Learning Django by Testing Documentation

Release 0.0.1

San Diego Python

Jul 19, 2017

Contents

Thank you for attending San Diego Python's workshop on test-driven development with the Django web framework. In this one-day workshop, you will learn to build a well-tested, Django-based website.

This workshop was made possible by a grant from the Python Software Foundation Outreach and Education Committee.

CHAPTER 1

Why test-driven development?

When creating a new application, at first you may not need tests. Tests can be difficult to write at first and they take time, but they can save an enormous amount of manual troubleshooting time.

As your application grows, it becomes more difficult to grow and to refactor your code. There's always the risk that a change in one part of your application will break another part. A good collection of automated tests that go along with an application can verify that changes you make to one part of the software do not break another.

CHAPTER 2

Prerequisites

- Python 2.6 or 2.7 (2.7 is recommended)
- Install Django 1.5
- The Django tutorials

You do not need to be a Django expert to attend this workshop or to find this document useful. However, the goal of getting a working website with tests in a single day is a lofty one and so we ask that attendees come with Python and Django installed. We also encourage people to go through the Django tutorials beforehand in order to get the most out of the workshop.

CHAPTER 3

The Project: building a blog

The right of passage for most web developers is their own blog system. There are hundreds of solutions out there. The features and requirements are generally well understood. Writing one with TDD becomes a kind of code kata that can help you work through all kinds of aspects of the Django framework.

CHAPTER 4

Contents

Getting started

Verifying setup

Before we get started, let's just make sure that Python and Django are installed correctly and are the appropriate versions.

Running the following command in the MacOS or Linux terminal or in the Windows command prompt should show the version of Python. For this workshop you should have a 2.6.x or 2.7.x version of Python.

```
$ python -V
```

You should also have pip installed on your machine, along with the requirements.txt file.

```
# In the same directory where you downloaded requirements.txt
$ pip install -r requirements.txt
```

Hint: Things you should type into your terminal or command prompt will always start with `$` in this workshop. Don't type the leading `$` though.

Running the next command will show the version of Django you have installed. For this workshop, a 1.5.x version is required. If instead you see a "No module named django" message, please follow the Django installation instructions.

```
$ python -c "import django; print(django.get_version())"
```

Creating the project

The first step when creating a new Django website is to create the project boilerplate files.

```
$ django-admin.py startproject myblog
$ cd myblog
```

Running this command created a new directory called `myblog/` with a few files and folders in it. Notably, there is a `manage.py` file which is a file used to manage a number of aspects of your Django application such as creating the database and running the development web server. Two other key files we just created are `myblog/settings.py` which contains configuration information for the application such as how to connect to the database and `myblog/urls.py` which maps URLs called by a web broser to the appropriate Python code.

Directory variables

Add the following to the top of your `myblog/settings.py` file:

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(__file__))
```

Setting up the database

One building block of virtually all websites that contain user-generated content is a database. Databases facilitate a good separation between code (Python and Django in this case), markup and scripts (HTML, CSS and JavaScript) and actual content (database). Django and other frameworks help guide developers to separate these concerns.

First we need to update the `DATABASES` variable in our settings file (`myblog/settings.py`).

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

Now let's create the database and a super user account for accessing the admin interface which we'll get to shortly:

```
$ python manage.py syncdb
```

After running this command, there will be a database file `myblog.sqlite3` in the same directory as `manage.py`. Right now, this database only has a few tables specific to Django. The command looks at `INSTALLED_APPS` in `myblog/settings.py` and creates database tables for models defined in those apps' `models.py` files.

Later in this workshop, we will create models specific to the blog we are writing. These models will hold data like blog posts and comments on blog posts.

Hint: SQLite is a self-contained database engine. It is inappropriate for a multi-user website but it works great for development. In production, you would probably use PostgreSQL or MySQL. For more info on SQLite, see the SQLite documentation.

Enabling the admin site

One of the killer features Django provides is an admin interface. An admin interface is a way for an administrator of a website to interact with the database through a web interface which regular website visitors are not allowed to use. On a blog, this would be where the author writes new blog posts.

We need to add `'django.contrib.admin'` to `INSTALLED_APPS` in our settings file (`myblog/settings.py`). Afterward it should look something like this:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.admin',        # we just added this
)
```

After adding the admin to our installed apps we need to have Django create the database tables for admin:

```
$ python manage.py syncdb
```

We also need to enable admin URLs and enable auto-discovery of `admin.py` files in our apps. We will create one of these `admin.py` files later to expose our blog post model and comment model to the admin interface. To enable auto-discovery, we need to uncomment some lines in our project's urls file (`myblog/urls.py`). Afterward our urls file should look something like this:

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

from django.contrib import admin
admin.autodiscover()

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

Checking our progress

Let's check our progress by running the Django test server and visiting the admin site.

In your terminal, run the Django development server:

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

Now visit the admin site in your browser (http://localhost:8000/admin/).

Hint: The Django development server is a quick and simple web server used for rapid development and not for long-term production use. The development server reloads any time the code changes but some actions like adding files do not trigger a reload and the server will need to be manually restarted.

Read more about the development server in the official documentation.

Quit the server by holding the control key and pressing C.

Models

Creating an app

It is generally a good practice to separate your Django projects into multiple specialized (and sometimes reusable) apps. Additionally every Django model must live in an app so you'll need at least one app for your project.

Let's create an app for blog posts and related models. We'll call the app `blog`:

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

This command should have created a `blog` directory with the following files:

```
__init__.py
models.py
tests.py
views.py
```

We'll be focusing on the `models.py` file below.

Before we can use our app we need to add it to our `INSTALLED_APPS` in our settings file (`myblog/settings.py`). This will allow Django to discover the models in our `models.py` file so they can be added to the database when running syncdb.

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

    'blog',
)
```

Note: Just to make sure we are on the same page, your project structure should look like this:

```
- blog
|   - __init__.py
|   - models.py
|   - tests.py
|   - views.py
- manage.py
- myblog
|   - __init__.py
|   - settings.py
|   - urls.py
|   - wsgi.py
- myblog.sqlite3
```

Creating a model

First let's create a blog post model. This will correspond to a database table which will hold our blog posts. A blog post will be represented by an instance of our `Post` model class and each `Post` model instance will identify a row in our database table.

```
from django.db import models


class Post(models.Model):
    title = models.CharField(max_length=500)
    author = models.ForeignKey('auth.User')
    body = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True, editable=False)
    modified_at = models.DateTimeField(auto_now=True, editable=False)
```

If you aren't already familiar with databases, this code may be somewhat daunting. A good way to think about a model (or a database table) is as a sheet in a spreadsheet. Each field like the `title` or `author` is a column in the spreadsheet and each different instance of the model – each individual blog post in our project – is a row in the spreadsheet.

To create the database table for our `Post` model we need to run syncdb again:

```
$ python manage.py syncdb
```

Tip: If you notice, this code is written in a very particular way. There are two blank lines between imports and class definitions and the code is spaced very particularly. There is a style guide for Python known as PEP8. A central tenet of Python is that code is read more frequently than it is written. Consistent code style helps developers read and understand a new project more quickly.

Creating posts from the admin site

We don't want to manually add posts to the database every time we want to update our blog. It would be nice if we could use a login-secured webpage to create blog posts. Fortunately Django's admin interface can do just that.

In order to create blog posts from the admin interface we need to register our Post model with the admin site. We can do this by creating a new `blog/admin.py` file with the following code:

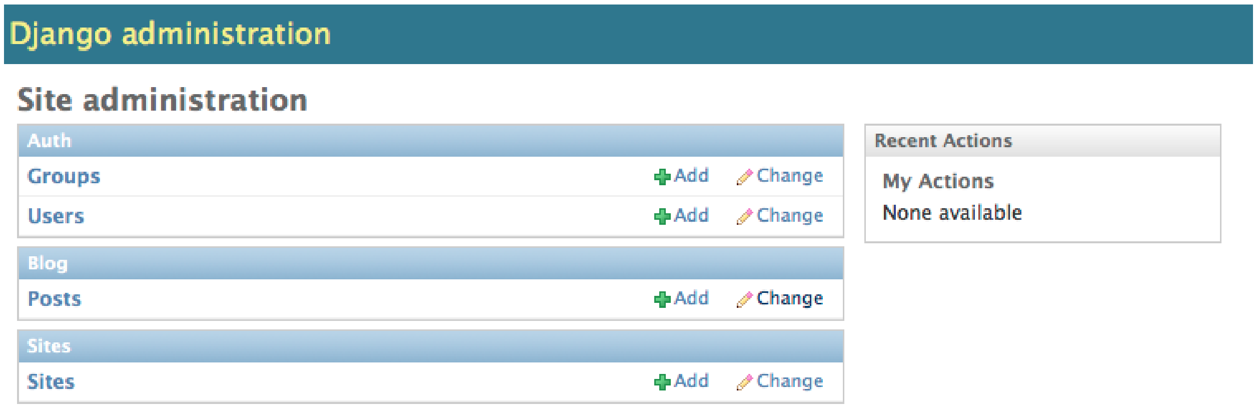
```
from django.contrib import admin
from .models import Post


admin.site.register(Post)
```

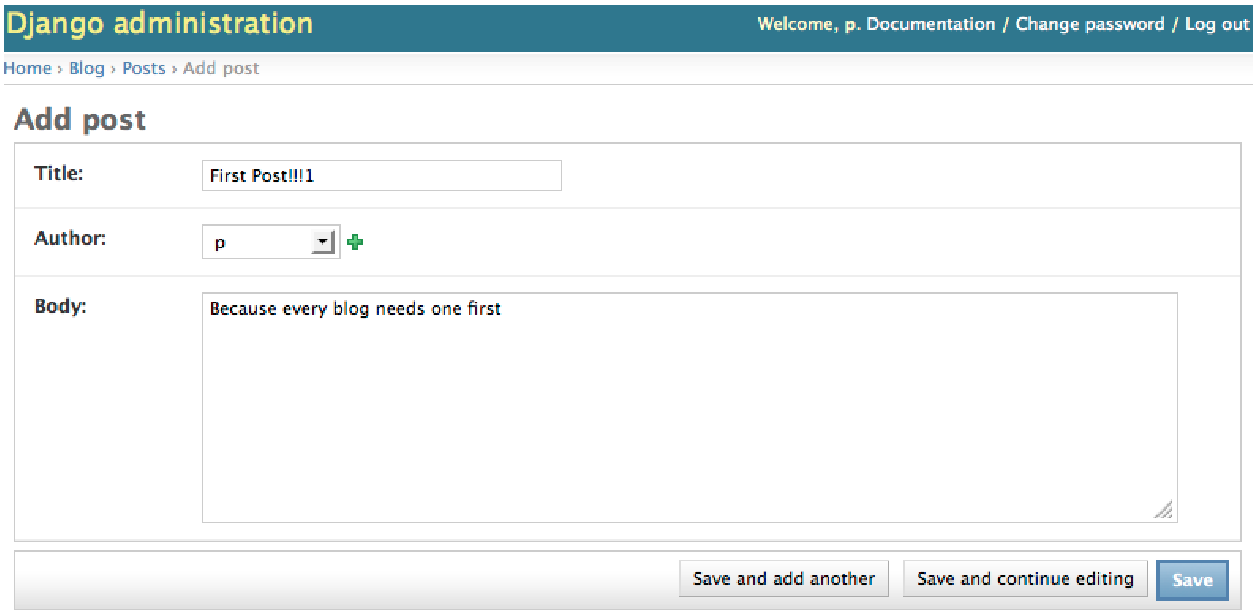
Now, start up the development server again and navigate to the admin site (http://localhost:8000/admin/) and create a blog post.

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

First click the "Add" link next to *Posts* in the admin site.



Next fill in the details for our first blog post and click the *Save* button.

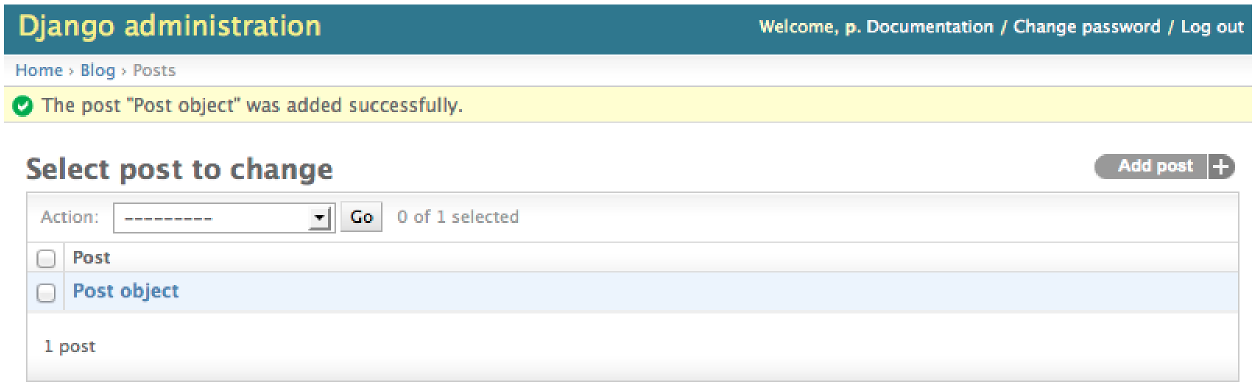

Django administration
Welcome, p. Documentation / Change password / Log out
Home › Blog › Posts › Add post
Add post
Save and add another
Save and continue editing
Save

Title: First Post!!!1

Author: p

Body: Because every blog needs one first

Our post was created



The post "Post object" was added successfully.

Our first test: __unicode__ method

In the admin change list our posts all have the unhelpful name *Post object*. We can customize the way models are referenced by creating a `__unicode__` method on our model class. Models are a good place to put this kind of reusable code that is specific to a model.

Let's first create a test demonstrating the behavior we'd like to see.

All the tests for our app will live in the `blog/tests.py` file. Delete everything in that file and start over with a failing test:

```
from django.test import TestCase


class PostModelTest(TestCase):

    def test_unicode_representation(self):
        self.fail("TODO Test incomplete")
```

Now run the test command to ensure our app's single test fails as expected:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_unicode_representation (blog.tests.PostModelTest)
----------------------------------------------------------------------
Traceback (most recent call last):
...
AssertionError: TODO Test incomplete

----------------------------------------------------------------------
Ran 1 test in 0.001s

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

If we read the output carefully, the `manage.py test` command did a few things. First, it created a test database. This is important because we wouldn't want tests to actually modify our real database. Secondly, it executed each "test" in `blog/tests.py`. If all goes well, the test runner isn't very chatty, but when failures occur like in our test, the test runner prints lots of information to help you debug your failing test.

Now we're ready to create a real test.

Tip: There are lots of resources on unit testing but a great place to start is the official Python documentation on the unittest module and the Testing Django applications docs. They also have good recommendations on naming conventions which is why our test classes are named like `SomethingTest` and our methods named `test_something`. Because many projects adopt similar conventions, developers can more easily understand the code.

Let's write our test to ensure that a blog post's unicode representation is equal to its title. We need to modify our tests file like so:

```
from django.test import TestCase
from .models import Post
```

```
class PostModelTest(TestCase):

    def test_unicode_representation(self):
        post = Post(title="My post title")
        self.assertEqual(unicode(post), post.title)
```

Hint: `__unicode__` may seem like a strange name, but Unicode is a standard for representing and encoding most of the world's writing systems. All strings that Django passes around are Unicode strings so that Django can be used for applications designed for different languages.

Now let's run our tests again:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
F
======================================================================
FAIL: test_unicode_representation (blog.tests.PostModelTest)
----------------------------------------------------------------------
Traceback (most recent call last):
...
AssertionError: u'Post object' != 'My post title'

----------------------------------------------------------------------
Ran 1 test in 0.001s

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

Our test fails again, but this time it fails because we haven't customized our `__unicode__` method yet so the unicode representation for our model is still the default *Post object*.

Let's add a `__unicode__` method to our model that returns the post title. Our `models.py` file should look something like this:

```
from django.db import models


class Post(models.Model):
    title = models.CharField(max_length=500)
    author = models.ForeignKey('auth.User')
    body = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True, editable=False)
    modified_at = models.DateTimeField(auto_now=True, editable=False)

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

If you start the development server and take a look at the admin interface (http://localhost:8000/admin/) again, you will see the post titles in the list of posts.

Now if we run our test again we should see that our single test passes:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

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

We've just written our first test and fixed our code to make our test pass.

Test Driven Development (TDD) is all about writing a failing test and then making it pass. If you were to write your code first, then write tests, it's harder to know that the test you wrote really does test what you want it to.

While this may seem like a trivial example, good tests are a way to document the expected behavior of a program. A great test suite is a sign of a mature application since bits and pieces can be changed easily and the tests will ensure that the program still works as intended. The Django framework itself has a massive unit test suite with thousands of tests.

Views and Templates

Now we can create blog posts and see them in the admin interface, but no one else can see our blog posts yet.

The homepage test

Every site should have a homepage. Let's write a failing test for that.

We can use the Django test client to create a test to make sure that our homepage returns an HTTP 200 status code (this is the standard response for a successful HTTP request).

Let's add the following to our `blog/tests.py` file:

```
class ProjectTests(TestCase):

    def test_homepage(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)
```

If we run our tests now this test should fail because we haven't created a homepage yet.

Hint: There's lots more information on the hypertext transfer protocol (HTTP) and its various status codes on Wikipedia. Quick reference, 200 = OK; 404 = Not Found; 500 = Server Error

Base template and static files

Let's start with base templates based on zurb foundation. First download and extract the Zurb Foundation files (direct link).

Zurb Foundation is a CSS, HTML and JavaScript framework for building the front-end of web sites. Rather than attempt to design a web site entirely from scratch, Foundation gives a good starting place on which to design and build an attractive, standards-compliant web site that works well across devices such as laptops, tablets and phones.

Static files

Create a `static` directory in our top-level directory (the one with the `manage.py` file). Copy the `css` directory from the foundation archive to this new `static` directory.

Now let's add this new `static` directory to our `myblog/settings.py` file:

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

For more details, see Django's documentation on static files.

Important: This workshop is focused on Python and Django and so out of necessity we are going to gloss over explaining HTML, CSS and JavaScript a little bit. However, virtually all websites have a front-end built with these fundamental building blocks of the open web.

Template files

Templates are a way to dynamically generate a number of documents which are similar but have some data that is slightly different. In the blogging system we are building, we want all of our blog posts to look visually similar but the actual text of a given blog post varies. We will have a single template for what all of our blog posts and the template will contain variables that get replaced when a blog post is rendered. This reuse that Django helps with and the concept of keeping things in a single place is called the DRY principle for Don't Repeat Yourself.

Create a `templates` directory in our top-level directory. Our directory structure should look like

```
- blog
|   - __init__.py
|   - admin.py
|   - models.py
|   - tests.py
|   - views.py
- manage.py
- myblog
|   - __init__.py
|   - settings.py
|   - urls.py
|   - views.py
|   - wsgi.py
- myblog.sqlite3
- static
|   - css
|       - foundation.css
|       - foundation.min.css
|       - normalize.css
- templates
```

Create a basic HTML file like this and name it `templates/index.html`:

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head>
    <title>Foundation 4</title>
    <link rel="stylesheet" href="{% static "css/foundation.css" %}">
</head>
<body>
    <section class="row">
        <header class="large-12 columns">
            <h1>Welcome to My Blog</h1>
            <hr>
        </header>
    </section>
</body>
</html>
```

Now let's add this new `templates` directory to our `myblog/settings.py` file:

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

For just about everything there is to know about Django templates, read the template documentation.

Tip: In our examples, the templates are going to be used to generate similar HTML pages. However, Django's template system can be used to generate any type of plain text document such as CSS, JavaScript, CSV or XML.

Views

Now let's create a homepage using the `index.html` template we added.

Let's start by creating a views file: `myblog/views.py` referencing the `index.html` template:

```
from django.views.generic.base import TemplateView


class HomeView(TemplateView):

    template_name = 'index.html'

home = HomeView.as_view()
```

Important: We are making this views file in the `myblog` project directory (next to the `myblog/urls.py` file we are about to change). We are **not** changing the `blog/views.py` file yet. We will use that file later.

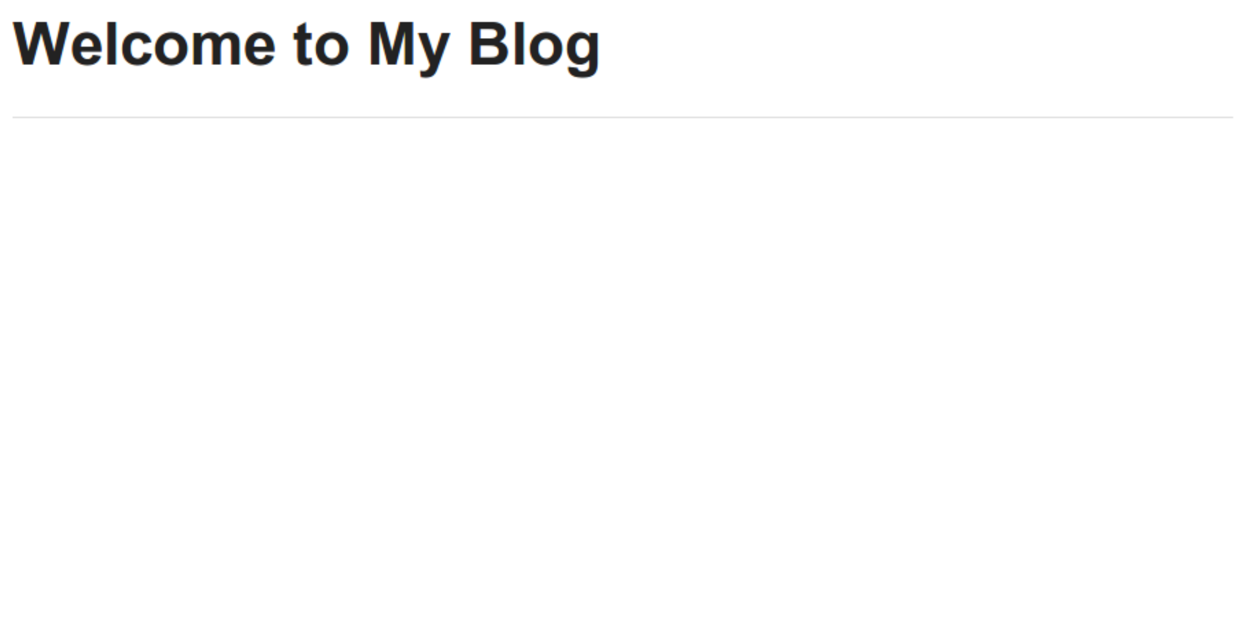
Django will be able to find this template in the `templates` folder because of our `TEMPLATE_DIRS` setting. Now we need to route the homepage URL to the home view. Our URL file `myblog/urls.py` should look something like this:

```
from django.conf.urls import patterns, include, url
from myblog import views

from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^$', views.home),
    url(r'^admin/', include(admin.site.urls)),
)
```

Now let's visit http://localhost:8000/ in a web browser to check our work. You should see a webpage that looks like this:


Welcome to My Blog

Great! Now let's make sure our new test passes:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
..
----------------------------------------------------------------------
Ran 2 tests in 0.021s

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

Hint: From a code flow perspective, we now have a working example of how Django creates dynamic web pages. When an HTTP request to a Django powered web site is sent, the `urls.py` file contains a series of patterns for matching the URL of that web request. The matching URL delegates the request to a corresponding view (or to a another set of URLs which map the request to a view). Finally, the view delegates the request to a template for rendering the actual HTML.

In web site architecture, this separation of concerns is variously known as a three-tier architecture or a model-view-controller architecture.

Using a base template

Templates in Django are generally built up from smaller pieces. This lets you include things like a consistent header and footer on all your pages. Convention is to call one of your templates `base.html` and have everything inherit from that.

We'll start with putting our header and a sidebar in `templates/base.html`:

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
<head>
    <title>Foundation 4</title>
    <link rel="stylesheet" href="{% static "css/foundation.css" %}">
</head>
<body>
    <section class="row">
        <header class="large-12 columns">
            <h1>Welcome to My Blog</h1>
            <hr>
        </header>
    </section>

    <section class="row">

        <div class="large-8 columns">
            {% block content %}{% endblock %}
        </div>

        <div class="large-4 columns">
            <h3>About Me</h3>
            <p>I am a Python developer and I like Django.</p>
        </div>

    </section>

</body>
</html>
```

Note: We will not explain the CSS classes we used above (e.g. `large-8`, `column`, `row`). More information on these classes can be found in the Zurb Foundation grid documentation.

There's a lot of duplicate code between our `templates/base.html` and `templates/index.html`. Django's templates provide a way of having templates inherit the structure of other templates. This allows a template to define only a few elements, but retain the overall structure of its parent template.

If we update our `index.html` template to extend `base.html` we can see this in action. Delete everything in `templates/index.html` and replace it with the following:

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

{% block content %}
Page body goes here.
{% endblock content %}
```

Now our `templates/index.html` just overrides the `content` block in `templates/base.html`. For more details on this powerful Django feature, you can read the documentation on template inheritance.

ListViews

We put a hard-coded title and article in our filler view. These post details should come from our models and database instead. Let's write a test for that.

The Django `test client` can be used for a simple test of whether text shows up on a page. Let's add the following to our `blog/tests.py` file:

```
from django.contrib.auth import get_user_model

class ListPostsOnHomePage(TestCase):

    """Test whether our blog posts show up on the homepage"""

    def setUp(self):
        self.user = get_user_model().objects.create(username='some_user')

    def test_one_post(self):
        Post.objects.create(title='1-title', body='1-body', author=self.user)
        response = self.client.get('/')
        self.assertContains(response, '1-title')
        self.assertContains(response, '1-body')

    def test_two_posts(self):
        Post.objects.create(title='1-title', body='1-body', author=self.user)
        Post.objects.create(title='2-title', body='2-body', author=self.user)
        response = self.client.get('/')
        self.assertContains(response, '1-title')
        self.assertContains(response, '1-body')
        self.assertContains(response, '2-title')
```

which should fail like this

```
Creating test database for alias 'default'...
FF..
======================================================================
FAIL: test_one_post (blog.tests.ListPostsOnHomePage)
----------------------------------------------------------------------
Traceback (most recent call last):
  ...
AssertionError: Couldn't find '1-title' in response

======================================================================
FAIL: test_two_posts (blog.tests.ListPostsOnHomePage)
----------------------------------------------------------------------
Traceback (most recent call last):
  ...
AssertionError: Couldn't find '1-title' in response

----------------------------------------------------------------------
Ran 4 tests in 0.201s

FAILED (failures=2)
Destroying test database for alias 'default'...
```

Updating our views

One easy way to get all our posts objects to list is to just use a `ListView`. That changes our `HomeView` only slightly.

```
from django.views.generic import ListView

from blog.models import Post
```

```python
class HomeView(ListView):
    template_name = 'index.html'
    queryset = Post.objects.order_by('-created_at')

home = HomeView.as_view()
```

Important: Make sure you update your `HomeView` to inherit from `ListView`. Remember this is still `myblog/views.py`.

That small change will provide a `post_list` object to our template `index.html` which we can then loop over. For some quick documentation on all the Class Based Views in django, take a look at Classy Class Based Views

The last change needed then is just to update our homepage template to add the blog posts. Let's replace our `templates/index.html` file with the following:

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

{% block content %}
    {% for post in post_list %}
        <article>

            <h2><a href="{{ post.get_absolute_url }}">{{ post.title }}</a></h2>

            <p class="subheader">
                <time>{{ post.modified_at|date }}</time>
            </p>

            <p></p>

            {{ post.body|linebreaks }}

        </article>
    {% endfor %}
{% endblock content %}
```

Tip: Notice that we didn't specify the name `post_list` in our code. Django's class-based generic views often add automatically-named variables to your template context based on your model names. In this particular case the context object name was automatically defined by the get_context_object_name method in the `ListView`. Instead of referencing `post_list` in our template we could have also referenced the template context variable `object_list` instead.

Running the tests here we see that all the tests pass!

Note: Read the Django built-in template tags and filters documentation for more details on the linebreaks and date template filters.

And now, if we add some posts in our admin, they should show up on the homepage. What happens if there are no posts? We should add a test for that

```
def test_no_posts(self):
    response = self.client.get('/')
    self.assertContains(response, 'No blog post entries yet.')
```

And that gives us the expected failure

```
Creating test database for alias 'default'...
F....
======================================================================
FAIL: test_no_posts (blog.tests.ListPostsOnHomePage)
----------------------------------------------------------------------
Traceback (most recent call last):
  ...
AssertionError: Couldn't find 'No blog post entries yet' in response

----------------------------------------------------------------------
Ran 5 tests in 0.044s

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

The easiest way to add this is to use the empty clause. See if you can add this in yourself to make the test pass.

What about viewing an individual blog post?

Blog Post Details

To save a bit of time let's make our urls look like `http://myblog.com/post/ID/` where ID is the database ID of the blog post we want to see.

Before we create this page, let's move the template content that displays our blog posts on our homepage into a separate template file so we can reuse it on our blog post details page.

Let's make a file called `templates/_post.html` and put the following in it:

```
<article>

    <h2><a href="{{ post.get_absolute_url }}">{{ post.title }}</a></h2>

    <p class="subheader">
        <time>{{ post.modified_at|date }}</time>
    </p>

    <p></p>

    {{ post.body|linebreaks }}

</article>
```

Note: The `post.get_absolute_url` reference doesn't do anything yet. Later we will add a `get_absolute_url` method to the post model which will make these links work.

Tip: The filename of our includable template starts with _ by convention. This naming convention is recommended by Harris Lapiroff in An Architecture for Django Templates.

Now let's change our homepage template (`templates/index.html`) to include the template file we just made:

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

{% block content %}
    {% for post in post_list %}
        {% include "_post.html" with post=post only %}
    {% empty %}
        <p>No blog post entries yet.</p>
    {% endfor %}
{% endblock content %}
```

Tip: We use the `with post=post only` convention in our `include` tag for better encapsulation (as mentioned in An Architecture for Django Templates). Check the Django documentation more information on the include tag.

Let's write a test for that:

```
from django.contrib.auth import get_user_model

class BlogPostViewTest(TestCase):

    def setUp(self):
        self.user = get_user_model().objects.create(username='some_user')
        self.post = Post.objects.create(title='1-title', body='1-body',
                                        author=self.user)

    def test_basic_view(self):
        response = self.client.get(self.post.get_absolute_url())
        self.assertEqual(response.status_code, 200)
```

This test fails beacuse we didn't define get_absolute_url (Django Model Instance Documentation). We need to create a URL and a view for blog post pages now. We'll need to create a `blog/urls.py` file and reference it in the `myblog/urls.py` file.

Our `blog/urls.py` file is the very short

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


urlpatterns = patterns('blog.views',
    url(r'^post/(?P<pk>\d+)/$', 'post_details'),
)
```

The urlconf in `myblog/urls.py` needs to reference `blog.urls`:

```
url(r'^', include('blog.urls')),
```

Now we need to define a `post_details` view in our `blog/views.py` file:

```
from django.http import HttpResponse


def post_details(request, pk):
    return HttpResponse('empty')
```

We'll be updating this view later to return something useful.

Finally we need to create the `get_absolute_url()` function which should return the post details URL for each posts. We should create a test first. Let's add the following test to our `PostModelTest` class:

```
def test_get_absolute_url(self):
    user = get_user_model().objects.create(username='some_user')
    post = Post.objects.create(title="My post title", author=user)
    self.assertIsNotNone(post.get_absolute_url())
```

Now we need to implement `get_absolute_url` in our `Post` class (found in `blog/models.py`):

```
from django.core.urlresolvers import reverse

# And in our Post model class...

def get_absolute_url(self):
    return reverse('blog.views.post_details', kwargs={'pk': self.pk})
```

We should now have passing tests again.

Let's make the blog post details page actually display a blog post. First we'll write some tests in our `BlogPostViewTest` class:

```
def test_blog_title_in_post(self):
    response = self.client.get(self.post.get_absolute_url())
    self.assertContains(response, self.post.title)

def test_blog_body_in_post(self):
    response = self.client.get(self.post.get_absolute_url())
    self.assertContains(response, self.post.body)
```

To implement our blog post page we'll use another class-based generic view: the DetailView. The `DetailView` is a view for displaying the details of an instance of a model and rendering it to a template. Let's replace our `blog/views.py` file with the following:

```
from django.views.generic import DetailView
from .models import Post


class PostDetails(DetailView):
    model = Post

post_details = PostDetails.as_view()
```

Now we'll see some `TemplateDoesNotExist` errors when running our tests again:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
EEE......
======================================================================
ERROR: test_blog_body_in_post (blog.tests.BlogPostViewTest)
----------------------------------------------------------------------
...
TemplateDoesNotExist: blog/post_detail.html

======================================================================
ERROR: test_blog_title_in_post (blog.tests.BlogPostViewTest)
----------------------------------------------------------------------
```

```
...
TemplateDoesNotExist: blog/post_detail.html

----------------------------------------------------------------------
Ran 9 tests in 0.071s

FAILED (errors=3)
Destroying test database for alias 'default'...
```

These errors are telling us that we're referencing a blog/post_detail.html template but we haven't created that file yet. Let's create a templates/blog/post_detail.html. The DetailView should provide us with a post context variable that we can use to reference our Post model instance. Our template should look similar to this:

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

{% block content %}
    {% include "_post.html" with post=post only %}
{% endblock %}
```

Now our tests should pass again:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
.......
----------------------------------------------------------------------
Ran 8 tests in 0.071s

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

More Views

Blogs should be interactive. Let's allow visitors to comment on each post.

Adding a Comment model

First we need to add a Comment model in blog/models.py.

```
class Comment(models.Model):
    post = models.ForeignKey(Post)
    name = models.CharField(max_length=100)
    email = models.EmailField()
    body = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True, editable=False)
    modified_at = models.DateTimeField(auto_now=True, editable=False)
```

Let's write a __unicode__ method for our Comment model like we did for our Post model earlier.

First we should create a test in blog/tests.py. Our test should look very similar to the __unicode__ test we wrote for posts earlier. This should suffice:

```
class CommentModelTest(TestCase):

    def test_unicode_representation(self):
        comment = Comment(body="My comment body")
        self.assertEqual(unicode(comment), "My comment body")
```

Don't forget to import our `Comment` model:

```
from .models import Post, Comment
```

Now let's run our tests to make sure our new test fails:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
F.
======================================================================
FAIL: test_unicode_representation (blog.tests.CommentModelTest)
----------------------------------------------------------------------
Traceback (most recent call last):
...
AssertionError: u'Comment object' != 'My comment body'

----------------------------------------------------------------------
Ran 10 tests in 0.077s

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

Great. After we implement our `__unicode__` method our tests should pass:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
..........
----------------------------------------------------------------------
Ran 10 tests in 0.072s

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

Since we have added a new model, we also need to make sure that this model gets synched to our SQLite database.

```
$ python manage.py syncdb
```

Adding comments on the admin site

Let's add the Comment model to the admin just like we did with the Post model. This involves editing `blog/admin.py` to look like this:

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

```
admin.site.register(Post)
admin.site.register(Comment)
```

If you start the development server again, you will see the Comment model in the admin and you can add comments to the blog posts. However, the point of a blog is to let other users and not only the admin post comments.

Displaying comments on the website

Now we can create comments in the admin interface, but we can't see them on the website yet. Let's display comments on the detail page for each blog post.

At the end of our `content` block in `templates/blog/post_detail.html` let's add the following:

```
<hr>
<h4>Comments</h4>
{% for comment in post.comment_set.all %}
    <p><em>Posted by {{ comment.name }}</em></p>
    {{ comment|linebreaks }}
{% empty %}
    No comments yet.
{% endfor %}
```

Important: We forgot to add a test for this! Why don't you add a test to make sure comments appear on the blog post page.

Now we can see our comments on the website.

Forms

Adding a Comment form

To allow users to create comments we need to accept a form submission. HTML forms are the most common method used to accept user input on web sites and send that data to a server. We can use Django's form framework for this task.

First let's write some tests. We'll need to create a blog `Post` and a `User` for our tests. Let's make a setup method for our tests which creates a post and adds it to the database. The setup method is called before each test in the given test class so that each test will be able to use the `User` and `Post`.

```
class CommentFormTest(TestCase):

    def setUp(self):
        user = get_user_model().objects.create_user('zoidberg')
        self.post = Post.objects.create(author=user, title="My post title")
```

Let's make sure we've imported `get_user_model` and `CommentForm` in our tests file. Our imports should look like this:

```
from django.test import TestCase
from django.contrib.auth import get_user_model
from .models import Post, Comment
from .forms import CommentForm
```

Before we start testing our form remember that we are writing our tests before actually writing our CommentForm code. In other words, we're pretending that we've already written our code in the way that we want it to work, then we're writing tests for that not-yet-written code. Once we've seen that the tests have failed, we then write the actual code. Lastly, we run the tests again against our implemented code and, if necessary, modify the actual code so the tests run successfully.

Our first test should ensure that our form's `__init__` accepts a `post` keyword argument:

```
def test_init(self):
    CommentForm(post=self.post)
```

We want to link our comments to posts by allowing our form to accept a `post` keyword argument. Assuming our `CommentForm` has been written this is how we'd like to use it (**you don't need to type this code anywhere**):

```
>>> form = CommentForm(post=post)   # Without form data
>>> form = CommentForm(request.POST, post=post)   # with form data
```

Important: `request.POST` refers to HTTP POST data and not to the blog post. This is the data accepted from user input.

Our next test should ensure that our test raises an exception if a `post` keyword argument isn't specified:

```
def test_init_without_post(self):
    with self.assertRaises(KeyError):
        CommentForm()
```

Let's run our tests:

```
$ python manage.py test blog
```

```
ImportError: No module named forms
```

We haven't created our forms file yet so our import is failing. Let's create an empty `blog/forms.py` file.

Now we get:

```
$ python manage.py test blog
```

```
ImportError: cannot import name CommentForm
```

We need to create our `CommentForm` model form in `blog/forms.py`. This form will process the data sent from users trying to comment on a blog post and ensure that it can be saved to our blog database. Let's start with something simple:

```
from django import forms
from .models import Comment


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

Here we have created a simple form associated with our Comment model and we have specified that the form handle only a subset of all of the fields on the comment.

Important: Django forms are a powerful way to handle HTML forms. They provide a unified way to check submissions against validation rules and in the case of `ModelForm` subclasses, share any of the associated model's validators. In our example, this will ensure that the Comment `email` is a valid email address.

Now our tests should fail because the `post` keyword argument is not accepted nor required:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
...EF.......
======================================================================
ERROR: test_init (blog.tests.CommentFormTest)
----------------------------------------------------------------------
Traceback (most recent call last):
...
TypeError: __init__() got an unexpected keyword argument 'post'

======================================================================
FAIL: test_init_without_post (blog.tests.CommentFormTest)
----------------------------------------------------------------------
Traceback (most recent call last):
...
AssertionError: KeyError not raised

----------------------------------------------------------------------
Ran 12 tests in 0.080s

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

Our two form tests fail as expected. Let's create a couple more tests for our form before we start fixing it. We should create at least two tests to make sure our form validation works:

1. Ensure that `form.is_valid()` is `True` for a form submission with valid data
2. Ensure that `form.is_valid()` is `False` for a form submission with invalid data (preferably a separate test for each type of error)

This is a good start:

```
def test_valid_data(self):
    form = CommentForm({
        'name': "Turanga Leela",
        'email': "leela@example.com",
        'body': "Hi there",
    }, post=self.post)
    self.assertTrue(form.is_valid())
    comment = form.save()
    self.assertEqual(comment.name, "Turanga Leela")
    self.assertEqual(comment.email, "leela@example.com")
    self.assertEqual(comment.body, "Hi there")
    self.assertEqual(comment.post, self.post)

def test_blank_data(self):
    form = CommentForm({}, post=self.post)
    self.assertFalse(form.is_valid())
    self.assertEqual(form.errors, {
        'name': ['required'],
        'email': ['required'],
        'body': ['required'],
    })
```

It's usually better to test too much than to test too little.

Okay now let's write finally write our form code.

```
from django import forms
from .models import Comment


class CommentForm(forms.ModelForm):

    def __init__(self, *args, **kwargs):
        self.post = kwargs.pop('post')    # the blog post instance
        super(CommentForm, self).__init__(*args, **kwargs)

    def save(self):
        comment = super(CommentForm, self).save(commit=False)
        comment.post = self.post
        comment.save()
        return comment

    class Meta:
        model = Comment
        fields = ('name', 'email', 'body')
```

The `CommentForm` class is instantiated by passing the blog post that the comment was written against as well as the HTTP POST data containing the remaining fields such as comment body and email. The `save` method is overridden here to set the associated blog post before saving the comment.

Let's run our tests again to see whether they pass:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
...F..........
======================================================================
FAIL: test_blank_data (blog.tests.CommentFormTest)
----------------------------------------------------------------------
Traceback (most recent call last):
...
AssertionError: {'body': [u'This field is required.'], 'name': [u'This field is␣
↪required.'], 'email': [u'This field is required.']} != {'body': ['required'], 'name
↪': ['required'], 'email': ['required']}

----------------------------------------------------------------------
Ran 14 tests in 0.086s

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

Our test for blank form data is failing because we aren't checking for the correct error strings. Let's fix that and make sure our tests pass:

```
$ python manage.py test blog
```

```
   Creating test database for alias 'default'...
   ..............
   ----------------------------------------------------------------------
   Ran 14 tests in 0.085s

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

Displaying the comment form

We’ve made a form to create comments, but we still don’t yet have a way for visitors to use the form. The Django test client cannot test form submissions, but WebTest can. We’ll use django-webtest to handle testing the form submission.

Let’s create a test to verify that a form is displayed on our blog post detail page.

First we need to import the `WebTest` class (in `blog/tests.py`):

```
from django_webtest import WebTest
```

Now let’s make our `BlogPostViewTest` class inherit from `WebTest`. Change our `BlogPostViewTest` to inherit from `WebTest` instead of from `TestCase`:

```
class BlogPostViewTest(WebTest):
```

Caution: **Do not** create a new `BlogPostViewTest` class. We already have a `BlogPostViewTest` class with tests in it. If we create a new one, our old class will be overwritten and those tests won’t run anymore. All we want to do is change the parent class for our test from `TestCase` to `WebTest`.

Our tests should continue to pass after this because `WebTest` is a subclass of the Django `TestCase` class that we were using before.

Now let’s add a test to this class:

```
def test_view_page(self):
    page = self.app.get(self.post.get_absolute_url())
    self.assertEqual(len(page.forms), 1)
```

Now let’s update our `PostDetails` view (in `blog/views.py`) to inherit from `CreateView` so we can use it to handle submissions to a `CommentForm`:

```
from django.views.generic import CreateView
from django.shortcuts import get_object_or_404
from .models import Post
from .forms import CommentForm


class PostDetails(CreateView):
    template_name = 'blog/post_detail.html'
    form_class = CommentForm

    def get_post(self):
        return get_object_or_404(Post, pk=self.kwargs['pk'])
    def dispatch(self, *args, **kwargs):
        self.blog_post = self.get_post()
        return super(PostDetails, self).dispatch(*args, **kwargs)

    def get_context_data(self, **kwargs):
        kwargs['post'] = self.blog_post
        return super(PostDetails, self).get_context_data(**kwargs)

post_details = PostDetails.as_view()
```

Now if we run our test we'll see 4 failures. Our blog post detail view is failing to load the page because we aren't passing a `post` keyword argument to our form:

```
$ python manage.py test
Creating test database for alias 'default'...
EEEE...........
======================================================================
ERROR: test_basic_view (blog.tests.BlogPostViewTest)
----------------------------------------------------------------------
...
KeyError: 'post'

----------------------------------------------------------------------
Ran 15 tests in 0.079s

FAILED (errors=4)
```

Let's get the `Post` from the database and pass it to our form. Our view should look something like this now:

```
class PostDetails(CreateView):
    template_name = 'blog/post_detail.html'
    form_class = CommentForm

    def get_post(self):
        return get_object_or_404(Post, pk=self.kwargs['pk'])

    def dispatch(self, *args, **kwargs):
        self.blog_post = self.get_post()
        return super(PostDetails, self).dispatch(*args, **kwargs)

    def get_form_kwargs(self):
        kwargs = super(PostDetails, self).get_form_kwargs()
        kwargs['post'] = self.blog_post
        return kwargs

    def get_context_data(self, **kwargs):
        kwargs['post'] = self.blog_post
        return super(PostDetails, self).get_context_data(**kwargs)
```

Now when we run our tests we'll see the following assertion error because we have not yet added the comment form to our blog detail page:

```
$ python manage.py test blog
```

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

```
======================================================================
FAIL: test_view_page (blog.tests.BlogPostViewTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/zoidberg/learning-django-by-testing/test/myblog/blog/tests.py", line 81,
↪ in test_view_page
    self.assertEqual(len(page.forms), 1)
AssertionError: 0 != 1

----------------------------------------------------------------------
Ran 15 tests in 0.099s

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

Let's add a comment form to the bottom of our `content` block in our blog post detail template (`templates/post_detail.html`):

```
<h5>Add a comment</h5>
<form method="post">
    {{ form.as_table }}
    <input type="submit" value="Create Comment">
</form>
```

Now our tests pass again:

```
$ python manage.py test blog
```

```
   Creating test database for alias 'default'...
   ...............
   ----------------------------------------------------------------------
   Ran 15 tests in 0.108s

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

Let's test that our form actually submits. We should write two tests: one to test for errors, and one to test a successful form submission.

```
def test_form_error(self):
    page = self.app.get(self.post.get_absolute_url())
    page = page.form.submit()
    self.assertContains(page, "This field is required.")

def test_form_success(self):
    page = self.app.get(self.post.get_absolute_url())
    page.form['name'] = "Phillip"
    page.form['email'] = "phillip@example.com"
    page.form['body'] = "Test comment body."
    page = page.form.submit()
    self.assertRedirects(page, self.post.get_absolute_url())
```

Now let's run our tests:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
...EE...........
======================================================================
ERROR: test_form_error (blog.tests.BlogPostViewTest)
----------------------------------------------------------------------
...
AppError: Bad response: 403 FORBIDDEN (not 200 OK or 3xx redirect for http://
↪localhost/post/1)
...

======================================================================
ERROR: test_form_success (blog.tests.BlogPostViewTest)
----------------------------------------------------------------------
...
AppError: Bad response: 403 FORBIDDEN (not 200 OK or 3xx redirect for http://
↪localhost/post/1)
...

----------------------------------------------------------------------
Ran 17 tests in 0.152s

FAILED (errors=2)
```

We got a HTTP 403 error because we forgot to add the cross-site request forgery token to our form. Every HTTP POST request made to our Django site needs to include a CSRF token. Let's change our form to add a CSRF token field to it:

```
<form method="post">
    {% csrf_token %}
    {{ form.as_table }}
    <input type="submit" value="Create Comment">
</form>
```

Now only one test fails:

```
$ python manage.py test blog
```

```
Creating test database for alias 'default'...
....E...........
======================================================================
ERROR: test_form_success (blog.tests.BlogPostViewTest)
----------------------------------------------------------------------
...
ImproperlyConfigured: No URL to redirect to.  Either provide a url or define a get_
↪absolute_url method on the Model.

----------------------------------------------------------------------
Ran 17 tests in 0.0.166s

FAILED (errors=1)
```

Let's fix this by adding a `get_success_url` to our view:

```
def get_success_url(self):
    return self.get_post().get_absolute_url()
```

Now our tests pass again and we can submit comments as expected.