board/static/board/js/views.js* 中更新自定义的 `showErrors` 方法，映射到表单上方的字段、非字段错误信息。

```
...
var LoginView = TemplateView.extend({
    id: 'login',
    templateName: '#login-template',
    errorTemplate:_.template('<span class="error"><%- msg %></span>'),      ❶
    events: {
        'submit form': 'submit'
    },
    submit: function (event) {
        var data = {};
        event.preventDefault();
        this.form= $(event.currentTarget);
        this.clearErrors();                                                  ❷
        data = {
            username: $(':input[name="username"]', this.form).val(),
            password: $(':input[name="password"]', this.form).val()
        };
        $.post(app.apiLogin, data)
            .done($.proxy(this.loginSuccess,this))
            .fail($.proxy(this.loginFailure, this));
    },
    loginSuccess: function (data) {
        app.session.save(data.token);
        this.trigger('login', data.token);
    },
    loginFailure: function (xhr, status, error) {
        var errors = xhr.responseJSON;
        this.showErrors(errors);
    },
    showErrors: function (errors) {
        _.map(errors, function (fieldErrors, name) {                         ❸
            var field = $(':input[name=' + name + ']', this.form),

                label = $('label[for=' + field.attr('id') + ']', this.form);
            if (label.length=== 0) {
                label = $('label', this.form).first();
            }
            function appendError(msg) {
                label.before(this.errorTemplate({msg: msg}));                ❹
            }
            _.map(fieldErrors, appendError, this);                           ❺
        }, this);
    },
    clearErrors: function () {                                               ❻
        $('.error', this.form).remove();
    }
});
...
```

❸❺ 这里循环所有返回的字段和错误信息，并把每个错误添加到字段标签之前的DOM中。如果没有发现匹配的字段，则把错误加到第一个标签之前。

❶❹ 每个错误信息都从一个新的错误模板中创建，用到了我们在脚本一开始创建的内联*Underscore.js*模板。

❷❻ 这个方法在每次提交时将所有存在的错误信息从表单中移除。

现在，我们的`LoginView`已经包含所有处理登录表单的渲染和提交所必需的逻辑了。我们需要将这个逻辑连接到路由上正确显示。在此之前，让我们看看能否将这个表单处理做得更加通用，这样之后就可以在其他的视图中使用它了。

通用表单视图

既然知道在应用中很可能会包含多个表单，那就让我们使用这一模式，基于通用的`FormView`创建一个更加洁净的向上扩展的代码。我们会需要一种方式在表单中包含提交以及错误处理功能，因此，让我们从将这些模块添加到*board/static/board/js/views.js*中开始我们的工作。

```
...
var TemplateView= Backbone.View.extend({
    templateName: '',
   initialize: function () {
        this.template= _.template($(this.templateName).html());
    },
    render: function () {
        var context = this.getContext(),
            html = this.template(context);
        this.$el.html(html);
    },

    getContext: function () {
        return {};
    }
});

var FormView = TemplateView.extend({
    events: {                                                          ❶
        'submit form': 'submit'
    },
    errorTemplate: _.template('<span class="error"><%- msg %></span>'),  ❷
    clearErrors: function () {                                         ❸
        $('.error', this.form).remove();
    },
    showErrors: function (errors) {                                    ❹
        _.map(errors, function (fieldErrors, name) {
            Var field = $(':input[name=' + name + ']', this.form),
```

```
                label = $('label[for=' + field.attr('id') + ']', this.form);
            if (label.length=== 0) {
                label = $('label', this.form).first();
            }
            function appendError(msg) {
                label.before(this.errorTemplate({msg: msg}));
            }
            _.map(fieldErrors, appendError, this);
        }, this);
    },
    serializeForm: function (form) {                                    ❺
        return _.object(_.map(form.serializeArray(), function (item) {
        // Convert object to tuple of (name, value)
        return [item.name, item.value];
        }));
    },
    submit: function (event) {                                          ❻
        event.preventDefault();
        this.form= $(event.currentTarget);
        this.clearErrors();
    },
    failure: function (xhr, status, error) {                            ❼
        var errors = xhr.responseJSON;
        this.showErrors(errors);
    },
    done: function (event) {                                            ❽
        if (event) {
            event.preventDefault();
        }
        this.trigger('done');
        this.remove();
    }
});
...
```

❶❷❸❹❼ events、errorTemplate、clearErrors、showErrors 和 failure 直接来自于我们原先实现的 LoginView。

❺ serializeForm 是一个更加通用的表单字段数值序列化方法，而非 LoginView 中含有的明确的版本。

❻ submit 在设置表单提交开始时，会防止默认提交事件和清除错误信息。扩展这一部分的视图需要进一步定义这个回调。

❽ done 也是新引入的，是一个新版本更通用的触发 login 事件。它也处理将表单从 DOM 中移除的工作，在绝大多数情况下都会用到。

像我们之前创建的 TemplateView，不需要把这个视图添加到 app.views 的映射中。我们来看看在 *board/static/board/js/views.js* 里这个新的 FormView 怎样被应用到原先的 LoginView 中。

```
...
var LoginView= FormView.extend({                                    ❶
    id: 'login',
    templateName: '#login-template',
    submit: function (event) {
        var data = {};
        FormView.prototype.submit.apply(this, arguments);           ❷
        data = this.serializeForm(this.form);                       ❸
        $.post(app.apiLogin, data)                                  ❹
            .success($.proxy(this.loginSuccess, this))
            .fail($.proxy(this.failure, this));
    },
    loginSuccess: function (data) {                                 ❺
        app.session.save(data.token);
        this.done();
    }
});
...
```

❶ LoginView 现在从 FormView 而非 TemplateView 中继承。

❷ `submit` 回调会调用原始的 `FormView` 提交以便阻止提交，并清除错误信息。

❸ 使用 `serializeForm` 助手序列化表单数据，取代之前手动获取用户名和密码字段的做法。

❹❺ 当保存完记号后，`loginSuccess` 回调开始使用 `done` 助手。这里使用默认的 `failure` 回调，不用重新定义。

JavaScript 不含有像 Python 那样的 `super` 调用。`FormView.prototype.submit.apply` 是一个有效的调用父类方法的等价方式。

如你所见，这项工作帮助我们简化了 `LoginView` 的实现。与此同时，它创建了供今后使用的基础 `FormView`。现在让我们转移到路由，看看如何将我们的视图和它们连接。

认证路由

正如本章之前定义的，Backbone 路由是一种通过 History API 为我们的应用创建 URL 的方式。除了 API 根节点和登录部分，API 需要一些验证来让用户与我们的应用交互。在没有验证记号的情况下，客户端不能做任何事。我们需要创建一种方式在记号不可用时显示登录表单，并可以获取记号。在服务器端，我们通过将用户重定向到登录 URL 进

行处理，并在登录完成后重定向回来。在客户端，`LoginView` 不需要自己的路由。我们可以通过使用 *router.js* 文件（位于 *board/static/board/js*）来劫持应用路由处理这个问题。

```
...
var AppRouter = Backbone.Router.extend({
    routes: {
        '': 'home'
    },
    initialize: function (options) {
        this.contentElement = '#content';
        this.current = null;
        Backbone.history.start();
    },
    home: function () {
        Var view = new app.views.HomepageView({el: this.contentElement});
        this.render(view);
    },
    route: function (route, name, callback) {                                    ❶
        // Override default route to enforce login on every page
        var login;
        callback = callback || this[name];
        callback = _.wrap(callback, function (original) {                        ❷
            var args = _.without(arguments, original);
            if (app.session.authenticated()) {
                original.apply(this, args);                                      ❸
            } else {
                // Show the login screen before calling the view
                $(this.contentElement).hide();
                // Bind original callback once the login is successful           ❹
                login = new app.views.LoginView();
                $(this.contentElement).after(login.el);
                login.on('done', function () {
                    $(this.contentElement).show();
                    original.apply(this, args);
                }, this);
                // Render the login form
                login.render();
            }
        });
      return Backbone.Router.prototype.route.apply(this, [route, name, callback]); ❺
    },
    render: function (view) {
        if (this.current) {
            this.current.undelegateEvents();
            this.current.$el= $();
            this.current.remove();
        }
        this.current= view;
        this.current.render();
    }
});
...
```

❶ 这里为路由重写了默认的 `route` 方法。它以哈希路由的形式传递回调函数的名字或直接回调函数。

❷ 原始的回调函数会被封装起来，用于在调用之前对验证状态作初始检查。

❸ 如果用户通过验证，将会简单地调用原先的回调。

❹ 如果用户没有通过验证，当前页面内容会被隐藏，并渲染一个登录视图作为替代。当登录视图触发了 `done` 事件，原先的回调就可以继续进行了。

❺ 现在使用新的封装后的回调来调用原先的 `route`。

既然用户可以登录，可以注销 (log out) 对他们也很有帮助。我们可以通过一个小视图来为页面渲染一个标头，创建一种方便的形式让用户在应用的地方都可以注销 (log out)。

创建标头视图

与 Django 视图不同，Backbone 视图不用控制整个渲染的页面，页面可以用多个视图来处理。`HomepageView` 会渲染主体内容，而新的 `HeaderView` 可以用来渲染显示在所有视图顶端的页面标头。让我们把这个模板添加到我们的 *index.html* 文件中（位于 *board/templates/board*）。

```
...
<script type="text/html" id="header-template">
    <span class="title">Scrum Board Example</span>
    <% if (authenticated ) { %>
        <nav>
            <a href="/" class="button">Your Sprints</a>
            <a href="#" class="logout button">Logout</a>
        </nav>
    <% } %>
</script>
</head>
...
```

和其他模板一起使用时，它会被加到 `<head>` 元素里。这个模板是第一个使用了 Underscore 的模板逻辑的模板。在 `<% .. %>`（有时被称作是蜜蜂蛰的记号）之内，Underscore 可以执行 JavaScript，在本例中是一个 if 语句。`authenticated` 的值会由用于渲染模板的数据来决定。

我们将在 *board/static/board/js/views.js* 中创建一个与这个模板关联的视图，并引入验证需求以便添加我们的注销交互。

```
...
var HeaderView = TemplateView.extend({                    ❶
```

```
    tagName: 'header',                                              ❷
    templateName: '#header-template',
    events: {
        'click a.logout': 'logout'                                  ❸
    },
    getContext: function () {                                       ❹
       return {authenticated: app.session.authenticated()};
    },
    logout: function (event) {                                      ❺
        event.preventDefault();
        app.session.delete();
        window.location= '/';
    }
});

app.views.HomepageView= HomepageView;
app.views.LoginView= LoginView;
app.views.HeaderView= HeaderView;                                   ❻
```

❶ HeaderView 从 TemplateView 中继承，与其他视图保存相似的逻辑。

❷ 不同于之前的视图，tagName 已经被定义好了。这意味着模板会被渲染到 <header> 元素中。

❸❺ 头部的导航栏会在用户通过验证后显示两个链接。一个是回到主页的链接，另一个是注销 (logout) 的链接。注销 (logout) 的逻辑是通过视图中 logout 回调来处理的。

❹ authenticated 的值将基于当前会话的状态传递给模板上下文。当状态改变时它不会自动更新。视图需要被再次渲染。

❻ 最后，把这个视图加入到 app.views 映射中，这样路由就可以使用它了。

和 LoginView 一样，HeaderView 并不关联某个特定路由，它将始终出现在那里。只有当登录状态改变时才需要改变它。这可以全部通过路由自身（例如在 *board/static/board/js/router.js* 中）来处理。

```
...
var AppRouter= Backbone.Router.extend({
    routes: {
        '': 'home'
    },
    initialize: function (options) {
        this.contentElement= '#content';
        this.current= null;
        this.header= new app.views.HeaderView();                    ❶
        $('body').prepend(this.header.el);
        this.header.render();
        Backbone.history.start();
    },
```

```
        home: function () {
            Var view = new app.views.HomepageView({el: this.contentElement});
            this.render(view);
        },
        route: function (route, name, callback) {
            // Override default route to enforce login on every page
            Var login;
            callback = callback || this[name];
            callback = _.wrap(callback, function (original) {
                var args= _.without(arguments, original);
                if (app.session.authenticated()) {
                    original.apply(this, args);
                } else {
                    // Show the login screen before calling the view
                    $(this.contentElement).hide();
                    // Bind original callback once the login is successful
                    login = new app.views.LoginView();
                    $(this.contentElement).after(login.el);
                    login.on('done', function () {
                        this.header.render();                          ❷
                        $(this.contentElement).show();
                        original.apply(this, args);
                    }, this);
                    // Render the login form
                    login.render();
                }
            });
            return Backbone.Router.prototype.route.apply(
                this, [route, name, callback]);
        },
        render: function (view) {
            if (this.current) {
                this.current.undelegateEvents();
                this.current.$el= $();
                this.current.remove();
            }
            this.current= view;
            this.current.render();
        }
    });
    ...
```

❶ 当路由创建时初始化标头视图，并把它的元素加入到 <body> 的起始处。

❷ 当登录完成后，标头会再次被渲染反映出新的状态。

通过将劫持的所有路由分隔，加强主体内容的逻辑和标头，帮助我们保证路由不再需要在视图或是模板中单独处理登录状态。

为什么没有登录路由

你也许会问“为什么不直接创建一个登录路由并重定向用户呢？”这是在服务器端MVC应用下的典型做法，但因为服务器是无状态的，在客户端并不需要。创建多客户端应用和服务器端应用相比，前者要更加接近桌面或本地移动应用。在进行诸如登录、注销或是删除等操作时，不需要使用客户端路由。路由是客户端管理状态的另一种方式。它们使客户端应用可以被链接，也可以被作为书签收藏。你会发现，在客户端上为操作添加路由或是做类似“重定向”的操作会破坏浏览器后退按钮的使用，或是在用户后退之前视图却没有得到期望的内容时，制造不必要的麻烦。

使用之前的方法，这些问题都可以被避免。在用户未被验证之前，登录视图永远不会被调用，同时`HomepageView`和我们之后将要创建的视图也只会在用户通过验证后才被调用。这样就把所有验证需求全部封装到了一起，不会在用户浏览器历史上创建不必要的`#login`条目。

在继续添加更多API交互之前，我们先给网站标头和登录表单添加一些基础样式。为了有个洁净的开始，我们先引入一个重置的样式表。这个样式表可以在*http://necolas.github.io/normalize.css/latest/normalize.css*上下载，并保存到*vendor*静态文件夹下。我们还需要在*board/static/board/css/board.css*里添加我们的网站CSS。

```
@import url(http://fonts.googleapis.com/css?family=Lato:300,400,700,900);

/* Generic
 ================== */
body {
    -moz-box-sizing: border-box;
         box-sizing: border-box;
    font-family: 'Lato', Helvetica, sans-serif;
}

a { text-decoration: none; }

button, a.button {
    font-size: 13px;
    padding: 5px 8px;
    border: 2px solid #222;
    background-color: #FFF;
    color: #222;
   transition: background 200ms ease-out;
}

button:hover, a.button:hover {
    background-color: #222;
    color: #FFF;
}
```

```
input, textarea {
    background: white;
    font-family: inherit;
    border: 1px solid #cccccc;
    box-shadow: inset 0 1px 2px rgba(0,0,0,0.1);
    display: block;
    font-size: 18px;
    margin: 0 0 16px 0;
    padding: 8px
}

#content {
    padding-left: 25px;
    padding-right: 25px;
}
.hide {
    display: none;
}

.error {
    color: #cd0000;
    margin: 8px 0;
    display: block;
}

/* Header
==================== */

header {
    height: 45px;
    line-height: 45px;
    border-bottom: 1px solid #CCC;
}

header .title {
    font-weight: 900;
    padding-left: 25px;
}

header nav {
    display: inline-block;
    float: right;
    padding-right: 25px;
}

header nav a {
    margin-left: 8px;
}

header nav a:hover{
    background: #222;
    color: #FFF;
}
```

```
/* Login
==================== */

#login {
    width: 100%;
}

#login form {
    width: 300px;
    margin: 0 auto;
    padding: 25px;
}

#login form input {
    margin-bottom: 15px;
}

#login form label, #login form input {
    display: block;
}
```

最后，要把这些加入到 *board/templates/board/index.html* 中。

```
{% load staticfiles %}
<!DOCTYPE html>
<html class="no-js">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <title>Scrum Board</title>
        <meta name="description" content="">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="{% static 'board/vendor/normalize.css' %}">  ❶
        <link rel="stylesheet" href="{% static 'board/css/board.css' %}">          ❷
        <script type="text/html" id="home-template">
            <h1>Welcome to my Scrum Board Project!</h1>
        </script>
        <script type="text/html" id="login-template">
          <form action="" method="post">
              <label for="id_username">Username</label>
              <input id="id_username" type="text" name="username"
                  maxlength="30" required />
              <label for="id_password">Password</label>
              <input id="id_password" type="password"
                  name="password" required />
            <button type="submit">Login</button>
          </form>
        </script>
        <script type="text/html" id="header-template">
          <span class="title">Scrum Board Example</span>
          <% if (authenticated ) { %>
             <nav>
                 <a href="/">Sprints</a>
                 <a class="logout" href="#">Logout</a>
             </nav>
```

```
        <% } %>
      </script>
    </head>
    <body>

      <div id="content"></div>

      <script src="{% static 'board/vendor/jquery.js' %}"></script>
      <script src="{% static 'board/vendor/underscore.js' %}"></script>
      <script src="{% static 'board/vendor/backbone.js' %}"></script>
      <script id="config" type="text/json">
          {
              "models": {},
              "collections": {},
              "views": {},
              "router": null,
              "apiRoot": "{% url 'api-root' %}",
              "apiLogin": "{% url 'api-token' %}"
          }
      </script>
      <script src="{% static 'board/js/app.js' %}"></script>
      <script src="{% static 'board/js/models.js' %}"></script>
      <script src="{% static 'board/js/views.js' %}"></script>
      <script src="{% static 'board/js/router.js' %}"></script>
  </body>
</html>
```

❶ *normalize.css* 提供了一个设置应用样式的跨浏览器的基础。

❷ *board.css* 是我们添加项目样式的地方。

如果在未验证的情况下渲染页面，会得到如图 5-3 所示的结果。

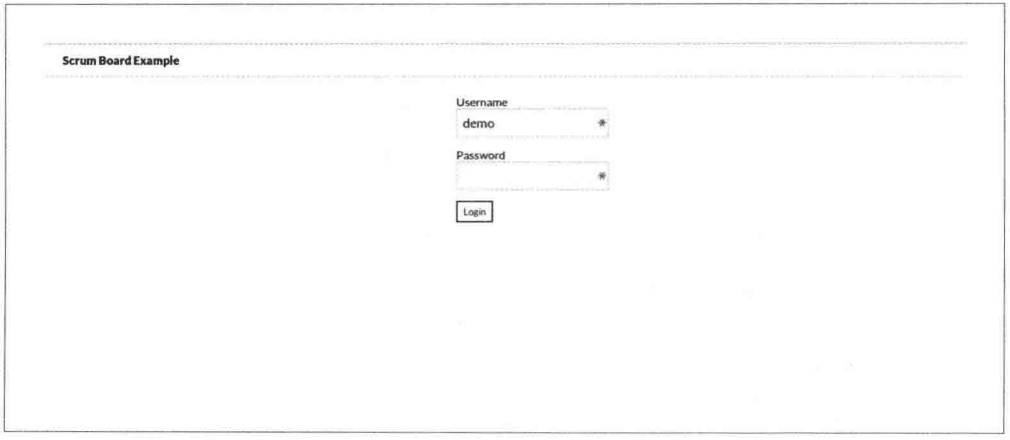


图 5-3：欢迎主页

当表单提交后，页面会渲染欢迎信息以及新的标头，如图 5-4 所示。

图 5-4：带有头部的欢迎主页

既然客户端可以使用 API 处理验证，主页可以由静态的欢迎页面改变成动态应用。回顾之前的 *index.html*，你会发现项目结构将客户端和服务器很好地分隔开来。我们除了 `static` 和 `url` 之外没有用到其他的 Django 模板，而它们也可以方便地用路径引用进行替换。还可以作为类似第 3 章中制作进程的一部分，以静态 HTML 的形式输出，并将 CSS 和 JavaScript 压缩精简。

记住这里的方法并不只针对 Backbone，许多关于定义配置和代码组织的相同的技巧也可以被用在 Angular 和 Ember 这样的框架上。下一章中，我们将继续创建 Backbone 应用，探索创作单页面 Web 应用的独特体验。

第 6 章

单页面 Web 应用

为 Web 创建丰富的体验是我们这些开发者所一直致力的。近些年来我们已经逐渐习惯使用多个语言和框架，这通常需要客户端和服务器架构之间有更好的通信。

随着近年来逐渐向重度客户端开发上转移，我们可能会更加在意用什么样的组织和结构来搭设项目的架构。在本章中，我们将创建一个简单的单页面 Web 应用，并逐一解释 Django 框架如何适应在这一开发上的新变化。

什么是单页面 Web 应用？

单页面 Web 应用（SPA）是这次客户端开发中的主要内容。有关什么是单页面 Web 应用和如何使用它们的问题，存在很多误区。有些开发者认为这一类应用只包含那些仅仅需要一个页面的应用，例如待办事项应用。这是一个合理的设想，但却不是“单页面”这个术语在这一类应用下的全部含义。

单页面应用中的“单页面”指的是服务器将所有需要的展示逻辑发送给客户端的单个页面。从这里开始，浏览器管理数据并基于“按需”的原则异步创建页面。在初始页面载入后，服务器通过 API 与客户端交互，只通过网络发送数据。减少了响应的大小，也增强了每个后续请求的可缓存性。因为浏览器不需要刷新整个页面，这一架构可以创建迅速响应用户交互的界面。

你会看到这个误解可能会导致某些 Django 开发者觉得 SPA 对他们项目的需求不具有延展性。在本章中，我们将会关注如何使用第 4 章中 REST API 提供的 JSON 数据结构，并将其与第 5 章的基本布局和架构结合起来。

发现 API

与 API 的交互由 Backbone 模型和集合来处理。这些概念被很好地反映到 Django 的模型和结果集上。

获取 API

目前我们已经使用 `django-rest-framework` 配置了 API 根路径，但还没有配置 API 里面的任何子资源。尽管我们可以使用同样的方法为 sprint、任务、用户端点等配置路径，以至于不能再使用 API 本身的可发现性。因此我们将改变路径，在 models.js 文件（位于 *board/static/board/js*）中的 API 根节点中获取可用资源的 API。

```
(function ($, Backbone, _, app) {
...
    app.models.Sprint = Backbone.Model.extend({});                          ❶
    app.models.Task = Backbone.Model.extend({});
    app.models.User = Backbone.Model.extend({});

    app.collections.ready = $.getJSON(app.apiRoot);                         ❷
    app.collections.ready.done(function (data) {                            ❸
          app.collections.Sprints = Backbone.Collection.extend({
              model: app.models.Sprint,
              url: data.sprints
          });
          app.sprints = new app.collections.Sprints();
          app.collections.Tasks = Backbone.Collection.extend({
              model: app.models.Task,
              url: data.tasks
          });
          app.tasks = new app.collections.Tasks();
          app.collections.Users = Backbone.Collection.extend({
              model: app.models.User,
              url: data.users
          });
          app.users = new app.collections.Users();
    });

})(jQuery, Backbone, _, app);
```

❶ 为我们需要的每个模型（`Sprint`、`Tasks` 和 `User`）创建存根代码。把每个都加入到 `app.models` 映射中以便在整个应用中使用。

❷ 取得 API 的根节点，AJAX 延时对象存储在 `app.collections.ready` 中。这就允许应用的其他部分等待直到集合准备就绪。

❸ 当资源返回时，结果 URL 被用于创建每个集合。集合的定义被加到 `app.collections` 映射中，并创建 `app.sprints`、`app.tasks`、`app.users` 的集合的共享实例。

目前为止，这些模型和集合还没有 Backbone 默认设置之外的自定义。我们需要做一些改变，通过使用 Backbone 创建自定义模型来让它更好地与 `django-rest-framework` 以及我们的 API 协作。

模型自定义

在 Backbone 中，由集合的 URL 创建模型的 URL，不包含结尾的斜杠。尽管这是 JavaScript 框架的基本模式，但 `django-rest-framework` 需要在 URL 结尾处有一个斜杠。这需要以某种方式把它合并到我们的 URL 结构中。

此外，API 会返回 links 值中模型的 URL。如果这个值已经给定，则要直接使用该值而非创建 URL。参考下面这段 *board/static/board/js/models.js* 中的代码片段：

```
...
var BaseModel = Backbone.Model.extend({
    url: function () {                                                    ❶
        var links = this.get('links'),
            url = links && links.self;
        if (!url) {
            url = Backbone.Model.prototype.url.call(this);                ❷
        }
        return url;
    }
});

    app.models.Sprint = BaseModel.extend({});                             ❸
    app.models.Task = BaseModel.extend({});
    app.models.User = BaseModel.extend({});
    ...
```

❶ 这段代码重载了默认的 URL 结构，开始时先从 `links` 属性中查找 `self` 的值。

❷ 如果 API 没有给出 URL，则使用原始的 Backbone 方法创建它。

❸ 修改模型来从 `BaseModel` 而非 `Backbone.Model` 中继承。

正如所提到的，默认 `Backbone.Model.url` 创建的 URL 没有结尾的斜杠。然而 `django-rest-framework` 默认包含结尾的斜杠。绝大多数时候我们使用 API 返回的 URL，但对于任何在客户端从基础 URL 和模型 id 来创建 URL 的地方，仍然需要消除这种不一致。我们在 board/urls.py 中的 `BaseModel.url` 函数里添加一个结尾的斜杠来处理这一问题。

```
from rest_framework.routers import DefaultRouter

from . import views


router = DefaultRouter(trailing_slash=False)
```

```
router.register(r'sprints', views.SprintViewSet)
router.register(r'tasks', views.TaskViewSet)
router.register(r'users', views.UserViewSet)
```

API 期望使用 `username` 而非 `id` 来引用用户。我们可以在 *board/static/board/js/models.js* 中使用 `idAttribute model` 选项来配置它。

```
...
app.models.User = BaseModel.extend({
    idAttributemodel: 'username'
});
...
```

有了 `TemplateView` 和 `FormView`，我们现在就可以用通用的方法来扩展和进一步自定义我们的应用。

集合自定义

默认设置下，Backbone 预期所有的模型列在一个 API 返回的数组中。API 实现的分页功能会将对象列表包裹起来，并包含有关页数和总页数的元数据。要在 Backbone 中使用它，我们需要改变每个集合的 `parse` 方法。和模型一样，我们可以通过 *board/static/board/js/models.js* 中的基础集合类来处理它。

```
...
var BaseCollection = Backbone.Collection.extend({
    parse: function (response) {                                    ❶
        this._next = response.next;
        this._previous = response.previous;
        this._count = response.count;
        return response.results || [];
    }
});

app.collections.ready = $.getJSON(app.apiRoot);
app.collections.ready.done(function (data) {
      app.collections.Sprints = BaseCollection.extend({             ❷
          model: app.models.Sprint,
          url: data.sprints
      });
      app.sprints = new app.collections.Sprints();
      app.collections.Tasks = BaseCollection.extend({               ❸
          model: app.models.Task,
          url: data.tasks
      });
      app.tasks = new app.collections.Tasks();
      app.collections.Users = BaseCollection.extend({               ❹
          model: app.models.User,
          url: data.users
      });
      app.users = new app.collections.Users();
```

```
});
...
```

❶ parse 重载了集合上的 next、previous 以及 count 元数据，并返回对象列表，附在 API 的 results 键上。

❷❸❹ 将所有的集合配置为从 BaseCollection 继承。

这是处理 API 分页的最简单的方法。关于更全面的做法，可以参考 backbone-paginator。

有了这些设置，我们可以开始在视图中使用这些模型了。

构建主页

主页在我们的应用中有两个作用：渲染当前的 sprint 和用于创建新的 sprint。一旦某个用户得到授权并重定向到该主页，能观看当前的 sprint，我们要先着手对其进行渲染。

显示当前的 sprint

在刚刚创建的 *views.js*（在 *board/static/board/js*）中，让我们来回顾一下当前的 HomepageView。

```
...
var HomepageView = TemplateView.extend({
    templateName: '#home-template'
});
...
```

如你所见，这里在渲染 home-template 时不用考虑额外的上下文。需要将当前 sprint 的集合传递给模板，并在页面上显示。

```
...
var HomepageView = TemplateView.extend({
    templateName: '#home-template',
    initialize: function (options) {                                    ❶
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        app.collections.ready.done(function () {
            var end = new Date();
            end.setDate(end.getDate() - 7);
            end = end.toISOString().replace(/T.*/g, '');
```

```
            app.sprints.fetch({
                data: {end_min: end},
                success: $.proxy(self.render, self)
            });
        });
    },
    getContext: function () {
        return {sprints: app.sprints || null};                    ❷
    }
});
...
```

❶ 在创建视图时，结束日期大于 7 天前的 sprint 会被获取。当 sprint 可用时，页面会再次渲染显示它们。

❷ 模板上下文现在包含来自于 `app.sprints` 当前的 sprint。在 `app.collections` 未就绪时，它可能未被定义。在这种情况下，模板会得到一个 `null` 值。

什么是 this？

对于那些对 Python 比 JavaScript 更加熟悉的人来说，在 `initialize` 函数中使用 `var self = this;` 可以保持当前 this 值（也就是视图实例）的引用，这样它就可以在后续的 `done` 回调函数中使用了。`this` 的值取决于如何调用函数或方法，要确保它的正确性可能会很困难，尤其是在嵌套回调函数中。另外将要在本书中会看到的一种模式是 `$.proxy`，可以用于为函数调用显式地设置 `this` 的上下文。Underscore 里有与之对应的助手叫做 `_.bind`。这两个都模仿了 ECMAScript 5 中引入的 `Function.prototype.bind`，并保证了跨浏览器的兼容性。

如果对理解函数上下文以及 `this` 的值有困难，建议阅读《*JavaScript: The Good Parts*》(O'Reilly)，或者访问 Mozilla Developer Network 文档，查看更加深入的介绍。

现在我们需要对主页 Underscore 模板（*board/templates/board/index.html*）进行更新，既要处理可用 sprint 的情况，也要处理加载 sprint 的情况。

```
...
<script type="text/html" id="home-template">
    <h2>Your Sprints</h2>
    <button class="add" type="submit">Add Sprint</button>
    <% if (sprints !== null) { %>
        <div class="sprints">
            <% _.each(sprints.models, function (sprint) { %>          ❶
                <a href="#sprint/<%- sprint.get('id') %>" class="sprint">   ❷
                    <%- sprint.get('name') %> <br>
                    <span>DUE BY <%- sprint.get('end') %></span>
                </a>
            <% }); %>
        </div>
```

```
    <% } else { %>
        <h3 class="loading">Loading...</h3>                    ❸
    <% } %>
</script>
...
```

❶ 当 sprint 不全为 null 时，循环并输出名称和结束日期。输出链接来显示 sprint 的详情，但在应用中还没有加入路由。这项内容在下一节中处理。

❸ 当 sprint 为 null 时，显示“加载中……”消息。

❷ 链接最终会为用户对 sprint 的详情进行导航。应用路由目前不能处理这个表单中的链接，要在本章之后的内容中进行更新。

我们还要在 *board/static/board/css/board.css* 中加入一些基础的 CSS 样式。

```
...
/* Sprint Listing
==================== */

.sprints {
    margin-top: 25px;
}

.sprints a.sprint {
    padding: 25px;
    float: left;
    text-align: center;
    background: #000;
    margin: 8px 8px 0 0;
    color: #FFF;
}

.sprints a.sprint span {
    font-size: 10px;
}
```

这是对原先的主页模板的全部重写，并加上了一些需要应用显示的动态数据。如你所见，我们还加入了一个添加新 sprint 的按钮。它目前还不能使用，我们在下一节中实现它的功能。

创建新 sprint

为用户创建一种生成内容的方式是所有应用开发的一部分工作。在主页里我们用按钮来提供这种方式，它会显示一个表单供用户添加新的 sprint。当表单完成后，它将会移除。

创建新 sprint 的逻辑将会由子视图来处理并渲染表单。与 `LoginView` 类似，这里也可以使用 `FormView`，这里的 *board/static/board/js/views.js* 如下所示。

```
...
var FormView = TemplateView.extend({
    ...
    modelFailure: function (model, xhr, options) {                    ❶
        var errors = xhr.responseJSON;
        this.showErrors(errors);
    }
});

var NewSprintView = FormView.extend({                                 ❷
    templateName: '#new-sprint-template',
    className: 'new-sprint',
    events: _.extend({                                                ❸
        'click button.cancel': 'done',
    }, FormView.prototype.events),
    submit: function (event) {
        var self = this,
        attributes = {};
        FormView.prototype.submit.apply(this, arguments);
        attributes = this.serializeForm(this.form);
        app.collections.ready.done(function () {
            app.sprints.create(attributes,{                           ❹
                wait: true,
                success: $.proxy(self.success, self),
                error: $.proxy(self.modelFailure, self)               ❺
            });
        });
    },
    success: function (model) {                                       ❻
        this.done();
        window.location.hash = '#sprint/' + model.get('id');
    }
});

var HomepageView = TemplateView.extend({                              ❼
...
```

❷❼ 视图继承了 FormView 帮助类，需要在 HomepageView 之前定义它。

❸ 单击添加按钮会触发表单的 submit 事件，这由 FormView 基类来处理。除了默认的 submit 事件处理器，视图还会处理取消按钮事件来调用 FormView 定义的 done 方法。

❹ 和 LoginView 一样，表单中的值被序列化了。视图使用 app.sprints.create 而不用手动调用 $.post。视图中对操作成功和操作失败的处理器进行了绑定。

❻ 当 sprint 被创建时，视图调用 done 并重定向到显示 sprint 详情的路由，这部分还未编写。

❶❺ 操作失败的回调会进入一个新加入 FormView 的 modelFailure。这是必要的，因为 $.ajax 失败回调的第一个参数是响应对象，而 Model.save 的第一个参数是 model 实例，第二个参数是响应。

现在我们可以在 *board/templates/board/index.html* 中为视图创建一个模板来包含这个表单了。

```
...
<script type="text/html" id="new-sprint-template">
    <form action="" method="post">
        <label for="id_name">Sprint Name</label>
        <input id="id_name" type="text" name="name" maxlength="100" required />
        <label for="id_end">End Date</label>
        <input id="id_end" type="date" name="end" />
        <label for="id_description">Description</label>
        <textarea id="id_description" name="description" cols="50"></textarea>
        <button class="cancel">Cancel</button>
        <button type="submit">Create</button>
    </form>
</script>
</head>
...
```

我们已经创建了视图和模板，但还没有对它们进行渲染。渲染操作是由 HomepageView（位于 *board/static/board/js/views.js*）处理的，它会添加事件绑定自定义方法。

```
...
var HomepageView = TemplateView.extend({
    templateName: '#home-template',
    events: {
        'click button.add': 'renderAddForm'                                    ❶
    },
    initialize: function (options) {
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        app.collections.ready.done(function () {
            var end = new Date();
            end.setDate(end.getDate() - 7);
            end = end.toISOString().replace(/T.*/g, '');
            app.sprints.fetch({
                data: {end_min: end},
                success: $.proxy(self.render, self)
            });
        });
    },
    getContext: function () {
        return {sprints: app.sprints || null};
    },
    renderAddForm: function (event) {                                          ❷
        var view = new NewSprintView(),
            link = $(event.currentTarget);
        event.preventDefault();
        link.before(view.el);
        link.hide();
        view.render();
        view.on('done', function () {
            link.show();
        });
```

```
    }
});
...
```

❶❷ 添加按钮的 `click` 事件现在由 `renderAddForm` 来处理。它创建一个 `NewSprintView` 实例，并在按钮上方进行渲染。当视图结束时，不管是通过添加按钮还是取消按钮，链接都会再次显示。

现在我们需要再稍加一些 CSS 来从列表中清晰地显示表单，如下面 *board/static/board/css/board.css* 中的代码所示。

```
...
.new-sprint {
    border: 1px solid #CCC;
    padding: 20px;
}
```

当新的 sprint 被创建后，它会将用户定向到 `#sprint/<id>`，不过还没有对那个路由进行处理。现在该用详情页面来显示每个 sprint 相关的数据了。

sprint 详情页面

从主页中我们已经实现了使用基础交互、路由、视图以及子视图来创作 Backbone 应用。sprint 的详情页面会在此基础上更进一步。让我们来花点时间回顾一下，我们计划要在这个页面上制作哪些东西。

当浏览一个 sprint 时，我们希望显示它的细节：名称、结束日期还有描述。还要带有将显示分配到该 sprint 任务的状态列。还要用列表显示未分配的“积压”任务以及创建新任务的地方。每个任务要显示一个简单的概要，当用户单击任务时会显示更加详细的信息。

完成这些粗略设计之后，我们开始在一个名为 `SprintView` 的新视图上渲染 sprint。

渲染 sprint

在开始为我们的 sprint 详情页面渲染数据之前，需要做几件事情。我们需要用某种方式从 API 中获取 sprint 数据并将它传递给模板以渲染细节。让我们在 *board/static/board/js/views.js* 中创建视图来指向这个模板并向它传递数据。

```
...
var SprintView = TemplateView.extend({                    ❶
        templateName: '#sprint-template',
    initialize: function (options) {
```

```
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprintId = options.sprintId;                                    ❷
        this.sprint = null;
        app.collections.ready.done(function () {
            self.sprint = app.sprints.push({id: self.sprintId});             ❸
            self.sprint.fetch({
                success: function () {
                    self.render();
                }
            });
        });
    },
    getContext: function () {
        return {sprint: this.sprint};
    }
});

app.views.HomepageView = HomepageView;
app.views.LoginView = LoginView;
app.views.HeaderView = HeaderView;
app.views.SprintView = SprintView;                                           ❹
...
```

❶ 从 TemplateView 继承并使用了已有的回调。

❷❸ app.sprints.push 会将一个新模型实例放入客户端集合中。该模型只知道模型的 id。后续的 fetch 方法会从 API 获取余下的详情。

❹ 要把 SprintView 加入 app.views，这样它就可以被路由使用了。

当视图初始化时从 API 获取 sprint，当它可用后，视图会再次渲染。当创建视图时需要传递 sprintId 给它，这样它就知道去获取哪一个 sprint 了。在将视图加入路由之前，需要将 sprint-template 加入 *index.html* 文件（位于 *board/templates/board*）。

```
...
<script type="text/html" id="sprint-template">
    <% if (sprint !== null) { %>
        <h2><%- sprint.get('name') %></h2>
        <span class="due-date">Due <%- sprint.get('end') %></span>
        <p class="description"><%- sprint.get('description') %></p>
        <div class="tasks"></div>
    <% } else { %>
        <h1 class="loading">Loading...</h1>
    <% } %>
</script>
</head>
...
```

与主页模板类似，我们创建了一个条件语句来处理 sprint 的多种情况：如果 sprint 不存在，

以及如果 sprint 正由 API 自身加载。通过使用 Underscore 模板和基本的 JavaScript，我们创造了一种呈现数据多个状态的简便方法。

现在我们可以将视图加入路由并为 sprint 详情页面创建一个 URL。

路由 sprint 详情

正如前面所提到的，当 `SprintView` 创建时，它需要得到 `sprintId`。这个值需要由路由来获得并传递给视图。和 Django 一样，Backbone 支持捕获路由中的一部分内容，但在语法上稍有区别。让我们将指向 sprint 详情页面的路由加入 *router.js* 文件（位于 *board/static/board/js*）。

```
...
var AppRouter = Backbone.Router.extend({
    routes: {
        '': 'home',
        'sprint/:id': 'sprint'                                    ❶
    },
    ...
    sprint: function (id) {                                       ❷
        var view = new app.views.SprintView({
            el: this.contentElement,
            sprintId: id
        });
        this.render(view);
    },
    route: function (route, name, callback) {
    ...
});
...
```

❶ 现在 `routes` 配置中添加一个新的条目，来映射 `sprint` 回调。它会捕获斜杠后面的值并将它以 `id` 的形式传递给回调函数。这也与之前主页模板以及 `NewSprintView` 的惯例一致。

❷ `sprint` 回调会接受 id 并创建和渲染一个新的 `SprintView`。

这里可以在 sprint 的详情 URL 上看到一小部分 sprint 详情，但无法看到相关的任务。这个视图存在的问题是，它没有利用客户端的状态，而这在应用中是非常关键的。让我们回顾一下这些客户端状态并考虑如何将其利用到这个过程中。

使用客户端状态

第一遍获取并渲染的 sprint 数据与要被写入服务器的内容非常一致。当我们处理活动的无状态数据时，要注意几件事情。要求服务器是无状态的，且所有的状态都要从数据库

中获取。然而，在客户端上，当我们需要模型对象，即本例中的 sprint 数据时，不需要每次都单击 API。

sprint 也有可能已经存在于 `app.sprints` 的集合中，只需要通过在 *board/static/board/js/models.js* 中添加一个简单的集合助手就可解决。

```
...
var BaseCollection = Backbone.Collection.extend({
    parse: function (response) {
        this._next = response.next;
        this._previous = response.previous;
        this._count = response.count;
        return response.results || [];
    },
    getOrFetch: function (id) {                                    ❶
        var result = new $.Deferred(),
            model = this.get(id);                                  ❷
        if (!model){
            model = this.push({id: id});                           ❸
            model.fetch({
                success: function (model, response, options) {
                    result.resolve(model);
                },
                error: function (model, response, options) {
                    result.reject(model, response);
                }
            });
        } else {
            result.resolve(model);
        }
        return result;
    }
});
...
```

❶ 这里将一个新的 `getOrFetch` 方法添加到了 `BaseCollection` 中。它返回一个延迟对象，并存放在模型实例中。

❷ 使用 `this.get` 在当前集合下用 ID 查找模型。调用 `this.get`，不向 API 服务器发送请求，只是查找匹配当前内存中的模型列表集合中的模型。如果在集合中找到模型，则立刻将延迟对象作为结果返回。

❸ 如果模型不在集合中，则从 API 获取。这一步使用了与原先视图实现相同的逻辑，先将空模型放入集合，再从 API 中取回。

现在可以把 `SprintView` 更新到 *board/static/board/js/views.js* 中，以便使用这个新方法。它会对在本地或 API 上成功找到 sprint 的情况进行处理，也会对给定 `sprintId` 不存在对应 sprint 的失败情况进行处理。

```
...
var SprintView = TemplateView.extend({
    templateName: '#sprint-template',
    initialize: function (options) {
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprintId = options.sprintId;
        this.sprint = null;
        app.collections.ready.done(function () {
            app.sprints.getOrFetch(self.sprintId).done(function (sprint) {    ❶
                self.sprint = sprint;
                self.render();
            }).fail(function (sprint) {                                       ❷
                self.sprint = sprint;
                self.sprint.invalid = true;
                self.render();
            });
        });
    },
    getContext: function () {
        return {sprint: this.sprint};
    }
});
...
```

❶ 原始的 `fetch` 模型被替换为 `getOrFetch`。由于它返回了一个延迟对象，在 sprint 可用时它必须与 done 回调相链接。

❷ 如果从 API 获取对象时出现错误，将执行 `fail` 回调。这种情况下，在渲染模板之前要将 sprint 标记为无效。

`fail` 处理器考虑了之前代码未涉及的问题：万一不存在给定 `id` 的 sprint 情况呢？最后一步所做的改进，就是更新模板（*board/templates/board/index.html*）处理 sprint 无效的情况。

```
...
<script type="text/html" id="sprint-template">
    <% if (sprint !== null) { %>
        <% if (!sprint.invalid) { %>                                          ❶
            <h2><%- sprint.get('name') %></h2>
            <span class="due-date">Due <%- sprint.get('end') %></span>
            <p class="description"><%- sprint.get('description') %></p>
            <div class="tasks"></div>
        <% } else { %>                                                        ❷
            <h1>Sprint <%- sprint.get('id') %> not found.</h1>
        <% } %>
     <% } else { %>
         <h1 class="loading">Loading...</h1> <% } %>
</script>
</head>
...
```

❶ 新增关于 sprint 是否设置了 `invalid` 标记的检查。

❷ 如果 sprint 标记为无效，要添加简单的消息让用户知道。

现在，主页已经可以渲染当前的 sprint 了，如图 6-1 所示。

图 6-1：sprint 主页

有了这项改进，用户可以在不初始化另一个 API 调用的情况下，从主页中调用 sprint 链接来查看详情。接下来我们将开始渲染与每个 sprint 相关的任务。

渲染任务

在现在的数据模型中，任务可被赋予四种状态之一：未开始、开发中、测试中和已完成。任务也可以脱离 sprint 存在。在这个应用中，希望以 5 个不同的列来呈现这些任务状态。让我们从处理这些列之一的视图开始，该视图位于 *board/static/board/js/views.js*。

```
...
var StatusView = TemplateView.extend({
    tagName: 'section',
    className: 'status',
    templateName: '#status-template',
    initialize: function (options) {
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprint = options.sprint;
        this.status = options.status;
        this.title = options.title;
    },
    getContext: function () {
        return {sprint: this.sprint, title: this.title};
    }
});

var SprintView = TemplateView.extend({
...
```

该视图与其他基于 TemplateView 的视图没有太大区别。它需要设置 3 个选项：sprint、status 和 title。将基础模板加入到 *board/templates/board/index.html*。

```
...
<script type="text/html" id="status-template">
    <h4><%- title %></h4>
    <div class="list"></div>
    <% if (sprint === null) { %>
        <button class="add" type="submit">Add Task</button>
    <% } %>
</script>
</head>
...
```

正如之前提到的，StatusView 一共有 5 个实例。它们会由 SprintView 来管理，一旦视图初始化好了，就在 *board/static/board/js/views.js* 中进行实现。

```
...
var SprintView = TemplateView.extend({
    templateName: '#sprint-template',
    initialize: function (options) {
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprintId = options.sprintId;
        this.sprint = null;
        this.statuses = {                                                    ❶
            unassigned: new StatusView({
                sprint: null, status: 1, title: 'Backlog'}),
            todo: new StatusView({
                sprint: this.sprintId, status: 1, title: 'Not Started'}),
            active: new StatusView({
                sprint: this.sprintId, status: 2, title: 'In Development'}),
            testing: new StatusView({
                sprint: this.sprintId, status: 3, title: 'In Testing'}),
            done: new StatusView({
                sprint: this.sprintId, status: 4, title: 'Completed'})
        };
        app.collections.ready.done(function () {
            app.sprints.getOrFetch(self.sprintId).done(function (sprint) {
                self.sprint = sprint;
                self.render();
            }).fail(function (sprint) {
                self.sprint = sprint;
                self.sprint.invalid = true;
                self.render();
            });
        });
    },
    getContext: function () {
        return {sprint: this.sprint};
    },
    render: function () {                                                    ❷
        TemplateView.prototype.render.apply(this, arguments);
```

```
            _.each(this.statuses, function (view, name) {
                $('.tasks', this.$el).append(view.el);
                view.delegateEvents();
                view.render();
            }, this);
        }
    });
    ...
```

❶ 当 SprintView 创建后，为每个可能的状态创建一个 StatusView。

❷ 渲染 SprintView 时会移除现有的子视图。这些子视图需要被重新加入 DOM 并再次绑定事件。

即使有了这些更新，现在还只能渲染列标题，因为我们还没有获取每个 sprint 的任务。要处理这个问题，需要在 *board/static/board/js/models.js* 中添加两个帮助方法。

```
...
app.models.Sprint = BaseModel.extend({
    fetchTasks: function () {                                          ❶
        var links = this.get('links');
        if (links && links.tasks) {
            app.tasks.fetch({url: links.tasks, remove: false});
        }
    }
});
...
app.collections.ready.done(function (data) {
    ...
    app.collections.Tasks = BaseCollection.extend({
        model: app.models.Task,
        url: data.tasks,
        getBacklog: function () {                                      ❷
            this.fetch({remove: false, data: {backlog: 'True'}});
        }
    });
    ...
});
...
```

❶ 第一个帮助方法是 Sprint 模型中的 fetchTasks，它使用了 API 提供的 tasks 链接。当获取完成后，任务会被添加到全局的 app.tasks 集合中。

❷ 第二个帮助方法是 Tasks 集合中的 getBacklog。它使用了 backlog = True 的过滤器来得到未分配给 sprint 的任务。与 fetchTasks 一致，结果会被添加到全局的 app.tasks 集合中。

这两个方法都没有返回值。作为替代，应用会在当这些项目被添加时使用 app.tasks 集合的事件触发。这些工作同样会由 *board/static/board/js/views.js* 中的 SprintView 来处理。

```
...
var SprintView = TemplateView.extend({
    templateName: '#sprint-template',
    initialize: function (options) {
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprintId = options.sprintId;
        this.sprint = null;
        this.statuses = {
            unassigned: new StatusView({
                sprint: null, status: 1, title: 'Backlog'}),
            todo: new StatusView({
                sprint: this.sprintId, status: 1, title: 'Not Started'}),
            active: new StatusView({
                sprint: this.sprintId, status: 2, title: 'In Development'}),
            testing: new StatusView({
                sprint: this.sprintId, status: 3, title: 'In Testing'}),
            done: new StatusView({
                sprint: this.sprintId, status: 4, title: 'Completed'})
        };
        app.collections.ready.done(function () {
            app.tasks.on('add', self.addTask, self);                          ❶
            app.sprints.getOrFetch(self.sprintId).done(function (sprint) {
                self.sprint = sprint;
                self.render();
                // Add any current tasks
                app.tasks.each(self.addTask, self);                           ❷
                // Fetch tasks for the current sprint
                sprint.fetchTasks();                                          ❸
            }).fail(function (sprint) {
                self.sprint = sprint;
                self.sprint.invalid = true;
                self.render();
            });
            // Fetch unassigned tasks
            app.tasks.getBacklog();                                           ❹
        });
    },
    getContext: function () {
        return {sprint: this.sprint};
    },
    render: function () {
        TemplateView.prototype.render.apply(this, arguments);
        _.each(this.statuses, function (view, name) {
            $('.tasks', this.$el).append(view.el);
            view.delegateEvents();
            view.render();
        }, this);
    },
    addTask: function (task) {
        // TODO: Handle the task
    }
});
...
```

❸❹ 当集合准备就绪时，就可以获取日志任务；当获取完sprint后，就可以获取相关的任务。

❶ 当结果返回时，app.tasks 会启动 add 事件，它绑定在视图的 addTask 上。

❷ 如果我们在 sprint 页面间进行浏览时，也有可能出现任务已经存在客户端的情况。尽管如此，也需要将它们添加进去。记住，并非所有任务都与该 sprint 相关，所以它们还要在 addTask 回调中进行过滤。

addTask 的回调会处理从 API 获取的任务以及已经存在于客户端的任务。只有当任务与 sprint 相关或是日志任务时，才会被渲染。渲染的地方取决于任务的状态。让我们在 *board/static/board/js/models.js* 的 Task 模型中添加一些新方法来帮助梳理这些逻辑。

```
...
app.models.Task = BaseModel.extend({
    statusClass: function () {                                          ❶
        var sprint = this.get('sprint'),
            status;
        if (!sprint) {
            status = 'unassigned';
        } else {
            status = ['todo', 'active', 'testing', 'done'][this.get('status') - 1];
        }
        return status;
    },
    inBacklog: function () {                                            ❷
        return !this.get('sprint');
    },
    inSprint: function (sprint) {                                       ❸
        return sprint.get('id') == this.get('sprint');
    }
});
...
```

❶ statusClass 帮助将任务映射到与其关联的 StatusView 中。

❷ inBacklog 决定任务是否出现在后台任务上。

❸ inSrpint 决定任务是否在一个给定的 sprint 上。

现在我们可以回到 *board/static/board/js/views.js* 中的 SprintView，继续添加任务。

```
...
var SprintView = TemplateView.extend({
    templateName: '#sprint-template',
    initialize: function (options) {
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprintId = options.sprintId;
        this.sprint = null;
        this.tasks = [];                                                ❶
```

```
    this.statuses = {
        unassigned: new StatusView({
            sprint: null, status: 1, title: 'Backlog'}),
        todo: new StatusView({
            sprint: this.sprintId, status: 1, title: 'Not Started'}),
        active: new StatusView({
            sprint: this.sprintId, status: 2, title: 'In Development'}),
        testing: new StatusView({
            sprint: this.sprintId, status: 3, title: 'In Testing'}),
        done: new StatusView({
            sprint: this.sprintId, status: 4, title: 'Completed'})
    };
    app.collections.ready.done(function () {
        app.tasks.on('add', self.addTask, self);
        app.sprints.getOrFetch(self.sprintId).done(function (sprint) {
            self.sprint = sprint;
            self.render();
            // Add any current tasks
            app.tasks.each(self.addTask, self);
            // Fetch tasks for the current sprint
            sprint.fetchTasks();
        }).fail(function (sprint) {
            self.sprint = sprint;
            self.sprint.invalid = true;
            self.render();
        });
        // Fetch unassigned tasks
        app.tasks.getBacklog();
    });
},
getContext: function () {
    return {sprint: this.sprint};
},
render: function () {
    TemplateView.prototype.render.apply(this, arguments);
    _.each(this.statuses, function (view, name) {
        $('.tasks', this.$el).append(view.el);
        view.delegateEvents();
        view.render();
    }, this);
    _.each(this.tasks, function (task) {                                    ❷
        this.renderTask(task);
    }, this);
},
addTask: function (task) {                                                  ❸
    if (task.inBacklog() || task.inSprint(this.sprint)) {
        this.tasks[task.get('id')] = task;
        this.renderTask(task);
    }
},
renderTask: function (task) {                                               ❹
    var column = task.statusClass(),
        container = this.statuses[column],
        html = _.template('<div><%- task.get("name") %></div>', {task: task});
    $('.list', container.$el).append(html);
```

```
    }
});
...
```

❶❸ 现在 addTask 过滤出与该视图相关的任务。并保存一个存放所有添加的任务的列表，以防视图再次进行渲染。

❷❹ 现在渲染任务由 Underscore 模板中的一个内联语句执行。我们之后将它重构为自己的子视图和模板。

有了 SprintView 和提供的一种处理方式以及与之相关的任务，我们现在可以编写所有组件来制作我们的任务视图了。

AddTaskView

Scrum 板应用的一个核心功能就是允许用户生成任务并把它们分配给 sprint。我们不仅需要创建模型、视图和模板，还要考虑管理数据的不同方式。

在这之前，我们向 *board/static/board/js/views.js* 中添加一个简单的 AddTaskView 来创建一种方式，供用户为 sprint 添加任务。

```
...
var AddTaskView = FormView.extend({                                    ❶
    templateName: '#new-task-template',
    events: _.extend({
        'click button.cancel': 'done'
    }, FormView.prototype.events),
    submit: function (event) {
        var self = this,
            attributes = {};
        FormView.prototype.submit.apply(this, arguments);
        attributes = this.serializeForm(this.form);                    ❷
        app.collections.ready.done(function () {
            app.tasks.create(attributes, {                             ❸
                wait: true,
                success: $.proxy(self.success, self),
                error: $.proxy(self.modelFailure, self)
            });
        });
    },
    success: function (model, resp, options) {
        this.done();
    }
});

var StatusView = TemplateView.extend({
...
```

❶ 我们将会拓展 FormView 来实现基本的模板渲染、表单提交和错误处理。

❷ 代码将表单序列化为 JSON 数据，供 API 使用。

❸ 这里我们在集合中创建了新任务，并分配多个与 API 交互的属性。因为使用的是模型的 save 方法，所以失败的情况被绑定到 FormView 中的 modelFailure 的回调上的。

和 NewSprintView 一样，这个视图也添加了一个处理取消按钮的事件处理器。让我们把它加入 FormView（位于 *board/static/board/js/views.js*）的默认设置中。

```
...
var FormView = TemplateView.extend({
    events: {
        'submit form': 'submit',
        'click button.cancel': 'done'                          ❶
    },
    ...
});
...
var NewSprintView = FormView.extend({
    templateName: '#new-sprint-template',
    className: 'new-sprint',                                   ❷
    ...
});
...
var AddTaskView = FormView.extend({
    templateName: '#new-task-template',                        ❸
    ...
});
...
```

❶ 现在 FormView 默认将所有 button.cancel 单击事件绑定到 done 方法。

❷❸ 现在 NewSprintView 和 AddTaskView 不用扩展 events，声明可以从每个视图中移除。

完成了这些细微的改进后，我们要在 *board/static/board/js/views.js* 中更新 StatusView 来渲染新的表单。

```
...
var StatusView = TemplateView.extend({
    tagName: 'section',
    className: 'status',
    templateName: '#status-template',
    events: {
        'click button.add': 'renderAddForm'                    ❶
    },
    initialize: function (options) {
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprint = options.sprint;
        this.status = options.status;
        this.title = options.title;
```

```
    },
    getContext: function () {
        return {sprint: this.sprint, title: this.title};
    },
    renderAddForm: function (event) {                                    ❷
        var view = new AddTaskView(),
            link = $(event.currentTarget);
        event.preventDefault();
        link.before(view.el);
        link.hide();
        view.render();
        view.on('done', function () {
            link.show();
        });
    }
});
...
```

❶ 现在，这个视图绑定到了一个单击 `add` 类按钮的事件处理器上。尽管这一视图的所有实例都会带有这个处理器，只有渲染按钮的模板是关于日志任务的 `StatusView` 实例。

❷ 与主页中添加 sprint 类似，当单击按钮时创建一个新的 `AddTaskView` 实例，无论是创建了新的任务还是用户单击了取消按钮，都在完成后将其移除。

现在可以将我们的 `new-task-template` 添加到 *index.html* 文件（位于 *board/templates/board*）来完成创建任务的工作流。

```
...
<script type="text/html" id="new-task-template">
  <form class="add-task" action="" method="post">
      <label for="id_name">Task Name</label>
      <input id="id_name" type="text" name="name" maxlength="100" required />
      <label for="id_description">Description</label>
      <textarea id="id_description" name="description"></textarea>
      <button class="create" type="submit">Create</button>
      <button class="cancel" type="submit">Cancel</button>
  </form>
</script>
</head>
...
```

为了让这个页面的结构更具可读性和易用性，我们在 *board/static/board/css/board.css* 中添加一些 CSS。

```
...
/* Tasks
==================== */

.tasks {
    display: -webkit-box;
    display: -moz-box;
```

```
    display: -ms-flexbox;
    display: -webkit-flex;
    display: flex;

    -webkit-flex-flow: row wrap;

    align-items: strech;
}

.tasks .status {
background: #EEE;
border: 1px solid #CCC;
padding: 0 10px 15px 10px;
margin-right: 8px;
width: 17%;
}

.tasks .status .list {
    display: -webkit-box;
    display: -moz-box;
    display: -ms-flexbox;
    display: -webkit-flex;
    display: flex;

    -webkit-flex-flow: row wrap;
    flex-direction: row;
    justify-content: space-around;
}

.task-detail input, .task-detail textarea {
    width: 90%;
}
```

我们需要添加一些其他的方式供用户与任务互动，在不同的阶段进行自定义。让我们使用四种不同的 CRUD（增删改查）操作，作为即将构建的交互基础。

CRUD 任务

在使用自定义 API 的时候，通过 Backbone 创建可编辑内容是开发 Web 应用众多体验中的一项强大工具。就我们的 Scrum 应用来说，可以创建 sprint 并给它添加任务。现在我们借助其他三种 CRUD 方法：查找、更新、删除，增加几种和数据的交互方式。

在 sprint 内渲染任务

开始之前，我们将先创建一个基本视图，把任务渲染到模板的各个不同组件。在 `TaskView` 的创建过程中，我们需要通过移除内联的 Underscore 模板并将 `renderTask` 方法指向新视图来重构 `SprintView` 模板。回顾一下 *board/static/board/js/views.js* 中目前的 `SprintView.renderTask` 方法。

```
var SprintView = TemplateView.extend({
...
    renderTask: function (task) {
        var column = task.statusClass(),
            container = this.statuses[column],
            html = _.template('<div><%- task.get("name") %></div>', {task: task});
        $('.list', container.$el).append(html);
    }
});
...
```

renderTask 有两个主要问题。一是内联模板不总与其他视图管理模板的方式一致，这在我们需要处理编辑任务的事件时会出现问题。二是 renderTask 将这个 HTML 插入一个由某个 StatusView 实例来渲染和控制的元素。更好的做法是让 StatusView 来决定放置任务的最佳位置。

这个内联模板可以很方便地与其他模板一起被放在 *board/templates/board/index.html* 的 <head> 中的模板上。

```
...
<script type="text/html" id="task-item-template">
    <%- task.get('name') %>
</script>
</head>
...
```

接下来，我们可以在 *board/static/board/js/views.js* 中创建 TaskItemView，用这个模板渲染每个任务。

```
...
var TaskItemView = TemplateView.extend({
    tagName: 'div',
    className: 'task-item',
    templateName: '#task-item-template',
    initialize: function (options) {
        TemplateView.prototype.initialize.apply(this, arguments);
        this.task = options.task;
        this.task.on('change', this.render, this);
        this.task.on('remove', this.remove, this);
    },
    getContext: function () {
        return {task: this.task};
    },
    render: function () {
        TemplateView.prototype.render.apply(this, arguments);
        this.$el.css('order', this.task.get('order'));
    }
});

var SprintView = TemplateView.extend({
...
```

TaskItemView遵循绝大多数已有的TempleteView示例模式，但有些许不同。在initialize中，它将自身绑定到模型的change和remove事件上。如果模型实例在客户端上有更新，视图会再次渲染自身以显示更新。如果模型被删除，视图会把自己从DOM中移除。render还在元素中设置了order的CSS，这会用于在伸缩盒布局（flexbox）中以正确的顺序进行渲染，不管DOM的位置。

现在既然我们已经有了基本的视图来显示任务，那么可以在*board/static/board/js/views.js*中的SprintView.renderTask里使用它了。

```
...
var StatusView = TemplateView.extend({
    ...
    addTask: function (view) {                                          ❶
        $('.list', this.$el).append(view.el);
    }
});
...
var SprintView = TemplateView.extend({
    templateName: '#sprint-template',
    initialize: function (options) {
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprintId = options.sprintId;
        this.sprint = null;
        this.tasks = {};                                                ❷
        this.statuses = {
            unassigned: new StatusView({
                sprint: null, status: 1, title: 'Backlog'}),
            todo: new StatusView({
                sprint: this.sprintId, status: 1, title: 'Not Started'}),
            active: new StatusView({
                sprint: this.sprintId, status: 2, title: 'In Development'}),
            testing: new StatusView({
                sprint: this.sprintId, status: 3, title: 'In Testing'}),
            done: new StatusView({
                sprint: this.sprintId, status: 4, title: 'Completed'})
        };
        app.collections.ready.done(function () {
            app.tasks.on('add', self.addTask, self);
            app.sprints.getOrFetch(self.sprintId).done(function (sprint) {
                self.sprint = sprint;
                self.render();
                // Add any current tasks
                app.tasks.each(self.addTask, self);
                // Fetch tasks for the current sprint
                sprint.fetchTasks();
            }).fail(function (sprint) {
                self.sprint = sprint;
                self.sprint.invalid = true;
                self.render();
            });
```

```
            // Fetch unassigned tasks
            app.tasks.getBacklog();
        });
    },
    getContext: function () {
        return {sprint: this.sprint};
    },
    render: function () {
        TemplateView.prototype.render.apply(this, arguments);
        _.each(this.statuses, function (view, name) {
            $('.tasks', this.$el).append(view.el);
            view.delegateEvents();
            view.render();
        }, this);
        _.each(this.tasks, function (view, taskId) {                    ❸
            var task = app.tasks.get(taskId);
            view.remove();
            this.tasks[taskId] = this.renderTask(task);
        }, this);
    },
    addTask: function (task) {
        if (task.inBacklog() || task.inSprint(this.sprint)) {
            this.tasks[task.get('id')] = this.renderTask(task);         ❹
        }
    },
    renderTask: function (task) {                                       ❺
        var view = new TaskItemView({task: task});
        _.each(this.statuses, function (container, name) {
            if (container.sprint == task.get('sprint') &&
                container.status == task.get('status')) {
                container.addTask(view);
            }
        });
        view.render();
        return view;
    }
});
...
```

❶ StatusView 有一个新的 addTask 任务来添加任务视图，并将其添加到 DOM 中。

❷❸ tasks 属性从列表被修改成了一个联合数组，这个数组将视图从 task.id 映射到子视图的实例中。更新 addTask 来进行分配。更新 render 在映射中做迭代。

❹

❺ renderTask 为新的 TaskItemView 创建了一个实例并遍历状态子视图。renderTask 现在将返回子视图，供 addTask 在视图映射中进行追踪。

我们在 *board/static/board/css/board.css* 中添加一些细微的样式来分隔任务。

```
...
.tasks .task-item {
    width: 100%;
    margin: 5px 0;
    padding: 3px;
}
```

第一个 sprint（"Something Sprint"）的详情应当如图 6-2 所示。

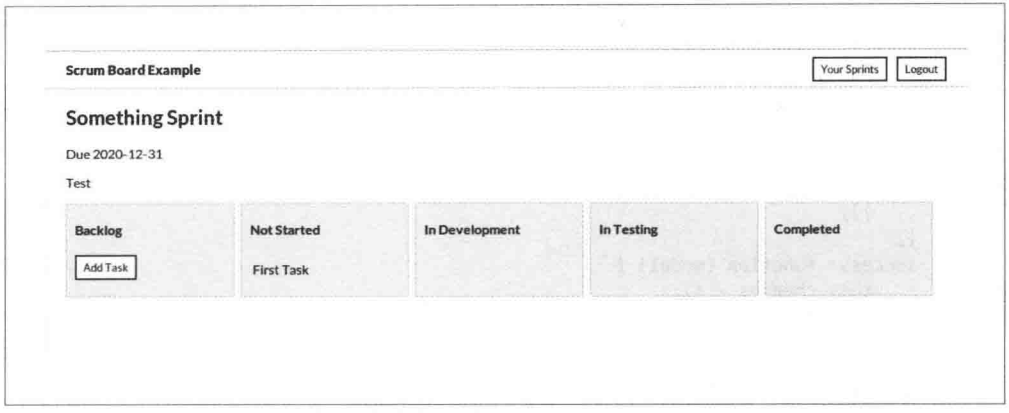


图 6-2：Sprint 详情

这里假设遵循了我们在第 4 章中探索 API 中创建和分配 sprint 以及任务的过程。

有了这些细小的结构，我们可以开始考虑用户如何查看任务的不同方面，以及视图中可能会发生的交互。

更新任务

由于任务是创建 sprint 内容的固有部分，很重要的一点是让用户能够查看和编辑任务的详情。实现这一功能的步骤类似添加任务。当用户单击任务项目时，将打开一个表单，显示任务的更多细节并允许进行编辑。用户可以保存编辑或点击取消按钮关闭表单。让我们从在 *board/static/board/js/views.js* 中创建一个详情视图开始。

```
...
var TaskDetailView = FormView.extend({
    tagName: 'div',
    className: 'task-detail',
    templateName: '#task-detail-template',
```

```
    initialize: function (options) {                                    ❶
        FormView.prototype.initialize.apply(this, arguments);
        this.task = options.task;
        this.changes = {};
        $('button.save', this.$el).hide();
        this.task.on('change', this.render, this);
        this.task.on('remove', this.remove, this);
    },
    getContext: function () {
        return {task: this.task, empty: '-----'};                       ❷
    },
    submit: function (event) {
        FormView.prototype.submit.apply(this, arguments);
        this.task.save(this.changes, {                                  ❸
            wait: true,
            success: $.proxy(this.success, this),
            error: $.proxy(this.modelFailure, this)
        });
    },
    success: function (model) {                                         ❹
        this.changes = {};
        $('button.save', this.$el).hide();
    }
});
var TaskItemView = TemplateView.extend({
...
```

❶ TaskDetailView 监听模型的改变与较小的 TaskItemView 类似。它同时也记录追踪用户对模型作出的改变。

❷ 除了任务之外，在没有内容与任务的某个特定属性关联时，模板会将“-----”作为值来渲染。

❸❹ 表单保存用户做出的改变。当保存成功时，那些改变会重置。

现在我们可以将任务详情模板添加到 *board/templates/board/index.html* 中，并引入相应的表单在 sprint 内创建一个新任务。

```
...
<script type="text/html" id="task-detail-template">
    <div data-field="name" class="name">
        <%- task.get('name') %>
    </div>
    <div data-field="description" class="description">
        <%- task.get('description') %>
    </div>
    <div class="with-label">
        <div class="label">Due:</div>
        <div data-field="due" class="due date">
            <%- task.get('due') || empty %>
        </div>
    </div>
```

```
    <div class="with-label">
        <div class="label">Assigned To:</div>
        <div data-field="assigned" class="assigned">
            <%- task.get('assigned') || empty %>
        </div>
    </div>
    <form>
        <button class="cancel" type="submit">Close</button>
        <button class="save" hidden type="submit">Save</button>
    </form>
</script>
</head>
...
```

保存按钮的默认状态是隐藏的，只有当保存发生变化时该按钮才会出现。当用户单击列表中的任务时，需要创建 TaskDetailView 实例。TaskItemView 需要监听这一事件。下面是我们在 *board/static/board/js/views.js* 中实现这些需求的方法。

```
...
var TaskItemView = TemplateView.extend({
    tagName: 'div',
    className: 'task-item',
    templateName: '#task-item-template',
    events: {
        'click': 'details'                                          ❶
    },
    initialize: function (options) {
        TemplateView.prototype.initialize.apply(this, arguments);
        this.task = options.task;
        this.task.on('change', this.render, this);
        this.task.on('remove', this.remove, this);
    },
    getContext: function () {
        return {task: this.task};
    },
    render: function () {
        TemplateView.prototype.render.apply(this, arguments);
        this.$el.css('order', this.task.get('order'));
    },
    details: function () {                                          ❷
        var view = new TaskDetailView({task: this.task});
        this.$el.before(view.el);
        this.$el.hide();
        view.render();
        view.on('done', function () {
            this.$el.show();
        }, this);
    }
});
...
```

❶ 现在，TaskItemView 将 click 事件绑定到新的 details 回调上。

❷ 在事件回调中，会为任务创建一个新的 TaskDetailView 实例。当前视图将会被隐藏直到编辑完成，这由 FormView 触发的 done 事件来指示。

现在我们可以通过单击展开任务详情，并使用关闭按钮再次隐藏它们，但现在任务还不能被编辑。视图中还没有东西更新用来追踪模型更新的 change 事件。接下来会处理这些事情。

内联编辑功能

在与编辑的内容保持内联的同时能够对其进行编辑是体现便于操作的关键部分。因为我们想要遵循应用中的这些模式，同时为用户创建交互，所以在叫做 contenteditable 的 HTML5 元素（位于 *board/static/board/js/views.js*）中使用自定义方法，以便进一步创建内联可编辑内容。

```
...
var TaskDetailView = FormView.extend({
    tagName: 'div',
    className: 'task-detail',
    templateName: '#task-detail-template',
    events: _.extend({
        'blur [data-field][contenteditable=true]': 'editField'                    ❶
    }, FormView.prototype.events),
...
    editField: function (event) {                                                 ❷
        var $this = $(event.currentTarget),
            value = $this.text().replace(/^\s+|\s+$/g,''),
            field = $this.data('field');
        this.changes[field] = value;
        $('button.save', this.$el).show();
    }
});
...
```

❶ 代码中添加了事件监听器，查找所有 contenteditable 在我们的模板中出现的地方。

❷ 这是个自定义方法，使特定内容字段可以编辑并保持到我们的 API 中。文本内容前后的空格会被移除。

有了新方法，现在可以加入模板（位于 *board/templates/board/index.html*）并将 contenteditable 分配给每个需要内联编辑的部分。DOM 的特别区域会通过 data-field 属性与模型关联。

```
...
<script type="text/html" id="task-detail-template">
    <div data-field="name" class="name" contenteditable="true">
        <%- task.get('name') %>
```

```
        </div>
        <div data-field="description" class="description" contenteditable="true">
            <%- task.get('description') %>
        </div>
        <div class="with-label">
            <div class="label">Due:</div>
            <div data-field="due" class="due date" contenteditable="true">
                <%- task.get('due') || empty %>
            </div>
        </div>
        <div class="with-label">
            <div class="label">Assigned To:</div>
            <div data-field="assigned" class="assigned" contenteditable="true">
                <%- task.get('assigned') || empty %>
            </div>
        </div>
        <form>
            <button class="cancel" type="submit">Close</button>
            <button class="save hide" type="submit">Save</button>
        </form>
    </script>
    </head>
    ...
```

这个方法存在的问题是，默认的 `FormView.showErrors` 不显示 API 中的错误信息。`FormView.showErrors` 依赖 `<input>` 和 `<label>` 标签的名称与模型名称相匹配。这些内容在目前的模板中未被显示。要处理这个问题，`TaskDetailView` 需要定义自己的 `showErrors`，并将错误与 `data-field` 属性中的正确区域关联，如下面 *board/static/board/js/views.js* 所示。

```
var TaskDetailView = FormView.extend({
...
    showErrors: function (errors) {
        _.map(errors, function (fieldErrors, name) {
            var field = $('[data-field=' + name + ']', this.$el);
            if (field.length === 0) {
                field = $('[data-field]', this.$el).first();
            }
            function appendError(msg) {
                var parent = field.parent('.with-label'),
                    error = this.errorTemplate({msg: msg});
                if (parent.length === 0) {
                    field.before(error);
                } else {
                    parent.before(error);
                }
            }
            _.map(fieldErrors, appendError, this);
        }, this);
    }
});
```

我们需要在 *board/static/board/css/board.css* 中添加额外的样式，以便将详情 / 编辑状态更好地与普通列表区分开。

```
...
/* Task Detail
==================== */
.task-detail {
    width: 100%;
    margin: 5px 0;
    background-color: #FFF;
    padding: 8px;
}

.task-detail div {
    margin-bottom: 10px;
}

.task-detail .name, .task-detail .description {
    border-bottom: 1px dotted #333;
}

.task-detail .with-label {
    display: -webkit-box;
    display: -moz-box;
    display: -ms-flexbox;
    display: -webkit-flex;
    display: flex;
}

[contenteditable=true]:hover {
    background: #EEE;
}

[contenteditable=true]:focus {
    background: #FFF;
}
```

现在我们已经有了功能齐全的 Backbone 应用，可以使用 REST API 来添加、编辑 sprint 和任务了。客户端和服务器被清楚地分离开，它们之间传递的配置最小。JavaScript 客户端从资源响应中提供的根节点和 links 信息发现 API 布局。当最初的页面加载完毕后，Django 应用只通过网络发送数据。由于负载小而集中，很节省带宽（对于移动设备是个好事），并可以很方便地做缓存（对所有人都是好事）。可以在客户端处理交互，不需要来回与服务器通信并刷新整个页面，为用户提供了无缝的操作体验。尽管在客户端上管理复杂的状态会变得很困难，Backbone 视图和模型配合 Underscore 模板，提供了一系列与 Django 类似的工具，让它们更易于管理。在接下来的章节中，我们会涵盖如何使用 websocket 来创建关于数据的实时效果，并提供更丰富的体验。

第 7 章

实时 Django

最近，创建实时 Web 应用或在已有应用中集成实时组件已经成为一个趋势。不过，Django 主要针对短暂的 HTTP 请求、响应周期，而且绝大多数 Python Web 服务器也不是为了支持实时应用而处理大量并发的长连接请求而生。Node.js 的兴起以及 Python 标准库中加入的 asyncio 使得许多开发者使用协同多任务以及事件循环作为高并发问题的解决方案。不过这一解决方案要求全栈都以这样的协同式风格来编写，以保证没有东西阻塞循环。除了低效长时间的轮询，没有什么明显的解决方案给 Django 应用加入实时特性。

本章中，我们探索如何将实时特性集成到同样基于 Django 的 REST API 的任务板应用上。实时特性会由一个新的服务器来处理，使用 Tornado 编写，使用异步 I/O 来处理大量的并发连接。我们将学习如何高效安全地允许 Django 向客户端发送更新。首先，为了更好地理解这个方法，我们将检查 HTML5 中定义的新的 Web API 集合。

HTML5 实时 API

我们将实时更新我们的应用，以便让用户看到在运动状态下顺序不断改变中的任务。对于我们的应用来说，客户端更新的协议已经由 REST API 定义好了。然而，还有一些很有必要知道但又不用保存下来的客户端更新，比如当用户开始移动任务时的更新。允许该应用通知正在查看 sprint 的其他客户端，告知他们有其他用户正在修改任务。由于客户端－服务器通信是双向的，因此我们要用 websocket 来实现这项功能。

实时 Web 从早期的长时间轮询和 Comet 发展到包括 websocket、服务器事件（server-sent events，SSE）以及 Web 实时通信（Web real-time communication，WebRTC）的 HTML5 新的标准。这些新标准每个都有不同的应用场合、不同的扩展需求以及当前适用的浏览

器。它们都可以在 Django 应用上使用一个类似本章中介绍的方法。下面我们来快速分析一下实时 Web API 的当前状态。

websocket

websocket 是浏览器和服务器之间双向通信的一个规范。其间的连接是持久的，意味着服务器必须要能够同时处理大量开通的连接。服务器端需要小心处理，并注意不要使用其他可能受限的资源。例如，如果每个 Web 服务器连接都开启一个数据库连接，那么连接的数量就不一定受限于 Web 服务器可以承载的连接数量，而是受限于数据库的连接数量。websocket 是人们最熟知的 HTML5 标准，拥有最好的浏览器支持。最新版本的主流桌面浏览器都支持 websocket。移动浏览器的支持也在提升中。

可以在 *http://caniuse.com/#feat=websockets* 上查看当前最新的浏览器对 websocket 的支持情况。

服务器事件（SSE）

像 websocket 一样，服务器事件也需要长时间的服务器连接。不过与 websocket 不同的是，它不是双向连接的。服务器事件连接允许服务器将新事件推送给客户端。正如你所期望的，客户端和服务器端的 API 都要比 websocket 协议简单很多。可以通过创建新的 HTTP 请求来处理任何来自客户端的更新。11.0 版的 IE 还不支持 SSE，但支持 websocket。

可以在 *http://caniuse.com/#feat=eventsource* 上查看当前最新的浏览器对服务器事件的支持情况。

WebRTC

不同于前面两种规范，WebRTC 是一个浏览器到浏览器的通信协议。尽管我们通常需要服务器来发现其他客户端并初始化握手，一旦连接建立后服务器就看不见通信了。最初它用于支持客户端间的音频和视频流的信道，但近来也开始用于发送任意数据了。这适用客户端彼此之间同步大量数据而服务器对这些数据并不在意的情况。WebRTC 可以与其他协议结合起来用于向服务器以及在客户端间作实时更新。这是 HTML5 标准中最晚

面世的协议，支持该协议的浏览器也最少，IE11.0 和 Safari8.0 都不支持。移动浏览器基本上都不支持。

可以在 *http://caniuse.com/#feat=rtcpeerconnection* 上查看最当前新的浏览器对 WebRTC 的支持情况。

在 Tornado 下使用 websocket

我们将使用 Python 编写的异步网络库和小型 Web 框架 Tornado 编写 websocket 服务器。这个服务器最初是由 FriendFeed 开发的，获得该服务器后一直由 Facebook 进行维护。它运行在基于 select/epoll 的单线程事件循环中，可以用很小的内存开销处理大量同时发生的连接。其核心的网络功能以及 Web 框架都简单易读，并有充足的文档和示例。目前，它拥有异步 Python 框架最成熟的 Python 3 的支持。

可以在项目网站上查看 Tornado 的更多特性，并阅读官方文档。

为什么我们不把所有东西全部搭设到一个 Django 服务器上？有以下几点原因。第一，Django 主要基于 WSGI 规范，并不只对处理长连接而制定。大多数 WSGI 服务器使用多线程处理高并发，但这一方案并不能直接扩展到大量长连接的场合。有输入输出限制的多线程 Python 应用也会经常因为全局解释器锁（Global Interpreter Lock，GIL）而出现问题。

有些人曾尝试使用轻量级或“绿色”线程代替原有线程。这种方法有一定优点，但其核心层面仍然工作在单线程事件的循环上，是合作多任务。如果应用中的某些部分在处理输入输出时不让步于事件循环，那么所有连接都会被阻塞。一些隐式的异步框架，如 gevent 和 eventlet，以及 Python 标准库中的 monkey-patch I/O 库，如对网络进行操作的 Python 的 socket 库的 .C 扩展，都没有针对这种情况进行改进。在这些隐式框架中追踪调试有关阻塞循环的程序错误可能非常困难，甚至是不可能的。

websocket 带有独立服务器的另一个原因是分离关注点。Django 期望接受短期连接，在这个 REST API 的情况下是完全无状态连接。websocket 连接是长期连接并且有可能与状态相关。这就导致它们有不同的扩展问题。此外，这个应用的实时特性更多是对于核心

功能的锦上添花，可能会存在客户端没办法或者不适合开启这一功能的情形，例如旧版本的浏览器或性能不足的移动设备。通过用分离进程来处理这些连接的方式，使得在实时服务器崩溃的情况下应用的核心功能依然可以使用。这就要求 Django 应用和实时服务器之间尽可能小地共享数据。所有需要共享的信息都将使用明确定义的 API 来完成。

Tornado 并非是这个服务器的唯一选项。实时服务器可以用 Node、Erlang、Haskell 或者其他任何看起来最适合的语言或框架编写。如果实现得当，在今后需要用另外一种不同的语言或框架替换时，可以保持 Django 不做任何改变。这里重要的内容是在两个服务器之间进行隔离，并进行明确的通信定义。

Tornado 简介

如前面提到的，我们将使用 Tornado 来搭建实时服务器组件。Tornado 可以使用 pip 通过 PyPi 来安装。

```
hostname $ pip install tornado==4.2
```

从 3.2 版本起，Tornado 加入了一个可选的 C 扩展来提升 websocket 连接的性能。然而，这需要安装一个 C 编译器。如果安装时在系统中找到了 C 编译器，扩展内容会自动安装。Tornado4.0 版附属加入了 certifi。

对于许多 Web 应用来说，无需深入底层了解 Tornado 提供的网络功能，只需要关注 Tornado 搭建的 Web 框架即可。这个框架主要由两个类构成：`RequestHandler` 和 `Application`。正如它的名字，`RequestHandler` 用于处理请求，它有着和 Django 的 `View` 类（所有视图的基类）相似的 API。`Application` 并不与 Django 中某个概念相对应。它包含 Django 的根 URL 配置和设置的功能，也就是说，`Application` 将 URL 映射到 `RequestHandler` 类上，同时也处理全局配置或共享资源。Tornado 使用子类 `tornado.websocket.WebSocketHandle` 来支持处理 websocket 连接。

为给定 sprint 处理更新内容的 Tornado 服务器的启动配置，叫做 *watercooler.py*，如下所示：

```
from urllib.parse import urlparse

from tornado.ioloop import IOLoop
from tornado.web import Application
from tornado.websocket import WebSocketHandler


class SprintHandler(WebSocketHandler):                                    ❶
    """Handles real-time updates to the board."""
```

```
    def check_origin(self, origin):                                    ❷
        allowed = super().check_origin(origin)
        parsed = urlparse(origin.lower())
        return allowed or parsed.netloc.startswith('localhost:')

    def open(self, sprint):
        """Subscribe to sprint updates on a new connection."""

    def on_message(self, message):
        """Broadcast updates to other interested clients."""

    def on_close(self):
        """Remove subscription."""


if __name__ == "__main__":
    application = Application([                                        ❸
        (r'/(?P<sprint>[0-9]+)', SprintHandler),
    ])
    application.listen(8080)
    IOLoop.instance().start()
```

❶ SprintHandler 是 websocket 连接的处理器。open、on_message 以及 on_close 方法是 WebSocketHandler 定义的 API，我们将加入所需的功能。

❸ Application 通过指向 websocket 处理器的单一路由来构建，并配置成监听 8080 端口。

❷ 重写 check_origin 来运行跨域名请求。它将允许所有运行在本地的服务器发送的连接。将其用于开发过程，并在以后进行更多的配置。

同样的，我们将它保存成 *watercooler.py*。现在它并不会做太多的事情，只是放置了一些方法。SprintHandler 将通过 open、on_message 和 on_close 方法来定义 websocket 交互，这些方法目前还都只是填充了一些文档字符串。

从 4.0 版开始，Tornado 通过 check_origin 默认拒绝交叉源间的 websocket 连接。前面提到的服务器要在 4.0 版之前的 Tornado 上运行，不过这项检查并不是强制的。

这个服务器在 8080 端口上运行。执行下面的脚本来启动服务器。

```
hostname $ python watercooler.py
```

和 Django 的开发服务器一样，这个服务器可以通过 Ctrl+C 来停止。因为没有东西捕获 keyboardInterrupt 异常，所以控制台会输出如下的堆栈跟踪：

```
hostname $ python watercooler.py
^CTraceback (most recent call last):
File "watercooler.py", line 31, in <module>
    IOLoop.instance().start()
...
KeyboardInterrupt
```

不用担心，这很正常，我们很快将会处理它。不同于 Django 的开发服务器，这个服务器默认不包含任何有用的输出。同时它也不会在代码更新后自动重载。我们可以在 *watercooler.py* 中使用应用设置中的 debug 标记来启用这些特性。

```
from urllib.parse import urlparse

from tornado.ioloop import IOLoop
from tornado.options import define, parse_command_line, options              ❶
from tornado.web import Application
from tornado.websocket import WebSocketHandler


define('debug', default=False, type=bool, help='Run in debug mode')          ❷
define('port', default=8080, type=int, help='Server port')
define('allowed_hosts', default="localhost:8080", multiple=True,
        help='Allowed hosts for cross domain connections')


class SprintHandler(WebSocketHandler):
    """Handles real-time updates to the board."""
    def check_origin(self, origin):
        allowed = super().check_origin(origin)                               ❸
        parsed = urlparse(origin.lower())
        matched = any(parsed.netloc == host for host in options.allowed_hosts)
        return options.debug or allowed or matched

    def open(self, sprint):
        """Subscribe to sprint updates on a new connection."""

    def on_message(self, message):
        """Broadcast updates to other interested clients."""

    def on_close(self):
        """Remove subscription."""


if __name__ == "__main__":
    parse_command_line()                                                     ❹
    application = Application([
        (r'/(?P<sprint>[0-9]+)', SprintHandler),
    ], debug=options.debug)                                                  ❺
    application.listen(options.port)
    IOLoop.instance().start()
```

❶ 这些是新引入的 Tornado 工具，用于处理命令行参数。

❷ 这里我们定义了可供命令行使用的选项。选项可以被映射到一个 Python 类型上。在本例中，debug 要么是 True 要么是 False，端口需要一个整数。allowed_hosts 可以被多次传递。

❹ 需要调用 parse_command_line 赋给 options。命令行中未指定的选项由 define 中给出的默认值赋给。

❺ 将 debug 传给应用实例，port 号用于启动应用。

❸ 更新 check_origin 以使用新的 allowed_hosts 和 debug 设置。

启用了 debug 后，会在有请求到达服务器时提供日志输出，包括收到的状态码以及响应时间。当 *watercooler.py* 保存后代码会重新加载。和 Django 一样，保存、重载时的语法错误会让程序崩溃，需要手动重新启动。

allowed_hosts 选项可以用于跨域连接。它允许连接到本地服务器或在命令行中使用 --allowed_hosts 选项配置的服务器。当启用了 debug 后，允许来自任何服务器的连接。

和 Django 一样，不推荐在产品环境中启用 debug 运行 Tornado 服务器。

要处理停止服务器的请求，服务器要监听键盘中断时发送的 SIGINT 信号。下面是需要对 *watercooler.py* 做的更新。

```
import logging                                                    ❶
import signal
import time

from urllib.parse import urlparse

from tornado.httpserver import HTTPServer                         ❷
...

def shutdown(server):                                             ❸
    ioloop = IOLoop.instance()
    logging.info('Stopping server.')                              ❹
    server.stop()

    def finalize():
        ioloop.stop()
        logging.info('Stopped.')                                  ❺


    ioloop.add_timeout(time.time() + 1.5, finalize)
```

```
if __name__ == "__main__":
    parse_command_line()
    application = Application([
        (r'/(?P<sprint>[0-9]+)', SprintHandler),
    ], debug=options.debug)
    server = HTTPServer(application)                                    ❻
    server.listen(options.port)
    signal.signal(signal.SIGINT, lambda sig, frame: shutdown(server))   ❼
    logging.info('Starting server on localhost:{}'.format(options.port))  ❽
    IOLoop.instance().start()
```

❶❷ 从标准库中导入的 Tornado 的 HTTP 服务器模块。

❸ 这个函数关闭HTTP服务器。它先让服务器停止接受新连接，然后在短暂的停顿之后，完全关闭 IOLoop。

❻ 代替 application.start，使用应用创建 HTTP 服务器实例并将其启动。

❼ 创建的服务器被绑定到一个 SIGINT 的处理器上。当信号被捕获时，调用之前的关闭代码。

❹❺❽ 代码设置登录机制，来提供有关当前服务器状态的更多信息。

这段代码源于 Tornado 社区中提供的有关优雅关闭的常见示例，还有一些瑕疵。它在完全停止之前要等待 1.5 秒，可能还不足以完成现有请求。更多有关停止 Tornado 服务器的内容，可以在 *https://groups.google.com/forum/#!topic/python-tornado/VpHp3kXHP7s* 以及 *https://gist.github.com/mywaiting/4643396* 中找到。

有了这些改进，可以在 debug 模式下使服务器输出更多信息，并能更优雅地退出。

```
hostname $ python watercooler.py --debug
[I 140620 21:56:23 watercooler:49] Starting server on localhost:8080
```

同样，我们可以使用 Ctrl+C 来停止：

```
hostname $ python watercooler.py --debug
[I 140620 21:56:23 watercooler:49] Starting server on localhost:8080
^C[I 140620 21:56:24 watercooler:31] Stopping server.
[I 140620 21:56:26 watercooler:36] Stopped.
```

在这里，我们可以做更多改进，如绑定到其他地址上，启用 HTTPS，或处理如 SIGTERM 或 SIGQUIT 在内的其他操作系统信号。在将这段代码放入实际产品前，很可能需要这些辅助功能。这些改进可以直接使用现有的例子，留给读者作为练习。现在我们已经为服务器添加功能做好了准备，使其能按一定要求处理客户端连接。

接下来，假设服务器开启 debug 标记在默认的 8080 端口下运行。

消息订阅

当一个新的客户端连接到 *http://localhost:8080/1/* 时，会调用 `SprintHandler.open`，并将 1 作为 sprint 的值传递给它。在 Django 中，使用正则表达式进行 URL 路由，命名的分组被解析成参数。任何时候都可以通过 `write_message` 方法将新消息发送给客户端。服务器需要订阅客户端来获取所有与 sprint 相关的和在 sprint 中任务被更新时的更新信息。在我们最初的实现中，可以在服务器的 `Application` 中追踪这些订阅。我们将会创建一个 `Application` 的子类来使用字典存放每个 sprint 的订阅者。

```
import logging
import signal
import time

from collections import defaultdict                                    ❶
from urllib.parse import urlparse
...
class ScrumApplication(Application):

    def __init__(self, **kwargs):
        routes = [
            (r'/(?P<sprint>[0-9]+)', SprintHandler),
        ]
        super().__init__(routes, **kwargs)
        self.subscriptions = defaultdict(list)                         ❷

    def add_subscriber(self, channel, subscriber):                     ❸
        self.subscriptions[channel].append(subscriber)

    def remove_subscriber(self, channel, subscriber):                  ❹
        self.subscriptions[channel].remove(subscriber)

    def get_subscribers(self, channel):                                ❺
        return self.subscriptions[channel]


def shutdown(server):
    ioloop = IOLoop.instance()
    logging.info('Stopping server.')
    server.stop()

    def finalize():
        ioloop.stop()
        logging.info('Stopped.')

    ioloop.add_timeout(time.time() + 1.5, finalize)
```

```
if __name__ == "__main__":
    parse_command_line()
    application = ScrumApplication(debug=options.debug)                    ❻
    server = HTTPServer(application)
    server.listen(options.port)
    signal.signal(signal.SIGINT, lambda sig, frame: shutdown(server))
    logging.info('Starting server on localhost:{}'.format(options.port))
    IOLoop.instance().start()
```

❶❷❻ 在创建应用实例时，将创建一个新的字典存储订阅。字典将把 sprint 的 ID 映射到一个连接列表上。当创建应用时，路由也会同时被创建而非被传递。

❸❹ 应用不会将包含的 `subscriptions` 字典显示出来，而是显示管理和查询可用订阅者的

❺ `add_subscriber`、`remove_subscriber` 和 `get_subscribers` 方法。这种提取操作便于之后因子的重新设置。

因为服务器应用是单线程和单进程的，所以这样管理订阅是可行的。每个处理器都可以通过它的 `application` 属性访问到应用实例。当连接一个新的客户端时，处理器会注册客户端获取有关 sprint 的更新，如下面 *watercooler.py* 中的代码所示。

```
...
class SprintHandler(WebSocketHandler):
    """Handles real-time updates to the board."""
...
    def open(self, sprint):
        """Subscribe to sprint updates on a new connection."""
        self.sprint = sprint
        self.application.add_subscriber(self.sprint, self)
...
```

如果某个客户端关闭连接，则需要将它从订阅者列表中移除。在 *watercooler.py* 中，我们可以这样实现：

```
...
class SprintHandler(WebSocketHandler):
    """Handles real-time updates to the board."""
...
    def on_close(self):
        """Remove subscription."""
        self.application.remove_subscriber(self.sprint, self)
...
```

这段代码注册客户端以获取更新，但还没有东西来处理这些更新。当服务器从它的一个客户端中获取到一条消息时，它需要把消息广播给所有感兴趣的客户端。因为应用追踪了所有的订阅，所以它也可以处理广播，像这里的 watercooler.py 所示。

```
...
from tornado.websocket import WebSocketHandler, WebSocketClosedError
```

```
...
class ScrumApplication(Application):
...
    def broadcast(self, message, channel=None, sender=None):
    if channel is None:
        for c in self.subscriptions.keys():
        self.broadcast(message, channel=c, sender=sender)
    else:
        peers = self.get_subscribers(channel)
        for peer in peers:
            if peer != sender:
              try:
                peer.write_message(message)
            except WebSocketClosedError:
                # Remove dead peer
                self.remove_subscriber(channel, peer)
...
```

在这里，我们直接将消息按原接收状态从客户端转发给所有同类端点。这里所谓的同类端点是指所有对同一个 sprint 感兴趣的客户端。如果没有这样的客户端，消息就会消失。如果尝试给一个已经关闭了的连接发送信息，会抛出 `WebSocketClosedError` 异常。当这个异常抛出时，这个同类端点则被直接从订阅者列表中移除。

我们也可以不考虑 sprint，而通过 `channel=None` 给所有客户端广播消息。websocket 处理器不会用到这个特性，而会在之后从 Django 应用中传送更新信息时用到。

当处理器接收消息时，它会调用应用广播消息并将自己作为发送者传递给它，以保证消息不会重新发送回来，如下面的 *watercooler.py* 所示：

```
...
class SprintHandler(WebSocketHandler):
    """Handles real-time updates to the board."""
...
    def on_message(self, message):
        """Broadcast updates to other interested clients."""
        self.application.broadcast(message, channel=self.sprint, sender=self)
...
```

这段代码在这里工作是正常的，不过它有许多限制。首先，它只能用于单个进程。所有的订阅都由内存中的一个应用实例来管理。另外，从 Django 应用向这个服务器发送与同类端点通信消息的途径也不够明确。有关这些问题将在本章后面进行处理。

客户端通信

目前为止我们只关注了 websocket 连接的服务器端，但实际上它是一个双向连接。浏览器客户端需要连接，向服务器端发送消息，并用服务器处理发送来的消息。首先我们来

看看标准 API 是如何定义的，接下来我们给它加上一个包装程序，以便更好地集成现有的 Backbone 应用代码。

简单示例

对于客户端来说，websocket 是个相对简单的 API。我们通过创建一个新的 WebSocket 来生成新的连接，并使用以下回调函数来处理套接字的动作：

- onopen
- onmessage
- onclose
- onerror

```
var socket = new WebSocket('ws://localhost:8080/123');
socket.onopen = function () {
    console.log('Connection is open!');
    socket.send('ping');
};
socket.onmessage = function (message) {
    console.log('New message: ' + message.data);
    if (message.data == 'ping') {
        socket.send('pong');
    }
};
```

由于目前服务器不会对传递的 sprint 进行验证，所以示例中使用了固定数值 123。要测试连接，请确保启动了 *watercooler.py* 服务器，并运行：

```
hostname $ python watercooler.py
[I 140629 11:47:58 watercooler:86] Starting server on localhost:8080
```

然后打开 *http://localhost:8080/*。可以将该代码粘贴到选用浏览器的开发工具控制台上对其进行测试。

http://localhost:8080/ 会返回 404 错误，因为我们的应用并没有定义该 URL。不过，仍旧可以从该页面中完成测试。

粘贴第一段代码时，会在控制台中显示“Connection is open!”的消息，如图 7-1 所示。

图 7-1：浏览器开发工具中 websocket 的初始连接

如果打开一个新的标签页并再次粘贴这段代码，将打开一个新的连接显示两个客户端的通信。最新的标签页将得到“pong”回复，如图 7-2 所示。

图 7-2：浏览器开发工具中 websocket 连接的第二个标签页

回到最初的标签页，将看到第二个标签页发来的“ping”消息，如图 7-3 所示。

图 7-3：浏览器开发工具中 websocket 连接的初始标签页

正如该例所示，当前服务器不加选择地将所有消息发送给相关订阅者。要想在接收端有意义地处理这些消息，需要更加结构化的格式。接着，我们为这个连接编写一个 JS 包装程序，将套接字的消息转换成 Backbone 事件。

套接字包装程序

当在前端进行操作，例如用户拖动一个任务时，我们会使用实时通信来通知其他客户端。要为消息添加结构，客户端将使用 JSON 来编写消息代码。每个消息都包含被操作模型的类型、模型的 ID 以及进行的操作。这些消息会被转换成 Backbone 事件，这样视图就可以订阅特定的事件，而无需从 websocket 获取所有消息。

对消息的编码和转换将由 *board/static/board/js/socket.js* 中创建的一个包装脚本来处理。

```
(function ($, Backbone, _, app) {
```

```
var Socket = function (server) {                                    ❶
    this.server = server;
    this.ws = null;
    this.connected = new $.Deferred();
    this.open();
};

Socket.prototype = _.extend(Socket.prototype, Backbone.Events, {
    open: function () {                                             ❷
        if (this.ws === null) {
            this.ws = new WebSocket(this.server);
            this.ws.onopen = $.proxy(this.onopen, this);
            this.ws.onmessage = $.proxy(this.onmessage, this);
            this.ws.onclose = $.proxy(this.onclose, this);
            this.ws.onerror = $.proxy(this.onerror, this);
        }
        return this.connected;
    },
    close: function () {
        if (this.ws && this.ws.close) {
            this.ws.close();
        }
        this.ws = null;
        this.connected = new $.Deferred();
        this.trigger('closed');
    },
    onopen: function () {                                           ❸
        this.connected.resolve(true);
        this.trigger('open');
    },
    onmessage: function (message) {                                 ❹
        var result = JSON.parse(message.data);
        this.trigger('message', result, message);
        if (result.model && result.action) {                        ❺
            this.trigger(result.model + ':' + result.action,
                result.id, result, message);
        }
    },
    onclose: function () {                                          ❻
        this.close();
    },
    onerror: function (error) {                                     ❼
        this.trigger('error', error);
        this.close();
    },
    send: function (message) {                                      ❽
        var self = this,
            payload = JSON.stringify(message);
        this.connected.done(function () {
            self.ws.send(payload);
        });
    }
});
```

```
    app.Socket = Socket;
})(jQuery, Backbone, _, app);
```

❶ 构造器使用套接字地址来开启套接字，设置初始状态来追踪和开启套接字。

❷ 如果还没有创建真正的 websocket 实例，使用 open 创建该 websocket 实例，并将所有套接字事件绑定到同名的本地方法上。

❸❹ onopen、onmessage、onclose 以及 onerror 方法处理 websocket 事件并将它们转译
❻❼ 成同名事件。它们同时也对套接字自身的连接状态进行维护。

❺ onmessage 对来自服务器的消息进行解码。在收到一条消息时它不仅发送一揽子消息事件，还为每个模型动作发送带命名空间的消息。

❽ send 处理发送给服务器消息的编码。它使用 connected 状态确保消息不被发送给一个已经关闭或还未开启的套接字。

它会将以下形式的消息：

```
{
    model: 'modelname',
    id: 'id',
    action: 'actionname'
}
```

转译命名空间为 *modelname:actionname* 的事件，并将 id 值作为事件处理器的第一个参数传递。之后，订阅者可以监听消息的特定子集以及所有其他消息。

要让这个工具在客户端可用，我们需要将它加入 *index.html* 模板（位于 *board/templates/board*）。

```
...
<script id="config" type="text/json">
    {
        "models": {},
        "collections": {},
        "session": null,
        "views": {},
        "router": null,
        "apiRoot": "{% url 'api-root' %}",
        "apiLogin": "{% url 'api-token' %}"
    }
</script>
<script src="{% static 'board/js/app.js' %}"></script>
<script src="{% static 'board/js/socket.js' %}"></script>                    ❶
...
```

❶ 这是一个对之前套接字包装程序的引用。它需要在 *app.js* 配置后面引用，但要在

views.js 之前使用。

下一步就是将包装程序集成到当前的视图中。

客户端连接

客户端需要知道 websocket 的地址来建立连接。服务器地址可以通过 `apiRoot` 和 `apiLogin` 的 url 这样的配置数据来传递。这是一个易于配置的静态方法。另一个动态方法是将这个地址作为 API 响应的一部分来引用。它可以是自身的资源，也可以是该应用中 sprint 的子资源。要将这个信息传递给客户端，需要通过更改 `SprintSerializer` 将它以另一个链接的形式传递。Django 应用需要获知 websocket 服务器的地址，为此我们可以在 *scrum/settings.py* 中添加一个新的设置。

```
...
STATIC_URL = '/static/'

WATERCOOLER_SERVER = os.environ.get('WATERCOOLER_SERVER', 'localhost:8080')

WATERCOOLER_SECURE = bool(os.environ.get('WATERCOOLER_SECURE', ''))
```

这里默认指向 *http://localhost:8080/*，但可以在今后部署产品时通过 `WATERCOOLER_SERVER` 环境变量来配置它。默认情况下，期望服务器不使用安全协议，但可以配置。

`SprintSerializer` 可以通过这一设置和请求的 sprint 来创建 websocket 的 URL，如下面这段 *board/serializers.py* 中的代码所示。

```
...
from django.conf import settings
from django.contrib.auth import get_user_model
...

class SprintSerializer(serializers.ModelSerializer):
...
    def get_links(self, obj):
    request = self.context['request']
    return {
        'self': reverse('sprint-detail',
            kwargs={'pk': obj.pk}, request=request),
        'tasks': reverse('task-list',
            request=request) + '?sprint={}'.format(obj.pk),
        'channel': '{proto}://{server}/{channel}'.format(
            proto='wss' if settings.WATERCOOLER_SECURE else 'ws',
            server=settings.WATERCOOLER_SERVER,
            channel=obj.pk
        ),
    }
```

这个方法的好处是：尽管目前服务器的地址是静态的，为了对 websocket 连接进行负载平衡，也可以将其做成动态的。该方法也允许在不修改客户的情况下修改路径配置，并使其更易于保护 websocket 端点的安全。另外，这也不是唯一可行的做法。这个应用着重强调 sprint 上 websocket 的用法，但如果用法超出这里的范围，也可以在 API 中将 websocket 频道作为它们自身的资源。

客户端会对用户在应用详情页面中拖曳任务的事件进行跟踪。不要忘了这个页面是由 *board/static/board/js/views.js* 中定义的 `SprintView` 来控制的。作为开始，套接字连接要在初始化视图时进行创建，在视图删除时被关闭。

```
...
var SprintView = TemplateView.extend({
    templateName: '#sprint-template',
    initialize: function (options) {
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprintId = options.sprintId;
        this.sprint = null;
        this.tasks = {};
        this.statuses = {
            unassigned: new StatusView({
                sprint: null, status: 1, title: 'Backlog'}),
            todo: new StatusView({
                sprint: this.sprintId, status: 1, title: 'Not Started'}),
            active: new StatusView({
                sprint: this.sprintId, status: 2, title: 'In Development'}),
            testing: new StatusView({
                sprint: this.sprintId, status: 3, title: 'In Testing'}),
            done: new StatusView({
                sprint: this.sprintId, status: 4, title: 'Completed'})
        };
        this.socket = null;                                              ❶
        app.collections.ready.done(function () {
            app.tasks.on('add', self.addTask, self);
            app.sprints.getOrFetch(self.sprintId).done(function (sprint) {
                self.sprint = sprint;
                self.connectSocket();                                    ❷
                self.render();
                // Add any current tasks
                app.tasks.each(self.addTask, self);
                // Fetch tasks for the current sprint
                sprint.fetchTasks();
            }).fail(function (sprint) {
                self.sprint = sprint;
                self.sprint.invalid = true;
                self.render();
            });
            // Fetch unassigned tasks
            app.tasks.getBacklog();
        });
    },
```

```
...
    connectSocket: function () {                                    ❸
        var links = this.sprint && this.sprint.get('links');
        if (links && links.channel) {
            this.socket = new app.Socket(links.channel);
        }
    },
    remove: function () {                                           ❹
        TemplateView.prototype.remove.apply(this, arguments);
        if (this.socket && this.socket.close) {
            this.socket.close();
        }
    }
});
...
```

❶ 创建视图时初始化它的 socket 属性。

❷❸ 一旦获取到 sprint，调用一个新方法来连接。connectSocket 将从 API 响应中得到 websocket 的地址。

❹ 重写默认的 remove 方法，关闭已开启的套接字。

websocket 连接已经可以使用，客户端现在可以开始广播详细描述用户行为的事件了。这些行为将使用之前介绍的格式在套接字上进行广播。

从客户端发送事件

当前，应用支持向后台日志中添加新的任务，但不允许用户改变任务的状态栏。用户能够将任务从一个栏拖到另一个栏。当用户拖曳任务时，应用会广播拖拽事件的开始和结束，以便其他查看 sprint 的客户端能够更新状态。

HTML5 中引入了一个针对元素拖曳的 JavaScript API。它对于大多数桌面浏览器都很支持，但对移动端浏览器的支持还很欠缺。API 以元素在不同阶段触发的新事件为中心。在处理拖曳过程中最后的释放操作时，还可以给拖曳的元素附带一些数据以获取更多的环境信息。

http://www.html5rocks.com/en/tutorials/dnd/basics/ 是 API 的一项很好的基本资源。*http://caniuse.com/#feat=dragndrop* 包含了更多的有关浏览器对固有拖曳功能的支持信息。

回顾一下，每个任务都是由一个定义在 *views.js* 中的 TaskItemView 来渲染的。这些任务

都是用户可以拖曳的元素，用户需要处理相关的事件。然而，TaskItemView 并没有访问套接字。要么需要让 SprintView 把套接字向下传递给这些子视图，要么让子视图触发可以被能够访问套接字的父视图捕获的事件。使用事件可以更好地分离各项职责，也是我们在 *board/static/board/js/views.js* 中要采用的方法。

```
...
var TaskItemView = TemplateView.extend({
    tagName: 'div',
    className: 'task-item',
    templateName: '#task-item-template',
    events: {
        'click': 'details',
        'dragstart': 'start',                                          ❶
        'dragenter': 'enter',
        'dragover': 'over',
        'dragleave': 'leave',
        'dragend': 'end',
        'drop': 'drop'
    },
    attributes: {                                                      ❷
        draggable: true
    },
    ...
    start: function (event) {                                          ❸
        var dataTransfer = event.originalEvent.dataTransfer;
        dataTransfer.effectAllowed = 'move';
        dataTransfer.setData('application/model', this.task.get('id'));
        this.trigger('dragstart', this.task);
    },
    enter: function (event) {                                          ❹
        event.originalEvent.dataTransfer.effectAllowed = 'move';
        event.preventDefault();
        this.$el.addClass('over');
    },
    over: function (event) {                                           ❺
        event.originalEvent.dataTransfer.dropEffect = 'move';
        event.preventDefault();
        return false;
    },
    end: function (event) {                                            ❻
        this.trigger('dragend', this.task);
    },
    leave: function (event) {                                          ❼
        this.$el.removeClass('over');
    },
    drop: function (event) {                                           ❽
        var dataTransfer = event.originalEvent.dataTransfer,
            task = dataTransfer.getData('application/model');
        if (event.stopPropagation) {
            event.stopPropagation();
        }
        task = app.tasks.get(task);
        if (task !== this.task) {
```

```
            // TODO: Handle reordering tasks.
        }
        this.trigger('drop', task);
        this.leave();
        return false;
    }
});
...
```

❶ 这些是由拖曳 API 定义的新事件，将被绑定到视图中名字类似的方法上。

❷ draggable=true 属性也是 API 的一部分，说明元素是可以拖曳的。

❸ 当用户选择元素并开始移动时，start 事件被触发。客户端将存储当前任务的 ID，在元素被释放后使用。

❹❺ 当另一个元素被拖曳到当前元素上方时，enter/over 事件会被触发。防止事件告诉浏览器当前元素可以被释放。它还要添加一个新类来直观地表示出当前位于该元素上方的元素。

❼ leave 事件对应着离开当前元素释放区域的元素。在元素移到上方时添加的类在此时被删除。

❻ end 事件在被拖曳的元素停止拖曳却没有被释放到新位置时被触发。

❽ 最后，当元素被释放到新位置时触发 drop 事件。

新添加的 over 类可以使用 *board/static/board/css/board.css* 中一小段 CSS 来处理：

```
...
[draggable=true] {
    cursor: move;
}

.tasks .task-item.over {
    opacity: 0.5;
    border: 1px black dotted;
}
```

任务元素现在可以被拖曳到另一个元素上。更新模型状态的逻辑还没有被实现，将在以后来处理，但现在需要用 SprintView 来处理一些新事件。在处理过程中每个任务都会触发 dragstart、dragend 和 drop 事件。SprintView 需要监听这些事件，并将它们广播到正在浏览相同 sprint 的客户端。我们需要在 *board/static/board/js/views.js* 里重新配置 renderTask 方法，来使用这些新的交互。

```
...
```

```
var SprintView = TemplateView.extend({
    templateName: '#sprint-template',
...
    renderTask: function (task) {
        var view = new TaskItemView({task: task});
        _.each(this.statuses, function (container, name) {
            if (container.sprint == task.get('sprint') &&
                container.status == task.get('status')) {
                container.addTask(view);
            }
        });
        view.render();
        view.on('dragstart', function (model) {                    ❶
            this.socket.send({
                model: 'task',
                id: model.get('id'),
                action: 'dragstart'
            });
        }, this);
        view.on('dragend', function (model) {                      ❷
            this.socket.send({
                model: 'task',
               id: model.get('id'),
               action: 'dragend'
            });
        }, this);
        view.on('drop', function (model) {                         ❸
            this.socket.send({
                model: 'task',
                id: model.get('id'),
                action: 'drop'
            });
        }, this);
        return view;
    },
...
```

❶❷ 当视图触发拖曳事件时，它们被传给套接字。记住任务是作为第一个参数传给事件
❸ 处理器的。

通过在子视图中使用事件，我们可以在单独一个地方保存处理套接字连接逻辑和消息格式。然而，还有另外一种拖曳情况需要处理。目前，任务只能被拖曳到其他任务上。还不允许把任务移动到新的状态栏上。我们需要在 *board/static/board/js/views.js* 中使用 `StatusView` 来处理一些拖曳事件，来支持这种情况。

```
...
var StatusView = TemplateView.extend({
    tagName: 'section',
    className: 'status',
    templateName: '#status-template',
    events: {
```

```
            'click button.add': 'renderAddForm',
            'dragenter': 'enter',                                          ❶
            'dragover': 'over',
            'dragleave': 'leave',
            'drop': 'drop'
        },
        ...
        enter: function (event) {                                          ❷
            event.originalEvent.dataTransfer.effectAllowed = 'move';
            event.preventDefault();
            this.$el.addClass('over');
        },
        over: function (event) {                                           ❸
            event.originalEvent.dataTransfer.dropEffect = 'move';
            event.preventDefault();
            return false;
        },
        leave: function (event) {                                          ❹
            this.$el.removeClass('over');
        },
        drop: function (event) {                                           ❺
            var dataTransfer = event.originalEvent.dataTransfer,
                task = dataTransfer.getData('application/model');
            if (event.stopPropagation) {
                event.stopPropagation();
        }
            // TODO: Handle changing the task status.
            this.trigger('drop', task);
            this.leave();
        }
    });
    ...
```

❶ 由于用户无法拖曳栏目本身，所以不用处理 dragstart 和 dragend 事件。此外，和 TaskItemView 不同，这个视图没有被标记成可拖曳。

❷❸❹ 和 TaskItemView 一样，当任务元素位于栏目上方时，会添加或删除一个新的类。防止 dragover 事件告诉浏览器这是一个合法的释放位置。

❺ 最后，状态栏需要使用事件中附带的数据来处理任务释放动作。更新任务的逻辑将在接下来处理。

任务被拖曳到栏目上的视觉表示是由 *board/static/board/css/board.css* 中添加到状态栏的 over 类实现的：

```
...
.tasks .status.over {
    border: 1px black dotted;
}
```

由于状态栏只是释放目标，自身无法移动，所以只有一个新的事件触发器。它表示任务已经被释放在这个栏目里。需要更新 *board/static/board/js/views.js* 中的 SprintView 来监听这个事件。

```
...
var SprintView = TemplateView.extend({
    templateName: '#sprint-template',
    initialize: function (options) {
        var self = this;
        TemplateView.prototype.initialize.apply(this, arguments);
        this.sprintId = options.sprintId;
        this.sprint = null;
        this.tasks = {};
        this.statuses = {
            unassigned: new StatusView({
                sprint: null, status: 1, title: 'Backlog'}),
            todo: new StatusView({
                sprint: this.sprintId, status: 1, title: 'Not Started'}),
            active: new StatusView({
                sprint: this.sprintId, status: 2, title: 'In Development'}),
            testing: new StatusView({
                sprint: this.sprintId, status: 3, title: 'In Testing'}),
            done: new StatusView({
                sprint: this.sprintId, status: 4, title: 'Completed'})
        };
        _.each(this.statuses, function (view, name) {
            view.on('drop', function (model) {                          ❶
                this.socket.send({
                    model: 'task',
                    id: model.get('id'),
                    action: 'drop'
                });
            }, this);
        }, this);
        ...
```

❶ 视图监听每一个 StatusView 实例的 drop 事件，当有任务被释放时，广播这一消息。

这个事件以和之前事件相同的格式被广播出去。在我们更新代码向 API 输出任务更新之前，先来看看在另一个连接的客户端上是如何处理这些事件的。

处理来自客户端的事件

当用户开始和结束拖曳动作后加载和广播事件时，SprintView 已经建立完成了 websocket 连接。如果有其他用户正在查看同一个 sprint，套接字会得到有关 websocket 的消息，不过 SprintView 中还没有东西监听这些消息。

当 SprintView 从 websocket 中得到消息时，它应当做什么呢？正如这里的 *board/static/board/js/views.js* 所示，它应该为最初的用户展示一些可见的提示，告诉他另外一个用户正在修改任务，以防止其他用户此时也去修改这个任务。

```
...
var TaskItemView = TemplateView.extend({
...
    lock: function () {                                                    ❶
        this.$el.addClass('locked');
    },
    unlock: function () {
        this.$el.removeClass('locked');
    }
});
...
var SprintView = TemplateView.extend({
...
    connectSocket: function () {
        var links = this.sprint && this.sprint.get('links');
        if (links && links.channel) {
            this.socket = new app.Socket(links.channel);
            this.socket.on('task:dragstart', function (task) {             ❷
                var view = this.tasks[task];
                if (view) {
                    view.lock();
                }
            }, this);
            this.socket.on('task:dragend task:drop', function (task) {     ❸
                var view = this.tasks[task];
                if (view) {
                    view.unlock();
                }
            }, this);
        }
    },
...
```

❶ lock 和 unlock 是 TaskItemView 中的新方法，它们添加或删除一个类来显示任务是否正在被其他用户修改。当任务被锁定时，如果还有防止修改的其他状态存在，可以在这里切换。

❷ 套接字监听任务开始时的拖曳事件，查询相关的子视图，并锁定它。

❸ 当拖曳完成时，无论是否被释放，子视图都被解锁。

针对这个新添加的类，*board.css* 文件（位于 *board/static/board/css*）需要少量更新。

```
...
.tasks .task-item.locked {
    opacity: 0.5;
}
```

这里将给用户明确的显示提示，告诉他们任务正在被别的用户移动。目前客户端没有其他强制性限制来防止其他用户修改相同任务，但可以在 **lock/unlock** 方法中实现。应用可以用同样的方法来处理任务编辑。TaskItemView 会触发一个事件来表明用户已经开始编辑，SprintView 可以将这一事件传递给 websocket，当其他客户端收到这一消息时，任务可以被锁定或以其他形式告知用户。当编辑完成后，会触发一个新的事件。

要查看这项工作情况，需要开启两个浏览器窗口查看同一个 sprint 页面。当在一个窗口中开始拖曳任务时，另一个窗口会改变正在被拖曳任务的透明度。当拖曳的任务被放下时，它将恢复正常状态。

task:drop 事件令客户端了解其他用户移动了某一任务，但它们并没有提供足够的相关信息让客户端确定任务被移动的去向。事实上，目前在事件触发时任务状态并没有更新。现在来看看这个问题。

更新任务状态

当我们在 TaskItemView 和 StatusView 中创建了释放事件的处理器时，有一些 TODO 项目来处理任务状态的更新。具体来说，有哪些状态需要被更新？任务被释放时的栏目决定了新的任务状态。改变状态可能会改变模型中追踪的某些日期，比如说起止日期。任务在每个栏目下都是排过序的，因此移动任务也要允许对它们进行重新排序。要捕获这些逻辑，我们要在 *board/static/board/js/modesl.js* 中的 Task 模型里新增一个帮助方法。

```
...
app.models.Task = BaseModel.extend({
...
    moveTo: function (status, sprint, order) {                          ❶
        var updates = {
            status: status,
            sprint: sprint,
            order: order
        },
        today = new Date().toISOString().replace(/T.*/g, '');
        // Backlog Tasks                                                ❷
        if (!updates.sprint) {
            // Tasks moved back to the backlog
            updates.status = 1;
        }
        // Started Tasks                                                ❸
        if ((updates.status === 2) ||
            (updates.status > 2 && !this.get('started'))) {
            updates.started = today;
        } else if (updates.status < 2 && this.get('started')) {
            updates.started = null;
        }
        // Completed Tasks                                              ❹
```

```
        if (updates.status === 4) {
            updates.completed = today;
        } else if (updates.status < 4 && this.get('completed')) {
            updates.completed = null;
        }
        this.save(updates);
    }
});
```

❶ 该方法用到了三个信息：新的状态、新的 sprint 和新的顺序。使用这些数值来计算任务状态的辅助更新。

❷ 被分配到后台日志的任务会把状态重设为 1，意思是“未开始”。

❸ 如果任务被标记为已开始，但还未指定开始日期，则会指定为当前日期。同样的，如果任务还没开始但却设定了开始日期，这个日期会被清除。

❹ 与新添加的任务类似，当任务被标记为已完成时，会设置完成日期。如果任务在以后从完成的状态栏中被移出，这个日期会被清除。

`TaskItemView.dorp` 处理将一个任务释放到其他任务上的情况。当一个任务被释放到另一个任务之上时，它应当排在当前任务之前。如果任务位于不同的状态栏中，该任务也需要相应地改变它的状态。从后台日志移出的任务应该分配给当前的 sprint。如下面这段 *board/static/board/js/views.js* 中的代码片段所示。

```
...
var TaskItemView = TemplateView.extend({
    drop: function (event) {
        var self = this,
            dataTransfer = event.originalEvent.dataTransfer,
            task = dataTransfer.getData('application/model'),
            tasks, order;
        if (event.stopPropagation) {
            event.stopPropagation();
        }
        task = app.tasks.get(task);
        if (task !== this.task) {
            // Task is being moved in front of this.task
            order = this.task.get('order');
            tasks = app.tasks.filter(function (model) {                    ❶
                return model.get('id') !== task.get('id') &&
                    model.get('status') === self.task.get('status') &&
                    model.get('sprint') === self.task.get('sprint') &&
                    model.get('order') >= order;
            });
            _.each(tasks, function (model, i) {                            ❷
                model.save({order: order + (i + 1)});
            });
            task.moveTo(                                                   ❸
                this.task.get('status'),
```

```
                this.task.get('sprint'),
                order);
        }
        this.trigger('drop', task);
        this.leave();
        return false;
    },
...
```

❶ 获取所有匹配当前 sprint 状态并在顺序上大于当前任务的任务。

❷ 每个匹配到的任务都把顺序增 1。

❸ 每个被释放的任务都会和它被释放时所在的任务进行 sprint、状态和顺序的匹配。

回顾一下，这一事件是由释放目标来触发的，`this.task` 是放置到它上面的任务，`task` 是被用户移动的任务。这里处理的是将一个任务移动到另一个之上的情况。下面 *board/static/board/js/views.js* 中显示的 `StatusView.drop` 是当任务被释放到栏目中的处理情况，这种情况可能有也可能没有。

```
...
var StatusView = TemplateView.extend({
    drop: function (event) {
        var dataTransfer = event.originalEvent.dataTransfer,
            task = dataTransfer.getData('application/model'),
            tasks, order;
        if (event.stopPropagation) {
            event.stopPropagation();
        }
        task = app.tasks.get(task);
        tasks = app.tasks.where({sprint: this.sprint, status: this.status}); ❶
        if (tasks.length) {
            order = _.min(_.map(tasks, function (model) {
                return model.get('order');
            }));
        } else {
            order = 1;
        }
        task.moveTo(this.status, this.sprint, order - 1);                    ❷
        this.trigger('drop', task);
        this.leave();
    }
});
...
```

❶ 查找所有与当前栏目的 sprint 和状态相匹配的任务，对这些任务进行迭代获取最小的顺序。

❷ 将状态视图中任务的 sprint 和状态进行更新。将任务的顺序设为比给定的最小顺序

小 1 的顺序，以便所有其他的任务就不用重新排序了。这项操作只允许向栏目中任务列表的顶部添加任务。

这项操作更新的是模型状态，实际并没有将任务的 HTML 从一个状态栏移动到另一个状态栏。要处理这个问题，我们需要更新 SprintView（位于 *board/static/board/js/views.js*）来处理任务模型的变化。

```
...
var SprintView = TemplateView.extend({
    templateName: '#sprint-template',
    initialize: function (options) {
        ...
        app.collections.ready.done(function () {
            app.tasks.on('add', self.addTask, self);
            app.tasks.on('change', self.changeTask, self);                    ❶
            app.sprints.getOrFetch(self.sprintId).done(function (sprint) {
                ...
        });
    },
    ...
    changeTask: function (task) {                                             ❷
        var changed = task.changedAttributes(),
            view = this.tasks[task.get('id')];
        if (view && typeof(changed.status) !== 'undefined' ||
            typeof(changed.sprint) !== 'undefined') {
            view.remove();
            this.addTask(task);
        }
    }
});
...
```

❶ 当 SprintView 被初始化时，它与 app.tasks 集合中的更新事件处理器相绑定。这是一个包含所有当前存储在客户端中任务的集合。

❷ 当任何任务改变时，会触发 changeTask 回调。如果任务对于当前 SprintView 实例是全新的，状态或 sprint 有所改变，那么该任务的当前视图被移除，新增事件处理器会被重新触发，并将它放置到正确的状态栏中。

随着这些改变的完成，当任务被释放时，它将在 DOM 中被移动，但这不会反映到其他客户端上。当这些释放事件被触发时，它们可能会传递新的状态、sprint 以及顺序值。然而，为了应对一个任务被放置在另一个任务之上的情况，栏目中的所有的任务都要被重新排序，并且可能都需要被更新。客户端可以获取所有这些任务，但在什么时候完成 API 更新和什么时候重新获取之间可能产生条件竞争。最好是在保存完成之后 Django 应用能够广播更新。在下一章中，我们将看到如何用 Django 应用通过 Tornado 服务器给客户端发送更新。

第 8 章

Django 与 Tornado 通信

在之前的章节中，我们创建了一个基于 Tornado 的服务器，用于处理 websocket 连接，在客户端之间发送消息。在本章中，我们将拓展该服务器，让它允许 Django 应用推送更新，增强 websocket 连接的安全性，并将 Redis 集成为消息代理来提升将来服务器的可伸缩性。

从 Tornado 接收更新

为了防止之前章节中介绍的当释放任务时 API 正在更新而出现的竞态条件，需要用某种机制让 Django 服务器向 Tornado 服务器推送更新。现在 Django 服务器可以通过 `WATERCOOLER_SERVER` 设置来获取服务器的地址，但正如之前所提到的，Tornado 服务器管理着所有的订阅并在内部进行广播，因此没有机制来广播 Tornado 外部的消息。

在后面的内容中我们将会更新服务器，使用 Redis 作为广播的消息代理。在这个案例中，Django 可以直接向 Redis 发布消息。然而，这需要求 Django 应用知道消息的格式以及 Tornado 是怎样广播消息的，不过这会打破我们之前定义好的两个服务器之间的交互。

怎么解决这一问题呢？为了让这些应用保持独立，Tornado 服务器会将自身的 HTTP API 暴露出来以从 Django 那里接收更新。当 Tornado 服务器接收到了更新，它会转化成必要的格式并广播给任何感兴趣的客户端。

这个端点会由 *watercooler.py* 中一个新的 `UpdateHandler` 来处理。

```
import json                    ❶
import logging
import signal
import time
...
```

```
from tornado.web import Application, RequestHandler                    ❷
...
class UpdateHandler(RequestHandler):
    """Handle updates from the Django application."""

    def post(self, model, pk):                                         ❸
        self._broadcast(model, pk, 'add')

    def put(self, model, pk):                                          ❹
        self._broadcast(model, pk, 'update')

    def delete(self, model, pk):                                       ❺
        self._broadcast(model, pk, 'remove')

    def _broadcast(self, model, pk, action):
        message = json.dumps({
            'model': model,
            'id': pk,
            'action': action,
        })
        self.application.broadcast(message)                            ❻
        self.write("Ok")


class ScrumApplication(Application):

    def __init__(self, **kwargs):
        routes = [
            (r'/(?P<sprint>[0-9]+)', SprintHandler),
            (r'/(?P<model>task|sprint|user)/(?P<pk>[0-9]+)', UpdateHandler),  ❼
        ]
        super().__init__(routes, **kwargs)
        self.subscriptions = defaultdict(list)
...
```

❶❷ 这里引用了标准库中 json，以及 Tornado 的 RequestHandler。

❸❹❺ UpdateHandler 接收 POST、PUT 和 DELETE 方法，分别对应添加、更新以及删除操作。

❻ 由于在调用 broadcast 时没有指定通道，消息会被发送给所有连接着的客户端。

❼ 将 UpdateHandler 加入到应用路由中。model 和 pk 参数由 URL 模式的正则表达式来进行验证。

现在 Tornado 服务器会在 */task/* 下显示一个端点，供 Django 发送有关新任务或修改任务的更新。尽量保持端点具有足够的通用性，以便供其他应用模型使用。

从 Django 发送更新

我们需要修改 Django 服务器，以便在添加、修改和删除某个模型时，向 websocket 服务器发送适当的请求。这项工作可由 `django-rest-framework` 中的 `post_save` 和 `pre_delete` 来实现。由于所做的这项修改会被添加到所有 API 视图上，因此要在 *board/views.py* 中用一个混合类来实现。

```
import requests                                                    ❶

from django.conf import settings
from django.contrib.auth import get_user_model
...
class UpdateHookMixin(object):
    """Mixin class to send update information to the websocket server."""

    def _build_hook_url(self, obj):
        if isinstance(obj, User):
            model = 'user'
        else:
            model = obj.__class__.__name__.lower()
        return '{}://{}/{}/{}'.format(
            'https' if settings.WATERCOOLER_SECURE else 'http',
            settings.WATERCOOLER_SERVER, model, obj.pk)

    def _send_hook_request(self, obj, method):
        url = self._build_hook_url(obj)
        try:
            response = requests.request(method, url, timeout=0.5)
            response.raise_for_status()
        except requests.exceptions.ConnectionError:
            # Host could not be resolved or the connection was refused
            pass
        except requests.exceptions.Timeout:
            # Request timed out
            pass
        except requests.exceptions.RequestException:
            # Server responsed with 4XX or 5XX status code
            pass

    def perform_create(self, serializer):
        super().perform_create(serializer)
        self._send_hook_request(serializer.instance, 'POST')

    def perform_update(self, serializer):
        super().perform_update(serializer)
        self._send_hook_request(serializer.instance, 'PUT')

    def perform_destory(self, instance):
        self._send_hook_request(instance, 'DELETE')
        super().perform_destroy(instance)

class SprintViewSet(DefaultsMixin, viewsets.ModelViewSet):
...
```

❶ 和前几章中使用的 Python 客户端一样，它使用 `requests` 库来构建发给 Tornado 服务器的请求。

这个混合类为 API 处理 `post_save` 和 `pre_delete` 事件，并将这些事件翻译成给 websocket 服务器的请求。它需要 `pre_delete` 而不是 `post_delete` 方法，以便保持对象 `pk` 有效。这里有一个短暂的停顿以防止钩子长时间地阻塞 API 请求。类似的，大多数异常都会被阻止。在这里，关闭了所有异常，以防止出现额外的登录或处理要求。

完成了这个混合类后，要把它加到 *board/views.py* 的 `SprintViewSet`、`TaskViewSet` 和 `UserViewSet` 中。

```
...
class SprintViewSet(DefaultsMixin, UpdateHookMixin, viewsets.ModelViewSet):   ❶
    """API endpoint for listing and creating sprints."""

    queryset = Sprint.objects.order_by('end')
    serializer_class = SprintSerializer
    filter_class = SprintFilter
    search_fields = ('name', )
    ordering_fields = ('end', 'name', )


class TaskViewSet(DefaultsMixin, UpdateHookMixin, viewsets.ModelViewSet):      ❷
    """API endpoint for listing and creating tasks."""

    queryset = Task.objects.all()
    serializer_class = TaskSerializer
    filter_class = TaskFilter
    search_fields = ('name', 'description', )
    ordering_fields = ('name', 'order', 'started', 'due', 'completed', )


class UserViewSet(DefaultsMixin, UpdateHookMixin, viewsets.ReadOnlyModelViewSet): ❸
    """API endpoint for listing users."""

    lookup_field = User.USERNAME_FIELD
    lookup_url_kwarg = User.USERNAME_FIELD
    queryset = User.objects.order_by(User.USERNAME_FIELD)
    serializer_class = UserSerializer
    search_fields = (User.USERNAME_FIELD, )
```

❶❷❸ 将另一个 `UpdateHookMixin` 基类赋给每个 `ViewSet`，触发更新 websocket 服务器。

另外一种可行的方法是使用 Django 的信号处理器，这些处理器使用类似的 API。使用这种方法，可以处理每个模型的保存和删除操作，而不仅仅处理由 API 请求所生成的那些操作。这个方法与我们的方案相比各有优劣。其中的一个优点是，客户端可以接收来自 API 之外的更新，就像后台任务一样。主要缺点是，在没有客户端监听下仍然广播更新，这样会增加对每个模型进行保存时的开销。而如果只广播由 API 创建的更新，那么仅会在至少有一个客户端使用 API 的情况下才会增加这笔开销。

在客户端上处理更新

目前 Django 服务器会在有模型更新时通知 Tornado 服务器，Tornado 服务器会将这些更新广播给所有连接的客户端。不过，目前客户端上还没有东西来监听 `<model>:add`、`<model>:update` 或 `<model>:remove` 事件。我们可以在 *board/static/board/js/views.js* 中的 `SprintView` 里添加这些功能。

```
...
var SprintView = TemplateView.extend({
...
    connectSocket: function () {
        var links = this.sprint && this.sprint.get('links');
        if (links && links.channel) {
            this.socket = new app.Socket(links.channel);
            this.socket.on('task:dragstart', function (task) {
                var view = this.tasks[task];
                if (view) {
                    view.lock();
                }
            }, this);
            this.socket.on('task:dragend task:drop', function (task) {
                var view = this.tasks[task];
                if (view) {
                    view.unlock();
                }
            }, this);
            this.socket.on('task:add', function (task, result) {                ❶
                var model = app.tasks.push({id: task});
                model.fetch();
            }, this);
            this.socket.on('task:update', function (task, result) {             ❷
                var model = app.tasks.get(task);
                if (model) {
                    model.fetch();
                }
            }, this);
            this.socket.on('task:remove', function (task) {                     ❸
                app.tasks.remove({id: task});
            }, this);
        }
    },
...
```

❶ 当发现 add 事件时获取任务。

❷ 当在集合中发现任务时，会对它们进行更新。由于更新是发送给所有客户端的，因此当前 sprint 之外的任务并不需要被更新。

❸ 删除任务后，将它们从集合中移除。

这些修改用以通过注册 SprintView 来处理任务更新。在应用中不允许出现修改用户的内容，尽管可以添加新的 sprint，但当查看 sprit 详情时，它并不和用户相关。如果需要这些事件，也可以用类似方式来添加订阅管理。

当一条任务被置成新的状态或者在状态栏中被重新排序时，所有连接的客户端都会对该任务进行更新和移动。要再次对这个任务进行测试的话，需要打开两个浏览器窗口并查看同一个 sprint。当在第一个浏览器中完成放置后，两个浏览器会同时将任务移至新的状态栏。

当客户端获取到关于新任务或对任务的编辑事件时，客户端会从 API 重新获取该任务。这对于进行最初添加或编辑的客户端是低效的，因为这样会重新获取已有的任务信息。还意味着所有客户端会同时进行 API 调用，这会对我们自己的服务器造成攻击，而拒绝提供微服务。这些问题将在后续内容中进行处理。

改善服务器

现在当我们查看 sprint 详情时，已经拥有了实时更新任务的全部功能。不过，正如前面所提到的，我们的 Tornado 服务器还有一些改进空间。轻量级并不意味着不安全，也不意味着不健壮。

健壮的订阅管理

Tornado 服务器实现中的问题之一是无法扩展到多个进程。所有的订阅信息和消息广播都是由内存中的任务实例来完成。当我们创建单个 Tornado 进程来处理大量并行的长连接时，单进程不具备很高的容错机制。在目前的配置下，如果超过一个实例，就可能无法通知到所有需要消息的客户端。

要解决这一问题，服务器会使用 Redis 作为消息分发器。Redis 还是一个流行的键值存储器，支持发布－订阅（publish-subscribe）频道。尽管 Redis 不是 RabbitMQ 或 ActiveMQ 那样真正意义上的消息分发器，但它便于安装和管理，在这里还被用作消息队列来进行简单的消息模式识别。同时，Tornado 对集成 Redis 有大量支持。可以查看 Redis 文档来获取在本地系统上安装和进行配置的方法。

我们需要 PyPi 中的 tornado-redis 库来与 Redis 交互。Tornado-redis 推荐使用同步的 Redis 客户端向一个频道发送消息，因此还需要安装 redis-py。

```
hostname $ pip install tornado-redis==2.4.18 redis==2.10.3
```

tornado-redis 内建了一些在使用 SockJS 或 Socket.IO 时管理发布－订阅频道的类。因

为本项目使用了不带这些抽象概念的 websocket，所以只需要对提供的 BaseSubscriber 进行简单扩展。我们需要把它加入到 *watercooler.py*。为了使用这个 RedisSubscriber，ScrumApplication 以及它的相关订阅方法也需要同时得到更新。

hiredis 是一个用 C 写的 Redis 客户端，它的 Python 封装可以在 PyPi 中下载得到。如果安装了 hiredis，redis-py 会将它作为语法分析程序，以便提升速度。所以在尝试安装之前需要有一个编译器和对应操作系统的 Python 头文件。

```
...
from redis import Redis
from tornado.httpserver import HTTPServer
from tornado.ioloop import IOLoop
from tornado.options import define, parse_command_line, options
from tornado.web import Application, RequestHandler
from tornado.websocket import WebSocketHandler
from tornadoredis import Client
from tornadoredis.pubsub import BaseSubscriber


class RedisSubscriber(BaseSubscriber):                                    ❶

    def on_message(self, msg):
        """Handle new message on the Redis channel."""
        if msg and msg.kind == 'message':
            subscribers = list(self.subscribers[msg.channel].keys())
            for subscriber in subscribers:
                try:
                    subscriber.write_message(msg.body)
                except tornado.websocket.WebSocketClosedError:
                    # Remove dead peer
                    self.unsubscribe(msg.channel, subscriber)
        super().on_message(msg)
...
class ScrumApplication(Application):

    def __init__(self, **kwargs):
        routes = [
            (r'/(?P<sprint>[0-9]+)', SprintHandler),
            (r'/(?P<model>task|sprint|user)/(?P<pk>[0-9]+)', UpdateHandler),
        ]
        super().__init__(routes, **kwargs)
        self.subscriber = RedisSubscriber(Client())                       ❷
        self.publisher = Redis()                                          ❸

    def add_subscriber(self, channel, subscriber):
        self.subscriber.subscribe(['all', channel], subscriber)           ❹

    def remove_subscriber(self, channel, subscriber):
        self.subscriber.unsubscribe(channel, subscriber)                  ❺
        self.subscriber.unsubscribe('all', subscriber)
```

```
    def broadcast(self, message, channel=None, sender=None):                    ❻
        channel = 'all' if channel is None else channel
        self.publisher.publish(channel, message)
...
```

❶ 新的 Redis 客户端要处理发布－订阅频道的订阅，并将相关信息传递给订阅的 websocket 处理器。这里假设 Redis 服务器运行在不带权限验证的本地服务器的默认端口上。复制订阅者列表，以防止在迭代过程中将它们移除。

❷❸ 用两个 Redis 连接替换之前位于内存中的订阅字典：一个管理异步客户端的订阅，另一个广播新消息的同步客户端。

❹ `add_subscriber` 的签名仍旧保持不变，但现在它订阅两个 Redis 频道的连接：一个是当前 sprint 的名字，另一个是要发送给所有客户端的消息。

❺ 与此类似，`remove_subscriber` 基本没有改变，但它必须解除对新的 `all` 频道的订阅。

❻ 调整 `broadcast` 方法以便在未指定频道时使用 `all` 频道。这里 `sender` 未被使用，我们将在之后修复这个问题。

由于 `ScrumApplication` 中的方法签名与之前保持一致，因此无需改变 `SprintHandler` 中的调用。

新的 `all` 频道的使用是必要的，因为在有多个 websocket 服务器的情况下，`ScrumApplication` 不再记得所有频道的名字。这会带来一个小问题，因为 `RedisSubscriber` 不知道哪个 `SprintHandler`（如果有的话）在频道里发送了消息，所以客户端会收到自己发送出去的消息。JS 客户端可以在消息主体中加入一个标识，以便忽略自己发送出去的消息。或者也可以在每个 `SprintHandler` 中进行同样处理。本项目中，我们会在服务器（*watercooler.py*）中处理这个问题，以减少 websocket 上的通信。

```
import json
import logging
import signal
import time
import uuid                                                                     ❶
...
class RedisSubscriber(BaseSubscriber):

    def on_message(self, msg):
        """Handle new message on the Redis channel."""
        if msg and msg.kind == 'message':
            try:                                                                ❷
                message = json.loads(msg.body)
                sender = message['sender']
                message = message['message']
            except (ValueError, KeyError):
```

```
                message = msg.body
                sender = None
            subscribers = list(self.subscribers[msg.channel].keys())
            for subscriber in subscribers:
                if sender is None or sender != subscriber.uid:
                    try:
                        subscriber.write_message(message)
                    except tornado.websocket.WebSocketClosedError:
                        # Remove dead peer
                        self.unsubscribe(msg.channel, subscriber)
            super().on_message(msg)
...
class SprintHandler(WebSocketHandler):
    """Handles real-time updates to the board."""

    def open(self, sprint):
        """Subscribe to sprint updates on a new connection."""
        # TODO: Validate sprint
        self.sprint = sprint.decode('utf-8')
        self.uid = uuid.uuid4().hex                                    ❸
        self.application.add_subscriber(self.sprint, self)
...
class ScrumApplication(Application):
...
    def broadcast(self, message, channel=None, sender=None):
        channel = 'all' if channel is None else channel
        message = json.dumps({                                         ❹
            'sender': sender and sender.uid,
            'message': message
        })
        self.publisher.publish(channel, message)
...
```

❶ 引入了新的标准库 uuid。

❸ 每个连接处理器在打开连接时生成一个唯一的 ID。

❹ 当消息被广播时，如果它有原始发送者，则把该信息作为注释包裹进去。

❷ 当在频道中接收消息时，将展开这个包裹来获取原始信息以及发送者。如果没有发送者，消息被发送给所有订阅者。否则，它会将消息发送给除原始发送者之外的所有订阅者。

我们通过使用 Redis 作为简单的消息分发器，可以将 Tornado 服务器拓展到多个进程中。不但服务器的可拓展性得到了提升，还可以使服务器变得更加安全。

websocket 权限验证

目前我们还没有针对发送给 SprintHandler 实例的 sprint ID 进行验证。这种做法并不好，有如下两点原因。首先，它开启服务器用于各种消息转发。当服务器被部署成生产服务

器并显示在公共互联网上时，任何两个客户端都可以连接到一个随机整数的 sprint 上并通过 socket 发送消息。其次，某个恶意客户端可以用一个真实存在的 sprint ID 连接到频道上，并广播错误的信息，致使在未出现更新时使连接的客户端误认为有更新出现。添加这个验证可以对用户进行鉴定以防止错用。

要解决这个问题，websocket 服务器和 Django 服务器将共享一个在彼此间传递消息的密钥。回顾一下在 sprint 中用于告知客户端 websocket 地址的 API 响应，这个响应允许 API 在 websocket 的 URL 上加入一个标记。当客户端打开 websocket 时，它可以使用共享的密钥来验证这个标记。

这里，我们在 *scrum/settings.py* 中新增一个设置用于跟踪 API 和 websocket 服务器之间共享的密钥。

```
...
WATERCOOLER_SERVER = os.environ.get('WATERCOOLER_SERVER', 'localhost:8080')

WATERCOOLER_SECURE = bool(os.environ.get('WATERCOOLER_SECURE', ''))

WATERCOOLER_SECRET = os.environ.get('WATERCOOLER_SECRET',
    'pTyz1dzMeVUGrbOSu4QXsP984qTlvQRHpFnnlHuH')
```

和 SECRET_KEY 设置一样，也应该是一个随机生成的字符串。上面的字符串只用于举例中。

SprintSerializer 可以使用这个密钥为 websocket 生成标记，并将其加入到响应中，如下面 *board/serializers.py* 所示。

```
...
from django.conf import settings
from django.contrib.auth import get_user_model
from django.core.signing import TimestampSigner                          ❶
...
class SprintSerializer(serializers.ModelSerializer):
...
    def get_links(self, obj):
        request = self.context['request']
        signer = TimestampSigner(settings.WATERCOOLER_SECRET)            ❷
        channel = signer.sign(obj.pk)
        return {
            'self': reverse('sprint-detail',
                kwargs={'pk': obj.pk}, request=request),
            'tasks': reverse('task-list',
                request=request) + '?sprint={}'.format(obj.pk),
            'channel': '{proto}://{server}/socket?channel={channel}'.format(
```

```
                proto='wss' if settings.WATERCOOLER_SECURE else 'ws',
                server=settings.WATERCOOLER_SERVER,
                channel=channel,
            ),
        } ❸
```

❶❷ 使用 Django 的密码签名工具来生成频道以及共享 websocket 密钥。

❸ 用于 sprint 通信的频道依旧使用主 pk 作为标识符。

`TimestampSigner` 用于设置频道名称，防止它在客户端被篡改。因为它携带时间标签，所以这个标记只会在短时间内有效，即使被别人获取也能防止被再次使用。这里我们更新了 URL 的格式，所以也要更新 Tornado 服务器来适应这个变化。不过，由于客户端直接使用来自 API 的 URL，所以该更新不需要对 JS 客户端做任何改变。

可以在 *https://docs.djangoproject.com/en/1.7/topics/signing/* 获取更多有关 Django 签名工具的信息。

watercooler.py 中的 Tornado 服务器需要更新来处理这个变化。它要从查询参数中获取频道并验证签名。同时应用程序也要用这个共享密钥进行配置。

```
import json
import logging
import os
...

from django.core.signing import TimestampSigner, BadSignature, SignatureExpired ❶
from redis import Redis
...

class SprintHandler(WebSocketHandler):
    """Handles real-time updates to the board."""

    def check_origin(self, origin):
        allowed = super().check_origin(origin)
        parsed = urlparse(origin.lower())
        matched = any(parsed.netloc == host for host in options.allowed_hosts)
        return options.debug or allowed or matched

    def open(self):                                                    ❷
        """Subscribe to sprint updates on a new connection."""
        self.sprint = None
        channel = self.get_argument('channel', None)
        if not channel:
            self.close()
        else:
            try:
```

```
                self.sprint = self.application.signer.unsign(                    ❸
                    channel, max_age=60 * 30)
            except (BadSignature, SignatureExpired):
                self.close()
            else:
                self.uid = uuid.uuid4().hex
                self.application.add_subscriber(self.sprint, self)

    def on_message(self, message):
        """Broadcast updates to other interested clients."""
        if self.sprint is not None:                                              ❹
            self.application.broadcast(message, channel=self.sprint, sender=self)

    def on_close(self):
        """Remove subscription."""
        if self.sprint is not None:                                              ❺
            self.application.remove_subscriber(self.sprint, self)
...
class ScrumApplication(Application):

    def __init__(self, **kwargs):
        routes = [
            (r'/socket', SprintHandler),                                         ❻
            (r'/(?P<model>task|sprint|user)/(?P<pk>[0-9]+)', UpdateHandler),
        ]
        super().__init__(routes, **kwargs)
        self.subscriber = RedisSubscriber(Client())
        self.publisher = Redis()
        self._key = os.environ.get('WATERCOOLER_SECRET',                         ❼
            'pTyz1dzMeVUGrbOSu4QXsP984qTlvQRHpFnnlHuH')
        self.signer = TimestampSigner(self._key)
...
```

❶❼ 创建一个新的 TimestampSigner，并在创建时把它加入全局应用中。TimestampSigner 从操作系统环境中获取共享密钥。该密钥必须要与 Django 应用使用的密钥相一致。

❷ 频道的名称不是从 URL 地址中来获取，而是从 channel 参数中获取。如果没有给出频道，则会立即关闭连接。

❸ 频道的值来源于签过名的值。如果签名验证失败或已经失效，则会关闭连接。

❻ URL 被更新成一个更加通用的 */socket/* 形式，因为现在频道名称是通过查询字符串来传递的。

❹❺ 更新 on_message 和 on_close 来处理 sprint 为 None 的情况。

当调用 open 方法时，已经接受连接，并且已经执行了客户端的 onopen 回调。此时会留下一个小的窗口期，以便一个新打开的客户端去尝试在频道验证签名之前发送信息。我们通过将 self.sprint 设置为 None，并在广播消息或添加、删除订阅之前检查设置是否正确来解决这个问题。

目前标记会在 30 分钟后过期，理想状况下这个时间可以更短些。但是由于标记是在 sprint 被从主页中获取后生成的，这在它被用于 sprint 详细页面之前可能会存在长时间的延迟。我们可以通过创建另一个 API 调用，恰巧在使用它之前即时生成该频道的 URL 来改善这种现象。这样，就可以缩短时间延迟。

这里的 Django 是作为一个库而非框架来使用的，我们只用到了它的签名工具。使用如 Tornado 一类的 Python 框架来处理 websocket 是代码共享的一个优点，它保证了实现的有条不紊。

也可以使用如 itsdangerours 之类通用的签名库来替代。

更新端点还需要被保护，来确保只有 Django 应用才能推送更新。我们将在接下来解决这个问题，并对其他有关 UpdateHandler 如何处理更新进行优化。

更好的更新

目前当用户编辑一项任务时，不管是内联编辑还是将它拖曳到一个新的状态栏上，都会将更新提交给 API。然后 Django 服务器会通知 Tornado 服务器，告诉所有客户出现了更新。当浏览器客户端得知有更新时，会从 API 获取任务信息。对于最初提交更新的用户来说，这是一条浪费的 API 调用。其他连接的用户都使用同样的 API 请求来获取相同的信息。更加高效的做法是在一开始通过 websocket 将信息发送给其他客户端。

需要对应用的所有三个部分进行更新。首先，需要更新 Django 服务器，将当前模型状态发送给 Tornado 服务器。回顾一下，这项工作是由 *board/views.py* 中的 UpdateHookMixin 实现的。

```
...
from rest_framework import authentication, filters, permissions, viewsets
from rest_framework.renderers import JSONRenderer
...
class UpdateHookMixin(object):
    """Mixin class to send update information to the websocket server."""

...
    def _send_hook_request(self, obj, method):
        url = self._build_hook_url(obj)
        if method in ('POST', 'PUT'):                                    ❶
            # Build the body
            serializer = self.get_serializer(obj)
            renderer = JSONRenderer()
```

```
            context = {'request': self.request}
            body = renderer.render(serializer.data, renderer_context=context)
        else:
            body = None
        headers = {
            'content-type': 'application/json',
        }
        try:
            response = requests.request(method, url,
                data=body, timeout=0.5, headers=headers)                    ❷
            response.raise_for_status()
                except requests.exceptions.ConnectionError:
                    # Host could not be resolved or the connection was refused
                    pass
                except requests.exceptions.Timeout:
                    # Request timed out
                    pass
                except requests.exceptions.RequestException:
                    # Server responded with 4XX or 5XX status code
                    pass
...
```

❶ 在新增或更新任务时，将当前模型序列化成 API 使用的相同格式，并作为 JSON 输出。

❷ 在这里把之前构造的消息体传递给 Tornado 的更新钩子。

之前，Tornado 服务器不会从请求主体中获取任何信息。我们现在需要更新 *watercooler.py* 中的 `UpdateHandler` 来将这个数据发送个客户端。

```
...
class UpdateHandler(RequestHandler):
    """Handle updates from the Django application."""

...
    def _broadcast(self, model, pk, action):
        try:
            body = json.loads(self.request.body.decode('utf-8'))            ❶
        except ValueError:
            body = None
        message = json.dumps({
            'model': model,
            'id': pk,
            'action': action,
            'body': body,                                                   ❷
        })
        self.application.broadcast(message)
        self.write("Ok")
...
```

❶ 如果可行，将请求的主体作为 JSON 进行解码。

❷ 解析后的主体数据会使用 websocket 广播给客户端。

最后我们需要在 *board/static/board/js/views.js* 中更新 `task:add` 和 `task:update` 处理器，以便在给出这种主体的情况下使用它。

```
...
var SprintView = TemplateView.extend({
...
    connectSocket: function () {
        ...
        this.socket.on('task:add', function (task, result) {
            var model
            if (result.body) {
                model = app.tasks.add([result.body]);                    ❶
            } else {
                model = app.tasks.push({id: task});
                model.fetch();
            }
        }, this);
        this.socket.on('task:update', function (task, result) {
            var model = app.tasks.get(task);
            if (model) {
                if (result.body) {
                    model.set(result.body);                              ❷
                } else {
                   model.fetch();
                }
            }
        }, this);
        ...
...
```

❶ 新的任务将直接从 websocket 数据中添加，而非从 API 中获取。

❷ 类似地，任务数据从 websocket 的数据中更新，而不是从另一个进行的更新中更新。

正如前面所提到的，Tornado 的 Web 钩子存在的一个关键问题是没有东西能保护这个端点的安全。任何知道服务器地址的人都可以向所有连接的客户端推送更新。这种操作不会直接影响到服务器，除非某个客户端接受了错误的模型数据并在之后进行了保存。在下一节中我们将会解决这个安全性问题。

安全的更新

除了应用之外，还有许多添加安全层的方法。HTTP 协议（如 Nginx）可以放在服务器的前端，强制进行基本的、摘要的或是客户端权限验证。我们需要把权限验证加入到 Django 应用生成的请求中。在两个服务器使用共享的密钥时，还可能要包含一些使用现有密钥的验证信息。以和频道验证签名相同的方式，对 Web 挂钩 (webhook) 请求进行签名来确保它们来自受信任的端点，并且在传输过程中未被修改。

这个消息验证并非是服务器间通信加密的替代品。在生产环境中，Tornado 服务器通过 HTTPS 来获取更新。

UpdateHookMixin 需要在发送请求时发送一个该请求的签名，UpdateHandler 要在广播更新之前验证该签名。修改后的值要包含关于模型类型、ID 以及操作（添加、更新、删除）和其他有关模型状态在内的信息。下面是这个功能在 *board/views.py* 中的基本情况。

```
import hashlib                                                          ❶
import requests
...
from django.conf import settings
from django.core.signing import TimestampSigner                         ❷
from django.contrib.auth import get_user_model
...
class UpdateHookMixin(object):
...
    def _send_hook_request(self, obj, method):
        url = self._build_hook_url(obj)
        if method in ('post', 'put'):
            # Build the body
            serializer = self.get_serializer(obj)
            renderer = JSONRenderer()
            context = {'request': self.request}
            body = renderer.render(serializer.data, renderer_context=context)
        else:
            body = None
        headers = {
            'content-type': 'application/json',
            'X-Signature': self._build_hook_signature(method, url, body)    ❸
        }
        try:
            response = requests.request(method, url,
                data=body, timeout=0.5, headers=headers)
            response.raise_for_status()
        except requests.exceptions.ConnectionError:
            # Host could not be resolved or the connection was refused
            pass
        except requests.exceptions.Timeout:
            # Request timed out
            pass
        except requests.exceptions.RequestException:
            # Server responded with 4XX or 5XX status code
            pass

def _build_hook_signature(self, method, url, body):                     ❹
    signer = TimestampSigner(settings.WATERCOOLER_SECRET)
    value = '{method}:{url}:{body}'.format(
        method=method.lower(),
        url=url,
```

```
            body=hashlib.sha256(body or b'').hexdigest()
        )
            return signer.sign(value)
...
```

❶❷ 使用 TimestampSigner 为基于请求方法、URL 和主体的请求生成一个签名。

❹

❸ 通过 X-Signature 头将签名传给 Tornado 端点。

签名的内容取决于当前 URL、请求方法以及请求主体。这就防止了把一个有效的请求签名用在另一个假冒请求上。这里只使用主体的哈希值以避免签名过长。再有，将签名打上时间戳以防止窗口过期之后重新展示该请求，这样将使得签名保持短小。

这个签名来源于如何在 Amazon Web Services 的 API 中进行请求签名的一个简化版本。有关签名过程的更多信息，请参看：*http://docs.aws.amazon.com/general/latest/gr/signature-version-4.html*。

要验证这个签名，UpdateHandler 需要在 *watercooler.py* 中更新。验证签名既需要检查签名的正确性，又需要对包含匹配当前请求信息的签名内容进行检查。

```
import hashlib                                                          ❶
import json
...
from django.core.signing import TimestampSigner, BadSignature, SignatureExpired
from django.utils.crypto import constant_time_compare                   ❷
...
from tornado.web import Application, RequestHandler, HTTPError          ❸
...
class UpdateHandler(RequestHandler):
...
    def _broadcast(self, model, pk, action):
        signature = self.request.headers.get('X-Signature', None)       ❹
        if not signature:
            raise HTTPError(400)
        try:
            result = self.application.signer.unsign(signature, max_age=60 * 1)  ❺
        except (BadSignature, SignatureExpired):
            raise HTTPError(400)
        else:
            expected = '{method}:{url}:{body}'.format(                  ❻
                method=self.request.method.lower(),
                url=self.request.full_url(),
                body=hashlib.sha256(self.request.body).hexdigest(),
            )
            if not constant_time_compare(result, expected):
                raise HTTPError(400)
```

```
            try:
                body = json.loads(self.request.body.decode('utf-8'))
            except ValueError:
                body = None
            message = json.dumps({
                'model': model,
                'id': pk,
                'action': action,
                'body': body,
            })
            self.application.broadcast(message)
            self.write("Ok")
    ...
```

❶❷ 需要新引入 hashlib、constant_time_compare 以及 HTTPError。

❸

❹ 从头信息中签名获取，如果未找到签名，则返回 400 响应。

❺ 验证签名有效且未过期。该签名将在 1 分钟后过期。

❻ 未签名的值与其期望值做比较。

现在签名确保更新钩子来源于一个可以使用共享密钥的服务器。另外，过期时间被设置得很短以防止同一个请求被重现。但由于时间非常短，所以必须保证两个服务器的时钟同步，否则这一步就会失败。

最终的 websocket 服务器

创建这个 websocket 服务器究竟需要多少工作量？让我们来看看 *watercooler.py* 中完整的应用代码。

```
import hashlib
import json
import logging
import os
import signal
import time
import uuid

from urllib.parse import urlparse

from django.core.signing import TimestampSigner, BadSignature, SignatureExpired
from django.utils.crypto import constant_time_compare
from redis import Redis
from tornado.httpserver import HTTPServer
from tornado.ioloop import IOLoop
from tornado.options import define, parse_command_line, options
from tornado.web import Application, RequestHandler, HTTPError
```

```
from tornado.websocket import WebSocketHandler
from tornadoredis import Client
from tornadoredis.pubsub import BaseSubscriber


define('debug', default=False, type=bool, help='Run in debug mode')
define('port', default=8080, type=int, help='Server port')
define('allowed_hosts', default="localhost:8080", multiple=True,
    help='Allowed hosts for cross domain connections')


class RedisSubscriber(BaseSubscriber):

    def on_message(self, msg):
        """Handle new message on the Redis channel."""
        if msg and msg.kind == 'message':
            try:
                message = json.loads(msg.body)
                sender = message['sender']
                message = message['message']
            except (ValueError, KeyError):
                message = msg.body
                sender = None
            subscribers = list(self.subscribers[msg.channel].keys())
            for subscriber in subscribers:
                if sender is None or sender != subscriber.uid:
                    try:
                        subscriber.write_message(message)
                    except tornado.websocket.WebSocketClosedError:
                        # Remove dead peer
                        self.unsubscribe(msg.channel, subscriber)
       super().on_message(msg)


class SprintHandler(WebSocketHandler):
    """Handles real-time updates to the board."""

    def check_origin(self, origin):
        allowed = super().check_origin(origin)
        parsed = urlparse(origin.lower())
        matched = any(parsed.netloc == host for host in options.allowed_hosts)
        return options.debug or allowed or matched

    def open(self):
        """Subscribe to sprint updates on a new connection."""
        self.sprint = None
        channel = self.get_argument('channel', None)
        if not channel:
            self.close()
        else:
            try:
                self.sprint = self.application.signer.unsign(
                    channel, max_age=60 * 30)
            except (BadSignature, SignatureExpired):
                self.close()
```

```
            else:
                self.uid = uuid.uuid4().hex
                self.application.add_subscriber(self.sprint, self)

    def on_message(self, message):
        """Broadcast updates to other interested clients."""
        if self.sprint is not None:
            self.application.broadcast(message, channel=self.sprint, sender=self)

    def on_close(self):
        """Remove subscription."""
        if self.sprint is not None:
            self.application.remove_subscriber(self.sprint, self)


class UpdateHandler(RequestHandler):
    """Handle updates from the Django application."""

    def post(self, model, pk):
        self._broadcast(model, pk, 'add')

    def put(self, model, pk):
        self._broadcast(model, pk, 'update')

    def delete(self, model, pk):
        self._broadcast(model, pk, 'remove')

    def _broadcast(self, model, pk, action):
        signature = self.request.headers.get('X-Signature', None)
        if not signature:
            raise HTTPError(400)
        try:
            result = self.application.signer.unsign(signature, max_age=60 * 1)
        except (BadSignature, SignatureExpired):
            raise HTTPError(400)
        else:
            expected = '{method}:{url}:{body}'.format(
                method=self.request.method.lower(),
                url=self.request.full_url(),
                body=hashlib.sha256(self.request.body).hexdigest(),
            )
            if not constant_time_compare(result, expected):
                raise HTTPError(400)
        try:
            body = json.loads(self.request.body.decode('utf-8'))
        except ValueError:
            body = None
        message = json.dumps({
            'model': model,
            'id': pk,
            'action': action,
            'body': body,
        })
        self.application.broadcast(message)
        self.write("Ok")
```

```
class ScrumApplication(Application):

    def __init__(self, **kwargs):
        routes = [
            (r'/socket', SprintHandler),
            (r'/(?P<model>task|sprint|user)/(?P<pk>[0-9]+)', UpdateHandler),
        ]
        super().__init__(routes, **kwargs)
        self.subscriber = RedisSubscriber(Client())
        self.publisher = Redis()
        self._key = os.environ.get('WATERCOOLER_SECRET',
            'pTyz1dzMeVUGrbOSu4QXsP984qTlvQRHpFnnlHuH')
        self.signer = TimestampSigner(self._key)

    def add_subscriber(self, channel, subscriber):
        self.subscriber.subscribe(['all', channel], subscriber)

    def remove_subscriber(self, channel, subscriber):
        self.subscriber.unsubscribe(channel, subscriber)
        self.subscriber.unsubscribe('all', subscriber)

    def broadcast(self, message, channel=None, sender=None):
        channel = 'all' if channel is None else channel
        message = json.dumps({
            'sender': sender and sender.uid,
            'message': message
        })
        self.publisher.publish(channel, message)


    def shutdown(server):
        ioloop = IOLoop.instance()
        logging.info('Stopping server.')
        server.stop()

    def finalize():
        ioloop.stop()
        logging.info('Stopped.')

    ioloop.add_timeout(time.time() + 1.5, finalize)
    if __name__ == "__main__":
        parse_command_line()
        application = ScrumApplication(debug=options.debug)
        server = HTTPServer(application)
        server.listen(options.port)
        signal.signal(signal.SIGINT, lambda sig, frame: shutdown(server))
        logging.info('Starting server on localhost:{}'.format(options.port))
        IOLoop.instance().start()
```

不到 200 行的代码就创建了一个既处理从客户端到客户端广播事件，又处理从服务器到客户端广播事件的服务器。我们的应用使用了现有的 Django 工具来验证连接的客户端和收到的请求。两个服务器之间共享一个密钥。

这个实时 Django 的轻量级方法包含许多活动的块，是在之前的几个章节中构建起来的。这里用到了 Django 构建的一个 REST 式的 API、一个 Backbone.js 制作的单页面应用，以及一个 Tornado 构建的 websocket 服务器。每个块都有它们各自的作用，每项服务之间都有定义良好的通信。Django 服务器与前端应用进行交互，尽可能小地通过模板以及 REST 的 API 传递配置信息。Django 服务器通过设置的 HTTP 端点给 Tornado 推送更新。客户端使用一个不了解 Django 的简单消息模式与 websocket 进行交互。所谓轻量级 Django 是将 Django 想象成一大块拼图中的一块，以及学习如何让 Django 与其他可以组合使用的服务器进行交互。

作者介绍

Julia Elman 既是设计师，又是开发者，还是北加利福尼亚技术教育的倡导者。她自2002年以来一直从事的主要工作就是网络技能教育。她的创造性天赋使她2007年在Hallmark Cards公司获得了工作，在那里她做过像产品（RED）开发活动和Hallmark网站重新设计等项目。从那时起她就潜心于Django工作，在堪萨斯劳伦斯的世界在线做过初级设计师和开发员。早在2013年，她就开始协助有关《Girl Develop It》一书中本地一章的撰写，并组织了850多会员学习计算机编程。她还帮助组织了2013年少年技术活动。在那里，有20个当地的少年用一天的时间学习了Python编程。

Mark Lavin 从2006年就开始从事软件开发了，第一份工作是在华尔街。他现在是美国最大的专业Django公司Caktus的技术主管。Mark Lavin是Django协会的积极分子，经常在研讨会上进行演讲，为开源项目贡献成果，并回答有关栈溢出的问题。在闲暇之余，他喜欢自己酿啤酒、跑步、参加铁人三项赛，以及与妻子、两个女儿待在北加利福尼亚的家里。

封面介绍

本书封面上的动物是马鞭草蜂鸟（小吸蜜蜂鸟），原产于牙买加、海地和多米尼加共和国。这些土产的蜂鸟很常见，它们喜好亚热带和热带的低地丛林。

虽然不具备其他种类蜂鸟光亮耀眼的羽毛，但马鞭草蜂鸟却是世界上第二小的鸟类，它仅大于古巴的蜂雀。马鞭草蜂鸟的长度大约有2.4英寸，包括它尖尖的喙，其重量仅为0.07～0.08盎司。这些鸟主要以花朵中的花蜜为食，用它们可伸缩的舌头以每秒钟13次的速度甜食花蜜。

蜂鸟是独居动物，只在繁殖期才会聚居。交配完，雄鸟立即离开，留下雌鸟构建巢穴和抚育幼鸟。蜂鸟产下的蛋是最小的蛋，只有1/3英寸长，平均重0.375克。鸟巢仅为胡桃壳大小的1/2。

尽管它很微小，马鞭草蜂鸟产生的响亮刺耳的声音常常说明它的存在，因为这种鸟大小很难被发现。当它们的头由一侧转到另一侧时会发出响亮的声音，如歌唱一般。

O'Reilly 书籍封面的很多动物是濒危动物，它们对世界都很重要。要学习如何为它们提供帮助，请登录网站 animals.oreilly.com。

封面图像来源于《Wood's Natural History》。

O'REILLY®

轻量级Django

怎样使用Django构架实现客户端的交互和实时特性与网络应用相融合？本书通过一系列简单小巧的应用开发项目，展示了熟练的Django开发者将REST API、WebSockets和Backbone.js这样的客户端MVC构架加入到新建或已有的项目中的方法。

通过选取用于创建轻量级应用组件的形式来理解进行Django解耦设计的方法。通过本书的学习，你将具备创建单页面响应实时交互应用的能力。如果你熟练掌握了Python和JavaScript，则可以开始编写应用程序了。

- 学习开始新建Django项目的轻量级方法。
- 将可重用应用分解成与其他应用通信的更细小的服务。
- 创建静态、便捷的原型站点作为网站和应用的支撑平台。
- 使用Django Rest Framework构建REST API。
- 学习如何使用带有Backbone.js的MVC框架的Django。
- 在REST API平台上创建单页面网络应用。
- 将WebSockets和Tornado网络库与实时特性相融合。
- 在项目开发中使用本书的代码驱动实例。

"本书是超越传统应用和学习Django如何提高后端单页面网络应用能力的一项重大资源。"

——Aymeric Augustin
Django核心开发者，
oscaro.com的CTO

"我认为利用这种好的想法能够大大降低成为开发者的门槛，阅读这本书让我很兴奋！。"

——Barbara Shaurette
Python开发者，Cox Media Group

Julia Elman，一名前端的开发者和技术教育的倡导者，2008年在World Online工作期间就开始了Django的学习。她是Girl Develop It RDU和PyLadies RDU组织的共同创办人，该组织帮助超过850名妇女学习了编程知识。

Mark Lavin是北加利福尼亚达勒姆Caktus咨询集团的技术主管。他是在华尔街进行衍生品定价工作数年后开始进行Python网络开发的。他负责对与Django开发相关的几个开源项目进行维护。

PYTHON/WEB DEVELOPMENT

O'Reilly Media, Inc.授权中国电力出版社出版

ISBN 978-7-5123-9396-7

9 787512 393967

定价：39.00元

2016