

目录

Growth In Action Django	5
准备工作和工具	5
深入浅出 Django	5
Django 简介	5
Django 应用架构	6
Django hello,world	8
安装 Django	8
创建项目	11
Django 后台	12
第一次提交	14
Django 创建博客应用	16
Tasking	16
创建 BlogpostAPP	17
生成 APP	17
创建 Model	17
配置 URL	20
创建 View	21
创建博客列表页	21
创建博客详情页	21
测试	22
测试首页	22
测试详情页	24
功能测试与持续集成	25
编写自动化测试	25
Selenium 与第一个 UI 测试	25

搭建持续集成	29
Jenkins 创建任务	32
创建 shell	33
更多功能	34
静态页面	35
安装 flatpages	35
创建模板	36
评论功能	38
Sitemap	40
站点地图介绍	41
创建首页的 Sitemap	43
创建静态页面的 Sitemap	45
创建博客的 Sitemap	46
提交到搜索引擎	47
前端框架	49
响应式设计	49
引入前端框架	50
页面美化	51
添加导航	51
添加标语	54
优化列表	54
添加 footer	55
API	56
博客列表	57
Django REST Framework	57
创建博客列表 API	58

测试 API	59
自动完成	61
搜索 API	62
页面实现	63
跨域支持	65
添加跨域支持	65
移动应用	66
hello,world	66
构建应用	71
博客列表页	72
列表页	72
详情页	75
Profile	79
Json Web Tokens	79
登录表单	80
Profile	82
创建博客	84
Mobile Web	87
移动设备处理	87
前后端分离	89
Riot.js	89
ReactiveJS 构建服务	89
创建博客列表页	90
博客详情页	92
添加导航	93
配置管理	95

local settings	95
--------------------------	----

Growth In Action Django

准备工作和工具

在开始写代码之前你需要保证你有一些 **Python** 基础，如果没有的话，请参阅其他相关书籍来一起学习。

并且你还需要在你的计算机上安装：

- **Python** 环境及其包管理工具 **pip**。
- **Firefox** 浏览器——用于运行功能测试。
- **Git** 版本控制器——用于代码版本控制。
- 一个开发工具。（**PS**: 在这里笔者使用的是 **PyCharm** 的社区版）

深入浅出 Django

Django 简介

Django 是一个高级的 **Python Web** 开发框架，它的目标是使得开发复杂的、数据库驱动的网站变得更加简单。

由于 **Django** 最初是被开发来用于管理劳伦斯出版集团旗下的一些以新闻内容为主的网站的。所以，我们可以发现在使用 **Django** 的很多网站里，都是用于作为 **CMS**（内容管理系统）来使用的。使用 **Django** 的一些比较知名的网站如下图所示：

Django 是一个 **MTV** 框架，其架构模板看上去与传统的 **MVC** 架构并没有太大的区别。其对比如下表所示：

传统的 MVC 架构	Django 架构
Model	Model(Data Access Logic)
View	Template(Presentation Logic)
View	View(Business Logic)
Controller	Django itself

在 **Django** 中 **View** 只用来描述你要看到的内容，**Template** 才是最后用于显示的内容。而在 **MVC** 架构中，这只相当于是 **View** 层。它的核心包含下面的四部分：

- 一个对象关系映射，作为数据模型和关系性数据库间的媒介（**Model** 层）；



图 1: 使用 Django 的网站

- 一个基于正则表达式的 URL 分发器（即 MVC 中的 Controller）；
- 一个用于处理 HTTP 请求的系统，含 web 模板系统 (View 层)；

其核心框架还包含：

- 一个轻量级的、独立的 Web 服务器，只用于开发和测试。
- 一个表单序列化及验证系统，用于将 HTML 表单转换成适用于数据库存储的数据。
- 一个缓存框架，并且可以从几种缓存方式中选择。
- 中间件支持，能对请求处理的各个阶段进行处理。
- 内置的分发系统允许应用程序中的组件采用预定义的信号进行相互间的通信。
- 一个序列化系统，能够生成或读取采用 XML 或 JSON 表示的 Django 模型实例。
- 一个用于扩展模板引擎的能力的系统。

Django 应用架构

Django 的每一个模块在内部都称之为 APP，在每个 APP 里都有自己的三层结构。如下图所示：

这样做不仅可以在开发的时候更容易理解系统，而且可以提高代码的可复用性——因为每一个 APP 都是独立的应用，在下次使用时我们只需要简单的复制和粘贴。

说了这么多，还不如从一个 hello,world 开始。

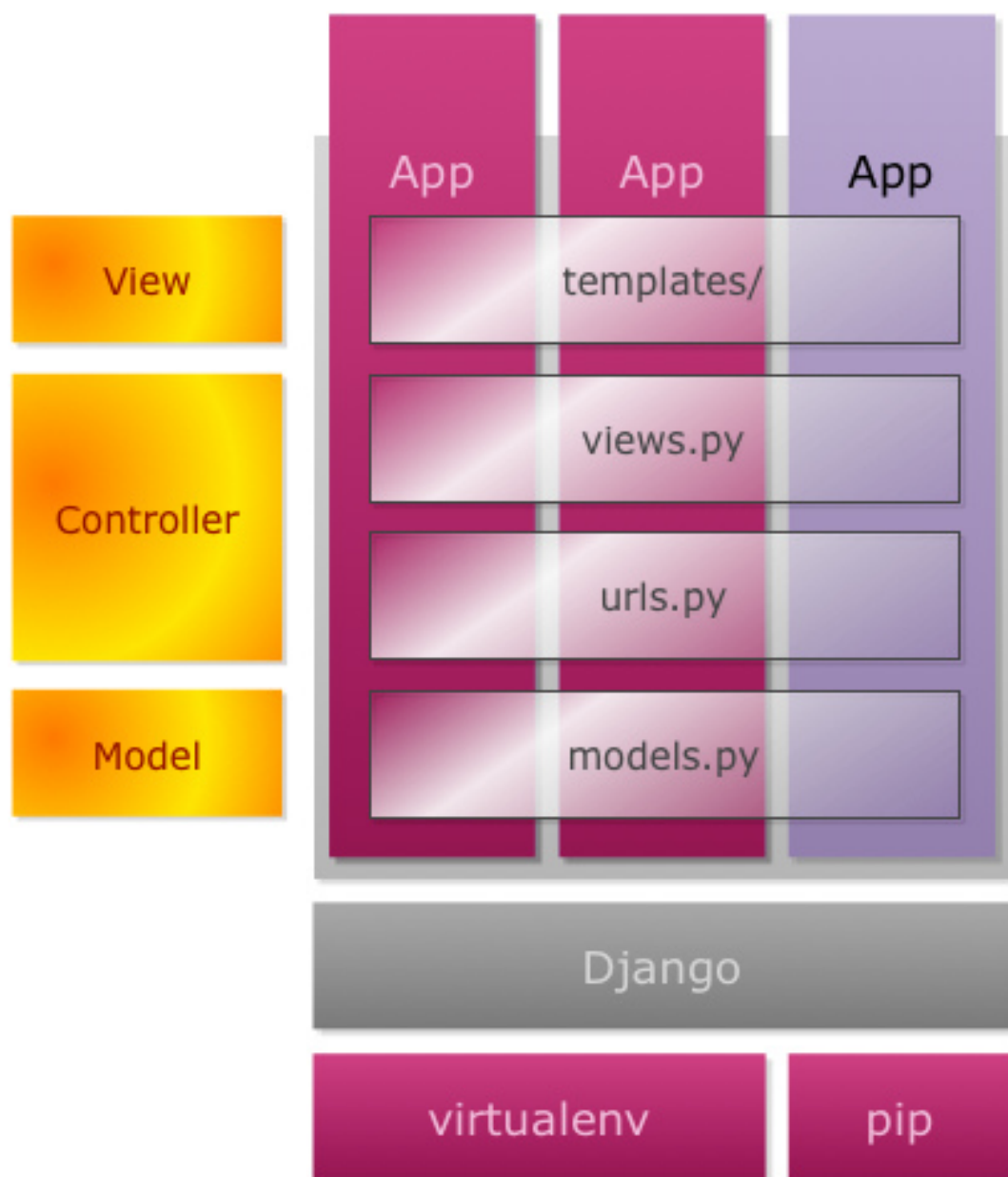


图 2: Django 应用架构

Django hello,world

安装 Django

安装 Django 之前，我们可以用 **virtualenv** 工具来创建一个虚拟的 Python 运行环境。环境问题是一个很复杂的问题，在我们使用 Python 的过程中，我们会不断地安装一些库，而这些库可能会有不同的版本。并且在安装 Python 库的过程中，我们会遇到权限问题——即我们需要超级用户的权限才能将库安装到系统的环境之下。随后在这个软件的生涯中，我们还需要保证这个项目所依赖的模块不会发生变动。而这些都是很棘手的一些事，这时候我们就需要创建一个虚拟的运行环境，而 **virtualenv** 就是这样的一个工具。

virtualenv 安装 Python 包我们需要用到 **pip** 命令，它是 Python 语言中的一个包管理工具。如果你没有安装的话，可以使用下面的命令来安装：

```
curl https://bootstrap.pypa.io/get-pip.py | python
```

在不同的 Python 环境中，我们可能需要使用不同的 **pip**，如下所示是笔者使用的 Python3 的 **pip** 命令 **pip3**

```
$ pip3 install virtualenv
```

如果是 Python2.7 的话，对应会有：

```
$ pip install virtualenv
```

需要注意的是这将会安装到 Python 所在的目录，如我的目录是：

```
$ /usr/local/bin/virtualenv
```

有的可能会是：

```
$ /usr/local/share/python3/virtualenv
```

在创建我们的这个虚拟环境之前，我们可以创建一个存储所有 **virtualenv** 的目录：

```
$ mkdir somewhere/virtualenvs
```


现在，我们就可以创建一个新的虚拟环境：

```
$ virtualenv somewhere/ virtualenvs/<project-name> -- no-site-packages
```

如果你想使用不同的 **Python** 版本的话，那么需要指定 **Python** 版本的路径

```
$ virtualenv --distribute -p /usr/local/bin/python3.3 somewhere/virtualenvs/<project-name>
```

通过到相应的目录下执行激活就可以使用这个虚拟环境了：

```
$ cd somewhere/virtualenvs/<project-name>/bin
$ source activate
```

停止使用只使用执行下面的命令即可：

```
$ deactivate
```

安装 **Django** 准备了这么久我们终于安装 **Django** 了，执行：

```
$ pip install django
```

那么我们将会开始下最新版本的 **Django**，如下所示：

```
Collecting django
```

```
Downloading Django-1.9.4-py2.py3-none-any.whl (6.6MB)
```

```
94% |██████████████████████████████████████████| 6.2MB 251kB/s eta 0:00:02
```

等下载完后，就会开始安装 **Django**。安装这完后，我们就可以使用 **Django** 自带的 **django-admin** 命令。**django-admin** 是 **Django** 自带的一个管理任务的命令行工具。

通过这个命令，我们不仅仅可以用它来创建项目、创建 **app**、运行服务、数据库迁移，还可以执行各种 **SQL** 工具等等。**django-admin** 用法如下：

```
$ django-admin <command> [options]
```

下面是 **django-admin** 自带的一些命令：

```
[django]
  check
  compilemessages
  createcachetable
  dbshell
  diffsettings
  dumpdata
  flush
  inspectdb
  loaddata
  makemessages
  makemigrations
  migrate
  runfcgi
  runserver
  shell
  sql
  sqlall
  sqlclear
  sqlcustom
  sqldropindexes
  sqlflush
  sqlindexes
  sqlinitialdata
  sqlmigrate
  sqlsequencereset
  squashmigrations
  startapp
  startproject
  syncdb
  test
  testserver
  validate
```

现在，让我们来看看这个强大的工具。

创建项目

在这些命令中 **startproject** 可以用于创建项目，在这里我们的项目名是 **blog**，那么我们的命令如下：

```
$ django-admin startproject blog
```

这个命令将创建下面的文件内容，而这些都是 **Django** 项目的一些必须文件。

```
.
└── blog
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── manage.py
```

blog 目录对应的就是 **blog** 这个项目，将会放置这个项目的一些相关配置：

1. **settings.py** 包含了这个项目的配置。如数据库环境、启用的插件等等。
2. **urls.py** 即 **URL Dispatcher** 的配置，指明了某个 **URL** 应该指向某个函数来处理。
3. **wsgi.py** 用于部署。**WSGI**（**Python Web Server Gateway Interface**，**Web** 服务器网关接口）是为 **Python** 语言定义的 **Web** 服务器和 **Web** 应用程序或框架之间的一种简单而通用的接口。
4. **init.py** 指明了这是一个 **Python** 模块。

manage.py 会在每个 **Django** 项目中自动生成，它可以和 **django-admin** 做类似的事。如我们可以用 **manage.py** 来启动测试环境的服务器：

```
$ python manage.py runserver
```

```
Performing system checks...
```

```
System check identified no issues (0 silenced).
```

```
You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.
```

```
March 24, 2016 - 03:07:34
```

```
Django version 1.9.4, using settings 'blog.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
Not Found: /
[24/Mar/2016 03:07:35] "GET / HTTP/1.1" 200 1767
Not Found: /favicon.ico
[24/Mar/2016 03:07:36] "GET /favicon.ico HTTP/1.1" 404 1934
```

现在，我们只需要在浏览器中打开<http://127.0.0.1:8000/>，便可以访问我们的应用程序。

Django 后台

Django 很适合 CMS 的另外一个原因，就是它自带了一个后台管理系统。为了启用这个后台管理系统，我们需要配置我们的数据库，并创建相应的超级用户。如下所示的是 `settings.py` 中的默认数据库配置：

```
# Database
# https://docs.djangoproject.com/en/1.7/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

上面的配置中我们使用的是 **SQLite3** 作为数据库，并使用了当前目录下的 `db.sqlite3` 作为数据库文件。**Django** 内建支持下面的一些数据库：

```
'django.db.backends.postgresql_psycopg2'
'django.db.backends.mysql'
'django.db.backends.sqlite3'
'django.db.backends.oracle'
```

如果我们想使用别的数据库，那么可以在网上寻找的解决方案，如用于支持使用 MongoDB 的 `django-nonrel` 项目。不同的数据库有不同的配置，如下所示的是使用 PostgreSQL 的配置。

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'mydatabase',
        'USER': 'mydatabaseuser',
        'PASSWORD': 'mypassword',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

接着，我们就可以运行数据库迁移，只需要运行相应的脚本即可：

\$ python manage.py migrate

Operations to perform:

Apply all migrations: sessions, admin, auth, contenttypes

Running migrations:

Rendering model states... DONE

Applying contenttypes.0001_initial... OK

Applying auth.0001_initial... OK

Applying admin.0001_initial... OK

Applying admin.0002_logentry_remove_auto_add... OK

Applying contenttypes.0002_remove_content_type_name... OK

Applying auth.0002_alter_permission_name_max_length... OK

Applying auth.0003_alter_user_email_max_length... OK

Applying auth.0004_alter_user_username_opts... OK

Applying auth.0005_alter_user_last_login_null... OK

Applying auth.0006_require_contenttypes_0002... OK

Applying auth.0007_alter_validators_add_error_messages... OK

Applying sessions.0001_initial... OK

(growth-django)

在上面的过程中，我们会创建相应的数据库模型，并依据迁移脚本来创建一些相应的数据，如默认的配置等等。

最后，我们可以创建一个相应的超级用户来登陆后台。

\$ python manage.py createsuperuser

```
Username (leave blank to use 'fdhuang'): root
Email address: h@phodal.com
Password:
Password (again):
Superuser created successfully.
```

输入相应的用户名和密码，即可完成创建。然后访问 <http://127.0.0.1:8000/admin>，输入上面的用户名和密码就可以来到后台：

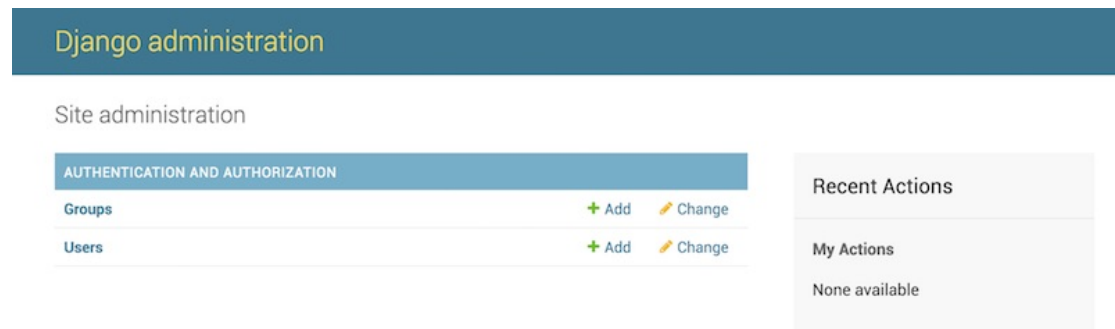


图 3: Django 后台

第一次提交

在创建完应用后，我们就可以进行第一次提交，通常这样的提交的提交信息 (**commit message**) 是 `init project`。如果在那之前，你没有执行 `git init` 来初始化 **git** 的话，那么我们就需要去执行这个命令。

```
git init
```

它将返回类似于下面的结果

```
Initialized empty Git repository in /Users/fdhuang/test/helloworld/.git/
```

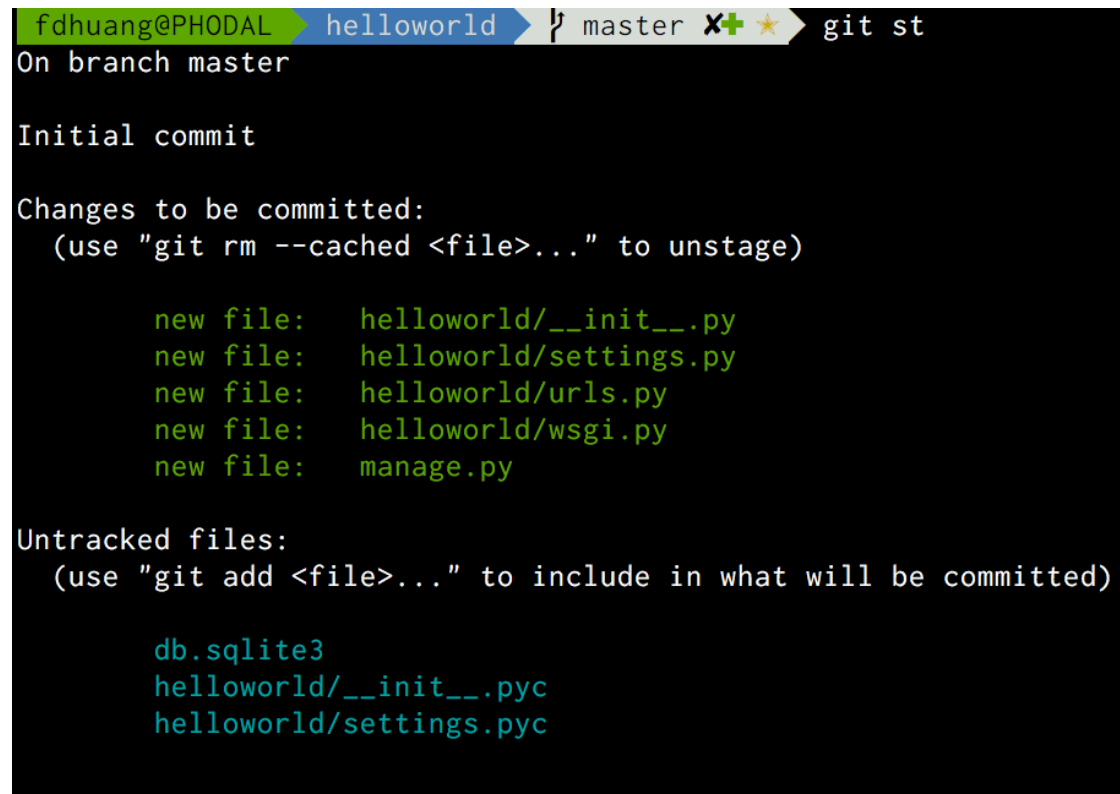
即初始化了一个空的 **Git** 项目，然后我们就可以执行 `add` 来添加上面的内容：

```
git add .
```

需要注意的是上在的数据库文件不应该添加到项目里，所以我们应该执行 **reset** 命令来重置这个状态：

```
git reset db.sqlite3
```

这时我们会将其变成下面的状态:



```
fdhuang@PHODAL > helloworld > master X+ ★ > git st
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   helloworld/__init__.py
        new file:   helloworld/settings.py
        new file:   helloworld/urls.py
        new file:   helloworld/wsgi.py
        new file:   manage.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        db.sqlite3
        helloworld/__init__.pyc
        helloworld/settings.pyc
```

图 4: 第一次提交前的 reset

上面的绿色文件代码这几个文件都被添加了进行, 蓝色则代表未添加的文件。为了避免手误产生一些问题, 我们需要添加一个名为.gitignore 文件用于将一些文件名入忽略名单, 如下是常用的 python 项目的.gitignore 文件中的内容:

```
*.pyc
*.db
*.sqlite3
```

当我们添加完这个文件完, git 就会识别到这个文件, 并忽略原来的那些文件, 如下图所示:

我们只需要添加这个文件即可:

```
git add .gitignore
```

如果你之前已经不小心添加了一些不应该添加的文件, 那么可以执行下面的命令来重置其状态:

```
fdhuang@PHODAL helloworld master X+★ git st
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   helloworld/__init__.py
    new file:   helloworld/settings.py
    new file:   helloworld/urls.py
    new file:   helloworld/wsgi.py
    new file:   manage.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore
```

图 5: 添加完 gitignore 文件后的效果

```
git reset .
```

然后再执行添加命令。

最后，我们就可以在本地提交我们的代码了：

```
git commit -m "init project"
```

如果你是将代码托管在 **GitHub** 上的话，那么你就可以执行 `git push` 来将代码提交到服务器上。

Django 创建博客应用

Tasking

在我们不了解 **Django** 的时候，要对这样一个任务进行 **Tasking**，有点困难。不过，我们还是可以简单地看看是应该如何去做：

- 生成 **APP**。对于大部分主流的 **Web** 框架来说，它们都可以手动地生成一些脚手架，如 **Ruby** 语言中的 **Ruby On Rails**、**Node.js** 中的 **Express** 等等。
- 创建对应的 **Model**，即其在数据库中存储的模型与我们在代码中要使用的模型。

- 创建程序对应的 **View**，用于处理数据。
- 创建程序的 **Template**，用于显示数据。
- 编写测试来保证功能。

对于其他应用来说也是差不多的。

创建 **BlogpostAPP**

生成 **APP**

现在我们可以开始创建我们的 **APP**，使用下面的代码来创建：

```
$ django-admin startapp blogpost
```

会在 **blogpost** 目录下，生成下面的文件：

```
.
├── __init__.py
├── admin.py
├── apps.py
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

创建 **Model**

现在，我们需要来创建博客的 **Model** 即可。对于一篇基本的博客来说，它会包含下面的几部分内容：

- 标题
- 作者
- 链接（中文更需要一个好的链接）
- 内容
- 发布日期

我们就可以按照上面的内容来创建我们的 **Blogpost model**：

```
from django.db import models
from django.db.models import permalink

class Blogpost(models.Model):
    title = models.CharField(max_length=100, unique=True)
    author = models.CharField(max_length=100, unique=True)
    slug = models.SlugField(max_length=100, unique=True)
    body = models.TextField()
    posted = models.DateField(db_index=True, auto_now_add=True)

    def __unicode__(self):
        return '%s' % self.title

    @permalink
    def get_absolute_url(self):
        return ('view_blog_post', None, { 'slug': self.slug })
```

上面的 `get_absolute_url` 方法就是用于返回博客的链接。之所以使用手动而不是自动生成，是因为自动生成不靠谱，而且不利

然后在 Admin 注册这个 Model

```
from django.contrib import admin
from blogpost.models import Blogpost

class BlogpostAdmin(admin.ModelAdmin):
    exclude = ['posted']
    prepopulated_fields = {'slug': ('title',)}

admin.site.register(Blogpost, BlogpostAdmin)
```

接着进入后台，我们就可以看到 **BLOGPOST** 的一栏里，就可以对其进行相关的操作。

点击 **Blogpost** 的 **Add** 后，我们会进入如下的添加博客界面：

实际上，这样做的意义是将删除 (Delete)、修改 (Update)、添加 (Create) 这些内容将给用户后台来做，当然它也不需要 View/Template 层来做。在我们的 Template 层



图 6: Django 后台界面

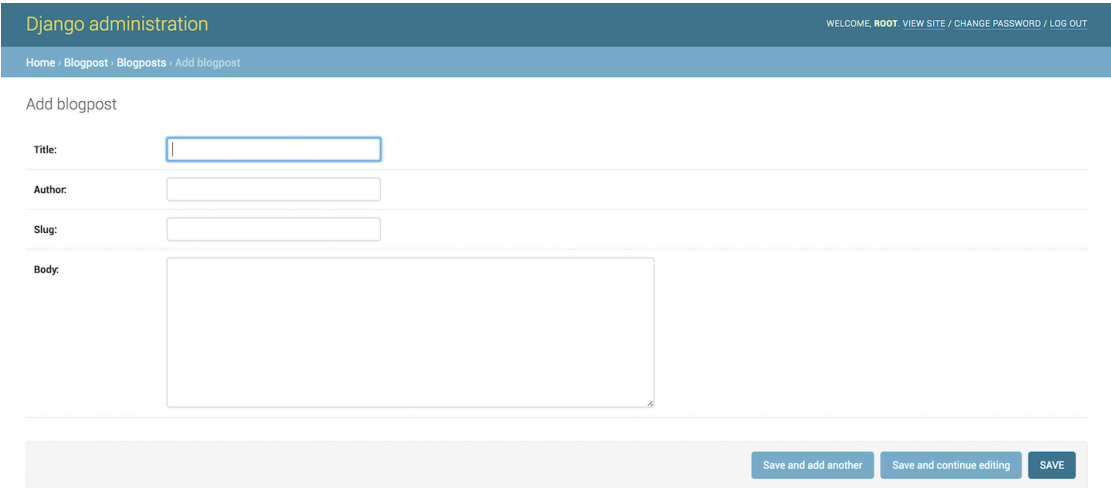


图 7: Django 添加博客

中，我们只需要关心如何来显示这些数据。

现在，我们可以执行一次新的代码提交——因为现在的代码可以正常工作。这样出现问题时，我们就可以即时的返回上一版本的代码。

```
git add .
git commit -m "create blogpost model"
```

然后再进行下一步地操作。

配置 URL

现在，我们就可以在我们的 `urls.py` 里添加相应的 **route** 来访问页面，代码如下所示：

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

apiRouter = routers.DefaultRouter()
apiRouter.register(r'blogpost', BlogpostSet)

urlpatterns = patterns('',
    (r'^$', 'blogpost.views.index'),
    url(r'^blog/(?P<slug>[^\.]+)\.html', 'blogpost.views.view_post', name='view_post'),
    url(r'^admin/', include(admin.site.urls))
) + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

在上面的代码里，我们创建了两个 **route**：

- 指向首页，其 **view** 是 **index**
- 指向博客详情页，其 **view** 是 **view_post**

指向博客详情页的 **URL** 正规 `r'^blog/(?P<slug>[^\.]+)\.html'`，会将形如 `blog/hello-world.html` 中的 `hello-world` 提取出来作为参数传给 `view_post` 方法。

接着，我们就可以创建两个 **view**。

创建 View

创建博客列表页

对于我们的首页来说，我们可以简单的只显示五篇博客，所以我们所需要做的就是从我们的 **Blogpost** 对象中，取出前五个结果即可。代码如下所示：

```
from django.shortcuts import render, render_to_response, get_object_or_404
from blogpost.models import Blogpost

def index(request):
    return render_to_response('index.html', {
        'posts': Blogpost.objects.all()[:5]
    })
```

Django 的 `render_to_response` 方法可以根据一个给定的上下文字典渲染一个给定的目标，并返回渲染后的 `HttpResponse`。即将相应的值，如这里的 `Blogpost.objects.all()[:5]`，填入相应的 `index.html` 中，再返回最后的结果。

因此，在我们的 `index.html` 中，我们就可以拿到前五篇博客。我们只需要遍历出 `posts`，拿出每个 `post` 相应的值，就可以完成列表页。

```
{% extends 'base.html' %}
{% block title %}Welcome to my blog{% endblock %}

{% block content %}
<h1>Posts</h1>
{% for post in posts %}
<h2><a href="{{ post.get_absolute_url }}">{{ post.title }}</a></h2>
<p>{{post.posted}} - By {{post.author}}</p>
<p>{{post.body}}</p>
{% endfor %}
{% endblock %}
```

在上面的模板里，我们还取出了博客的链接用于跳转到详情页。

创建博客详情页

依据上面拿到的 `slug`，我们就可以创建对应的详情页的 `view`，代码如下所示：

```
def view_post(request, slug):
    return render_to_response('blogpost_detail.html', {
        'post': get_object_or_404(Blogpost, slug=slug)
    })
```

这里的 `get_object_or_404` 将会根据 `slug` 来获取相应的博客，如果取不出相应的博客就会返回 **404**。因此，我们的详情页和上面的列表页也是类似的。

```
{% extends 'base.html' %}
{% block head_title %}{{ post.title }}{% endblock %}
{% block title %}{{ post.title }}{% endblock %}

{% block content %}
<h2>{{ post.title }}</h2>
<p>{{ post.posted }} - By {{ post.author }}</p>
<p>{{ post.body }}</p>
{% endblock %}
```

随后，我们就可以再提交一次代码了。

测试

TDD 虽然是一个非常好的实践，但是那是对于那些已经习惯写测试的人来说。如果你写测试的经历非常小，那么我们就可以从写测试开始。

在这里我们使用的是 **Django** 这个第三方框架来完成我们的工作，所以我们并不对这个框架的功能进行测试。虽然有些时候正是因为这些第三方框架的问题而导致的 **Bug**，但是我们仅仅只是使用一些基础的功能。这些基础的功能也已经在他们的框架中测试过了。

测试首页

先来做一个简单的测试，即测试我们访问首页的时候，调用的函数是上面的 `index` 函数

```
from django.core.urlresolvers import resolve
from django.http import HttpRequest
```

```
from django.test import TestCase

from blogpost.views import index, view_post


class HomePageTest(TestCase):
    def test_root_url_resolves_to_home_page_view(self):
        found = resolve('/')
        self.assertEqual(found.func, index)
```

但是这样的测试看上去没有多大意义，不过它可以保证我们的 **route** 可以和我们的 **URL** 对应上。在编写完测试后，我们就可以命令提示行中运行：

```
python manage.py test
```

来查看测试的结果：

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

```
.
```

```
-----
Ran 1 test in 0.031s
```

```
OK
```

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

```
(growth-django)
```

运行通过，现在我们可以进行下一个测试了——我们可以测试页面的标题是不是我们想要的结果：

```
def test_home_page_returns_correct_html(self):
    request = HttpRequest()
    response = index(request)
    self.assertIn(b'<title>Welcome to my blog</title>', response.content)
```

这里我们需要去请求相应的页面来获取页面的标题，并用 **assertIn** 方法来断言返回的首页的 **html** 中含有 `<title>Welcome to my blog</title>`。

测试详情页

同样的我们也可以用测试是否调用某个函数的方法，来看博客的详情页的 **route** 是否正确？

```
class BlogpostTest(TestCase):  
    def test_blogpost_url_resolves_to_blog_post_view(self):  
        found = resolve('/blog/this_is_a_test.html')  
        self.assertEqual(found.func, view_post)
```

与上面测试首页不一样的是，在我们的 **Blogpost** 测试中，我们需要创建数据，以确保这个流程是没有问题的。因此我们需要用 `Blogpost.objects.create` 方法来创建一个数据，然后访问相应的页面来看是否正确。

```
def test_blogpost_create_with_view(self):  
    Blogpost.objects.create(title='hello', author='admin', slug='this_is_a_test',  
                             posted=datetime.now)  
    response = self.client.get('/blog/this_is_a_test.html')  
    self.assertIn(b'This is a blog', response.content)
```

或许你会疑惑这个数据会不会被注入到数据库中，请看运行测试时返回的结果的第一句：

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

Django 将会创建一个数据库用于测试。

同理，我们也可以为首页添加一个相似的测试：

```
def test_blogpost_create_with_show_in_homepage(self):  
    Blogpost.objects.create(title='hello', author='admin', slug='this_is_a_test',  
                             posted=datetime.now)  
    response = self.client.get('/')  
    self.assertIn(b'This is a blog', response.content)
```

我们用同样的方法创建了一篇博客，然后在首页测试返回的内容中是否含有 `This is a blog`。

功能测试与持续集成

在上一章最后，我们写的测试可以算得上是单元测试，接着我们可以写一些自动化测试。

编写自动化测试

接着我们就可以用 **Selenium** 来做自动化测试。这是 **ThoughtWorks** 出品的一个强大的基于浏览器的开源自动化测试工具，它通常用来编写 **Web** 应用的自动化测试。

Selenium 与第一个 **UI** 测试

先让我们来看一个自动化测试的例子：

```
from django.test import LiveServerTestCase
from selenium import webdriver

class HomepageTestCase(LiveServerTestCase):
    def setUp(self):
        self.selenium = webdriver.Firefox()
        self.selenium.maximize_window()
        super(HomepageTestCase, self).setUp()

    def tearDown(self):
        self.selenium.quit()
        super(HomepageTestCase, self).tearDown()

    def test_visit_homepage(self):
        self.selenium.get(
            '%s%s' % (self.live_server_url, "/")
        )

        self.assertIn("Welcome to my blog", self.selenium.title)
```

在 **setUp**——即开始的时候，我们会用 **selenium** 起一个 **Firefox** 浏览器的进程，并执行 **maximize_window** 来将窗口最大化。在 **tearDown**——即结束的时候，我们就会关

闭这个浏览器的进程。我们的主要测试代码就在 `test_visit_homepage` 这个方法里，我们在里面访问首页，并判断标题是不是 `Welcome to my blog`。

运行上面的测试就会启动一个浏览器，并且会在浏览器上进行相应的操作。如下图所示：

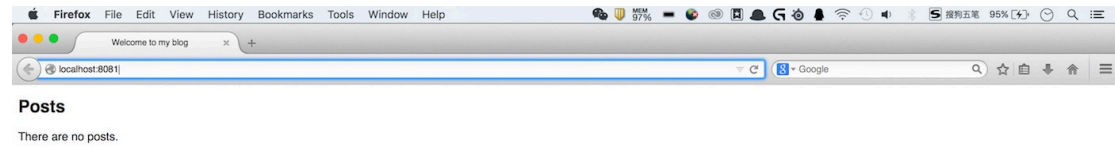


图 8: Selenium Demo

这时你可能会产生一些疑惑，这些内容我们不是已经测试过了么？两者从测试看是差不多的，但是从流程上来说并不是如此。下图是页面渲染的时间线：

请求从浏览器传到服务器要有一系列的过程，如重定向、缓存、DNS 等等，最后直至返回对应的 **Response**。我们用 **Django** 的测试框架只能实现到这一步，随后页面请求对应的静态资料，再对页面进行渲染，在这个过程中页面的内容会发生一些变化。

为了避免页面的内容被替换掉，那么我们就需要对这部分内容进行测试。

如下的代码也是可以用于测试页面内容的代码：

```
class BlogpostDetailCase(LiveServerTestCase):
    def setUp(self):
        Blogpost.objects.create(
            title='hello',
            author='admin',
            slug='this_is_a_test',
            body='This is a blog',
```

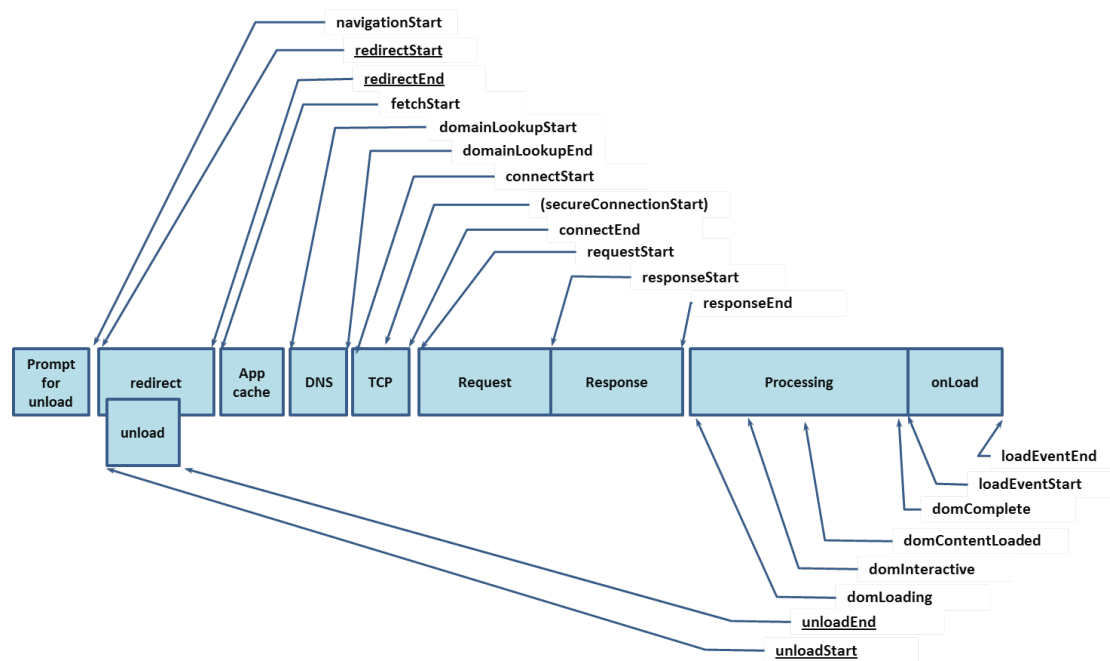


图 9: 页面渲染时间线

```

        posted=datetime.now
    )

    self.selenium = webdriver.Firefox()
    self.selenium.maximize_window()
    super(BlogpostDetailCase, self).setUp()

    def tearDown(self):
        self.selenium.quit()
        super(BlogpostDetailCase, self).tearDown()

    def test_visit_blog_post(self):
        self.selenium.get(
            '%s%s' % (self.live_server_url, "/blog/this_is_a_test.html")
        )

        self.assertIn("hello", self.selenium.title)

```

虽然在这里我们要测试的只是页面的标题，而实际上我们要测试的是页面的元素是否存在。

同样的，我们也可以对博客的内容进行测试。这些稍有不同的是，我们更多地是要测试用户的行为，如我们在首页点击某个链接，那么我应该中转到对应的博客详情页，如下代码所示：

```
class BlogpostFromHomepageCase(LiveServerTestCase):
    def setUp(self):
        Blogpost.objects.create(
            title='hello',
            author='admin',
            slug='this_is_a_test',
            body='This is a blog',
            posted=datetime.now
        )

        self.selenium = webdriver.Firefox()
        self.selenium.maximize_window()
        super(BlogpostFromHomepageCase, self).setUp()

    def tearDown(self):
        self.selenium.quit()
        super(BlogpostFromHomepageCase, self).tearDown()

    def test_visit_blog_post(self):
        self.selenium.get(
            '%s%s' % (self.live_server_url, "/")
        )

        self.selenium.find_element_by_link_text("hello").click()
        self.assertIn("hello", self.selenium.title)
```

需要注意的是，如果我们的单元测试如果可以测试到页面的内容——即没有使用 JavaScript 对页面的内容进行修改，那么我们应该使用单元测试即可。如测试金字塔所说，越底层的测试越快。

在我们编写完这些测试后，我们就可以搭建好相应的持续集成来运行这些测试了。

搭建持续集成

这里我们将使用 **Jenkins** 来完成这部分的工具，它是一个用 **Java** 编写的开源的持续集成工具。

它提供了软件开发的持续集成服务。它运行在 **Servlet** 容器中（例如 **Apache Tomcat**）。它支持软件配置管理（**SCM**）工具（包括 **AccuRev SCM**、**CVS**、**Subversion**、**Git**、**Perforce**、**Clearcase** 和 **RTC**），可以执行基于 **Apache Ant** 和 **Apache Maven** 的项目，以及任意的 **Shell** 脚本和 **Windows** 批处理命令。

要使用 **Jenkins**，只需要从 **Jenkins** 的主页上 (<https://jenkins.io/>) 下载最新的 **jenkins.war** 文件。然后运行

```
java -jar jenkins.war
```

便可以启动:

```
Running from: /Users/fdhuang/repractise/growth-ci/jenkins.war
webroot: $user.home/.jenkins
May 12, 2016 10:55:18 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: Logging initialized @489ms
May 12, 2016 10:55:18 PM winstone.Logger logInternal
INFO: Beginning extraction from war file
May 12, 2016 10:55:20 PM org.eclipse.jetty.util.log.JavaUtilLog warn
WARNING: Empty contextPath
May 12, 2016 10:55:20 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: jetty-9.2.z-SNAPSHOT
May 12, 2016 10:55:20 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: NO JSP Support for /, did not find org.eclipse.jetty.jsp.JettyJspServlet
Jenkins home directory: /Users/fdhuang/.jenkins found at: $user.home/.jenkins
May 12, 2016 10:55:21 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: Started w.@68c34b0{/,file:/Users/fdhuang/.jenkins/war/,AVAILABLE}/{/User
May 12, 2016 10:55:21 PM org.eclipse.jetty.util.log.JavaUtilLog info
INFO: Started ServerConnector@733a9ac6{HTTP/1.1}{0.0.0.0:8080}
```

接着，打开<http://0.0.0.0:8080/>就可以进行后续的安装，如下图所示：

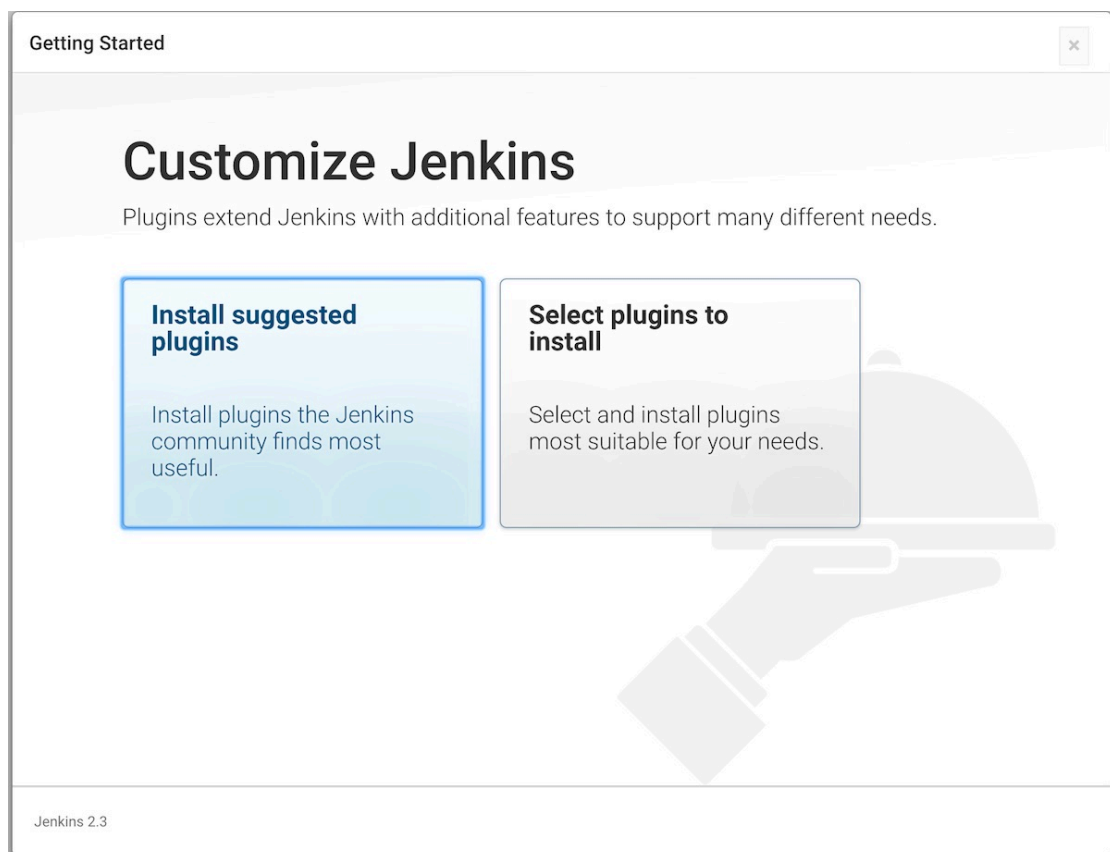


图 10: Jenkins 安装过程

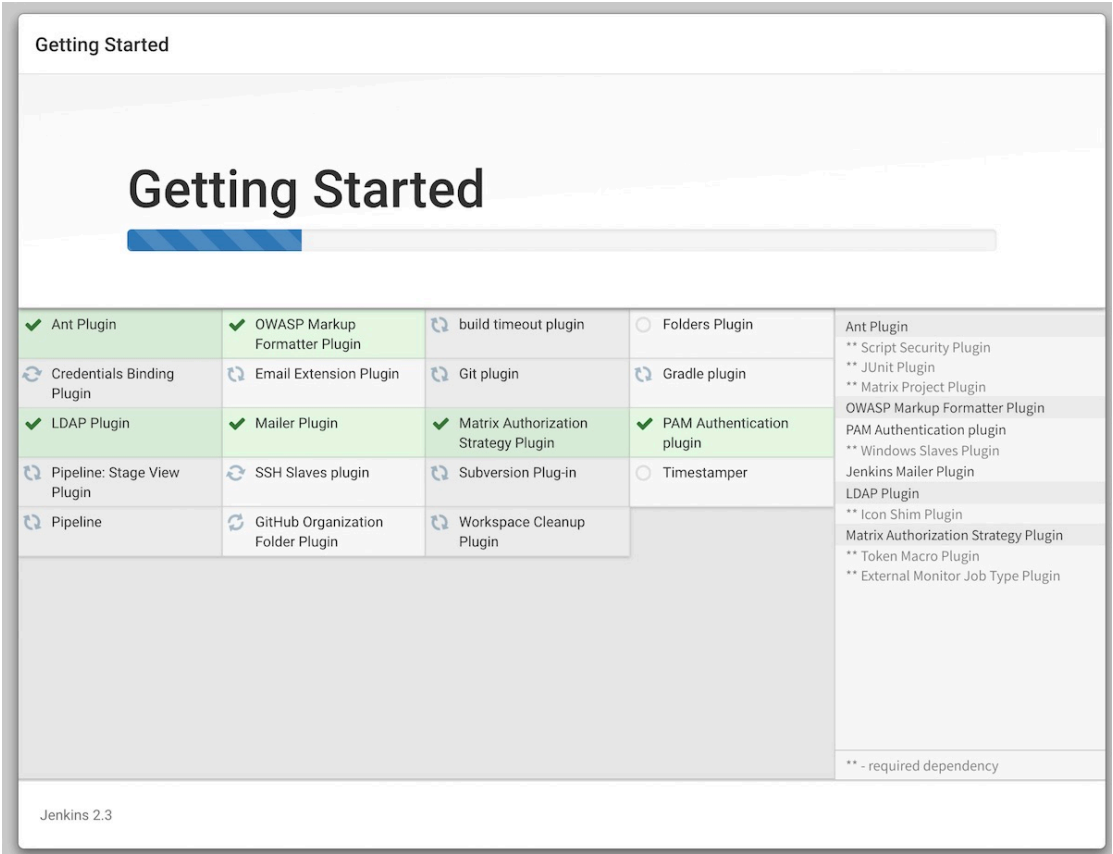


图 11: Jenkins 安装完成

慢慢等其安装完成：

等安装完成后，我们就可以开始使用 **Jenkins** 来创建我们的任务了。

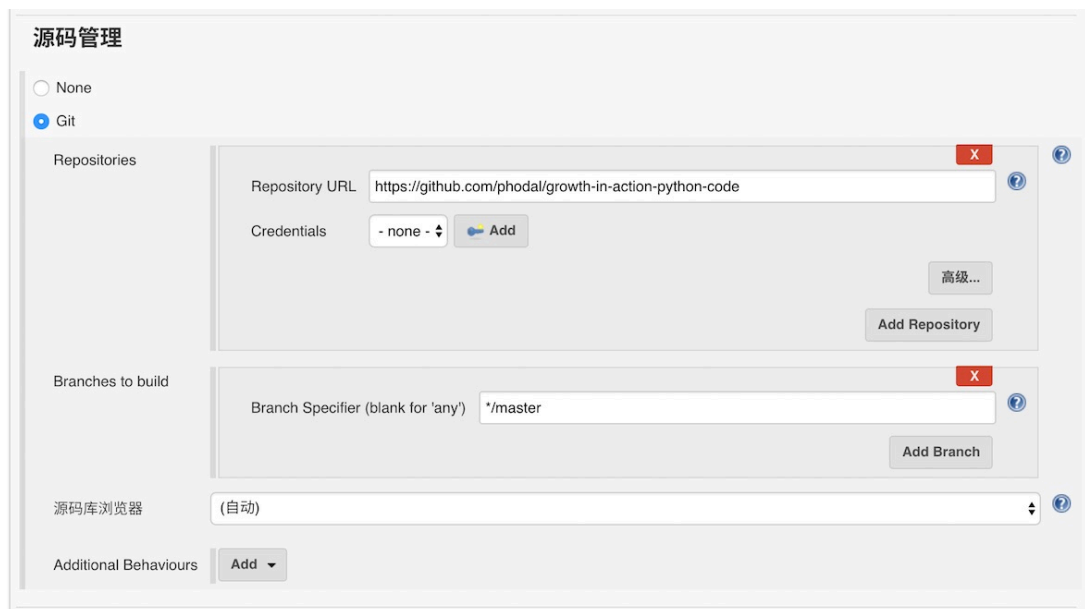
Jenkins 创建任务

在首页，我们会看到“开始创建一个新任务”的提示，点击它。

源码管理中选择 **Git**，并填入我们代码的地址：

```
[https://github.com/phodal/growth-in-action-python-code] (https://github.com/phodal/growth-in-action-python-code)
```

如下图所示：



The image shows the Jenkins 'Source Management' configuration page. It has a sidebar with 'None' and 'Git' (selected). The main area is divided into sections: 'Repositories' with a text input for 'Repository URL' (containing 'https://github.com/phodal/growth-in-action-python-code') and a dropdown for 'Credentials' (set to '- none -'), with buttons for '高级...' and 'Add Repository'; 'Branches to build' with a text input for 'Branch Specifier (blank for 'any')' (containing '*/master') and an 'Add Branch' button; '源码库浏览器' (Source Code Browser) with a dropdown set to '(自动)'; and 'Additional Behaviours' with an 'Add' button.

图 12: Jenkins 设计 Repo

然后就是构建触发器，一共有五种类型的触发器，意思也很容易理解：

- 触发远程构建 (例如, 使用脚本)
- Build after other projects are built
- Build periodically
- Build when a change is pushed to GitHub
- Poll SCM

在这里，我们要使用的是 **GitHub** 这个，它的原理是：

This job will be triggered if jenkins will receive PUSH GitHub hook from repo defined in scm section

即 Jenkins 在监听 GitHub 上对应的 PUSH hook，当发生代码提交时，就会运行我们的测试。

由于，我们暂时不需要一些特殊的构建环境配置，我们就可以将这个放空。接着，我们就可以配置构建了。

创建 shell

在这里我们需要添加的构建步骤是：execute shell，先让我们写一个简单的安装依赖的 shell

```
virtualenv --distribute -p /usr/local/bin/python3.5 growth-django
source growth-django/bin/activate
pip install -r requirements.txt
```

然后在保存后，我们可以尝试立即构建这个项目：



图 13: 控制台输出

在编写 shell 的过程中，我们要经过一些尝试，在这其中会经历一些失败的情形——即使是大部分有相关经验的程序员。如下图就是一次编写构建脚本引起的构建失败的例子：

最后，我们就得到下面的一个 shell 脚本，我们就可以将其变成相应的运行 CI 的脚本。以便于它可以在其他环境中使用：

```
Traceback (most recent call last):
  File "manage.py", line 10, in <module>
    execute_from_command_line(sys.argv)
  File "/usr/local/lib/python2.7/site-packages/django/core/management/__init__.py", line 385, in execute_from_command_line
    utility.execute()
  File "/usr/local/lib/python2.7/site-packages/django/core/management/__init__.py", line 354, in execute
    django.setup()
  File "/usr/local/lib/python2.7/site-packages/django/__init__.py", line 21, in setup
    apps.populate(settings.INSTALLED_APPS)
  File "/usr/local/lib/python2.7/site-packages/django/apps/registry.py", line 85, in populate
    app_config = AppConfig.create(entry)
  File "/usr/local/lib/python2.7/site-packages/django/apps/config.py", line 87, in create
    module = import_module(entry)
  File "/usr/local/Cellar/python/2.7.10_2/Frameworks/Python.framework/Versions/2.7/lib/python2.7/importlib/__init__.py", line 37, in import_module
    __import__(name)
ImportError: No module named rest_framework
Build step 'Execute shell' marked build as failure
Finished: FAILURE
```

图 14: Jenkins 失败的构建

```
#!/usr/bin/env bash

virtualenv --distribute -p /usr/local/bin/python3.5 growth-django
source growth-django/bin/activate
pip install -r requirements.txt
python manage.py test
python manage.py test test
```

记得给你的 **shell** 文件，加上执行的标志：

```
chmod u+x ./scripts/ci.sh
```

最后，我们就可以修改 **CI** 上相应的构建环境的配置。

更多功能

在 **Django** 框架中，内置了很多应用在它的“**contrib**”包中，这些包括：

- 一个可扩展的认证系统
- 动态站点管理页面
- 一组产生 **RSS** 和 **Atom** 的工具
- 一个灵活的评论系统
- 产生 **Google** 站点地图（**Google Sitemaps**）的工具
- 防止跨站请求伪造（**cross-site request forgery**）的工具
- 一套支持轻量级标记语言（**Textile** 和 **Markdown**）的模板库
- 一套协助创建地理信息系统（**GIS**）的基础框架

这意味着，我们可以直接用 **Django** 一些内置的组件来完成很多功能，先让我们来看看怎么完成一个简单的评论功能。

静态页面

Django 带有一个可选的“**flatpages**”应用，可以让我们存储简单的“扁平化 (**flat**)”页面在数据库中，并且可以通过 **Django** 的管理界面以及一个 **Python API** 来处理要管理的内容。这样的一个静态页面，一般包含下面的几个属性：

- 标题
- URL
- 内容 (**Content**)
- **Sites**
- 自定义模板（可选）

为了使用它来创建静态页面，我们需要在数据库中存储对应的映射关系，并创建对应的静态页面。

安装 **flatpages**

为此我们需要添加两个应用到 `settings.py` 文件的 `INSTALLED_APPS` 中：

- `django.contrib.sites`——“**sites**”框架，它用于将对象和功能与特定的站点关联。同时，它还是域名和你的 **Django** 站点名称之间的对应关系所保存的位置，即我们需要在这个地方设置我们的网站的域名。
- `django.contrib.flatpages`，即上文说到的内容。

在添加 `django.contrib.sites` 的时候，我们需要创建一个 `SITE_ID`。通过这个值等于 `1`，除非我们打算用这个框架去管理多个站点。代码如下所示：

```
SITE_ID = 1

INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
```

```

        'django.contrib.messages',
        'django.contrib.staticfiles',
        'django.contrib.sites',
        'django.contrib.flatpages',
        'blogpost'
    )

```

接着,还添加对应的中间件 `django.contrib.flatpages.middleware.FlatpageFallbackMiddleware` 到 `settings.py` 文件的 `MIDDLEWARE_CLASSES` 中。

然后,我们需要创建对应的 **URL** 来管理所有的静态页面。下面的代码是将静态页面都放在 **pages** 路径下,即如果我们有一个 **about** 的页面,那么的 **URL** 会变成 `http://localhost/pages/about/`。

```
url(r'^pages/', include('django.contrib.flatpages.urls')),
```

当然我们也可以将其配置为类似于 `http://localhost/about/` 这样的 **URL**:

```

urlpatterns += [
    url(r'^(?P<url>.*\/)$', views.flatpage),
]

```

最后,我们还需要做一个数据库迁移:

Operations to perform:

Apply all migrations: contenttypes, auth, admin, sites, blogpost, sessions,

Running migrations:

Rendering model states... DONE

Applying flatpages.0001_initial... OK

创建模板

接着,我们可以在 `templates` 目录下创建 `flatpages` 文件,用于存放我们的模板文件,下面是一个简单的模板:

```

{% extends 'base.html' %}
{% block title %}关于我{% endblock %}

```

```
{% block content %}
<div>
<h2>关于 博客</h2>
  <p>一方面，找到更多志同道合的人；另一方面，扩大影响力。</p>
  <p>内容包括</p>
  <ul>
    <li>成长记录</li>
    <li>技术笔记</li>
    <li>生活思考</li>
    <li>个人试验</li>
  </ul>
</div>
{% endblock %}
```

当我们完成模板后，我们就需要登录后台，并添加对应的静态页面的配置：

The screenshot shows the 'Flat Pages' admin interface. At the top, there is a breadcrumb trail: 'Home > Flat Pages > Flat pages > /about/ -- About'. Below this, the title 'Change flat page' is displayed. The form consists of several fields: 'URL:' with a text input containing '/about/' and a small example note below it; 'Title:' with a text input containing 'About'; 'Content:' with a large text area containing 'Something'; and 'Sites:' with a list box containing 'example.com' and 'localhost', a green plus icon for adding more sites, and a note at the bottom about holding down 'Control' or 'Command' on a Mac to select multiple sites.

图 15: 管理员界面创建 flatpage

然后从高级选项中填写我们的静态页面的路径，我们就可以完成静态页面的创建。

如下图所示：

图 16: flatpage 高级选项

最后，还要所个链接加到首页的导航中：

```
<li>
    <a href="/pages/resume/">简历</a>
</li>
```

下面让我们为我们的博客添加一个简单的评论功能吧！

评论功能

在早期的 **Django** 版本 (1.6 以前) 中，**Comments** 是自带的组件，但是后来它被从标准组件中移除了。因此，我们需要安装 **comments** 这个包：

```
pip install django-contrib-comments
```

再把它及它的版本添加到 `requirements.txt`，如下所示：

```
django==1.9.4
selenium==2.53.1
fabric==1.10.2
djangoestframework==3.3.3
djangoestframework-jwt==1.7.2
django-cors-headers==1.1.0
django-contrib-comments==1.7.1
```

接着,将 `django.contrib.sites` 和 `django_comments` 添加到 `INSTALLED_APPS`，如下：

```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'django.contrib.sites',  
    'django_comments',  
    'rest_framework',  
    'blogpost'  
)
```

然后做一下数据库迁移我们就可以完成对其的初始化:

Operations to perform:

Apply all migrations: contenttypes, admin, blogpost, auth, sites, sessions,

Running migrations:

Rendering model states... DONE

Applying sites.0001_initial... OK

Applying django_comments.0001_initial... OK

Applying django_comments.0002_update_user_email_field_length... OK

Applying django_comments.0003_add_submit_date_index... OK

Applying sites.0002_alter_domain_unique... OK

(growth-django)

然后再添加 URL 到 `urls.py`:

```
url(r'^comments/', include('django_comments.urls')),
```

现在,我们就可以登录后台,来创建对应的评论,但是这时候评论是不会显示到页面上的。所以我们需要对我们的博客详情页的模板进行修改,在其中添加一句:

```
{% render_comment_list for post %}
```

用于显示对应博客的评论,最近我们的模板文件如下面的内容所示:

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

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

{% block content %}
<div class="mdl-card mdl-shadow--2dp">
    <div class="mdl-card__title">
        <h2 class="mdl-card__title-text"><a href="{{ post.get_absolute_url }}">
a</h2>
    </div>
    <div class="mdl-card__supporting-text">
        {{post.body}}
    </div>
    <div class="mdl-card__actions">
        {{post.posted}} - By {{post.author}}
    </div>
</div>

{% render_comment_list for post %}

{% endblock %}
```

遗憾的是，当我们刷新页面的时候，页面报错了，原因如下所示：

我们还需要定义一个 `SITE_ID`，添加下面的代码到 `settings.py` 文件中即可：

```
SITE_ID = 1
```

然后，我们就可以从后台创建评论：

Sitemap

我们在之前的文章中提到过 **SEO** 的重要性，这里只是简单地对 **Sitemap** 的内容进行展开。

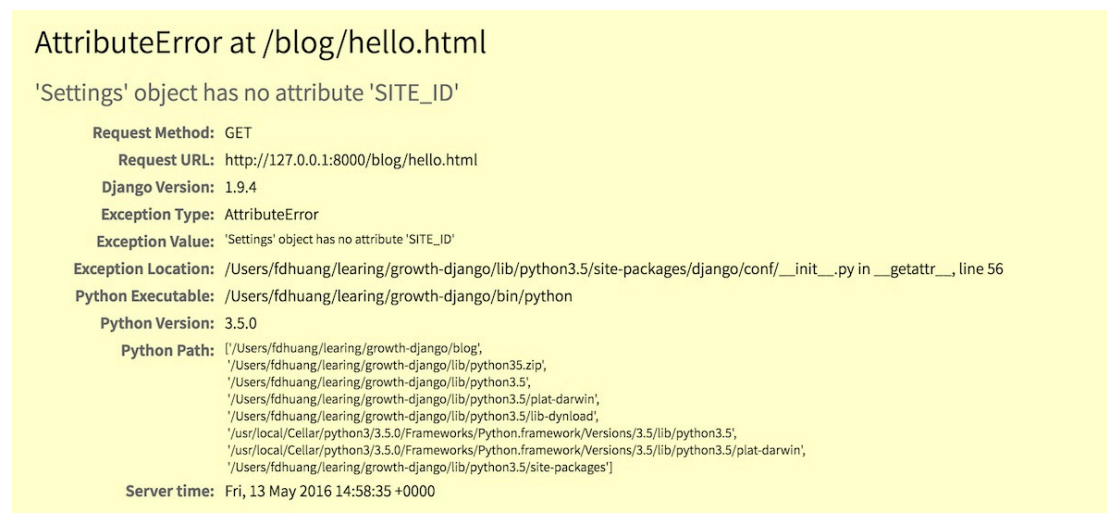


图 17: SITE_ID 报错

站点地图介绍

Sitemap 译为站点地图，它用于告诉搜索引擎他们网站上有哪些可供抓取的网页。常见的 Sitemap 的形式是以 xml 出现了，如下是我博客的 sitemap.xml 的一部分内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
<url>
  <loc>https://www.phodal.com/blog/mezzanine-add-new-page/</loc>
  <lastmod>2014-08-03</lastmod>
  <changefreq>Monthly</changefreq>
  <priority>0.2</priority>
</url>
</urlset>
```

从上面的内容中，我们可以发现它包含了下面的一些 XML 标签：

- urlset，封装该文件，并指明当前协议的标准。
- url，每个 URL 实体的父标签。
- loc，指明页面的 URL
- lastmod（可选），内容最后的修改时间
- changefreq（可选），内容的修改频率，用于告知搜索引擎抓取频率。它包含的值有：always、hourly、daily、weekly、monthly、yearly、never
- priority（可选），范围是从 0.0~1.0，搜索引擎用于对你网站在搜索结果的排序，

hello

还记得公瑾当年：当时CSS3才刚刚出来，于是你可以在上图中看到之中的CSS3的效果，只是做得很渣。唯一还不错的，算是在左上角做的LOGO。当时我们用着Django + Appspot 玩得不亦乐乎。当时我还能写些文字，找到了写的一个小故事《新上海滩》。当时只有一点基础，却总以为技术比大二大三的。我们需要自己去一点点写CSS样式，还不会使用Git。但是已经在使用Google App Engine，当时还试过Sina App Engine，只是当时恨铁不成钢啊。网站的一点点历史

March 24, 2016 - By world

May 13, 2016, 3:04 p.m. - huang

lfsdfa

图 18: 后台创建评论

即内部的优先级排序。需要注意的是如果你把所有页面的优先级设置为 **1**，那么它就和没有设置的效果是一样的。

从上面的内容中，我们可以发现：

站点地图能够提供与其中列出的网页相关的宝贵元数据：元数据是网页的相关信息，例如网页的最近更新时间、网页的更改频率以及网页相较于网站中其他网址的重要程度。——内容来自 **Google Sitemap** 帮助文档。

现在，我们一共有三种类型的页面：

- 首页，通常来说首页的 **priority** 应该是最高的，而它的 **changefreq** 可以设置为 **daily**、**weekly**，这取决于你的博客的更新频率。如果你是做一些 **UGC**(用户生成内容) 的网站，那么你应该设置为 **always**、**hourly**。
- 动态生成的博客详情页，这些内容一般很少进行改变，所以这的 **changefreq** 会比较低，如 **yearly** 或者 **monthly**——并且没有高的必要性，它会导致搜索引擎一直抓取你的内容。这会对服务器造成一定的压力，并且无助于你网站的排名。
- 静态页面，如 **About** 页面，它可以有一个高的 **priority**，但是它的 **changefreq** 也不一定很高。

下面就让我们从首页说起。

创建首页的 **Sitemap**

与上面创建静态页面时一样，我们也需要添加 `django.contrib.sitemaps` 到 `INSTALLED_APPS` 中。

然后，我们需要指定一个 **URL** 规则。通常来说，这个 **URL** 是叫 `sitemap.xml`——一个约定俗成的标准。我们需要创建一个 **sitemaps** 对象来存储所有的 **sitemaps**：

```
url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps}, name='django.contrib.sitemaps')
```

因此，我们需要创建几种不同类型的 **sitemap**，如下是首页的 **Sitemap**，它继承自 Django 的 **Sitemap** 类：

```
class PageSitemap(Sitemap):  
    priority = 1.0  
    changefreq = 'daily'
```

```
def items(self):
    return ['main']

def location(self, item):
    return reverse(item)
```

它定义了自己的 **priority** 是最高的 **1.0**，同时每新频率为 **daily**。然后在 **items** 里面去取它所要获取的 **URL**，即 `urls.py` 中对应的 **name** 的 **main** 的 **URL**。在这里我们只返回了 **main** 一个值，依据于下面的 **location** 方法中的 **reverse**，它找到了 **main** 对应的 **URL**，即首页。

最后结合首页 **sitemap.xml** 的 `urls.py` 代码如下所示：

```
from sitemap.sitemaps import PageSitemap

sitemaps = {
    "page": PageSitemap
}

urlpatterns = patterns('',
    url(r'^$', blogpostViews.index, name='main'),
    url(r'^blog/(?P<slug>[^\.]*)\.html', 'blogpost.views.view_post', name='view_post'),
    url(r'^comments/', include('django_comments.urls')),
    url(r'^admin/', include(admin.site.urls)),
    url(r'^pages/', include('django.contrib.flatpages.urls')),
    url(r'^sitemap\.xml$', sitemap, {'sitemaps': sitemaps}, name='django.contrib.sitemaps.views.sitemap')
) + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

除此，我们还需要创建自己的 **sitemap.xml** 模板——自带的系统模板比较简单。

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
{% spaceless %}
{% for url in urlset %}
<url>
    <loc>{{ url.location }}</loc>
```

```

    {% if url.lastmod %}<lastmod>{{ url.lastmod|date:"Y-m-d" }}</lastmod>{% e
    {% if url.changefreq %}<changefreq>{{ url.changefreq }}</changefreq>{% en
    {% if url.priority %}<priority>{{ url.priority }}</priority>{% endif %}
</url>
{% endfor %}
{% endspaceless %}
</urlset>

```

最后，我们访问<http://localhost:8000/sitemap.xml>，我们就可以获取到我们的 sitemap.xml：

```

<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.phodal.com/</loc>
    <changefreq>daily</changefreq>
    <priority>1.0</priority>
  </url>
</urlset>

```

下一步，我们仍可以直接创建出对应的静态页面的 Sitemap。

创建静态页面的 Sitemap

相似的，我们也需要从 `items` 访法中，定义出我们所要创建页面的对象。

```

from django.contrib.sitemaps import Sitemap
from django.core.urlresolvers import reverse
from django.apps import apps as django_apps

class FlatPageSitemap(Sitemap):
    priority = 0.8

    def items(self):
        Site = django_apps.get_model('sites.Site')
        current_site = Site.objects.get_current()
        return current_site.flatpage_set.filter(registration_required=False)

```

只不过这个方法可能会稍微麻烦一些，我们需要从数据库中取中当前的站点。再取出当前站点中的 **flatpage** 集合，过滤出那些不需要注册的页面，即代码中的 `registration_required=False`。

最近再将这个对象放入 **sitemaps** 即可：

```
from sitemap.sitemaps import PageSitemap, FlatPageSitemap

sitemaps = {
    "page": PageSitemap,
    'flatpages': FlatPageSitemap
}
```

现在，我们可以完成博客的 **Sitemap** 了。

创建博客的 **Sitemap**

同上面一样的是，我们依然需要在 **items** 方法中返回所有的博客内容。并且在 **lastmod** 中，返回这篇博客的发表日期——以免他们返回的是同一个日期：

```
class BlogSitemap(Sitemap):
    changefreq = "never"
    priority = 0.5

    def items(self):
        return Blogpost.objects.all()

    def lastmod(self, obj):
        return obj.posted
```

最近我们的 **Sitemap.xml**，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.phodal.com/about/</loc>
    <priority>0.8</priority>
```

```
</url>
<url>
  <loc>http://www.phodal.com/</loc>
  <changefreq>daily</changefreq>
  <priority>1.0</priority>
</url>
<url>
  <loc>http://www.phodal.com/blog/hello.html</loc>
  <lastmod>2016-03-24</lastmod>
  <changefreq>never</changefreq>
  <priority>0.5</priority>
</url>
</urlset>
```

提交到搜索引擎

这里我们以 Google Webmaster 为例简单的介绍一下如何使用各种站长工具来提交 sitemap.xml。

我们可以登录 Google 的 Webmaster: <https://www.google.com/webmasters/tools/home?hl=zh-cn>, 然后点击添加属性来创建一个新的网站:



图 19: 添加网站

这时候 Google 需要确认这个网站是你的, 所以它提供几点方法来验证, 除了下面的推荐方法:

我们可以使用下面的这一些方法:

我个人比较喜欢用 HTML Tag 的方式来实现

在我们完成验证之后, 我们就可以在后台手动提交 Sitemap.xml 了。

点击上方的添加/测试站点地图即可。

推荐的方法	备用方法	历史记录
-------	------	------

推荐：Google Analytics（分析）

使用您的 Google Analytics（分析）帐户。

- 您必须使用[异步跟踪代码](#)。
- 您的跟踪代码应该位于网页的 <head> 部分。
- 您必须拥有对 Google Analytics（分析）网络媒体资源的“修改”权限。

Google Analytics（分析）跟踪代码仅用于验证网站所有权，而不适用于访问任何 Google Analytics（分析）数据。

验证

暂不验证

图 20: 推荐的验证方式

推荐的方法	备用方法	历史记录
-------	------	------

☐ **HTML 文件上传**
将 HTML 文件上传到您的网站。

☐ **HTML 标记**
向您网站的首页添加元标记。

☐ **域名提供商**
登录到您的域名提供商网站。

☐ **Google 跟踪代码管理器**
使用您的 Google 跟踪代码管理器帐户。

验证

暂不验证

图 21: 备选的方法



图 22: HTML 标签验证



图 23: 提交 Sitemap.xml

前端框架

我们的前端样式实在是太丑了，让我们想办法来美化一下它们吧——这时候我们就需要一个前端框架来帮助我们做这件事。这里的前端框架并不是指那种 **MV*** 框架，而是 **UI** 框架。

响应式设计

考虑到易学程度，以其响应式设计的问题，我们决定用 **Bootstrap** 来作为这里的前端框架。**Bootstrap** 是 **Twitter** 推出的一个用于前端开发的开源工具包，似乎也是当前“最受欢迎”的前端框架。它提供了全面、美观的文档。你能在这里找到关于 **HTML** 元素、**HTML** 和 **CSS** 组件、**jQuery** 插件方面的所有详细文档。并且我们能在 **Bootstrap** 的帮助下通过同一份代码快速、有效适配手机、平板、**PC** 设备。

它是一个支持响应式设计的框架，即页面的设计与开发应当根据用户行为以及设备环境（系统平台、屏幕尺寸、屏幕定向等）进行相应的响应和调整。如下图所示：

我们在不同的设计上看到的是不是同的布局，这会依据我们的设备大小做出调



图 24: 响应式设计

整——使用媒体查询 (media queries) 实现。

引入前端框架

下好 **Bootstrap**，将里面的内容复制到 `static/` 目录，如下所示：

```
.
├── css
│   ├── bootstrap-theme.css
│   ├── bootstrap-theme.css.map
│   ├── bootstrap-theme.min.css
│   ├── bootstrap-theme.min.css.map
│   ├── bootstrap.css
│   ├── bootstrap.css.map
│   ├── bootstrap.min.css
│   ├── bootstrap.min.css.map
│   └── styles.css
├── fonts
│   ├── glyphicons-halflings-regular.eot
│   ├── glyphicons-halflings-regular.svg
│   ├── glyphicons-halflings-regular.ttf
│   ├── glyphicons-halflings-regular.woff
│   └── glyphicons-halflings-regular.woff2
```

```
└── js
    ├── bootstrap.js
    ├── bootstrap.min.js
    └── npm.js
```

它包含了 **JavaScript**、**CSS** 还有字体，需要注意的一点是 **bootstrap** 依赖于 **jquery**。因此，我们需要下载 **jquery** 并放到这个目录里。然后在我们的 **head** 里引入这些 **css**

```
<head>
  <title>{% block head_title %}Welcome to my blog{% endblock %}</
title>
  <link rel="stylesheet" type="text/css" href="{% static 'css/
bootstrap.min.css' %}">
</head>
```

在我们的 **body** 结尾的地方：

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

在这里，将 **Script** 放在 **body** 的尾部有利于用户打开页面的速度。而对于一些纯前端的框架来说，它们就需要放在页面开始的地方。

页面美化

现在，我们就可以创建一个导航了。

添加导航

根据 **Bootstrap** 的官方文档的 **Demo**，我们可以创建对应的导航。

```
<header class="navbar navbar-static-top bs-docs-nav" id="top" role="banner">
  <div class="container">
    <div class="navbar-header">
      <button class="navbar-toggle collapsed" type="button" data-toggle="collapse">
```

```

        data-target=".bs-navbar-collapse">
        <span class="sr-only">切换视图</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
    </button>
    <a href="/" class="navbar-brand">Growth 博客</a>
</div>
<nav class="collapse navbar-collapse bs-navbar-collapse" role="naviga
    <ul class="nav navbar-nav">
        <li>
            <a href="/pages/about/">关于我</a>
        </li>
        <li>
            <a href="/pages/resume/">简历</a>
        </li>
    </ul>
    <ul class="nav navbar-nav navbar-right">
        <li><a href="/admin" id="loginLink">登入</a></li>
    </ul>

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

```

它在桌面下的效果大致如下图所示：

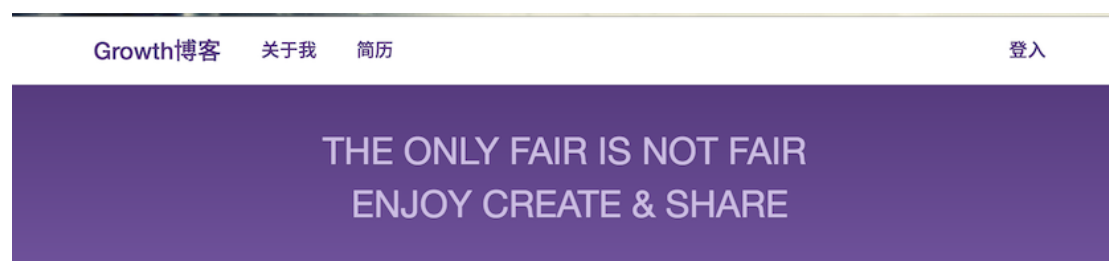


图 25: 桌面浏览器下的 Bootstrap 导航

而在移动浏览器下则是这样的效果：

当我们点击右上角的菜单按钮时，会出现我们的菜单

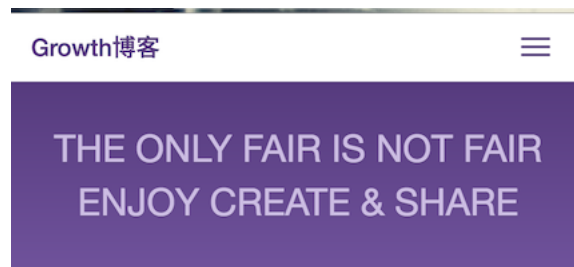


图 26: 移动设备上的导航



图 27: 点击导航后的结果

添加标语

接着，我们可以快速的创建一个标语：

```
<main class="bs-docs-masthead" id="content" role="main">
  <div class="container">
    <div id="carbonads-container">
      THE ONLY FAIR IS NOT FAIR <br>
      ENJOY CREATE & SHARE
    </div>
  </div>
</main>
```

这里的代码都比较简单，我想也不需要太多的解释。

优化列表

接着，我们可以简单的对首页的博客列表做一个优化，方法比较简单：

- 为博客列表添加一个 `row` 的 `class`，表示它可以滚动
- 在每一篇博客里添加 `col-sm-4` 的 `class`，在不同的大小下会有不同的布局

代码如下所示：

```
{% extends 'base.html' %}
{% block title %}Welcome to my blog{% endblock %}

{% block content %}
<h2>博客</h2>
<div class="row">
  {% if posts %}
  {% for post in posts %}
    <div class="col-sm-4">
      <h2><a href="{{ post.get_absolute_url }}">{{ post.title }}</a></h2>
      {{post.body | slice:"":80}}
      {{post.posted}} - By {{post.author}}
    </div>
  {% endfor %}
</div>
```

```

    </div>

    {% endfor %}

    {% else %}

    <p>There are no posts.</p>

    {% endif %}

</div>

{% endblock %}

```

它在桌面和自动设备上的效果如下图所示：

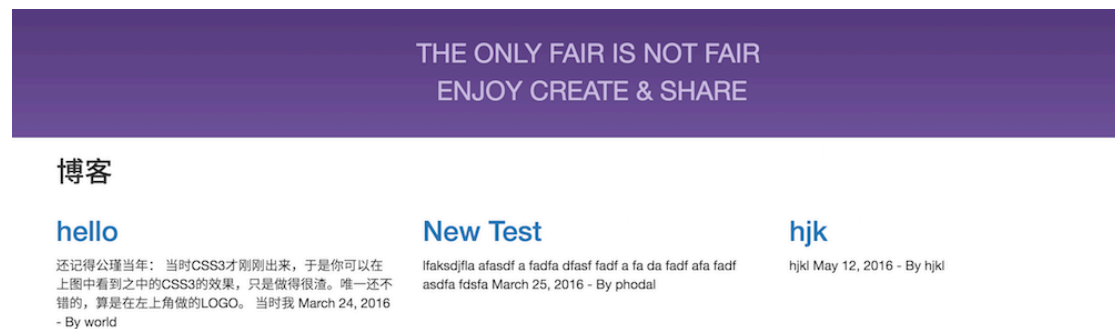


图 28: 桌面设备效果

添加 footer

最后，我们可以在页面的最下方添加一个 **footer**，来做一些版权声明：

```

<footer class="footer">
  <div class="container">
    <p class="text-muted">©Copyright Phodal.com</p>
  </div>
</footer>

```

它拥有一些简单的样式，来将 **footer** 固定在页面的最下方：

```

.footer {
  position: absolute;
  bottom: 0;
  width: 100%;
}

```

博客

hello

还记得公瑾当年：当时CSS3才刚刚出来，于是你可以在上图中看到之中的CSS3的效果，只是做得很渣。唯一还不错的，算是在左上角做的LOGO。当时我 March 24, 2016 - By world

New Test

Ifaksdjfla afasdf a fadfa dfasf fadf a fa da fadf afa fadf asdfa fdsfa March 25, 2016 - By phodal

hjk

hjkl May 12, 2016 - By hjkl

图 29: 移动设备效果

```
/* Set the fixed height of the footer here */
height: 60px;
background-color: #f5f5f5;
}
.footer .container {
  width: auto;
  max-width: 680px;
  padding: 0 15px;
}

.footer .container .text-muted {
  margin: 20px 0;
}
```

API

在下一章开始之前，我们先来搭建一下 API 平台，不仅仅可以提供一些额外的功能，还可以为我们的 APP 提供 API。

博客列表

Django REST Framework

在这里，我们需要用到一个名为 Django REST Framework 的 RESTful API 库。通过这个库，我们可以快速创建我们所需要的 API。

Django REST Framework 这个名字很直白，就是基于 Django 的 REST 框架。因此，首先我们仍是要安装这个库：

```
pip install djangorestframework
```

然后把它添加到 INSTALLED_APPS 中：

```
INSTALLED_APPS = (  
    ...  
    'rest_framework',  
)
```

如下所示：

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

接着我们可以在我们的 API 中创建一个 URL，用于匹配它的授权机制。

```
urlpatterns = [  
    ...  
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))  
]
```

不过这个 **API**，目前并没有多大的用途。只有当我们在制作一些需要权限验证的接口时，它才会突显它的重要性。

创建博客列表 **API**

为了方便我们继续展开后面的内容，我们先来创建一个博客列表 **API**。参考 **Django REST Framework** 的官方文档，我们可以很快地创建出下面的 **Demo**：

```
from django.contrib.auth.models import User
from rest_framework import serializers, viewsets
from blogpost.models import Blogpost


class BlogpsotSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Blogpost
        fields = ('title', 'author', 'body', 'slug')


class BlogpostSet(viewsets.ModelViewSet):
    queryset = Blogpost.objects.all()
    serializer_class = BlogpsotSerializer
```

在上面这个例子中，**API** 由两个部分组成：

- **ViewSet**，用于定义视图的展现形式——如返回哪些内容，需要做哪些权限处理
- **Serializers**，用于定义 **API** 的表现形式——如返回哪些字段，返回怎样的格式

我们在我们的 **URL** 中，会定义相应的规则到 **ViewSet**，而 **ViewSet** 则通过 `serializer_class` 找到对应的 **Serializers**。我们将 **Blogpost** 的所有对象赋予 `queryset`，并返回这些值。在 **BlogpsotSerializer** 中，我们定义了我们要返回的几个字段：`title`、`author`、`body`、`slug`。

接着，我们可以在我们的 `urls.py` 配置 **URL**。

...

```
from rest_framework import routers
```

```
from blogpost.api import BlogpostSet

apiRouter = routers.DefaultRouter()
apiRouter.register(r'blogpost', BlogpostSet)

urlpatterns = patterns('',
    ...
    url(r'^api/', include(apiRouter.urls)),
) + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

我们使用默认的 **Router** 来配置我们的 **URL**，即 **DefaultRouter**，它提供了一个非常简单的机制来自动检测 **URL** 规则。因此，我们只需要注册好我们的 **url**——**blogpost** 以及它值 **BlogpostSet** 即可。随后，我们再为其定义一个根 **URL** 即可：

```
url(r'^api/', include(apiRouter.urls))
```

测试 API

现在，我们可以访问<http://127.0.0.1:8000/api/>来访问我们现在的 **API**。由于 **Django REST Framework** 提供了一个 **UI** 机制，所以我们可以直接在网页上看到我们所有的 **API**：



图 30: Django REST Framework 列表

然后，点击页面中的<http://127.0.0.1:8000/api/blogpost/>，我们就可以访问博客相关的 **API** 了，如下图所示：

在页面上显示了所有的博客内容，在页面的下面有一个表单可以先让我们来创建数据：

直接在表单中添加数据，我们就可以完成数据创建了。

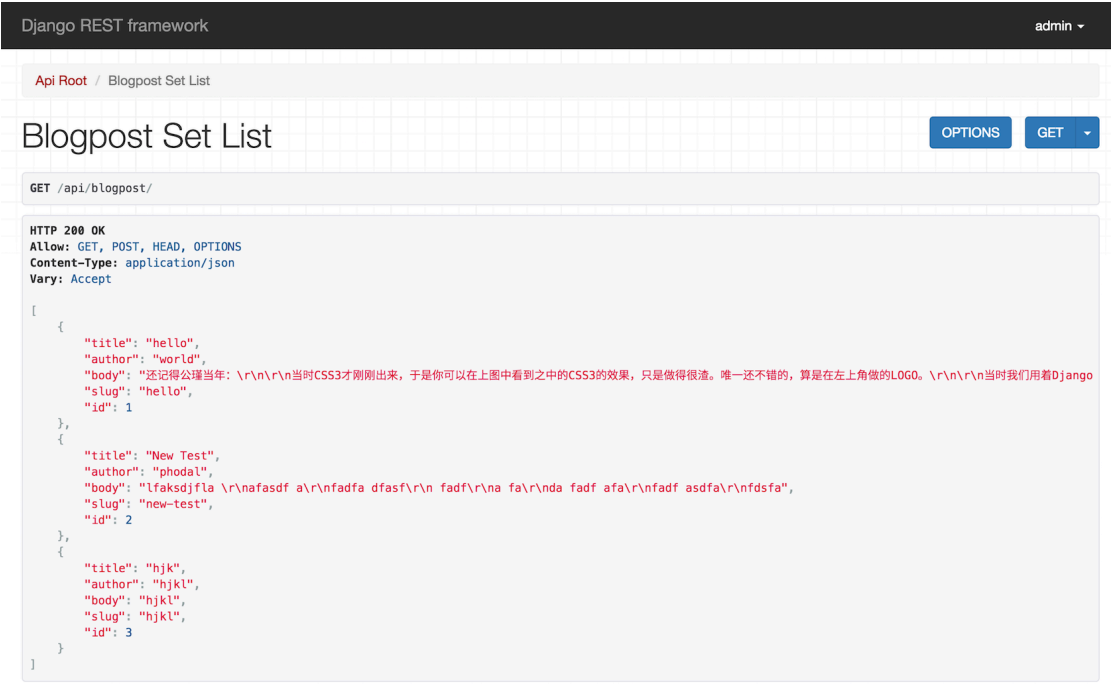


图 31: 博客 API

Raw data HTML form

Title

Author

Body

Slug

POST

图 32: 创建博客的表单

当我们输入某一些关键字的时候，就会出现文章的标题，随后我们只需要点击相应的标题即可跳转到文章。

搜索 API

为了实现这个功能我们需要对之前的博客 API 做一些简单的改造——可以支持搜索博客标题。这里我们需要稍微扩展一下我们的博客 API 即可：

```
class BlogpostSet(viewsets.ModelViewSet):
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)
    serializer_class = BlogpsotSerializer
    search_fields = 'title'

    def list(self, request):
        queryset = Blogpost.objects.all()

        search_param = self.request.query_params.get('title', None)
        if search_param is not None:
            queryset = Blogpost.objects.filter(title__contains=search_param)

        serializer = BlogpsotSerializer(queryset, many=True)
        return Response(serializer.data)
```

我们添加了一个名为 `search_fields` 的变量，顾名思义就是定义搜索字段。接着我们覆写了 `ModelViewSet` 的 `list` 方法，它是用于列出 (`list`) 所有的结果。我们会尝试在我们的请求中获取搜索参量，如果没有的话我们就返回所有的结果。如果搜索的参数中含有标题，则从所有博客中过滤出标题中含有搜索标题中的内容，再返回这些结果。如下是一个搜索的 URL: <http://127.0.0.1:8000/api/blogpost/?format=json&title=test>，我们搜索标题中含有 `test` 的内容。

同时，我们还需要为我们的 `apiRouter` 设置一个 `basename`，即下面代码中最后的 `Blogpost`

```
apiRouter.register(r'blogpost', BlogpostSet, 'Blogpost')
```

页面实现

接着，我们就可以在页面上实现这个功能。在这里我们使用一个名为 **Bootstrap-3-Typeahead** 的插件来实现，下载这个插件以及它对应的 **CSS**: <https://github.com/bassjobsen/typeahead.js-bootstrap-css>，并添加到 `base.html` 中，然后创建一个 `main.js` 文件负责相关的逻辑处理。

```
<script src="{% static 'js/jquery.min.js' %}"></script>
<script src="{% static 'js/bootstrap.min.js' %}"></script>
<script src="{% static 'js/bootstrap3-typeahead.min.js' %}"></script>
<script src="{% static 'js/main.js' %}"></script>
```

接着我们需要在页面上创建对应的 **UI**，我们可以直接在登录后面添加这个搜索按钮：

```
<nav class="collapse navbar-collapse bs-navbar-collapse" role="navigation">
  <ul class="nav navbar-nav">
    <li>
      <a href="/pages/about/">关于我</a>
    </li>
    <li>
      <a href="/pages/resume/">简历</a>
    </li>
  </ul>
  <ul class="nav navbar-nav navbar-right">
    <li><a href="/admin" id="loginLink">登录</a></li>
  </ul>
  <div class="col-sm-3 col-md-3 pull-right">
    <form class="navbar-form" role="search">
      <div class="input-group">
        <input type="text" id="typeahead-input" class="form-control" placeholder="Search" name="search" data-provide="typeahead">
        <div class="input-group-btn">
          <button class="btn btn-default search-button" type="submit">搜索</button>
        </div>
      </div>
    </form>
  </div>
```

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

我们主要是使用 **input** 标签，标签上对应有一个 **id**

```
<input type="text" id="typeahead-input" class="form-control" placeholder="Sea
```

对应于这个 ID，我们就可以开始编写我们的功能了：

```
$(document).ready(function () {
    $('#typeahead-input').typeahead({
        source: function (query, process) {
            return $.get('/api/blogpost/?format=json&title=' + query, function (data) {
                return process(data);
            });
        },
        updater: function (item) {
            return item;
        },
        displayText: function (item) {
            return item.title;
        },
        afterSelect: function (item) {
            location.href = 'http://localhost:8000/blog/' + item.slug + ".html";
        },
        delay: 500
    });
});
```

`$(document).ready()` 方法可以是在 DOM 完成加载后，运行其中的函数。接着我们开始监听 `#typeahead-input`，对应的便是 `id` 为 `typeahead-input` 的元素。可以看到在这其中有五个对象：

- **source**，即搜索的来源，我们返回的是我们搜索的 URL。
- **updater**，即每次更新要做的事
- **displayText**，显示在页面上的内容，如在这里我们返回的是博客的标题

- `afterSelect`，每用户选中某一项后做的事，这里我们直接中转到对应的博客。
- `delay`，延时 `500ms`。

虽然我们使用的是插件来完成我们的功能，但是总体的处理逻辑是：

1. 监听我们的输入文本
2. 获取 `API` 的返回结果
3. 对返回结果进行处理——如高亮输入文本、显示到页面上
4. 处理用户点击事件

跨域支持

当我们想为其他的网页提供我们的 `API` 时，可能会报错——原因是不支持跨域请求。为了方便我们下一章更好的展开，内容我们在这里对跨域进行支持。

添加跨域支持

有一个名为 `django-cors-headers` 的插件用于实现对跨域请求的支持，我们只使用安装它，并进行一些简单的配置即可。

```
pip install django-cors-headers
```

安装过程如下：

```
Collecting django-cors-headers
  Downloading django-cors-headers-1.1.0.tar.gz
Building wheels for collected packages: django-cors-headers
  Running setup.py bdist_wheel for django-cors-headers ... done
  Stored in directory: /Users/fdhuang/Library/Caches/pip/wheels/b0/75/89/7b17f134fc01b74e10523f3128e45b917da0c5f8638213e073
Successfully built django-cors-headers
Installing collected packages: django-cors-headers
Successfully installed django-cors-headers-1.1.0
```

我们还需要添加到 `django-cors-headers=1.1.0` 到 `requirements.txt` 文件中，以及添加到 `settings.py` 中：

```
INSTALLED_APPS = (  
    ...  
    'corsheaders',  
    ...  
)
```

以及对应的中间件:

```
MIDDLEWARE_CLASSES = (  
    ...  
    'corsheaders.middleware.CorsMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    ...  
)
```

同时还有对应的配置:

```
CORS_ALLOW_CREDENTIALS = True
```

现在, 让我们进行下一步, 开始 **APP** 吧!

移动应用

依照国庆惯例, 我们还将用 **Ionic 2** 继续创建 **hello,world**。

hello,world

开始之前我们需要先安装 **Ionic** 的命令行工具, 后来我们需要用这个工具来创建工程。

```
npm install -g ionic@beta
```

如果没有意外, 我们将安装成功, 然后可以使用 `ionic` 命令:

它自带了一系列的工具来加速我们的开发, 这些工具可以在后面的章节中学习。

```
Available tasks: (use --help or -h for more info)
```

```

start ..... Starts a new Ionic project in the specified PATH
serve ..... Start a local development server for app dev/
testing
platform ..... Add platform target for building an Ionic app
run ..... Run an Ionic project on a connected device
emulate ..... Emulate an Ionic project on a simulator or emulator
build ..... Build (prepare + compile) an Ionic project for a given p

plugin ..... Add a Cordova plugin
resources ..... Automatically create icon and splash screen resources (I
    Put your images in the ./resources directory, named splash or i
    Accepted file types are .png, .ai, and .psd.
    Icons should be 192x192 px without rounded corners.
    Splashscreens should be 2208x2208 px, with the image centered i

upload ..... Upload an app to your Ionic account
share ..... Share an app with a client, co-worker, friend, or custom
lib ..... Gets Ionic library version or updates the Ionic library
setup ..... Configure the project with a build tool (beta)
io ..... Integrate your app with the ionic.io platform services
security ..... Store your app's credentials for the Ionic Platform (alp
push ..... Upload APNS and GCM credentials to Ionic Push (alpha)
package ..... Use Ionic Package to build your app (alpha)
config ..... Set configuration variables for your ionic app (alpha)
browser ..... Add another browser for a platform (beta)
service ..... Add an Ionic service package and install any required p
add ..... Add an Ion, bower component, or addon to the project
remove ..... Remove an Ion, bower component, or addon from the proje
list ..... List Ions, bower components, or addons in the project
info ..... List information about the users runtime environment
help ..... Provides help for a certain command
link ..... Sets your Ionic App ID for your project
hooks ..... Manage your Ionic Cordova hooks
state ..... Saves or restores state of your Ionic Application using
docs ..... Opens up the documentation for Ionic

```

```
generate ..... Generate pages and components
```

现在，我们就可以用第一个命令 `start` 来创建我们的项目。

```
ionic start growth-blog-app --v2
```

在这个过程中，它将下载 **Ionic 2** 项目的基础项目，并执行安装命令。

```
Creating Ionic app in folder /Users/fdhuang/repractise/growth-
blog-app based on tabs project
Downloading: https://github.com/driftyco/ionic2-app-base/
archive/master.zip
[=====] 100% 0.0s
Downloading: https://github.com/driftyco/ionic2-starter-tabs/
archive/master.zip
[=====] 100% 0.0s
Installing npm packages...
```

然后到 `growth-blog-app` 目录，我们会看到类似于下面的内容：

```
.
├── README.md
├── app
│   ├── app.js
│   ├── pages
│   │   ├── page1
│   │   │   ├── page1.html
│   │   │   ├── page1.js
│   │   │   └── page1.scss
│   │   ├── page2
│   │   │   ├── page2.html
│   │   │   ├── page2.js
│   │   │   └── page2.scss
│   │   └── page3
│   │       ├── page3.html
│   │       ├── page3.js
│   │       └── page3.scss
```

```

|   |   L—— tabs
|   |       |—— tabs.html
|   |       L—— tabs.js
|   L—— theme
|       |—— app.core.scss
|       |—— app.ios.scss
|       |—— app.md.scss
|       |—— app.variables.scss
|       L—— app.wp.scss
|—— config.xml
|—— gulpfile.js
|—— hooks
|   |—— README.md
|   L—— after_prepare
|       L—— 010_add_platform_class.js
|—— ionic.config.json
|—— package.json
L—— www
    L—— index.html

```

在这 **2.0** 版本的 **Ionic**，页面开始以目录来划分，一个页面路径下有自己的 `html`、`js`、`scss`。

- `tabs` 负责这些页面间跳转
- `theme` 则负责系统相应样式的修改
- `config.xml` 带有相应的 **Cordova** 配置
- `hooks` 则对系统添加和编译时进行一些预处理
- `ionic.config.json` 则是 **ionic** 的一些相关配置选项
- `package.json` 则存放相应的 **node.js** 的包的依赖
- `www` 目录用于存放最后构建出来的内容，以及一些静态资源

由于 **Angular 2.0** 使用的是 **Typescript**，所以在这里我们将用 **typescript** 进行展示，因此我们的执行命令变成 ~~：

```
ionic start growth-blog-app --v2 --ts
```

--ts 表示使用的是 **typescript** 来创建项目，安装的过程是一样的，不一样的是后面写的代码。

执行相应的 `serve` 命令，我们就可以开始我们的项目了：

```
ionic serve
```

这时候 Ionic 将做一些额外的事，才能启动我们的服务，如：

- 删除 `www/build` 目录下的文件
- 编译 SASS 到 CSS
- 编译文件到 HTML
- 编译字体
- 等等

最后，它将启动一个 Web 服务，URL 为 <http://localhost:8100>

```
Running 'serve:before' gulp task before serve
[20:59:16] Starting 'clean'...
[20:59:16] Finished 'clean' after 6.07 ms
[20:59:16] Starting 'watch'...
[20:59:16] Starting 'sass'...
[20:59:16] Starting 'html'...
[20:59:16] Starting 'fonts'...
[20:59:16] Starting 'scripts'...
[20:59:16] Finished 'scripts' after 43 ms
[20:59:16] Finished 'html' after 51 ms
[20:59:16] Finished 'fonts' after 54 ms
[20:59:16] Finished 'sass' after 738 ms
7.6 MB bytes written (5.62 seconds)
[20:59:22] Finished 'watch' after 6.62 s
[20:59:22] Starting 'serve:before'...
[20:59:22] Finished 'serve:before' after 3.87 μs

Running live reload server: http://localhost:35729
Watching: www/**/*, !www/lib/**/*
√ Running dev server: http://localhost:8100
Ionic server commands, enter:

  restart or r to restart the client app from the root
  goto or g and a url to have the app navigate to the given url
```

```
consolelogs or c to enable/disable console log output
serverlogs or s to enable/disable server log output
quit or q to shutdown the server and exit
ionic $
```

接着，就可以打开相应的 Web 页面，如下图所示：

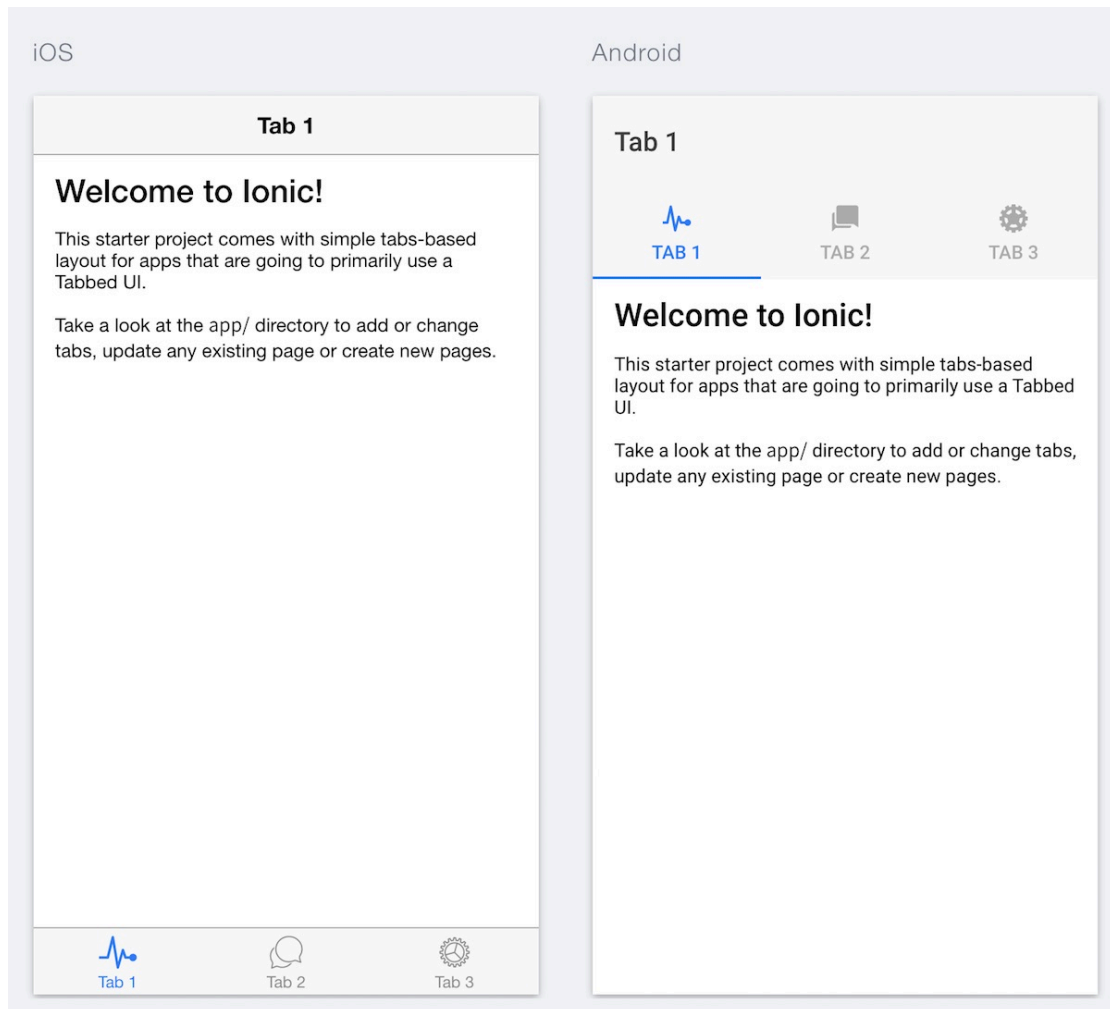


图 36: Ionic Web 预览界面

构建应用

由于 Ionic 是基于 Cordova 的，我们需要安装 Cordova 来完成后续的工作。

```
sudo npm install -g cordova
```

为了构建不同的平台的应用，我们就需要添加不同的平台，如：

```
ionic platform add android
```

上面的命令可以为项目添加 **Android** 平台的支持，过程如下面的日志所示：

```
Adding android project...
Creating Cordova project for the Android platform:
  Path: platforms/android
  Package: io.ionic.starter
  Name: V2_Test
  Activity: MainActivity
  Android target: android-23
Android project created with cordova-android@5.1.1
Running command: / Users/ fdhuang/ repractise/ growth- blog- app/
hooks/ after_prepare/ 010_add_platform_class.js / Users/ fdhuang/
repractise/growth-blog-app
```

最近，再执行 `run` 就可以在对应的平台上运行，如：

```
ionic run android
```

博客列表页

现在，让我们来结合我们的博客 **APP**，做一个相应的展示博客的 **APP**。在这一步我们所要做的事情比较简单：

- 获取博客列表 **API**
- 渲染博客列表

列表页

在上一个章节里我们已经有了一个博客详细的 **API**，我们只需要获取这个 **API** 并显示即可。不过，让我们简单地熟悉一下显示数据的这部分内容：

```
<ion-navbar *navbar>
  <ion-title>博客</ion-title>
</ion-navbar>
```



```

<ion-content class="blog-list">
  <ion-item *ngFor="#blogpost of blogposts">
    <h1 *ngIf="blogpost">
      {{blogpost.title}}
    </h1>

    <p *ngIf="blogpost">
      {{blogpost.body}}
    </p>
  </ion-item>
</ion-content>

```

上面是一个基本的详情页的模板，其中定义了一系列的 **Ionic** 自定义标签，如：

- 显示在导航栏中的内容
- 显示 APP 的内容
- 即将博客成每一项

而从上面的内容中，我们可以看到：我们在 **ngFor** 中遍历了 **blogposts**，然后显示每篇文章的标题和内容。对应的代码也就比较简单了：

```

import {Page} from 'ionic-angular';

@Page({
  templateUrl: 'build/pages/blog/list/index.html',
  providers: [BlogpostServices]
})
export class BlogList {
  public blogposts;

  constructor() {

  }
}

```

但是我们要去哪里获取博客的值呢，先我们我们看完改造后听 **BlogList** 的 **Controller**：

```
import {Page} from 'ionic-angular';
import {BlogpostServices} from '../../services/BlogpostServices';

@Page({
  templateUrl: 'build/pages/blog/list/index.html',
  providers: [BlogpostServices]
})
export class BlogList {
  private blogListService;
  public blogposts;

  constructor(blogpostServices:BlogpostServices) {
    this.blogListService = blogpostServices;
    this.initService();
  }

  private initService() {
    this.blogListService.getBlogpostLists().subscribe(
      data => {this.blogposts = JSON.parse(data._body);},
      err => console.log('Error: ' + JSON.stringify(err)),
      () => console.log('Get Blogpost')
    );
  }
}
```

我们初始化了一个 **blogListService**，然后我们调用这个服务去获取博客列表。

```
this.blogListService.getBlogpostLists().subscribe(
  data => {this.blogposts = JSON.parse(data._body);},
  err => console.log('Error: ' + JSON.stringify(err)),
  () => console.log('Get Blogpost')
);
```

当我们获取到数据的时候，我们就解析这个数据，并将这个值赋予 **blogposts**。如果这其中遇到什么错误，就会显示相应的错误信息。

现在，让我们创建一个获取博客的服务：

```
import {Inject} from 'angular2/core';
import {Http} from 'angular2/http';
import 'rxjs/add/operator/map';

export class BlogpostServices {
  private http;

  constructor(@Inject(Http) http:Http) {
    this.http = http
  }

  getBlogpostLists() {
    var url = 'http://127.0.0.1:8000/api/blogpost/?format=json';
    return this.http.get(url).map(res => res);
  }
}
```

我们将通过这个 **API** 来获取相关的数据，并将数据返回到 **BlogList** 类中。接着将更新 **blogposts** 的值，并重新渲染页面。

详情页

在我们的博客 **API** 中，每个内容都对应有一个 **id**，如下所示：

```
{
  "title": "这是一个标题",
  "author": "Phodal2",
  "body": "这是一个测试的内容",
  "slug": "this-is-a-test",
  "id": 3
}
```

我们只需要访问这个 **id**，就可以获取这个结果，如：<http://localhost:8000/api/blogpost/3/>

因此，我们所需要的就是：

- 在渲染博客列表的时候，为每一项赋予一个 **ID**

- 点击某一项时，将跳转到详情页，并去获取相应的 **API** 的数据，并渲染到页面上。

好了，我们可以用 **ionic** 的生成命令来创建博客详情页。

```
ionic g page blog-detail --ts
```

它将在 **app/pages** 目录下，生成下面的内容：

```
app/pages/blog-detail/  
├── blog-detail.html  
├── blog-detail.ts  
└── blog-detail.scss
```

我们可以遵循之前添加 **Django App** 的习惯，先添加 **Router**。因此我们可以在 **app.ts** 添加新的 **Route**：

```
const ROUTES = [  
  {path: '/app/blog/:id', component: BlogDetailPage}  
];  
  
@App({  
  template: '<ion-nav [root]="rootPage"></ion-nav>',  
  config: {}  
})  
@RouteConfig(ROUTES)  
export class MyApp {  
  rootPage:any = TabsPage;  
  
  constructor(platform:Platform) {  
    this.rootPage = TabsPage;  
    this.initializeApp(platform)  
  }  
  
  private initializeApp(platform:Platform) {  
    platform.ready().then(() => {  
      StatusBar.styleDefault();  
    })  
  }  
}
```

```
    });  
  }  
}
```

我们用 **RouteConfig** 来关联我们的 **URL** 和 **App Component**。

同上面的博客列表页面一样，我们也可以直接添加我们的 **API** 服务。有所区别的是，我们需要依据 **id** 去获取我们的博客内容。

```
getBlogpostDetail(id) {  
  var url = 'http://localhost:8000/api/blogpost/' + id + '?format=json';  
  return this.http.get(url).map(res => res);  
}
```

和之前的博客列表一样，我们需要几乎一样的方法来获取数据：

```
import {Page, NavController, NavParams} from 'ionic-angular';  
import {BlogpostServices} from "../../services/BlogpostServices";
```

```
@Page({  
  templateUrl: 'build/pages/blog-detail/blog-detail.html',  
  providers: [BlogpostServices]  
})  
export class BlogDetailPage {  
  private navParams;  
  private blogServices;  
  private blogpost;  
  
  constructor(public nav:NavController, navParams:NavParams, blogServices:BlogpostServices) {  
    this.nav = nav;  
    this.navParams = navParams;  
    this.blogServices = blogServices;  
  
    this.initService();  
  }  
}
```

```

private initService() {
  let id = this.navParams.get('id');
  this.blogServices.getBlogpostDetail(id).subscribe(
    data => {
      this.blogpost = JSON.parse(data._body);
      console.log(this.blogpost);
    },
    err => console.log('Error: ' + JSON.stringify(err)),
    () => console.log('Get Blogpost')
  );
}
}

```

现在我们几乎已经完成了博客详情页的工作，我们可以直接通过 URL 来访问博客详情页：<http://localhost:8100/#/app/blog/1>。结果如下图所示：



图 37: 访问博客详情页

不过，这时候我们的列表页并没有和详情页关联到一起。我们还需要做一些额外的工作：

- 在列表页的每一项中添加对点击事件的处理

在我们的模板页里 `ion-item` 里添加一个 `click` 事件，这个事件将调用 `navigate` 函数，并把博客 `id` 传到这个函数里。

```

<ion-item *ngFor="#blogpost of blogposts" (click)="navigate(blogpost.id)">
  <h1 *ngIf="blogpost">

```

```
        {{blogpost.title}}  
</h1>  
  
<p *ngIf="blogpost">  
    {{blogpost.body}}  
</p>  
</ion-item>
```

随后在我们的博客详情页的初始化里，我们要初始化一个 **NavController**:

```
constructor(nav: NavController, blogpostServices: BlogpostServices) {  
    this.blogListService = blogpostServices;  
    this.nav = nav;  
    this.initService();  
}
```

接着，在 **navigate** 里我们只需要将 **BlogDetailPage** 页面及参数 **push** 给 **navController**，并交由它来渲染页面。

```
navigate(id) {  
    this.nav.push(BlogDetailPage, {  
        id: id  
    });  
}
```

现在，我们可以试试从首页跳转到这个博客详情页。

Profile

现在，我们要做一个更有意思的东西了。不过这个内容是为后面的创建文章提供一个技术基础。在用户授权这一部分，我们使用不同的技术来实现，如 **Cookies**、**HTTP** 基本认证等等。而在手机端继续 **Cookie** 来进行用户授权，不是一件简单的事。因此我们就需要 **JSON Web Tokens**，这是一种基于 **token** 的认证方案。

Json Web Tokens

同样，为了实现这部分功能，我们仍然可以使用其他框架来帮助我们完成基础功能。这里我们就用到了一个名为 **djangorestframework-jwt** 的库，从它的名字上我们

就可以知道，它就是基于 **Django REST Framework** 之上的 **JWT** 实现。还是继续使用 **pip** 来安装这个库，记得把它添加到 `requirements.txt` 中。

```
pip install djangorestframework-jwt
```

接着，我们需要在我们的 URL 中配置用于获取 token 的 API 即可使用。

```
urlpatterns = patterns(
    '',
    # ...

    url(r'^api-token-auth/', 'rest_framework_jwt.views.obtain_jwt_token'),
)
```

在我们完成了上面的步骤之后，我们可以用 `curl` 命令或者 **Chrome** 浏览器的 **Postman** 来做测试：

- 向服务器发送我们的用户名和密码，获取对应的 Token。

如下是 curl 创建的请求，在这其中我们发送了我们的用户和密码。

```
curl -H "Content-Type: application/json" -X POST -d '{"username":"admin","password":"1234567890"}' http://localhost:8000/api-token-auth
```

然后服务端我们返回了对应的 **Token**，它可以用于后面的创建文章、获取用户信息等的功能。下面是一个 **Token** 的示例：

```
{"token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJlbWFpbCI6ImhAcGhvZGFsLmNvbS"}
```

登录表单

现在，我们要先做的一件事就是，创建一个用于登录的表单。

```
<form #loginCreds="ngForm" (ngSubmit)="login(loginCreds.value)">
  <ion-item>
    <ion-label>Username</ion-label>
    <ion-input type="text" ngControl="username"></ion-input>
```



```

</ion-item>

<ion-item>
  <ion-label>Password</ion-label>
  <ion-input type="password" ngControl="password"></ion-
input>
</ion-item>

<div padding>
  <button block type="submit">登 录</button>
</div>
</form>

```

我们创建一个名为 `loginCreds` 的 `ngForm`，在我们提交的时候我们就调用 `login` 方法，并把其中的值（`username`、`password`）传过去。而在我们的代码里，我们所要做的就是和上面一样将数据 `post` 到之前的 `Auth API` 的地址：

```

constructor(http: Http, nav:NavController) {
  this.nav = nav;
  this.http = http;
  this.local.get('id_token').then(
    (data) => {
      this.user = this.jwtHelper.decodeToken(data).username;
    }
  );
}

login(credentials) {
  this.contentType = new Headers({"Content-Type": "application/
json"});
  this.http.post(this.LOGIN_URL, JSON.stringify(credentials), {headers: this.
    .map(res => res.json())
    .subscribe(
      data => this.authSuccess(data.token),
      err => console.log(err)
    )
  );
}

```

```

}

authSuccess(token) {
  this.local.set('id_token', token);
  this.user = this.jwtHelper.decodeToken(token).username;
}

```

在我们成功的获取到 **Token** 的时候，保存这个 **Token**，并调用 **jwtHelper** 来解码 **Token**，并从中获取我们的 **username**。

同时，对于我们来说要登出就是一件容易的，删除这个 **token**，将清空用户名。

```

logout() {
  this.local.remove('id_token');
  this.user = null;
}

```

Profile

当我们获取到这个 **Token**，我们也可以顺便获取用户的用户名、邮件等等的信息给用户。我们所要做的就是再获取一次 **API**，但是在获取这次 **API** 的时候，我们需要上传我们的 **Token**。因此我们需要一个简单的 **AuthHelper** 来帮助我们。

AuthHttp 虽然我们要做的仅仅只是在我们的 **Header** 中，添加一个字段，它的值就是 **Token** 的值。但是这部分的逻辑交给 **Angular2-JWT** 来做可能会好一点，它提供了一个 **AuthHTTP** 方法可以让每次请求都带上这个 **Header**。首先我们需要安装这个库：

```
npm install angular2-jwt
```

然后在我们 **app.ts** 中添加这个 **provider**，并指明它的 **header** 前缀是 **JWT**。

```

@App({
  template: '<ion-nav [root]="rootPage"></ion-nav>',
  config: {}, // http://ionicframework.com/docs/v2/api/config/Config/
  providers: [
    provide(AuthHttp, {
      useFactory: (http) => {

```

```

        var authConfig = new AuthConfig({headerPrefix: 'JWT'});
        return new AuthHttp(authConfig, http);
    },
    deps: [Http]
  })
]
})

```

现在，我们就可以用和 **Http** 一样的方式去获取用户信息。

获取用户信息 现在我们所需要做的就是发出我们的 **API** 去获取用户的信息：

```

authSuccess(token) {
  this.local.set('id_token', token);
  this.user = this.jwtHelper.decodeToken(token).username;
  let params:URLSearchParams = new URLSearchParams();
  params.set('username', this.user);

  this.authHttp.request('http://localhost:8000/api/user/', {
    search: params
  })
  .map(res => res.text())
  .subscribe(
    data => this.local.set('user_info', JSON.stringify(JSON.parse(data)[0]))
    err => console.log(err)
  );
}

```

只是我们的 **API** 似乎还不支持这样的功能。它的实现方式和我们之前的 **AutoComplete** 是一样的，也是搜索用户名：

```

def list(self, request):
    search_param = self.request.query_params.get('username', None)
    if search_param is not None:
        queryset = User.objects.filter(username__contains=search_param)

```

```
serializer = UserSerializer(queryset, many=True)
return Response(serializer.data)
```

然后再显示这些数据:

```
<div *ngIf="auth.authenticated()">
  <div padding>
    <h1>Welcome, {{ user }}</h1>
    <h2 *ngIf="user_info">Last login {{user_info.last_login}}</h2>
    <button block (click)="logout()">Logout</button>
  </div>
</div>
```

不过由于我们 **Django** 自带的用户管理模块只有这点信息，我们也就只能显示这些信息了。下一步，我们就可以实现在我们的 **APP** 里去创建博客。

创建博客

在我们开始在 **APP** 端实现这个功能之前，我们先要实现一个高级点的用户授权管理——即只有用户登录或者用户的请求中带有 **Token** 的时候，我们才能创建博客。于是在这里我们所要做的就是实现一个 `IsAuthenticatedOrReadOnly`，来判断用户是否有权限，如果没有的话，那么只让用户看到博客的内容。代码如下所示：

```
SAFE_METHODS = ['GET', 'HEAD', 'OPTIONS']

class IsAuthenticatedOrReadOnly(BasePermission):
    """
    The request is authenticated as a user, or is a read-only request.
    """

    def has_permission(self, request, view):
        if (request.method in SAFE_METHODS or
            request.user and
            request.user.is_authenticated()):
            return True
        return False
```

```
class BlogpsotSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Blogpost
        fields = ('title', 'author', 'body', 'slug', 'id')
```

ViewSets define the view behavior.

```
class BlogpostSet(viewsets.ModelViewSet):
    permission_classes = (permissions.IsAuthenticatedOrReadOnly,)
    queryset = Blogpost.objects.all()
    serializer_class = BlogpsotSerializer
```

接着，我们可以创建一个 **Modal** 来做这个工作。对于我们的博客表单来说，和登录没有太大的区别。

```
<form #blogpostForm="ngForm" (ngSubmit)="create(blogpostForm.value)">
  <ion-item>
    <ion-label>标题</ion-label>
    <ion-input type="text" ngControl="title"></ion-input>
  </ion-item>

  <ion-item>
    <ion-label>作者</ion-label>
    <ion-input type="text" ngControl="author"></ion-input>
  </ion-item>

  <ion-item>
    <ion-label>URL</ion-label>
    <ion-input type="text" ngControl="slug"></ion-input>
  </ion-item>

  <ion-item>
    <ion-label>内容</ion-label>
    <ion-textarea type="text" ngControl="body"></ion-textarea>
  </ion-item>
```

```

    <div padding>
      <button block type="submit">创建</button>
    </div>

</form>

```

稍有不同的是在我们的标题栏里会有一个关闭按钮。

```

<ion-toolbar>
  <ion-title>
    创建博客
  </ion-title>
  <ion-buttons start>
    <button (click)="close()">
      <span primary showWhen="ios">取消</span>
      <ion-icon name="md-close" showWhen="android, windows"></
ion-icon>
    </button>
  </ion-buttons>
</ion-toolbar>

```

对于我们的实现代码来说，也是类似的，除了我们在发表成功的时候做的事情不一样——关闭这个 **Modal**。

```

close() {
  this.viewCtrl.dismiss();
}

create(value) {
  this.contentType = new Headers({"Content-Type": "application/
json"});
  this.authHttp.post('http://127.0.0.1:8000/api/blogpost/', JSON.stringify(
    ).map(res => res.json())
    .subscribe(
      data => this.postSuccess(data),
      err => console.log(err)
    )
  )
}

```

```

    );
}

postSuccess(data) {
    this.close()
}

```

同时，我们需要在我们的首页里添加这样的一个入口。

```

<button fab fab-bottom fab-right (click)="createBlog()" calm>
    <ion-icon name="add"></ion-icon>
</button>

```

以及它的处理逻辑：

```

createBlog() {
    let modal = Modal.create(CreateBlogModal);
    this.nav.present(modal)
}

```

Mobile Web

为了实现在移动设备上的访问，这里就以 **riot.js** 为例做一个简单的 **Demo**。不过，首先我们需要在后台判断用户是来自于某种设备，再对其进行特殊的处理。

移动设备处理

幸运的是我们又找到了一个库名为 `django_mobile`，可以根据用户的 **User-Agent** 来区别设备，并为其分配一个移动设备专用的模板。因此，我们需要安装这个库：

```
pip install django_mobile
```

并将 `'django_mobile.middleware.MobileDetectionMiddleware'` 和 `'django_mobile.middleware.SetFlavourMiddleware'` 添加 **MIDDLEWARE_CLASSES** 中：

```
MIDDLEWARE_CLASSES = (  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'corsheaders.middleware.CorsMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
    'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware',  
    'django_mobile.middleware.MobileDetectionMiddleware',  
    'django_mobile.middleware.SetFlavourMiddleware'  
)
```

修改 **Template** 配置，添加对应的 **loader** 和 **context_processor**，如下所示的内容即是修改完后的结果：

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [  
            'templates/'  
        ],  
        'LOADERS': [  
            'django_mobile.loader.Loader',  
            'django.template.loaders.filesystem.Loader',  
            'django.template.loaders.app_directories.Loader'  
        ],  
        'APP_DIRS': True,  
        'OPTIONS': {  
            'context_processors': [  
                'django.template.context_processors.debug',  
                'django.template.context_processors.request',  
                'django.contrib.auth.context_processors.auth',  
                'django.contrib.messages.context_processors.messages',  
                'django_mobile.context_processors.flavour'
```



```

        ],
      },
    },
  ],
]

```

我们在 **LOADERS** 中添加了 `'django_mobile.loader.Loader'`，在 `context_processors` 中添加了 `django_mobile.context_processors.flavour`。

然后在 **template** 目录中创建 `template/mobile/index.html` 文件，即可。

前后端分离

为了方便我们讲述模块化，也不改变系统原有架构，我决定挖个大坑使用 **Riot.js** 来展示这一部分的内容。

Riot.js

Riot 拥有创建现代客户端应用的所有必需的成分：

- “响应式”视图层用来创建用户界面
- 用来在各独立模块之间进行通信的事件库
- 用来管理 **URL** 和浏览器回退按钮的路由器 (**Router**)

等等。

接着让我们引入 **riot.js** 这个库，顺便也引入 **rxjs** 吧：

```

<script src="{% static 'js/mobile/riot+compiler.min.js' %}"></script>
<script src="{% static 'js/mobile/rx.core.min.js' %}"></script>

```

ReactiveJS 构建服务

由于我们所要做的服务比较简单，并且我们也更愿意使用 **Promise** 来加载 **API** 服务，因此我们引入了这个库来加速我们的开发。下面是我们用于获取博客 **API** 的代码：

```

var responseStream = function (blogId) {
  var url = '/api/blogpost/?format=json';

```

```
if(blogId) {
    url = '/api/blogpost/' + blogId + '?format=json';
}

return Rx.Observable.create(function (observer) {
    jQuery.getJSON(url)
        .done(function (response) {
            observer.onNext(response);
        })
        .fail(function (jqXHR, status, error) {
            observer.onError(error);
        })
        .always(function () {
            observer.onCompleted();
        });
});
};
```

当我们想访问特定博客的时候，我们就传博客 ID 进去——这时会使用 `'/api/blogpost/' + blogId + '?format=json'` 作为 URL。接着我们创建了创建自己定制的事件流——使用 jQuery 去获取 API：

- 成功的时候 (**done**)，我们将用 `onNext()` 来通知观察者
- 失败的时候 (**fail**)，我们就调用 `onError()` 来通知观察者
- 不论成功或者失败，都会执行 `always`

在使用的时候，我们只需要调用其 `subscribe` 方法即可：

```
responseStream().subscribe(function (response) {

})
```

创建博客列表页

现在，我们可以修改原生的博客模板，将其中的 `container` 内容变为：

```
<div class="container" id="container">
  <blog></blog>
</div>
```

接着，我们可以创建一个 `blog.tag` 文件，添加加载这个文件：

```
<script src="{% static 'riot/ blog.tag' %}" type="riot/ tag"></script>
```

为了调用这个 **tag** 的内容，我们需要在我们的 `main.js` 加上一句：

```
riot.mount("blog");
```

随后我们可以在我们的 **tag** 文件中，来对 **blog** 的内容进行操作。

```
<blog class="row">
  <div class="col-sm-4" each={ opts }>
    <h2><a href="#/blogDetail/{id}" onclick={ parent.click }>{ title }</a>
    { body }
    { posted } - By { author }
  </div>
  <script>
    var self = this;
    this.on('mount', function (id) {
      responseStream().subscribe(function (response) {
        self.opts = response;
        self.update();
      })
    })

    click(event)
    {
      this.unmount();
      riot.route("blogDetail/" + event.item.id);
    }
  </script>
</blog>
```

在 **Riot** 中，变量默认是以 **opts** 的方式传递起来的，因此我们也遵循这个方式。在模板方面，我们遍历每个博客取出其中的内容：

```
<div class="col-sm-4" each={ opts }>
  <h2><a href="#/blogDetail/{id}" onclick={ parent.click }>{ title }</a></h2>
  { body }
  { posted } - By { author }
</div>
```

而博客的数据需要依赖于我们监听 **mount** 事件才会去获取——即我们加载了这个 **tag**。

```
this.on('mount', function (id) {
  responseStream().subscribe(function (response) {
    self.opts = response;
    self.update();
  })
})
```

在这个页面中，还有一个单击事件 **onclick={ parent.click }**，即当我们点击某个博客的标题时执行的函数：

```
click(event)
{
  this.unmount();
  riot.route("blog/" + event.item.id);
}
```

我们将卸载当前的 **tag**，然后加载 **blogDetail** 的内容。

博客详情页

在我们加载之前，我们需要先配置好 **blogDetail**。我们仍然使用正规表达式 **blogDetail/*** 来获取博客的 **id**：

```
riot.route.base('#');
```

```
riot.route('blog/*', function(id) {
    riot.mount("blogDetail", {id: id})
});

riot.route.start();
```

然后将由相应的 **tag** 来执行：

```
<blogDetail class="row">
    <div class="col-sm-4">
        <h2>{ opts.title }</h2>
        { opts.body }
        { opts.posted } - By { opts.author }
    </div>
    <script>
        var self = this;

        this.on('mount', function (id) {
            responseStream(this.opts.id).subscribe(function (response) {
                self.opts = response;
                self.update();
            })
        })
    </script>
</blogDetail>
```

同样的，我们也将去获取这篇博客的内容，然后显示。

添加导航

在上面的例子里，我们少了一部分很重要的内容就是在页面间跳转，现在就让我们来创建 `navbar.tag` 吧。

首先，我们需要重新规则一下 **route**，在系统初始化的时候我们将使用的路由是 **blog**，在进入详情页的时候，我们用 **blog/***。

```
riot.route.base('#');

riot.route('blog/*', function (id) {
    riot.mount("blogDetail", {id: id})
});

riot.route('blog', function () {
    riot.mount("blog")
});

riot.route.start();

riot.route("blog");
```

然后将我们的 **navbar** 标签放在 `blog` 和 `blogDetail` 中，如下所示：

```
<blogDetail class="row">
    <navbar title="{ opts.title }"></navbar>
    <div class="col-sm-4">
        <h2>{ opts.title }</h2>
        { opts.body }
        { opts.posted } - By { opts.author }
    </div>
</blogDetail>
```

当我们到了博客详情页，我们将把标题作为参数传给title。接着，我们在navbar中我们

```
    <li><a href="#" onclick={parent.clickTitle}>Home</a></li>
    <li if="opts.title">{ opts.title } </li>
</ol>
```

最后可以在我们的`blogDetail`标签中添加一个点击事件来跳转到首页：

```
clickTitle(event) { self.unmount(true); riot.route("blog"); } “
```

配置管理

local settings

作为一个开源项目，我们在这方面做得并不是特别好——当然是有意如此的。不过，这里我们还是做一些简单的介绍。对于我们的项目来说，我们需要一些额外的配置，如我们的数据库中的 `DATABASES`、`DEFAULT_AUTHENTICATION_CLASSES`、`CORS_ORIGIN_ALLOW_ALL`、`SECRET_KEY` 应该在不同的环境中都有不同的配置。

我们可以一个创建 `local_settings.py`，在里面放置一些关键的服务器相关的配置，如：

```
# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'hpi!zb8!(j%40)r55@+_5k*^9qcjf9sx0o_it*jlp3=x9^2ak@'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

TEMPLATE_DEBUG = True

# Database
# https://docs.djangoproject.com/en/1.7/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'db.sqlite3',
    }
}

REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (

    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.SessionAuthentication',
```

```
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework_jwt.authentication.JSONWebTokenAuthentication',
    ),
}

CORS_ORIGIN_ALLOW_ALL = True
```

接着，我们只需要在我们的主 `settings.py` 中引用即可：