Getting Started With Django: A Crash Course Documentation

Release 1.0

Kenneth Love

Sep 22, 2017

Contents

CHAPTER 1

Prerequisites:

- A text editor or IDE
- A terminal
- Git
- Python 2.7
- A Heroku account and the Heroku toolbelt installed
- Completed the Polls tutorial

Contents:

virtualenv

We want at least `virtualenv` installed so we don't have to pollute our global site-packages with our project-specific packages. This also lets us use our `requirements.txt` file locally and on Heroku when we deploy later.

`pip install virtualenv` will install `virtualenv` for us. You may have to `sudo` this command or tell `pip` to install to your user space with the `--user` argument.

virtualenvwrapper

If you're on a compatible system, install `virtualenvwrapper` with `pip install virtualenvwrapper` and add:

```
export WORKON_HOME=$HOME/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

to whatever config file your shell uses (e.g. `~/.bashrc` or `~/.zshrc`). You may then need to restart your terminal or source the config file to make this active.

Make the `virtualenv`

Now we want to actually create the `virtualenv`.

With `virtualenv`:

```
virtualenv pycon-venv
source pycon-venv/bin/activate
```

With `virtualenvwrapper`:

```
mkvirtualenv pycon
workon pycon
```

Either way, your prompt should now change to show `(pycon-venv)` or `(pycon)`.

Django

Now that we have an active `virtualenv`, we need to install Django. `pip install django==1.6.2` will install the version of Django that we want for this project and give us the `django-admin.py` command. To start our project, we then run `django-admin.py startproject survivalguide`. Then `cd` into the `survivalguide` directory.

Git

This directory (`pycon/survivalguide/`) is where we want the base of our project to be as far as git and Heroku are concerned, so we'll go ahead and do a `git init`. We also should add the following `.gitignore` file:

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

Database

For the purposes of this demo, we aren't going to use a *real* database like Postgres locally so we don't have to install `psycopg2`. We'll stick with SQLite3, but feel free to swap it out for a local Postgres database if you want.

We do need to run `python manage.py syncdb` to get our default tables set up. Go ahead and create a superuser, too.

Template Dirs

We'll need some site-wide templates before long so we'll create a directory to hold them all with `mkdir templates`. We need to add that to `survivalguide/settings.py` as such:

```
TEMPLATE_DIRS = (
    os.path.join(BASE_DIR, 'templates'),
)
```

We have to be sure and include the trailing comma since `TEMPLATE_DIRS` must be a tuple.

Global Layouts

My convention for site-wide templates (and partials, both site-wide and app-specific) is to prepend the file or directory name with an _, so inside `templates` make a new directory named `_layouts`.

Inside there, we need to touch `base.html` and give it the following code:

```
<!DOCTYPE>
<html>
<head>
    <title>{% block title %}PyCon Survival Guide{% endblock title %}</title>
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.1.1/css/
↪bootstrap.min.css">
    <style>
        body {
            padding-bottom: 20px;
            padding-top: 70px;
        }
        .messages {
            list-style: none;
        }
    </style>
    {% block css %}{% endblock css %}
</head>
<body>
    <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="#">PyCon Survival Guide</a>
        </div>
        <div class="navbar-collapse collapse">
        </div><!--/.navbar-collapse -->
      </div>
    </div>
    <div class="jumbotron">
        <div class="container">{% block headline %}{% endblock headline %}</div>
    </div>
    <div class="container">
        {% block content %}{% endblock content %}
    </div>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></
↪script>
    {% block js %}{% endblock js %}
</body>
</html>
```

Auth views

HomeView

Before we start any authentication views, we should probably have a home page. So, let's make one. Our stub, `pycon/survivalguide/survivalguide/` doesn't have a `views.py`, so let's go ahead and create it with `touch survivalguide/views.py`. Let's create our first view here:

```
from django.views import generic


class HomePageView(generic.TemplateView):
    template_name = 'home.html'
```

Template

Now we need to `touch templates/home.html` and open it up for editing. It'll be a pretty simple view so let's just put the following into it:

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

{% block headline %}<h1>Welcome to the PyCon Survival Guide!</h1>{% endblock headline
↪%}

{% block content %}
<p>Howdy{% if user.is_authenticated %} {{ user.username }}{% endif %}!</p>
{% endblock %}
```

URL

Finally, we need an URL so we can reach the view. In `survivalguide/urls.py`, add the following:

```
[...]
from .views import HomePageView
[...]

url('^$', HomePageView.as_view(), name='home'),
```

Now any time we go to `/` on our site, we'll get our template.

SignUpView

Now, we need to make a view for users to be able to signup at. Let's update our `survivalguide/views.py` file like so:

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


class SignUpView(generic.CreateView):
    form_class = UserCreationForm
    model = User
    template_name = 'accounts/signup.html'
```

URL

Since we want to be able to get to the view from a URL, we should add one to `survivalguide/urls.py`.

```
[...]
from .views import SignUpView
[...]

url(r'^accounts/register/$', SignUpView.as_view(), name='signup'),
[...]
```

Template

Since we told the view that the template was in an `accounts` directory, we need to make one in our global `templates` directory. We have to make this directory because `accounts` isn't an app. `mkdir templates/accounts` and then `touch templates/accounts/signup.html`.

`signup.html` should look like:

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

{% block title %}Register | {{ block.super }}{% endblock %}

{% block headline %}<h1>Register for the PyCon Survival Guide</h1>{% endblock %}

{% block content %}
    <form action='' method="POST">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Sign Up">
    </form>
{% endblock %}
```

This default form doesn't render the most beautiful HTML and, thinking about our future forms, we'll have to do a lot of HTML typing just to make them work. None of this sounds like fun work and we're not using a Python web framework in order to have to write a bunch of HTML, so let's save ourselves some time and trouble by using `django-crispy-forms`.

django-crispy-forms

Like pretty much everything, first we need to install `django-crispy-forms` with `pip install django-crispy-forms`. Then we need to add `'crispy_forms'` to `INSTALLED_APPS` in our settings file and provide a new setting:

```
CRISPY_TEMPLATE_PACK = 'bootstrap3'
```

We have to tell `django-crispy-forms` what set of templates to use to render our forms.

New form

`touch survivalguide/forms.py` and open it in your editor. We need to create a new, custom form, based off of Django's default `UserCreationForm`.

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

from crispy_forms.helper import FormHelper
from crispy_forms.layout import Layout, ButtonHolder, Submit

class RegistrationForm(UserCreationForm):
    def __init__(self, *args, **kwargs):
        super(RegistrationForm, self).__init__(*args, **kwargs)

        self.helper = FormHelper()
        self.helper.layout = Layout(
            'username',
            'password1',
            'password2',
            ButtonHolder(
                Submit('register', 'Register', css_class='btn-primary')
            )
        )
```

View changes

In `survivalguide/views.py`, we need to change our form import from:

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

to

```
from .forms import RegistrationForm
```

Since we're using relative imports, we should add:

```
from __future__ import absolute_import
```

to the top of the file to ensure that our imports work like we want.

Change the `form_class` in the view to `RegistrationForm` and the view should be done.

Template change

Finally, in the template, change the `<form>` area to be: `{% crispy form %}` and load the `django-crispy-forms` tags with `{% load crispy_forms_tags %}` near the top of the file. If we refresh the page, we should now see a decent looking form that works to sign up our user.

LogInView

Most of `LogInView` is the same work as `SignUpView`. Since we know we're going to need a custom form, because we want to use `django-crispy-forms`, let's start there.

Form

Back in `survivalguide/forms.py`:

```
from django.contrib.auth.forms import AuthenticationForm


class LoginForm(AuthenticationForm):
    def __init__(self, *args, **kwargs):
        super(LoginForm, self).__init__(*args, **kwargs)

        self.helper = FormHelper()
        self.helper.layout = Layout(
            'username',
            'password',
            ButtonHolder(
                Submit('login', 'Login', css_class='btn-primary')
            )
        )
```

View

Then, we should create a view.

```
[...]
from django.contrib.auth import authenticate, login, logout
from django.core.urlresolvers import reverse_lazy
[...]
from .forms import LoginForm
[...]

class LoginView(generic.FormView):
    form_class = LoginForm
    success_url = reverse_lazy('home')
    template_name = 'accounts/login.html'

    def form_valid(self, form):
        username = form.cleaned_data['username']
        password = form.cleaned_data['password']
        user = authenticate(username=username, password=password)

        if user is not None and user.is_active:
            login(self.request, user)
            return super(LoginView, self).form_valid(form)
        else:
            return self.form_invalid(form)
```

URL

In our `survivalguide/urls.py` file, we need to add a route to our new login view.

```
from .views import LoginView
[...]
url(r'^accounts/login/$', LoginView.as_view(), name='login'),
[...]
```

Template

And, of course, since we gave our view a template name, we have to make sure the template exists. So, in `templates/accounts/` go ahead and touch `login.html` and fill the file with:

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

{% load crispy_forms_tags %}

{% block title %}Login | {{ block.super }}{% endblock %}

{% block headline %}<h1>Login to the PyCon Survival Guide</h1>{% endblock %}

{% block content %}
{% crispy form %}
{% endblock %}
```

LogOutView

We should also provide a quick and easy way for users to log out. Thankfully Django makes this pretty simple and we just need a view and an URL.

View

In `survivalguide/views.py`:

```
class LogOutView(generic.RedirectView):
    url = reverse_lazy('home')

    def get(self, request, *args, **kwargs):
        logout(request)
        return super(LogOutView, self).get(request, *args, **kwargs)
```

URL

And in our `survivalguide/urls.py`, we'll import the new view and create an URL:

```
[...]
from .views import LogOutView
[...]

url(r'^accounts/logout/$', LogOutView.as_view(), name='logout'),
[...]
```

Global template changes

Finally, though, we should have the ability to see if we're logged in or not, and have some links for logging in, signing up, and logging out. So open up `templates/_layouts/base.html` and add the following to the `.navbar-collapse` area:

```
{% if not user.is_authenticated %}
<a href="{% url 'signup' %}" class="btn btn-default navbar-btn">Register</a>
<a href="{% url 'login' %}" class="btn btn-default navbar-btn">Login</a>
{% else %}
<a href="{% url 'logout' %}" class="btn btn-default navbar-btn">Logout</a>
{% endif %}
```

django-braces

Our views are complete and pretty solid but it's a little weird that logged-in users can go to the login view and signup view and that logged-out users can go to the logout view. It would also be nice to send the users messages when something happens. Writing code to do all of these things is easy enough but there are already packages out there that provide this functionality. Namely `django-braces`.

As usual, install it with `pip install django-braces`. Since `braces` doesn't provide any models or templates, we don't have to add it to `INSTALLED_APPS`, but, as we want to show messages, we should update our `base.html` file to provide a place for them.

Messages

Open up `templates/_layouts/base.html` and add:

```
{% if messages %}
<ul class="messages">
    {% for message in messages %}
    <li{% if message.tags %} class="alert alert-{{ message.tags }}"{% endif %}>{{␣
↪message }}</li>
    {% endfor %}
</ul>
{% endif %}
```

before the `.jumbotron` area. This snippet will show any messages in the session in a way that Bootstrap expects.

Views

Now, back in `survivalguide/views.py`, we need to import `django-braces`, so add:

```
from braces import views
```

to the imports area near the top of the file. We need to add a few mixins and attributes to a few of the views, so let's do that now.

SignUpView

Add `views.AnonymousRequiredMixin` and `views.FormValidMessageMixin` to the class's signature. We should also add a `form_valid_message` attribute to the class which'll be shown to the user when they have successfully signed up.

The `AnonymousRequiredMixin` prevents authenticated users from accessing the view.

LogInView

Add the same two mixins to this view as well and set a `form_valid_message` that tells the user that they're logged in.

LogOutView

`LogOutView` needs the `views.LoginRequiredMixin` and the `views.MessageMixin` added to it.

The `LoginRequiredMixin` prevents this view from being accessed by anonymous users.

We also need to update the `get` method on the view and add:

```
self.messages.success("You've been logged out. Come back soon!")
```

to it before the `super()` call.

Now all of our views should be properly protected and give useful feedback when they're used.

The `talks` app

Now we can get to the meat of our project, the talks app.

`startapp`

To start our app, we can to tell Django to give us some boilerplate right away with `python manage.py startapp talks`. We want app names to be plural, generally, as they usually concern themselves with multiple model instances and work around them.

This will give us a structure similar to:

```
/talks
- __init__.py
- admin.py
- models.py
- tests.py
- views.py
```

We'll start off with the `models.py` in here.

Note: I'll be giving paths relative to the `talks/` directory from here on out, so be sure to adjust them in your head as needed. Most text editors seem to offer a fuzzy file finder now, so editing should be fairly painless.

`TalkList` model

We want to be able to organize our talks into lists, things like "Attend or Else" and "Watch Online". So the first thing we should probably have is a model for the list. Open up `models.py` and add the following:

```
from django.contrib.auth.models import User
from django.core.urlresolvers import reverse
from django.db import models
from django.template.defaultfilters import slugify


class TalkList(models.Model):
    user = models.ForeignKey(User, related_name='lists')
    name = models.CharField(max_length=255)
    slug = models.SlugField(max_length=255, blank=True)

    class Meta:
        unique_together = ('user', 'name')

    def __unicode__(self):
        return self.name

    def save(self, *args, **kwargs):
        self.slug = slugify(self.name)
        super(TalkList, self).save(*args, **kwargs)

    def get_absolute_url(self):
        return reverse('talks:lists:detail', kwargs={'slug': self.slug})
```

Our model ties `TalkList` instances to a user, makes sure each list has a unique name per user, runs `slugify` on the name so we can use it in our URL, and provides an URL to get an individual list. But, what about that URL? What's with the colons in it?

URLs

To make our URLs work like that, we need to set up three things and bring in a couple of namespaces.

First, let's make a placeholder in our `views.py` file.

```
from django.http import HttpResponse
from django.views import generic


class TalkListDetailView(generic.View):
    def get(self, request, *args, **kwargs):
        return HttpResponse('A talk list')
```

Now, we need to create `urls.py` inside `talks/` and add the followng:

```
from __future__ import absolute_import

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

from . import views


lists_patterns = patterns(
    '',
    url(r'^$', views.TalkListDetailView.as_view(), name='detail'),
)

urlpatterns = patterns(
```

```
    '',
    url(r'^lists/', include(lists_patterns, namespace='lists')),
)
```

This line sets up an internal namespace of `lists` for all of our `TalkList`-specific URLs. Now we need to add the `talks` namespace that our `get_absolute_url` mentioned.

Open up `survivalguide/urls.py` and add:

```
url(r'^talks/', include('talks.urls', namespace='talks')),
```

This sets up the `talks` namespace.

south

Now, before we actually create the model, we should add `south` into the mix. `pip install south` will install it and we need to add `'south'` to our `INSTALLED_APPS`. Since `south` has a model of its own, we also need to run `python manage.py syncdb` again to add it.

We should now add our `'talks'` app to `INSTALLED_APPS` and, instead of running `syncdb`, we should run `python manage.py schemamigration --initial talks`. `south` will create a migration that generates our database table and put it in the `migrations/` directory. Then we apply it with `python manage.py migrate`.

Note: `python manage.py schemamigration` is a really long command to have to type repeatedly, so I recommend creating a shell alias for it to save yourself some time.

Warning: Django 1.7 introduces an internal migration tool much like `south`. This tutorial does not cover that tool. While `south` will likely work with Django 1.7, you should use the new tool instead.

Default list

Now that we have a model and a database table, let's make our `SignUpView` automatically create a default list for everyone. Open `survivalguide/views.py` and change `SignUpView` to match this:

```
[...]
from talks.models import TalkList
[...]

 class SignUpView(views.AnonymousRequiredMixin, views.FormValidMessageMixin,
                  generic.CreateView):
     form_class = RegistrationForm
     form_valid_message = "Thanks for signing up! Go ahead and login."
     model = User
     success_url = reverse_lazy('login')
     template_name = 'accounts/signup.html'

     def form_valid(self, form):
         resp = super(SignUpView, self).form_valid(form)
         TalkList.objects.create(user=self.object, name='To Attend')
         return resp
```

TalkList Views

Our `TalkLists` need a few different views, so let's look at creating those.

TalkListListView

We should have a view to show all of our lists. In `views.py`:

```
[...]
from braces import views

from . import models

class TalkListListView(views.LoginRequiredMixin, generic.ListView):
    model = models.TalkList

    def get_queryset(self):
        return self.request.user.lists.all()
```

There's not really anything fancy about this view other than overriding `get_queryset`. We want users to only be able to view lists that they own, so this does that for us. We didn't specify a template, so Django will look for the default one at `talks/talklist_list.html`.

Template

Since this is an app and the templates are only for this app, I think it's best to put them in the app. This makes it easier to focus on the files for a specific app and it also makes it easier to make an app reusable elsewhere.

```
mkdir -p talks/templates/talks
touch talks/templates/talks/talklist_list.html
```

We need to namespace the template inside of a directory named the same as our app. Open up the template file and add in:

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

{% block title %}Lists | {{ block.super }}{% endblock title %}

{% block headline %}<h1>Your Lists</h1>{% endblock headline %}

{% block content %}
<div class="row">
    <div class="col-sm-6">
        <ul class="list-group">
            {% for object in object_list %}
            <li class="list-group-item">
                <a href="{{ object.get_absolute_url }}">{{ object }}</a>
            </li>
            {% empty %}
            <li class="list-group-item">You have no lists</li>
            {% endfor %}
        </ul>
    </div>
    <div class="col-sm-6">
        <p><a href="#" class="btn">Create a new list</a></p>
```

```
    </div>
</div>
{% endblock %}
```

URL

Now in our `urls.py` file, we need to update our `list_patterns` set of patterns.

```
[...]
url(r'^$', views.TalkListListView.as_view(), name='list'),
url(r'^d/(?P<slug>[-\w]+)/$', views.TalkListDetailView.as_view(),
    name='detail'),
[...]
```

You'll notice that we replaced our old default URL (`r'^$'`) with our `TalkListListView` and put the `TalkListDetailView` under a new regex that captures a slug. Our model's `get_absolute_url` should still work fine.

TalkListDetailView

Let's build out our actual detail view now. Back in `views.py`:

```
class TalkListDetailView(
    views.LoginRequiredMixin,
    views.PrefetchRelatedMixin,
    generic.DetailView
):
    model = models.TalkList
    prefetch_related = ('talks',)

    def get_queryset(self):
        queryset = super(TalkListDetailView, self).get_queryset()
        queryset = queryset.filter(user=self.request.user)
        return queryset
```

This mixin from `django-braces` lets us do a `prefetch_related` on our queryset to, hopefully, save ourselves some time in the database. Again, we didn't specify a template so we'll make one where Django expects.

Template

Create the file `talks/templates/talks/talklist_detail.html` and add in:

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

{% block title %}{{ object.name }} | Lists | {{ block.super }}{% endblock title %}

{% block headline %}
<h1>{{ object.name }}</h1>
<h2>Your Lists</h2>
{% endblock headline %}

{% block content %}
<div class="row">
    <div class="col-sm-6">
```

```
        <p>Talks go here</p>
    </div>

    <div class="col-sm-6">
        <p><a href="{% url 'talks:lists:list' %}">Back to lists</a></p>
    </div>
</div>
{% endblock %}
```

A pretty standard Django template. We already have the URL so this should be completely wired up now.

RestrictToUserMixin

We had to override `get_queryset` in both of our views, which is kind of annoying. It would be nice to not have to do that, especially two different ways both times. Let's write a custom mixin to do this work for us.

```
class RestrictToUserMixin(views.LoginRequiredMixin):
    def get_queryset(self):
        queryset = super(RestrictToOwnerMixin, self).get_queryset()
        queryset = queryset.filter(user=self.request.user)
        return queryset
```

This does the same work as our `get_queryset` in `TalkListDetailView`. Let's use it. In both views, add `RestrictToUserMixin` and take out the `views.LoginRequiredMixin` from `django-braces` since our new mixin provides that functionality too. Also remove the overrides of `get_queryset` from both views.

TalkListCreateView

We want to be able to create a new `TalkList`, of course, so let's create a `CreateView` for that. In `views.py`, still, add a new class:

```
1  [...]
2  import .forms
3
4  class TalkListCreateView(
5      views.LoginRequiredMixin,
6      views.SetHeadlineMixin,
7      generic.CreateView
8  ):
9      form_class = forms.TalkListForm
10     headline = 'Create'
11     model = models.TalkList
12
13     def form_valid(self, form):
14         self.object = form.save(commit=False)
15         self.object.user = self.request.user
16         self.object.save()
17         return super(TalkListCreateView, self).form_valid(form)
```

This view has a `form_class` that we haven't created yet, so we'll need to do that soon. Also, we override `form_valid`, which is called when the submitted form passes validation, and in there we create an instance in memory, assign the current user to the model instance, and then save for real and call `super()` on the method.

This view also brings in the `SetHeadlineMixin` and provides the `headline` attribute. We do this because we'll be using the same template for both create and update views and we don't want them to both have the same title and

headline. This way we can control that from the view instead of having to create new templates all the time.

Let's create the form now.

TalkListForm

We don't yet have a `talks/forms.py` so go ahead and create it with the following content:

```
1  from __future__ import absolute_import
2
3  from django import forms
4
5  from crispy_forms.helper import FormHelper
6  from crispy_forms.layout import Layout, ButtonHolder, Submit
7
8  from . import models
9
10
11 class TalkListForm(forms.ModelForm):
12     class Meta:
13         fields = ('name',)
14         model = models.TalkList
15
16     def __init__(self, *args, **kwargs):
17         super(TalkListForm, self).__init__(*args, **kwargs)
18         self.helper = FormHelper()
19         self.helper.layout = Layout(
20             'name',
21             ButtonHolder(
22                 Submit('create', 'Create', css_class='btn-primary')
23             )
24         )
```

Nothing really different here from our earlier forms except for line 13 which restricts the fields that the form cares about to just the `name` field.

URL

As with all of our other views, we need to make an URL for creating lists. In `talks/urls.py`, add the following line:

```
url(r'^create/$', views.TalkListCreateView.as_view(), name='create'),
```

Template

And, again, we didn't specify a specific template name so Django expects `talks/talklist_form.html` to exist. Django will use this form for both `CreateView` and `UpdateView` views that use the `TalkList` model unless we tell it otherwise.

```
{% extends '_layouts/base.html' %}
{% load crispy_forms_tags %}

{% block title %}{{ headline }} | Lists | {{ block.super }}{% endblock title %}

{% block headline %}
```

```
<h1>{{ headline }}</h1>
<h2>Your Lists</h2>
{% endblock headline %}

{% block content %}
{% crispy form %}
{% endblock content %}
```

You can see here were we use the `{{ headline }}` context item provided by the `SetHeadlineMixin`. Now users should be able to create new lists.

TalkListUpdateView

Anything we can create, we should be able to update so let's create a `TalkListUpdateView` in `views.py`.

```
class TalkListUpdateView(
    RestrictToOwnerMixin,
    views.LoginRequiredMixin,
    views.SetHeadlineMixin,
    generic.UpdateView
):
    form_class = forms.TalkListForm
    headline = 'Update'
    model = models.TalkList
```

There isn't anything in this view that we haven't covered already. All that's left for it is to create the URL pattern.

URL

You should be getting the hang of this by now, so let's just add this line to our `urls.py`:

```
url(r'^update/(?P<slug>[-\w]+)/$', views.TalkListUpdateView.as_view(),
    name='update'),
```

Global template changes

It's great that we've created all of these views but now there's no easy way to get to your views. Let's fix that by adding the following into `_layouts/base.html` next to our other navigation items inside the `{% else %}` clause:

```
<a href="{% url 'talks:lists:list' %}" class="btn btn-primary navbar-btn">Talk lists</
↪a>
```

App template changes

Our `talks/talklist_list.html` template should have a link to the `TalkListCreateView` so let's add that into the sidebar:

```
<p><a href="{% url 'talks:lists:create' %}" class="btn">Create a new list</a></p>
```

DeleteView?

So what about a `DeleteView` for `TalkList`? I didn't make one for this example but it shouldn't be too hard of an exercise for the reader. Be sure to restrict the queryset to the logged-in user.

Talks model

We need a model for our actual talks which will belong to a list. Eventually we'll want ratings and notes and such, but let's start with a simple model.

Model, take one

```
from django.core.urlresolvers import reverse
from django.db import models
from django.template.defaultfilters import slugify
from django.utils.timezone import utc


class Talk(models.Model):
    ROOM_CHOICES = (
        ('517D', '517D'),
        ('517C', '517C'),
        ('517AB', '517AB'),
        ('520', '520'),
        ('710A', '710A')
    )
    talk_list = models.ForeignKey(TalkList, related_name='talks')
    name = models.CharField(max_length=255)
    slug = models.SlugField(max_length=255, blank=True)
    when = models.DateTimeField()
    room = models.CharField(max_length=10, choices=ROOM_CHOICES)
    host = models.CharField(max_length=255)

    class Meta:
        ordering = ('when', 'room')
        unique_together = ('talk_list', 'name')

    def __unicode__(self):
        return self.name

    def save(self, *args, **kwargs):
        self.slug = slugify(self.name)
        super(Talk, self).save(*args, **kwargs)
```

Like with our `TalkList` model, we want to `slugify` the name whenever we save an instance. We also provide a tuple of two-tuples of choices for our `room` field, which makes sure that whatever talks get entered all have a valid room and saves users the trouble of having to type the room number in every time.

Also, we want default ordering of the model to be by when the talk happens, in ascending order, and then by room number.

Migration

Since we've added a model, we need to create and apply a migration.

```
python manage.py schemamigration --auto talks
python manage.py migrate talks
```

Form

We should create a form for creating talks. In `forms.py`, let's add `TalkForm`:

```
import datetime

from django.core.exceptions import ValidationError
from django.utils.timezone import utc
[...]

class TalkForm(forms.ModelForm):
    class Meta:
        fields = ('name', 'host', 'when', 'room')
        model = models.Talk

    def __init__(self, *args, **kwargs):
        super(TalkForm, self).__init__(*args, **kwargs)
        self.helper = FormHelper()
        self.helper.layout = Layout(
            'name',
            'host',
            'when',
            'room',
            ButtonHolder(
                Submit('add', 'Add', css_class='btn-primary')
            )
        )

    def clean_when(self):
        when = self.cleaned_data.get('when')
        pycon_start = datetime.datetime(2014, 4, 11).replace(tzinfo=utc)
        pycon_end = datetime.datetime(2014, 4, 13, 17).replace(tzinfo=utc)
        if not pycon_start < when < pycon_end:
            raise ValidationError("'when' is outside of PyCon.")
        return when
```

This `ModelForm` should look pretty similar to the other ones we've created so far, but it adds a new method, `clean_when`, which is called during the validation process and only on the `when` field.

We get the current value of `when`, then check it against two `datetime` objects that represent the start and end dates of PyCon. So long as our submitted date is between those two `datetimes`, we're happy.

Update `TalkListDetailView`

So now we need to be able to add a `Talk` to a `TalkList`. If you noticed on the `TalkForm`, we don't pass through the `talk_list` field because we'll do this in the view. But we aren't going to create a custom view for this, even though we could. We'll just extend the `TalkListDetailView` to handle this new bit of functionality.

So, back in `views.py`, let's update `TalkListDetailView`:

```
[...]
from django.shortcuts import redirect
[...]
```

```
4
5  class TalkListDetailView(
6      RestrictToUserMixin,
7      views.PrefetchRelatedMixin,
8      generic.DetailView
9  ):
10     form_class = forms.TalkForm
11     http_method_names = ['get', 'post']
12     model = models.TalkList
13     prefetch_related = ('talks',)
14
15     def get_context_data(self, **kwargs):
16         context = super(TalkListDetailView, self).get_context_data(**kwargs)
17         context.update({'form': self.form_class(self.request.POST or None)})
18         return context
19
20     def post(self, request, *args, **kwargs):
21         form = self.form_class(request.POST)
22         if form.is_valid():
23             obj = self.get_object()
24             talk = form.save(commit=False)
25             talk.talk_list = obj
26             talk.save()
27         else:
28             return self.get(request, *args, **kwargs)
29         return redirect(obj)
```

So, what are we doing here? We set a `form_class` attribute on the view, and, if this was a `FormView` derivative, it would know what to do with that, but it's not so we're really just providing this for our own convenience.

Then, in `get_context_data`, we set up the normal context dictionary before adding a `self.request.POST or None`-seeded instance of the form to the dict.

And, finally, in `post`, which is now allowed by the `http_method_names` attribute, we build a new instance of the form, check to see if it's valid, and save it if it is, first adding the `TalkList` to the `Talk`.

Template

Now we need to update the template for the `TalkListDetailView`, so open up `talks/templates/talks/talklist_detail.html` and add the following:

```
{% load crispy_forms_tags %}
[...]

<div class="panel panel-default">
    <div class="panel-heading">
        <h1 class="panel-title">Add a new talk</h1>
    </div>
    <div class="panel-body">
        {% crispy form %}

    </div>
</div>
```

The `.panel` div goes in the sidebar near the "Back to lists" and "Edit this list" links.

We're not doing anything interesting in this new snippet, just having `django-crispy-forms` render the form for us.

TalkListListView

Now that we can add talks to lists, we should probably show a count of the talks that a list has.

Pop open `talks/templates/talks/talklist_list.html` and, where we have a link to each `TalkList`, add:

```
<span class="badge">{{ object.talks.count }}</span>
```

Now, while this works, this adds an extra query for every `TalkList` our user has. If someone has a ton of lists, this could get very expensive.

Note: This is normally where I'd add in `django-debug-toolbar` and suggest you do the same. Install it with `pip` and follow the instructions online.

In `views.py`, let's fix this extra query.

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

class TalkListListView(
    RestrictToUserMixin,
    generic.ListView
):
    model = models.TalkList

    def get_queryset(self):
        queryset = super(TalkListListView, self).get_queryset()
        queryset = queryset.annotate(talk_count=Count('talks'))
        return queryset
```

We're using Django's `Count` annotation to add a `talk_count` attribute to each instance in the queryset, which means all of the counting is done by our database and we don't ever have to touch the `Talk` related items.

Go back to the template and change `{{ object.talks.count }}` to `{{ object.talk_count }}`.

Show the talks on a list

We aren't currently showing the talks that belong to a list, so let's fix that.

In `talks/templates/talks/talklist_detail.html`, the leftside column should contain:

```
<div class="col-sm-6">
    {% for talk in object.talks.all %}
        {% include 'talks/_talk.html' %}
    {% endfor %}
</div>
```

This includes a new template, `talks/templates/talks/_talk.html` for every talk on a list. Here's that new template:

```
<div class="panel panel-info">
    <div class="panel-heading">
        <a class="close" aria-hidden="true" class="pull-right" href="#">&times;</a>
        <h1 class="panel-title"><a href="{{ talk.get_absolute_url }}">{{ talk.name }}
↪</a></h1>
```

```
    </div>
    <div class="panel-body">
        <p class="bg-primary" style="padding: 15px"><strong>{{ talk.when }}</strong>
↪in <strong>{{ talk.room }}</strong></p>
        <p>by <strong>{{ talk.host }}</strong>.</p>
    </div>
</div>
```

TalkListRemoveTalkView

Since we can add talks to a list, we should be able to remove them. Let's make a new view in `views.py`

```
from django.contrib import messages

class TalkListRemoveTalkView(
    views.LoginRequiredView,
    generic.RedirectView
):
    model = models.Talk

    def get_redirect_url(self, *args, **kwargs):
        return self.talklist.get_absolute_url()

    def get_object(self, pk, talklist_pk):
        try:
            talk = self.model.objects.get(
                pk=pk,
                talk_list_id=talklist_pk,
                talk_list__user=self.request.user
            )
        except models.Talk.DoesNotExist:
            raise Http404
        else:
            return talk

    def get(self, request, *args, **kwargs):
        self.object = self.get_object(kwargs.get('pk'),
                                      kwargs.get('talklist_pk'))
        self.talklist = self.object.talk_list
        messages.success(
            request,
            u'{0.name} was removed from {1.name}'.format(
                self.object, self.talklist))
        self.object.delete()
        return super(TalkListRemoveTalkView, self).get(request, *args,
                                                       **kwargs)
```

Since we're using a `RedirectView`, we need to supply a `redirect_url` for the view to send requests to once the view is finished, and since we need it to be based off of a related object that we won't know until the view is resolved, we supply this through the `get_redirect_url` method.

Normally `RedirectViews` don't care about models or querysets, but we provide `get_object` on our view which expects the `pk` and `talklist_pk` that will come in through our URL (when we build it in a moment). We, again, check to make sure the current user owns the list and that the talk belongs to the list.

And, we've overridden `get` completely to make this all work. `get` gets our object with the URL `kwargs`, grabs the `TalkList` instance for later use, gives the user a message, and then actually deletes the `Talk`.

URL

Like all views, our new one needs a URL.

```
url(r'^remove/(?P<talklist_pk>\d+)/(?P<pk>\d+)/$',
    views.TalkListRemoveTalkView.as_view(),
    name='remove_talk'),
```

We add this to `list_patterns`, still, and then update `talks/templates/talks/_talk.html`, replacing the `'#'` in the `.close` link with `{% url 'talks:lists:remove_talk` talk.talk_list_id talk.id %}`. We can now remove talks from a list.

TalkListScheduleView

The views we've been creating are handy but aren't necessarily the cleanest for looking at, printing off, or keeping up on a phone, so let's make a new view that expressly aimed at those purposes.

In `views.py`, we're going to add:

```
class TalkListScheduleView(
    RestrictToUserMixin,
    views.PrefetchRelatedMixin,
    generic.DetailView
):
    model = models.TalkList
    prefetch_related = ('talks',)
    template_name = 'talks/schedule.html'
```

This view is very similar to our `TalkListDetailView` but has a specific template, no added form, and no `post` method. To round it out, let's set up the URL and the template.

URL

```
url(r'^s/(?P<slug>[-\w]+)/$', views.TalkListScheduleView.as_view(),
    name='schedule'),
```

Almost identical to the URL for our `TalkListDetailView`, it just changes the `d` to an `s`.

Note: This could be done entirely through arguments to the view from the url or querystring, but that would required more conditional logic in our view and/or our template, which I think is, in this case, a completely unnecessary complication.

Template

Our new template file is, of course, `talks/templates/talks/schedule.html`

```
1 {% extends '_layouts/base.html' %}
2
3 {% block title %}{{ object.name }} | Lists | {{ block.super }}{% endblock title %}
4
5 {% block headline %}
6 <h1>{{ object.name }}</h1>
```

```
<h2>Your Lists</h2>
{% endblock headline %}

{% block content %}
{% regroup object.talks.all by when|date:"Y/m/d" as day_list %}
{% for day in day_list %}
<div class="panel panel-default">
    <div class="panel-heading">
        <h1 class="panel-title">{{ day.grouper }}</h1>
    </div>
    <table class="table">
        <thead>
            <tr>
                <th>Room</th>
                <th>Time</th>
                <th>Talk</th>
                <th>Presenter(s)</th>
            </tr>
        </thead>
        <tbody>
            {% for talk in day.list %}
            <tr>
                <td>{{ talk.room }}</td>
                <td>{{ talk.when|date:"h:i A" }}</td>
                <td>{{ talk.name }}</td>
                <td>{{ talk.host }}</td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
</div>
{% endfor %}
{% endblock %}
```

The special thing about this template is how we regroup the talks. We want them grouped and sorted by their dates. Using `{% regroup %}` gives us this ability and a new object that is a list of dictionaries with two keys, `grouper` which holds our day; and `list`, which is a list of the instances in that group.

TalksDetailView

We have one last section to address in the development of this project, which is showing the detail of a `Talk` and being able to do stuff with it, like move it to a new list, rate the talk, and leave notes about it. If had more time, we'd probably want to add in some sharing abilities, too, so people could coordinate their talk attendance, but I'll leave that as an exercise for the reader.

View

While our view will definitely get more complicated as it goes, let's start with something very basic.

```
class TalkDetailView(views.LoginRequiredMixin, generic.DetailView):
        model = models.Talk

        def get_queryset(self):
            return self.model.objects.filter(talk_list__user=self.request.user)
```

We, of course, need an URL and template for it.

URL

Since this URL is all about a talk, let's make a new section in our `talks/urls.py`:

```
talks_patterns = patterns(
    '',
    url('^d/(?P<slug>[-\w]+)/$', views.TalkDetailView.as_view(),
        name='detail'),
)
```

And add it to our `urlpatterns` with its own namespace.

```
url(r'^talks/', include(talks_patterns, namespace='talks')),
```

While this is, admittedly, a bit much for just one view, most of the time you'd end up with many views related to this one model and want to have them all in a common location.

Template

And now let's set up `talks/templates/talks/talk_detail.html`:

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

{% block title %}{{ object.name }} | Talks | {{ block.super }}{% endblock title %}

{% block headline %}
<h1>{{ object.name }}</h1>
<h2>
    <span class="text-primary">{{ object.host }}</span>
    <strong>at {{ object.when }}</strong>
    in <span class="text-info">Room {{ object.room }}</span>
</h2>
{% endblock headline %}

{% block content %}

    <div class="row">
        <div class="col-sm-8">
        </div>
        <div class="col-sm-4">
            <p><a href="{{ object.talk_list.get_absolute_url }}">Back to list</a></p>
        </div>
    </div>

{% endblock content %}
```

It might seem like we have some strange bits of HTML and spacing, but we're going to fill those up soon.

Ratings

We said earlier we wanted to be able to rate the talks. I think there are two criteria that are most useful for rating a talk: the talk itself, including slides and materials; and how well the speaker performed and seemed to know their subject. So let's add two fields to our `Talk` model:

```
talk_rating = models.IntegerField(blank=True, default=0)
speaker_rating = models.IntegerField(blank=True, default=0)
```

We want both of these to be `blank`-able because we want to be able to save talks without ratings without any forms complaining at us. We also want them to have a `default` of 0 for our existing items and just as a sane default.

Let's add a `property` to our model, too, to calculate the average of these two ratings:

```
[...]
@property
def overall_rating(self):
    if self.talk_rating and self.speaker_rating:
        return (self.talk_rating + self.speaker_rating) / 2
    return 0
```

Migration

Since we've changed the model, we need to create a migration for it.

```
python manage.py schemamigration --auto talks
python manage.py migrate talks
```

Form

If give a model ratings, it's going to want a form.

```
1  [...]
2  from crispy_forms.layout import Field, Fieldset
3  [...]
4
5  class TalkRatingForm(forms.ModelForm):
6      class Meta:
7          model = models.Talk
8          fields = ('talk_rating', 'speaker_rating')
9
10     def __init__(self, *args, **kwargs):
11         super(TalkRatingForm, self).__init__(*args, **kwargs)
12         self.helper = FormHelper()
13         self.helper.layout = Layout(
14             Fieldset(
15                 'Rating',
16                 Field('talk_rating', css_class='rating'),
17                 Field('speaker_rating', css_class='rating')
18             ),
19             ButtonHolder(
20                 Submit('save', 'Save', css_class='btn-primary')
21             )
22         )
```

As you can see on line 8, we limit the fields to just the two rating fields. We also add them to a `Fieldset` with a caption of "Rating". We also gave both fields a `css_class` of `'rating'`. We'll use this to apply some CSS and Javascript soon.

View

Since we want to rate talks from the `TalkDetailView`, we need to update that view to include the form we just created.

```
class TalkDetailView(views.LoginRequiredMixin, generic.DetailView):
    http_method_names = ['get', 'post']
    model = models.Talk

    def get_queryset(self):
        return self.model.objects.filter(talk_list__user=self.request.user)

    def get_context_data(self, **kwargs):
        context = super(TalkDetailView, self).get_context_data(**kwargs)
        obj = context['object']
        rating_form = forms.TalkRatingForm(self.request.POST or None,
                                           instance=obj)
        context.update({
            'rating_form': rating_form,
        })
        return context

    def post(self, request, *args, **kwargs):
        self.object = self.get_object()
        talk_form = forms.TalkRatingForm(request.POST or None,
                                         instance=self.object)
        if talk_form.is_valid():
            talk_form.save()

        return redirect(self.object)
```

Template

And, finally, of course, we have to update `talks/templates/talks/talk_detail.html` to render the form.

```
[...]
{% load crispy_forms_tags %}
[...]

<div class="col-sm-8">
    {% crispy rating_form %}
</div>
[...]
```

You should now be able to type in a rating and save that on the model. If both fields are there, the `overall_rating` property should give you their average.

jQuery Plugin

But I'm not really happy with typing in a number. I'd rather click a star and have that set the rating. So we'll visit http://plugins.krajee.com and get their star rating plugin and put it to use.

When you download it, you'll get a directory of Javascript and a directory of CSS. Since this is, like our templates, app-specific, we'll create a `static` directory in our app to put these files into.

```
mkdir -p talks/static/talks/{css,js}
```

Move the `star-rating.min.css` file into the `css` directory we just created and do the same with the `star-rating.min.js` file and the `js` directory. Back in our template, let's add in the necessary blocks and tags to load these items.

```
{% load static from staticfiles %}
[...]

{% block css %}
<link href="{% static 'talks/css/star-rating.min.css' %}" rel="stylesheet">
{% endblock css %}

{% block js %}
<script src="{% static 'talks/js/star-rating.min.js' %}"></script>
{% endblock %}
```

Why use the `{% static %}` tag? This tag helps us if our files don't end up exactly in these directories after being pushed to a CDN or through some other process. It adds a slight bit of overhead compared to hardcoding the path to the file, but it's worth it for the convenience, I think.

Since we gave our fields the `'rating'` class, they should both show up with clickable stars for the ratings now.

Notes

We also said we wanted to be able to write notes for the talks. I like to take notes in Markdown, so we'll save a field of Markdown, convert it to HTML, and save both of those in the model.

First, we need to change our `Talk` model. We'll add two fields, one to hold the Markdown and one to hold the HTML.

```
notes = models.TextField(blank=True, default='')
notes_html = models.TextField(blank=True, default='', editable=False)
```

These fields are `blank`-able like our ratings fields for much the same reasons, same with giving them a `default`. The `notes_html` field gets `editable=False` because we don't want this field to be directly edited in `ModelForms` or the admin.

Of course, now that we've added fields to the model, we need to do another migration.

```
python manage.py schemamigration --auto talks
python manage.py migrate talks
```

But since we know we'll be getting Markdown and we know we want to convert it, we should come up with a way to do that easily and automatically.

First, let's `pip install mistune`. `mistune` is a super-fast Python Markdown library that we can use to convert it to HTML. It's also super-easy to use. We need to import it at the top of the file and then we'll override the `save` method of our `Talk` model.

```
1 class Talk(models.Model):
2     [...]
3     def save(self, *args, **kwargs):
4         self.slug = slugify(self.name)
5         self.notes_html = mistune.markdown(self.notes)
6         super(Talk, self).save(*args, **kwargs)
```

Template

Now let's update our `talks/templates/talks/talk_detail.html` template to show the notes and the ratings. Add the following block before the `.row` div, at the top of `{% block content %}`.

```
{% if object.notes_html %}
<div class="row">
    <div class="col-sm-8">
        <h3>Notes</h3>
        {{ object.notes_html|safe }}
    </div>
    <div class="col-sm-4">
        <div class="well">
            <table class="table table-condensed">
                <thead>
                    <tr>
                        <th>Category</th>
                        <th>Rating</th>
                    </tr>
                </thead>
                <tbody>
                    <tr>
                        <th>Talk</th>
                        <td>{{ object.talk_rating }}</td>
                    </tr>
                    <tr>
                        <th>Speaker</th>
                        <td>{{ object.speaker_rating }}</td>
                    </tr>
                    <tr>
                        <th>Overall</th>
                        <td>{{ object.overall_rating }}</td>
                    </tr>
                </tbody>
            </table>
        </div>
    </div>
</div>
<hr>
{% endif %}
```

This will show any notes we've saved and our ratings. Currently the display of the ratings depends on us having notes saved, but that's something to fix later. Especially since we're likely to save notes **during** a talk but not save ratings until **after**.

Stars template tag

We're only printing out the number of stars something was given, though. While that's good information, it's not the most useful or attractive of outputs. Let's make a template tag to render a total number of stars and color some of them based on the rating.

First, we need to make a place to write the template tag. Tags always live with an app and are usually named for the app, so let's start with that.

```
mkdir -p talks/templatetags/
touch talks/templatetags/{__init__,talks_tags}.py
```

This will create the `templatetags` directory for us and stick in two files, `__init__.py`, as usual, and `talks_tags.py`, which is where we'll write the tag. Open that file in your editor and add in:

```
from django import template

register = template.Library()


@register.inclusion_tag('talks/_stars.html')
def show_stars(count):
    return {
        'star_count': range(count),
        'leftover_count': range(count, 5)
    }
```

This tag is an inclusion tag which means it will render a template whenever we call it. Since it renders a template, we need to create that template. So open up `talks/templates/talks/_stars.html` and add:

```
{% for star in star_count %}
<i class="glyphicon glyphicon-star" style="color:#fc0; font-size:24px"></i>
{% endfor %}
{% if leftover_count %}
    {% for star in leftover_count %}
    <i class="glyphicon glyphicon-star-empty" style="font-size:24px"></i>
    {% endfor %}
{% endif %}
```

Nothing really fancy happening here, just printing out some stars based on the `range` that we created in the tag. We have some "magic numbers" here, but for the purposes of a demo, they're OK. In an actual production project, you'd want to set these rating upper limits in `settings.py`.

Now let's open up `talks/templates/talks/talk_detail.html` and replace the three places where we print out `{{ object.talk_rating }}`, etc, with `{% show_stars object.talk_rating %}`. We also need to add `{% load talks_tags %}` at the top of the template.

Move talks between lists

We'd also like to be able to move talks from one list to another, since we might change our minds about what list a talk should be on. We don't need to modify our models at all, since the `ForeignKey` between `Talk` and `TalkList` already exists, but we do need a new form and to modify our view and template.

Form

In `forms.py`, we're going to create a form called `TalkTalkListForm` and it'll look like:

```
class TalkTalkListForm(forms.ModelForm):
    class Meta:
        model = models.Talk
        fields = ('talk_list',)

    def __init__(self, *args, **kwargs):
        super(TalkTalkListForm, self).__init__(*args, **kwargs)
        self.fields['talk_list'].queryset = (
            self.instance.talk_list.user.lists.all())

        self.helper = FormHelper()
```

```
        self.helper.layout = Layout(
            'talk_list',
            ButtonHolder(
                Submit('move', 'Move', css_class='btn-primary')
            )
        )
```

The only thing special that we're doing in this form is restricting the queryset for our `talk_list` field to the lists related to the user that owns the list that our current talk belongs to. This means we can't move our talk to someone else's list.

View

Now we need to update the `TalkDetailView`. Our final version of this view could be better refined, likely by moving things to other views that just redirect back to this one, but in the interest of keeping this demo short, we'll do it a slightly messier way.

In the view's `get_context_data`, we need to instantiate the form we just created and add it to the context dictionary.

```
[...]
list_form = forms.TalkTalkListForm(self.request.POST or None,
                                   instance=obj)
context.update({
    'rating_form': rating_form,
    'list_form': list_form
})
```

We also need to update the `post` method and add in some logic for handling *which* form was submitted. This is the part that would benefit from being separated out to other views.

```
def post(self, request, *args, **kwargs):
    self.object = self.get_object()
    if 'save' in request.POST:
        talk_form = forms.TalkRatingForm(request.POST or None,
                                         instance=self.object)
        if talk_form.is_valid():
            talk_form.save()

    if 'move' in request.POST:
        list_form = forms.TalkTalkListForm(request.POST or None,
                                           instance=self.object,
                                           user=request.user)
        if list_form.is_valid():
            list_form.save()

    return redirect(self.object)
```

Template

Finally, we need to actually render the new form into the template. Open up `talks/templates/talks/talk_detail.html` and add `{% crispy list_form %}` in the `.col-sm-4` div near the "Back to list" link.

Deployment

Now we want to put this thing like on Heroku. We have several steps required before we can do this, so let's get started.

Postgres

Since Heroku uses Postgres, we need to provide an adapter library for it. `pip install psycopg2`. We also need to set up our database adapter in the project, so we need to also `pip install dj-database-url` and then open up `settings.py`. We need to change `DATABASES` to look like:

```
import dj_database_url
[...]

DATABASES = {
    'default': dj_database_url.config(
        default='sqlite:////{0}'.format(os.path.join(BASE_DIR, 'db.sqlite3'))
    )
}
```

This will let us keep using our local SQLite database but use Heroku's database there.

WSGI and static files

We also want to be able to serve our static files on Heroku, so we need to install `dj-static` or `whitenoise`. Since `dj-static` doesn't support Python 3 yet, and we'd like to make sure our code is as future-friendly as possible, let's use `whitenoise`.

`pip install whitenoise`, then open `survivalguide/wsgi.py` and, **after** the `os.environ` call, add `from whitenoise.django import DjangoWhiteNoise`.

Then change the `application` line to wrap `DjangoWhiteNoise` around the `get_wsgi_application()` call.

`requirements.txt`

We know that we're going to be running on gunicorn on Heroku, so we should `pip install gunicorn` before we go any further.

Now, we need to make sure Heroku can install our requirements. Back in the directory that contains `manage.py`, we need to create a file named `requirements.txt` that holds all of the packages we've installed and their version. The easiest way to do this is:

```
pip freeze --local > requirements.txt
```

If you look at this file, it should contain entries like:

```
Django==1.6.2
mistune==0.2.0
```

Procfile

The last thing we need to do before we send things to Heroku is to create the `Procfile` that tells Heroku what to run. Ours just needs one process which looks like:

```
web:  gunicorn survivalguide.wsgi
```

This tells Heroku to run `gunicorn` with our `wsgi.py` file.

Settings

In our `settings.py` file, we need to set `DEBUG` to `False` and change `ALLOWED_HOSTS` to `['*']` since we don't yet know our Heroku URL.

Deploy

Now that we have everything collected and added into Git, we're ready to send our project to Heroku.

`heroku create` will make Heroku create a new installation for us and set the Git remote locally. Now we can do `git push heroku master` and send all of our files to Heroku.

Once the process finishes, if you don't see any errors, you'll need to sync the database with `heroku run python manage.py syncdb` and create a superuser. Then `heroku open` will open your site in your browser.

Resources

Below are some handy packages, books, tutorials, etc to check out if you want to learn more about Django:

- Two Scoops of Django
- Hello Web App (Not released yet, but coming soon)
- Getting Started with Django
- Tango with Django
- GoDjango
- django-debug-toolbar
- PDF of this class
- HTML version of this class

CHAPTER 2

Indices and tables

- genindex
- modindex
- search