
Learning Django by Testing Documentation

Release v3.0

San Diego Python

Aug 07, 2017

Contents

1	Why test-driven development?	3
2	Prerequisites	5
3	The Project: building a blog	7
4	Contents	9
4.1	Getting started	9
4.2	Models	12
4.3	Views and Templates	19
4.4	More Views	30
4.5	Forms	32
4.6	The Testing Game	40
4.7	Custom template tags	44
4.8	Database Migrations	48
4.9	Readable URLs	49
4.10	Adding Gravatars	52
5	Getting Help & Contributing	55

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](#).

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 3](#) (3.4 is recommended)
- [Install Django 1.7](#)
- [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.

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 Mac OS or Linux terminal or in the Windows command prompt should show the version of Python. For this workshop you should have a 3.x version of Python.

```
$ python -V
```

You should also have `pip` installed on your machine. Pip is a dependency management tool for installing and managing Python dependencies. First let's install Django 1.7:

```
$ pip install Django==1.7
Downloading/unpacking Django==1.7
  Downloading Django-1.7-py2.py3-none-any.whl (7.4MB): 7.4MB downloaded
Installing collected packages: Django
Successfully installed Django
Cleaning up...
```

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. You should have Django 1.7 installed.

```
$ python -c "import django; print(django.get_version())"
1.7
```

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 browser to the appropriate Python code.

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

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying sessions.0001_initial... OK
$ python manage.py createsuperuser
Username (leave blank to use 'zoidberg'):
Email address: zoidberg@example.com
Password: ***
Password (again): ***
Superuser created successfully.
```

After running this command, there will be a database file `db.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 entries and comments on blog entries.

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](#).

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 entries.

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.

Python Package Requirements File

We want to use a few more Python packages besides Django. We'll plan to use [WebTest](#) and [django-webtest](#) for our functional tests. Let's install those also:

```
$ pip install webtest django-webtest
Downloading/unpacking webtest
  Downloading WebTest-2.0.16.zip (88kB): 88kB downloaded
  ...
Downloading/unpacking django-webtest
  Downloading django-webtest-1.7.7.tar.gz
  ...
Successfully installed webtest django-webtest six WebOb waitress beautifulsoup4
Cleaning up...
```

We don't want to manually install our dependencies every time. Let's create a [requirements file](#) listing our dependencies so we don't have to type them all out every time we setup our website on a new computer or anytime a package version updates.

First let's use [pip freeze](#) to list our dependencies and their versions:

```
$ pip freeze
Django==1.7
WebOb==1.4
WebTest==2.0.16
beautifulsoup4==4.3.2
django-webtest==1.7.7
six==1.8.0
waitress==0.8.9
```

We care about the Django, WebTest, and django-webtest lines here. The other packages are sub-dependencies that were automatically installed and don't need to worry about them. Let's create our `requirements.txt` with instructions for installing these packages with the versions we have installed now:

```
Django==1.7
WebTest==2.0.16
django-webtest==1.7.7
```

This file will allow us to install all Python dependencies at once with just one command. Whenever our dependency files are upgraded or if we setup a new development environment for our Django website we'll need to run:

```
$ pip install -r requirements.txt
```

Note: Note that we do not need to type this command right now since we have already installed all dependencies.

Hint: If you are using `virtualenvwrapper` (or just `virtualenv`), you can create a new `virtualenv`, and test your `requirements.txt` file. With `virtualenvwrapper`:

```
$ mkvirtualenv tddd-env2
$ workon tddd-env2
$ pip install -r requirements.txt
$ pip freeze
$ deactivate
$ workon YOUR_ORIGINAL_VENV
```

Or with `virtualenv`:

```
$ virtualenv venv2
$ source venv2/bin/activate
$ pip install -r requirements.txt
$ pip freeze
$ deactivate
$ source venv/bin/activate # or whatever your original virtualenv was
```

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 entries 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 and a subdirectory, `migrations`:

```
__init__.py
admin.py
migrations
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 `migrate`.

```
INSTALLED_APPS = (
    'django.contrib.admin',
```



```
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',

'blog',
)
```

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

```
+-- blog
|   +- admin.py
|   +- __init__.py
|   +- migrations
|   |   +- __init__.py
|   +- models.py
|   +- tests.py
|   +- views.py
+-- db.sqlite3
+-- manage.py
+-- myblog
|   +- __init__.py
|   +- settings.py
|   +- urls.py
|   +- wsgi.py
+-- requirements.txt
```

Creating a model

First, let's create a blog entry model by writing the code below in our *blog/models.py* file. Models are objects used to interface with your data, and are described in the [Django model documentation](#). Our model will correspond to a database table which will hold the data for our blog entry. A blog entry will be represented by an instance of our *Entry* model class and each *Entry* model instance will identify a row in our database table.

```
from django.db import models

class Entry(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 entry in our project – is a row in the spreadsheet.

To create the database table for our *Entry* model we need to make a migration and run migrate again:

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

Don't worry about the details of migrations just yet, we'll learn about them in a later section of the tutorial. For now, just think of migrations as Django's way of managing changes to models and the corresponding database.

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 entries from the admin site

We don't want to manually add entries 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 entries. Fortunately Django's admin interface can do just that.

In order to create blog entries from the [admin interface](#) we need to register our `Entry` model with the admin site. We can do this by modifying our `blog/admin.py` file to register the `Entry` model with the admin interface:

```
from django.contrib import admin

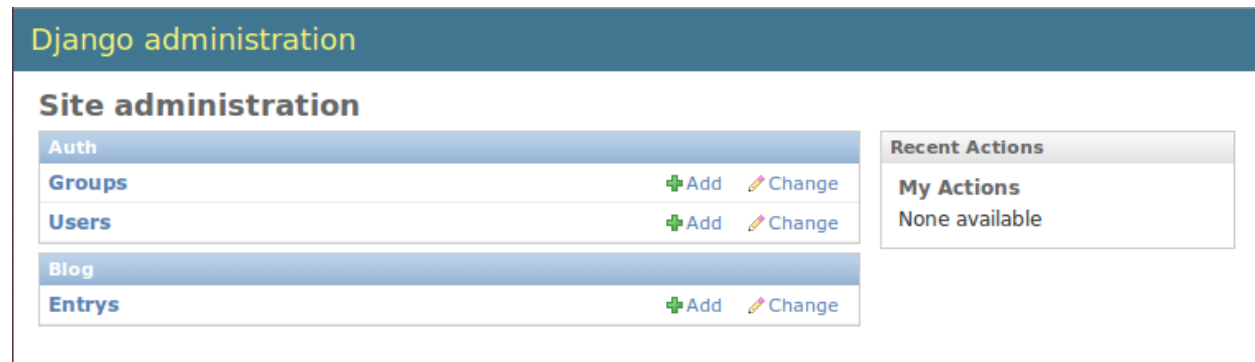
from .models import Entry

admin.site.register(Entry)
```

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

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

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



The screenshot shows the Django administration interface. At the top, there's a header "Django administration" in a blue bar. Below it, the "Site administration" section is visible. It contains two main categories: "Auth" and "Blog". Under "Auth", there are "Groups" and "Users", each with a green plus icon for "Add" and a pencil icon for "Change". Under "Blog", there is "Entries" with similar "Add" and "Change" links. To the right, a "Recent Actions" sidebar shows "My Actions" as "None available".

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

Django administration Welcome, **littlepony**. [Change password](#) / [Log out](#)

[Home](#) > [Blog](#) > [Entrys](#) > Add entry

Add entry

Title:

Author: [+](#)

Body:

There is a first to everything

Our blog entry was created

[Home](#) > [Blog](#) > [Entrys](#)

Select entry to change Add entry [+](#)

Action: 0 of 1 selected

<input type="checkbox"/>	Entry
<input type="checkbox"/>	Entry object

1 entry

Our first test: `__str__` method

In the admin change list our entries have the unhelpful title *Entry object*. Add another entry just like the first one, they will look identical. We can customize the way models are referenced by creating a `__str__` 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 EntryModelTest(TestCase):

    def test_string_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_string_representation (blog.tests.EntryModelTest)
-----
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.

Note: `django.test.TestCase` extends the `unittest.TestCase` class. Anything you would do in the base `unittest` class will work in Django's `TestCase` as well.

You can read more about `django.test.TestCase` in the Django documentation and the `unittest.TestCase` parent class in the Python documentation.

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

```
from django.test import TestCase

from .models import Entry

class EntryModelTest(TestCase):

    def test_string_representation(self):
        entry = Entry(title="My entry title")
        self.assertEqual(str(entry), entry.title)
```

Now let's run our tests again:

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

```

Creating test database for alias 'default'...
F
=====
FAIL: test_string_representation (blog.tests.EntryModelTest)
-----
Traceback (most recent call last):
...
AssertionError: 'Entry object' != 'My entry title'
- Entry object
+ My entry title

-----
Ran 1 test in 0.002s

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

```

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

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

```

from django.db import models

class Entry(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 __str__(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 entry titles in the list of entries.

The screenshot shows the Django administration interface. At the top, it says "Django administration" and "Welcome, littlepony. Change password / Log out". Below that, there's a breadcrumb "Home > Blog > Entrys". A green message box says "The entry 'First Entry' was added successfully." Below that, there's a section titled "Select entry to change" with a button "Add entry +". There's a form with "Action:" and a dropdown menu, a "Go" button, and "0 of 1 selected". Below that, there's a table with two rows: "Entry" and "First Entry". The "First Entry" row is highlighted. At the bottom, it says "1 entry".

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 0.000s

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

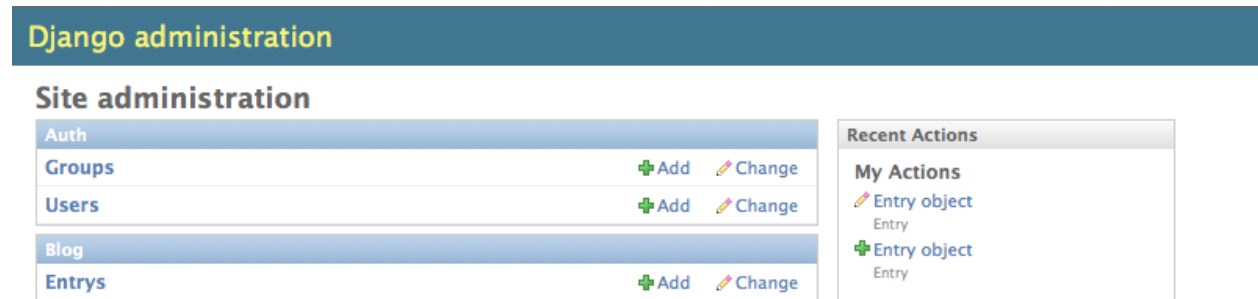
We’ve just written our first test and fixed our code to make our test pass.

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.

Another Test: Entries

Did you notice that the pluralization of entry is misspelled in the admin interface? “Entrys” should instead read “Entries”. Let’s write a test to verify that when Django correctly pluralizes “entry” to “entries”.



Let’s add a test to our `EntryModelTest` class:

```
def test_verbose_name_plural(self):
    self.assertEqual(str(Entry._meta.verbose_name_plural), "entries")
```

Note: This test uses the model `_meta` class (created based on the `Meta` class we will define). This is an example of an advanced Django feature. The `_meta` class is currently undocumented.

Now let’s make our test pass by specifying the verbose name for our model.

Add a `Meta` inner class inside our `Entry` model, like this:

```
class Entry(models.Model):

    # The rest of our model code

    class Meta:
        verbose_name_plural = "entries"
```

Hint: See the Django documentation for information on `verbose_name_plural` in the Meta class.

Views and Templates

Now we can create blog entries and see them in the admin interface, but no one else can see our blog entries 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 definition to the bottom of 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 entries to look visually similar but the actual text of a given blog entry varies. We will have a single template for what all of our blog entries and the template will contain variables that get replaced when a blog entry 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
|   +- admin.py
|   +- __init__.py
|   +- migrations
|   |   +- 0001_initial.py
|   |   +- __init__.py
|   +- models.py
|   +- tests.py
|   +- views.py
+-- db.sqlite3
+-- manage.py
+-- myblog
|   +- __init__.py
|   +- settings.py
|   +- urls.py
|   +- wsgi.py
+-- requirements.txt
+-- 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>My Blog</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 inform Django of this new `templates` directory by adding this at the bottom of our `myblog/settings.py` file:

```
# Template files
# https://docs.djangoproject.com/en/1.7/topics/templates/

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'
```

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 include, url
from django.contrib import admin

from . import views

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

Now let's visit <http://localhost:8000/> in a web browser to check our work. (Restart your server with the command `python manage.py runserver`.) 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 3 tests in 0.032s

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 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. Here is more information on [template inheritance with blocks](#).

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

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

```

<head>
  <title>My Blog</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 entry information 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 HomePageTests(TestCase):

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

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

    def test_one_entry(self):
        Entry.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_entries(self):
        Entry.objects.create(title='1-title', body='1-body', author=self.user)
        Entry.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_entry (blog.tests.HomePageTests)
-----
Traceback (most recent call last):
...
AssertionError: False is not true : Couldn't find '1-title' in response
=====
FAIL: test_two_entries (blog.tests.HomePageTests)
-----
Traceback (most recent call last):
...
AssertionError: False is not true : Couldn't find '1-title' in response
-----

Ran 5 tests in 0.052s

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

Updating our views

One easy way to get all our entries 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 Entry
```

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

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

That small change will provide a `entry_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 entries. Let's replace our `templates/index.html` file with the following:

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

{% block content %}
    {% for entry in entry_list %}
        <article>

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

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

            <p></p>

            {{ entry.body|linebreaks }}

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

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

Tip: Notice that we didn't specify the name `entry_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 `entry_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 entries in our admin, they should show up on the homepage. What happens if there are no entries? We should add a test for that

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

This test gives us the expected failure

```
Creating test database for alias 'default'...
..F...
=====
FAIL: test_no_entries (blog.tests.HomePageTests)
-----
Traceback (most recent call last):
...
AssertionError: False is not true : Couldn't find 'No blog entries yet.' in response
-----
Ran 6 tests in 0.044s

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

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

Hint: Remember that the phrase in the `empty` clause must contain the same phrase we check for in our test (“No blog entries yet.”).

What about viewing an individual blog entry?

Blog Entries, URLs, and Views

For simplicity, let’s agree on a project guideline to form our urls to look like `http://myblog.com/ID/` where `ID` is the database ID of the specific blog entry that we want to display. In this section, we will be creating a *blog entry detail* page and using our project’s url guideline.

Before we create this page, let’s move the template content that displays our blog entries on our homepage (`templates/index.html`) into a new, separate template file so we can reuse the blog entry display logic on our *blog entry details* page.

Let’s make a template file called `templates/_entry.html` and put the following in it:

```
<article>

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

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

    <p></p>

    {{ entry.body|linebreaks }}

</article>
```

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 entry in entry_list %}
        {% include "_entry.html" with entry=entry only %}
    {% empty %}
        <p>No blog entries yet.</p>
    {% endfor %}
{% endblock content %}
```

Tip: We use the `with entry=entry 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.

Great job. Now, let's write a test our new blog entry pages:

```
class EntryViewTest(TestCase):

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

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

This test fails because we didn't define the `get_absolute_url` method for our `Entry` model ([Django Model Instance Documentation](#)). We will need an absolute URL to correspond to an individual blog entry.

We need to create a URL and a view for blog entry pages now. We'll make a new `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 url

from . import views

urlpatterns = [
    url(r'^(?P<pk>\d+)/$', views.EntryDetail.as_view(), name='entry_detail'),
]
```

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

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

import blog.urls
from . import views
```

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

Remember, we are working on creating a way to see individual entries. Now we need to define an `EntryDetail` view class in our `blog/views.py` file. To implement our blog entry 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 the contents of `blog/views.py` file with the following:

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

class EntryDetail(DetailView):
    model = Entry
```

Let's look at how to create the `get_absolute_url()` function which should return the individual, absolute entry detail URL for each blog entry. We should create a test first. Let's add the following test to our `EntryModelTest` class:

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

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

```
from django.core.urlresolvers import reverse

# And in our Entry model class...

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

Tip: For further reading about the utility function, `reverse`, see the Django documentation on `django.core.urlresolvers.reverse`.

Now, run the tests again. We should have passing tests since we just defined a `get_absolute_url` method.

Let's make the blog entry detail view page actually display a blog entry. First we'll write some tests in our `EntryViewTest` class:

```
def test_title_in_entry(self):
    response = self.client.get(self.entry.get_absolute_url())
    self.assertContains(response, self.entry.title)

def test_body_in_entry(self):
    response = self.client.get(self.entry.get_absolute_url())
    self.assertContains(response, self.entry.body)
```

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_basic_view (blog.tests.EntryViewTest)
-----
Traceback (most recent call last):
...
django.template.base.TemplateDoesNotExist: blog/entry_detail.html
=====
ERROR: test_body_in_entry (blog.tests.EntryViewTest)
-----
Traceback (most recent call last):
...
django.template.base.TemplateDoesNotExist: blog/entry_detail.html
=====
ERROR: test_title_in_entry (blog.tests.EntryViewTest)
-----
Traceback (most recent call last):
...
django.template.base.TemplateDoesNotExist: blog/entry_detail.html
-----
Ran 10 tests in 0.136s

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

These errors are telling us that we're referencing a `blog/entry_detail.html` template but we haven't created that file yet.

We're very close to being able to see individual blog entry details. Let's do it. First, create a `templates/blog/entry_detail.html` as our blog entry detail view template. The `DetailView` will use an `entry` context variable to reference our `Entry` model instance. Our new blog entry detail view template should look similar to this:

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

{% block content %}
    {% include "_entry.html" with entry=entry only %}
{% endblock %}
```

Now our tests should pass again:

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

```
Creating test database for alias 'default'...
.....
-----
Ran 10 tests in 0.139s

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

More Views

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

Adding a Comment model

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

```
class Comment(models.Model):
    entry = models.ForeignKey(Entry)
    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)
```

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

```
$ python manage.py makemigrations
Migrations for 'blog':
  0002_auto_20141019_0232.py:
    - Create model Comment
    - Change Meta options on entry
$ python manage.py migrate
Operations to perform:
  Apply all migrations: contenttypes, blog, admin, auth, sessions
Running migrations:
  Applying blog.0002_auto_20141019_0232... OK
```

Before we create a `__str__` method for our Comment model similar to the one we previously added for our Entry model, let's create a test in `blog/tests.py`.

Our test should look very similar to the `__str__` test we wrote in `EntryModelTest` earlier. This should suffice:

```
class CommentModelTest(TestCase):

    def test_string_representation(self):
        comment = Comment(body="My comment body")
        self.assertEqual(str(comment), "My comment body")
```

Don't forget to import our Comment model:

```
from .models import Entry, 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_string_representation (blog.tests.CommentModelTest)
-----
Traceback (most recent call last):
  ...
AssertionError: 'Comment object' != 'My comment body'
```

```
- Comment object
+ My comment body

-----

Ran 11 tests in 0.154s

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

Great. So it looks like our test fails. Now we should implement the `__str__` method for the comment body, an exercise we leave to the reader. After implementing the method, run the test again to see it pass:

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

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

Ran 11 tests in 0.085s

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

Adding comments on the admin site

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

```
from django.contrib import admin

from .models import Entry, Comment

admin.site.register(Entry)
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 entries. 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 entry.

After the `<hr>` element inside of our content block in `templates/blog/entry_detail.html` let's add the following:

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

Important: We forgot to add tests for this! Why don't you add a test to make sure comments appear on the blog entry page and a test to make sure the "No comments yet" message shows up appropriately. These tests should probably be added to our `EntryViewTest` class.

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 `Entry` and a `User` for our tests. Let's make a `setup` method for our tests which creates an entry 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 `Entry`.

```
class CommentFormTest(TestCase):

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

Let's make sure we've imported `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 .forms import CommentForm
from .models import Entry, Comment
```

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 an `entry` keyword argument:

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

We want to link our comments to entries by allowing our form to accept an `entry` 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(entry=entry) # Without form data
>>> form = CommentForm(request.POST, entry=entry) # with form data
```

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

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

Let's run our tests:

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

```
ImportError: No module named 'blog.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 entry 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 `entry` 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 'entry'
=====
FAIL: test_init_without_entry (blog.tests.CommentFormTest)
-----
```

```
Traceback (most recent call last):
...
AssertionError: KeyError not raised

-----
Ran 15 tests in 0.097s

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",
    }, entry=self.entry)
    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.entry, self.entry)

def test_blank_data(self):
    form = CommentForm({}, entry=self.entry)
    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 finally write our form code.

```
from django import forms

from .models import Comment

class CommentForm(forms.ModelForm):

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

    def __init__(self, *args, **kwargs):
        self.entry = kwargs.pop('entry') # the blog entry instance
```

```

    super().__init__(*args, **kwargs)

    def save(self):
        comment = super().save(commit=False)
        comment.entry = self.entry
        comment.save()
        return comment

```

The `CommentForm` class is instantiated by passing the blog entry 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 entry 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: {'name': ['This field is required.'], 'email': ['Thi[55 chars]d.']} !
↳= {'name': ['required'], 'email': ['required'], 'body': ['required']}
-----

Ran 17 tests in 0.178s

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 17 tests in 0.179s

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 entry detail page.

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

```
from django_webtest import WebTest
```

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

```
class EntryViewTest(WebTest):
```

Caution: Do not create a new `EntryViewTest` class. We already have an `EntryViewTest` 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.entry.get_absolute_url())
    self.assertEqual(len(page.forms), 1)
```

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

```
from django.shortcuts import get_object_or_404
from django.views.generic import CreateView

from .forms import CommentForm
from .models import Entry

class EntryDetail(CreateView):
    model = Entry
    template_name = 'blog/entry_detail.html'
    form_class = CommentForm
```

Now if we run our test we'll see 6 failures. Our blog entry detail view is failing to load the page because we aren't passing an `entry` keyword argument to our form:

```
$ python manage.py test
Creating test database for alias 'default'...
.....EEEEEE....
=====
ERROR: test_basic_view (blog.tests.EntryViewTest)
-----
...
KeyError: 'entry'
-----

Ran 18 tests in 0.079s

FAILED (errors=6)
Destroying test database for alias 'default'...
```

Let's get the `Entry` from the database and pass it to our form. We need to add a `get_form_kwargs` method, and a `get_context_data` method to our view:

```
def get_form_kwargs(self):
    kwargs = super().get_form_kwargs()
```



```

kwargs['entry'] = self.get_object()
return kwargs

def get_context_data(self, **kwargs):
    d = super().get_context_data(**kwargs)
    d['entry'] = self.get_object()
    return d

```

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.EntryViewTest)
-----
Traceback (most recent call last):
...
AssertionError: 0 != 1
-----

Ran 18 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 entry detail template (templates/blog/entry_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
```

```

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

Ran 18 tests in 0.106s

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

```

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

```

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

```

```
def test_form_success(self):
    page = self.app.get(self.entry.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.entry.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.EntryViewTest)
-----
Traceback (most recent call last):
...
webtest.app.AppError: Bad response: 403 FORBIDDEN (not 200 OK or 3xx redirect for_
↳http://localhost/1/)
b'\n<!DOCTYPE html>\n<html lang="en">\n<head>\n  <meta http-equiv="content-type"
↳content="text/html; charset=utf-8">\n  <meta name="robots" content="NONE,NOARCHIVE">
↳\n  <title>403 Forbidden</title>\n  <style type="text/css">\n    html * { padding:0;
↳margin:0; }\n    body * { padding:10px 20px; }\n    body * * { padding:0; }\n
↳body { font:small sans-serif; background:#eee; }\n    body>div { border-bottom:1px
↳solid #ddd; }\n    h1 { font-weight:normal; margin-bottom:.4em; }\n    h1 span {
↳font-size:60%; color:#666; font-weight:normal; }\n    #info { background:#f6f6f6; }
↳\n    #info ul { margin: 0.5em 4em; }\n    #info p, #summary p { padding-top:10px; }
↳\n    #summary { background: #ffc; }\n    #explanation { background:#eee; border-
↳bottom: 0px none; }\n  </style>\n</head>\n<body>\n<div id="summary">\n  <h1>
↳Forbidden <span>(403)</span></h1>\n  <p>CSRF verification failed. Request aborted.</
↳p>\n\n\n  <p>You are seeing this message because this site requires a CSRF cookie
↳when submitting forms. This cookie is required for security reasons, to ensure that
↳your browser is not being hijacked by third parties.</p>\n  <p>If you have
↳configured your browser to disable cookies, please re-enable them, at least for
↳this site, or for &#39;same-origin&#39; requests.</p>\n\n</div>\n\n<div id=
↳"explanation">\n  <p><small>More information is available with DEBUG=True.</small></
↳p>\n</div>\n\n</body>\n</html>\n'
```

```
=====
ERROR: test_form_success (blog.tests.EntryViewTest)
-----
```

```
Traceback (most recent call last):
...
webtest.app.AppError: Bad response: 403 FORBIDDEN (not 200 OK or 3xx redirect for_
↳http://localhost/1/)
b'\n<!DOCTYPE html>\n<html lang="en">\n<head>\n  <meta http-equiv="content-type"
↳content="text/html; charset=utf-8">\n  <meta name="robots" content="NONE,NOARCHIVE">
↳\n  <title>403 Forbidden</title>\n  <style type="text/css">\n    html * { padding:0;
↳margin:0; }\n    body * { padding:10px 20px; }\n    body * * { padding:0; }\n
↳body { font:small sans-serif; background:#eee; }\n    body>div { border-bottom:1px
↳solid #ddd; }\n    h1 { font-weight:normal; margin-bottom:.4em; }\n    h1 span {
↳font-size:60%; color:#666; font-weight:normal; }\n    #info { background:#f6f6f6; }
↳\n    #info ul { margin: 0.5em 4em; }\n    #info p, #summary p { padding-top:10px; }
↳\n    #summary { background: #ffc; }\n    #explanation { background:#eee; border-
↳bottom: 0px none; }\n  </style>\n</head>\n<body>\n<div id="summary">\n  <h1>
↳Forbidden <span>(403)</span></h1>\n  <p>CSRF verification failed. Request aborted.</
↳p>\n\n\n  <p>You are seeing this message because this site requires a CSRF cookie
↳when submitting forms. This cookie is required for security reasons, to ensure that
↳your browser is not being hijacked by third parties.</p>\n  <p>If you have
↳configured your browser to disable cookies, please re-enable them, at least for
↳this site, or for &#39;same-origin&#39; requests.</p>\n\n</div>\n\n<div id=
↳"explanation">\n  <p><small>More information is available with DEBUG=True.</small></
↳p>\n</div>\n\n</body>\n</html>\n'
```

38 your browser is not being hijacked by third parties.</p>\n <p>If you have
Chapter 4. Contents

```
↳configured your browser to disable cookies, please re-enable them, at least for
↳this site, or for &#39;same-origin&#39; requests.</p>\n\n</div>\n\n<div id=
↳"explanation">\n  <p><small>More information is available with DEBUG=True.</small></
↳p>\n</div>\n\n</body>\n</html>\n'
```

```
-----
Ran 20 tests in 0.110s
FAILED (errors=2)
Destroying test database for alias 'default'...
```

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.EntryViewTest)
-----
Traceback (most recent call last):
...
AttributeError: 'Comment' object has no attribute 'get_absolute_url'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
...
django.core.exceptions.ImproperlyConfigured: No URL to redirect to. Either provide a
↳url or define a get_absolute_url method on the Model.

-----
Ran 20 tests in 0.141s
FAILED (errors=1)
Destroying test database for alias 'default'...
```

Let's fix this by adding a `get_success_url` to our view, `EntryDetail`, in `blog/views.py`:

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

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

The Testing Game

Test Coverage

It's important to test all your code. Code coverage is frequently used as a measuring stick for a developer's success in creating quality tests. The basic rule of thumb is comprehensive tests should execute every line of code.

`Coverage`, a tool that measures code coverage for Python code, will be used to check what percentage of the tutorial code is being tested.

Installing Coverage

First let's install coverage:

```
$ pip install coverage
```

Before we continue, we need to remember to add this new dependency to our `requirements.txt` file. Let's use `pip freeze` to discover the version of `coverage` we installed:

```
$ pip freeze
Django==1.7
WebOb==1.4
WebTest==1.4.3
beautifulsoup4==4.3.2
coverage==3.7.1
django-webtest==1.7.7
six==1.8.0
sqlparse==0.1.13
waitress==0.8.9
```

Now let's add `coverage` to our `requirements.txt` file:

```
coverage==3.7.1
Django==1.7
WebTest==2.0.16
django-webtest==1.7.7
```

Using Coverage

Now let's run our tests. As we run our tests from the command line, `coverage` records and creates a coverage report:

```
$ coverage run --include='./*' manage.py test
Creating test database for alias 'default'...
.....
-----
Ran 20 tests in 0.163s

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

Let's take a look at our code coverage report:

```
$ coverage report
```

Name	Stmts	Miss	Cover
blog/__init__	0	0	100%
blog/admin	4	0	100%
blog/forms	14	0	100%
blog/migrations/0001_initial	6	0	100%
blog/migrations/0002_auto_20141019_0232	5	0	100%
blog/migrations/__init__	0	0	100%
blog/models	23	0	100%
blog/tests	101	0	100%
blog/urls	3	0	100%
blog/views	18	0	100%
manage	6	0	100%
myblog/__init__	0	0	100%
myblog/settings	19	0	100%
myblog/urls	5	0	100%
myblog/views	5	0	100%
TOTAL	209	0	100%

Let's take a look at the coverage report. On the left, the report shows the name of the file being tested. `Stmts`, or code statements, indicate the number of lines of code that could be tested. `Miss`, or Missed lines, indicates the number of lines that are not executed by the unit tests. `Cover`, or Coverage, is the percentage of code covered by the current tests (equivalent to $(\text{Stmts} - \text{Miss}) / \text{Stmts}$). For example, `myblog/views` has 18 code statements that can be tested. We see that our tests did not miss testing any statements for a Code Coverage of 100%.

Important: Note that code coverage can only indicate that you've forgotten tests; it will not tell you whether your tests are good. Don't use good code coverage as an excuse to write lower quality tests.

HTML Coverage Report

Our current command-line coverage reports are useful, but they aren't very detailed. Fortunately coverage includes a feature for generating HTML coverage reports that visually demonstrate coverage by coloring our code based on the results.

Let's prettify the coverage report above into HTML format by running the following command:

```
$ coverage html
```

This command will create a `htmlcov` directory containing our test coverage. The `index.html` is the overview file which links to the other files. Let's open up our `htmlcov/index.html` in our web browser.

Our HTML coverage report should look something like this:

Coverage report: 100%

<i>Module</i>	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>coverage</i>
blog/__init__	0	0	0	100%
blog/admin	4	0	0	100%
blog/forms	14	0	0	100%
blog/models	21	0	0	100%
blog/tests	87	0	0	100%
blog/urls	2	0	0	100%
blog/views	22	0	0	100%
manage	6	0	0	100%
myblog/__init__	0	0	0	100%
myblog/settings	29	0	0	100%
myblog/urls	5	0	0	100%
myblog/views	6	0	0	100%
Total	196	0	0	100%

coverage.py v3.7.1

Hot-keys on this page

n s m x c change column sorting

Branch Coverage

So far we've been testing statement coverage to ensure we execute every line of code during our tests. We can do better by ensuring every code branch is taken. The coverage documentation contains a good description of [branch coverage](#).

From now on we will add the `--branch` argument when we record code coverage. Let's try it on our tests:

```
$ coverage run --include='./*' --branch manage.py test
$ coverage report
Name                               Stmts  Miss Branch BrMiss  Cover
-----
blog/__init__                       0      0      0      0   100%
blog/admin                          4      0      0      0   100%
blog/forms                          14      0      0      0   100%
blog/migrations/0001_initial         6      0      0      0   100%
blog/migrations/0002_auto_20141019_0232 5      0      0      0   100%
blog/migrations/__init__            0      0      0      0   100%
blog/models                         23      0      0      0   100%
blog/tests                          101     0      0      0   100%
blog/urls                           3      0      0      0   100%
blog/views                          18      0      0      0   100%
manage                              6      0      2      1    88%
myblog/__init__                     0      0      0      0   100%
myblog/settings                     19      0      0      0   100%
myblog/urls                          5      0      0      0   100%
myblog/views                         5      0      0      0   100%
-----
TOTAL                              209     0      2      1    99%
```

Notice the new `Branch` and `BrMiss` columns and note that we are missing a branch in our `manage.py` file. We'll take a look at that later.

Coverage Configuration

Coverage allows us to specify a configuration file (`.coveragerc` files) to specify default coverage attributes. The documentation explains how `.coveragerc` work.

Let's add a `.coveragerc` file to our project that looks like this:

```
[run]
include = ./*
branch = True
```

Now we can run coverage without any extra arguments:

```
$ coverage run manage.py test
```

Inspecting Missing Coverage

Now let's figure out why our branch coverage is not 100%. First we need to regenerate the HTML coverage report and have a look at it:

```
$ coverage html
```

Coverage report: 99%

<i>Module</i>	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>branches</i>	<i>partial</i>	<i>coverage</i>
blog/___init___	0	0	0	0	0	100%
blog/admin	4	0	0	0	0	100%
blog/forms	14	0	0	0	0	100%
blog/models	21	0	0	0	0	100%
blog/tests	87	0	0	0	0	100%
blog/urls	2	0	0	0	0	100%
blog/views	22	0	0	0	0	100%
manage	6	0	0	2	1	88%
myblog/___init___	0	0	0	0	0	100%
myblog/settings	29	0	0	0	0	100%
myblog/urls	5	0	0	0	0	100%
myblog/views	6	0	0	0	0	100%
Total	196	0	0	2	1	99%

coverage.py v3.7.1

Let's click on `manage` to see why our `manage.py` file has 88% coverage:

Coverage for **manage** : 88%

6 statements 6 run 0 missing 0 excluded 1 partial

```
1 #!/usr/bin/env python
2 import os
3 import sys
4
5 if __name__ == "__main__":
6     os.environ.setdefault("DJANGO_SETTINGS_MODULE", "myblog.settings")
7
8     from django.core.management import execute_from_command_line
9
10    execute_from_command_line(sys.argv)
```

◀ [index](#) [coverage.py v3.7.1](#)

We're missing the `False` case for that `if` statement in our `manage.py` file. We always run `manage.py` from the command line so that code is always executed.

We don't intend to ever test that missing branch, so let's ignore the issue and accept our imperfect coverage statistics.

Tip: For extra credit, figure out how we can exclude that `if __name__ == "__main__":` line from our coverage count. Check out the [.coveragerc](#) documentation for help.

Custom template tags

Let's make our blog list recent entries in the sidebar.

How are we going to do this? We could loop through blog entries in our `base.html` template, but that means we would need to include a list of our recent blog entries in the template context for all of our views. That could result in duplicate code and we don't like duplicate code.

Tip: If you didn't fully understand the last paragraph, that's okay. **DRY** or "Don't Repeat Yourself" is a rule of thumb for good programming practice. You might want to read through the Django [template documentation](#) again later.

To avoid duplicate code, let's create a [custom template tag](#) to help us display recent blog entries in the sidebar on every page.

Note: A custom template tag that itself fires a SQL query enables our HTML templates to add more SQL queries to our view. That hides some behavior. It's too early at this point, but that query should be cached if we expect to use

this often.

Where

Let's create a template library called `blog_tags`. Why `blog_tags`? Because naming our tag library after our app will make our template imports more understandable. We can use this template library in our templates by writing `{% load blog_tags %}` near the top of our template file.

Create a `templatetags` directory in our `blog` app and create two empty Python files within this directory: `blog_tags.py` (which will hold our template library code) and `__init__.py` (to make this directory into a Python package).

We should now have something like this:

```
blog
+- admin.py
+- forms.py
+- __init__.py
+- migrations
|   +- 0001_initial.py
|   +- 0002_auto_20141019_0232.py
|   +- __init__.py
+- models.py
+- templatetags
|   +- blog_tags.py
|   +- __init__.py
+- tests.py
+- urls.py
+- views.py
```

Creating an inclusion tag

Let's create an `inclusion tag` to query for recent blog entries and render a list of them. We'll name our template tag `entry_history`.

Let's start by rendering an empty template with an empty template context dictionary. First let's create a `templates/blog/_entry_history.html` file with some dummy text:

```
<p>Dummy text.</p>
```

Now we'll create our `blog/templatetags/blog_tags.py` module with our `entry_history` template tag:

```
from django import template

register = template.Library()

@register.inclusion_tag('blog/_entry_history.html')
def entry_history():
    return {}
```

Let's use our tag in our base template file. In our `base.html` file, import our new template library by adding the line `{% load blog_tags %}` near the top of the file.

Then modify our second column to use our `entry_history` template tag:

```
<div class="large-4 columns">
  <h3>About Me</h3>
  <p>I am a Python developer and I like Django.</p>
  <h3>Recent Entries</h3>
  {% entry_history %}
</div>
```

Restart the server and make sure our dummy text appears.

Make it work

We just wrote code without writing any tests. Let's write some tests now.

At the top of `blog/tests.py` we need to add `from django.template import Template, Context`. We need those imports because we will be manually rendering template strings to test our template tag.

Now let's add a basic test to our `blog/tests.py` file:

```
class EntryHistoryTagTest(TestCase):

    TEMPLATE = Template("{% load blog_tags %} {% entry_history %}")

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

    def test_entry_shows_up(self):
        entry = Entry.objects.create(author=user, title="My entry title")
        rendered = self.TEMPLATE.render(Context({}))
        self.assertIn(entry.title, rendered)
```

The tricky bits here are `TEMPLATE`, `Context({})` and that `render()` call. These should all look somewhat familiar from the [django tutorial part 3](#). `Context({})` in this case just passes no data to a `Template` that we're rendering directly in memory. That last `assert` just checks that the title of the entry is in the text.

As expected, our test fails because we are not actually displaying any entries with our `entry_history` template tag:

```
$ python manage.py test blog
Creating test database for alias 'default'...
.....F.....
=====
FAIL: test_entry_shows_up (blog.tests.EntryHistoryTagTest)
-----
Traceback (most recent call last):
...
AssertionError: 'My entry title' not found in '<p>Dummy text.</p>'
-----
Ran 21 tests in 0.132s

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

Let's make our template tag actually display entry history. First we will import our `Entry` model at the top of our template tag library module:

```
from ..models import Entry
```

Note: For more information on the `..` syntax for imports see the Python documentation on [relative imports](#).

Now let's send the last 5 entries in our sidebar:

```
def entry_history():
    entries = Entry.objects.all()[:5]
    return {'entries': entries}
```

Now we need to update our `_entry_history.html` file to display the titles of these blog entries:

```
<ul>
    {% for entry in entries %}
        <li>{{ entry.title }}</li>
    {% endfor %}
</ul>
```

Let's run our tests again and make sure they all pass.

Making it a bit more robust

What happens if we don't have any blog entries yet? The sidebar might look a little strange without some text indicating that there aren't any blog entries yet.

Let's add a test for when there are no blog posts:

```
def test_no_posts(self):
    rendered = self.TEMPLATE.render(Context({}))
    self.assertIn("No recent entries", rendered)
```

The above test is for an edge case. Let's add a test for another edge case: when there are more than 5 recent blog entries. When there are 6 posts, only the last 5 should be displayed. Let's add a test for this case also:

```
def test_many_posts(self):
    for n in range(6):
        Entry.objects.create(author=self.user, title="Post #{0}".format(n))
    rendered = self.TEMPLATE.render(Context({}))
    self.assertIn("Post #5", rendered)
    self.assertNotIn("Post #6", rendered)
```

The `{% for %}` template tag allows us to define an `{% empty %}` tag which we will be displayed when there are no blog entries (see [for loops](#) documentation).

Update the `_entry_history.html` template to utilize the `{% empty %}` tag and make sure the tests pass.

```
<ul>
    {% for entry in entries %}
        <li>{{ entry.title }}</li>
    {% empty %}
        <li>No recent entries</li>
    {% endfor %}
</ul>
```

It looks like we still have some problems because our tests still fail:

```
$ python manage.py test blog
Creating test database for alias 'default'...
.....EE.....
=====
ERROR: test_entry_shows_up (blog.tests.EntryHistoryTagTest)
-----
Traceback (most recent call last):
...
AttributeError: 'EntryHistoryTagTest' object has no attribute 'entry'
=====
ERROR: test_many_posts (blog.tests.EntryHistoryTagTest)
-----
Traceback (most recent call last):
...
AttributeError: 'EntryHistoryTagTest' object has no attribute 'user'
-----
Ran 23 tests in 0.164s

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

Try to fix the bugs on your own but don't be afraid to ask for help.

Hint: There are multiple bugs in our test code. Let's give you a couple of hints on how you can approach debugging and resolving them.

First of all, for the `test_no_posts`, think about what is initially being set up in the function `setUp`. How many entries have been created? What could we do to have no entries created when `test_no_posts` is called and executed?

Secondly, for `test_many_posts`, read about [slicing](#) and the `range` function to resolve the errors that appear during testing.

Database Migrations

If you have a database, you should be using [migrations](#) to manage changes in your database schema.

Let's learn about migrations to future-proof our website against database changes.

Making Migrations

We created our project using migrations, so let's look at the migrations we already have.

Right now we only have one app called `blog`. We can find the migrations in that app's `migrations` package:

```
migrations
+- 0001_initial.py
+- 0002_auto_20141019_0232.py
+- __init__.py
```

Now let's look at the migrations we have so far

```
$ python manage.py migrate --list
admin
  [X] 0001_initial
auth
  [X] 0001_initial
blog
  [X] 0001_initial
  [X] 0002_auto_20141019_0232
contenttypes
  [X] 0001_initial
sessions
  [X] 0001_initial
```

We actually have quite a few. Since migrations are a feature of Django itself, each reusable app distributed with Django contains migrations as well, and will allow you to automatically update your database schema when their models change.

Each of those migration files stores instructions on how to correctly alter the database with each change.

Using Migrate

Whenever we make a change to our models that would require a change in our database (e.g. adding a model, adding a field, removing a field, etc.) we need to create a schema migration file for our change.

To do this we will use the `makemigrations` command. Let's try it out right now:

```
$ python manage.py makemigrations blog
No changes detected in app 'blog'
```

No migration was created because we have not made any changes to our models.

Tip: For more information check out [migrations](#) in the Django documentation.

Readable URLs

Our current URL structure doesn't tell us much about the blog entries, so let's add date and title information to help users and also search engines better identify the entry.

For this purpose, we're going to use the URL scheme: `/year/month/day/pk-slug/`

Slug is a term coined by the newspaper industry for a short identifier for a newspaper article. In our case, we'll be using the Django's `slugify` method to convert our text title into a slugified version. For example, "This Is A Test Title" would be converted to lowercase with spaces replaced by dashes resulting in "this-is-a-test-title" and the complete URL might be `"/2014/03/15/6-this-is-a-test-title/"`.

First, let's update our Model to handle the new slug field.

Model

In our `Entry` model, we need to automatically create or update the slug of the entry after saving the entry. First, let's add the slug field to our `Entry` model. Add this after the `modified_at` field declaration:

```
slug = models.SlugField(default='')
```

Next, we update the save function. We import the slugify method at the top of the file:

```
from django.template.defaultfilters import slugify
```

Now create a save method in our Entry model that slugifies the title upon saving:

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

After this, we will update our `get_absolute_url()` method to do a reverse of the new URL using our new year, month, day, and slug parameters:

```
def get_absolute_url(self):
    kwargs = {'year': self.created_at.year,
             'month': self.created_at.month,
             'day': self.created_at.day,
             'slug': self.slug,
             'pk': self.pk}
    return reverse('entry_detail', kwargs=kwargs)
```

We now have to run South to migrate the database, since we have changed the model. Run the command to migrate your database. First, we create the new migration (assuming you have finished the previous tutorial where you created your initial migration):

```
$ python manage.py makemigrations blog
```

Next, we run the new migration that we just created:

```
$ python manage.py migrate blog
```

Write the Test

The first step is to define our test for the title. For this purpose, we'll:

1. Create a new blog entry
2. Find the slug for the blog entry
3. Perform an HTTP GET request for the new descriptive URL `/year/month/day/pk-slug/` for the blog entry
4. Check that the request succeeded with a code 200

First we need to import the Python `datetime` package and the `slugify` function into our tests file:

```
from django.template.defaultfilters import slugify
import datetime
```

Now let's write our test in the `EntryViewTest` class:

```
def test_url(self):
    title = "This is my test title"
    today = datetime.date.today()
    entry = Entry.objects.create(title=title, body="body",
```

```

        author=self.user)

    slug = slugify(title)
    url = "{year}/{month}/{day}/{pk}-{slug}/".format(
        year=today.year,
        month=today.month,
        day=today.day,
        slug=slug,
        pk=entry.pk,
    )
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
    self.assertTemplateUsed(response,
                             template_name='blog/entry_detail.html')

```

Try running the tests again, and you should see one failure for the test we just added:

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

URL Pattern

Next we are going to change our `blog/urls.py` file. Replace your code with this:

```

from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^(?P<year>\d{4})/(?P<month>\d{1,2})/(?P<day>\d{1,2})/(?P<pk>\d+)-(?P<slug>[-
↪\w]*)/$', views.EntryDetail.as_view(), name='entry_detail'),
]

```

Let’s break this down. For this URL pattern `(?P<year>\d{4})`, the outer parentheses are for “capturing” the input. The `?P<year>` specifies that we should capture this into a parameter named “year.” And the `\d{4}` means the value we are capturing should be four digits. The next part is the month, where we capture `\d{1,2}`, which captures either one or two digits for the month (January would be 1, December would be 12, so 1 or 2 digits will represent the month). And for the day, we also capture one or two digits.

We capture the `pk` (i.e. the “primary key” for accessing a Django model) with `(?P<pk>\d+)`.

The next part is capturing the slug in `(?P<slug>[-\w]*)`. For this part, we name the captured variable “slug” and look for alphanumeric characters or a dash/hyphen (-).

As you can see from the last part of the pattern, we are using the view `EntryDetail`.

Now save the file and try running the tests again. You should see all of the tests passing.

Another Test

What would happen if we changed the slug or an invalid date was given in the URL? This shouldn’t matter because we only check for the model `pk`.

Let’s write a couple more tests for this case to make sure the correct page is displayed in this case, and for when the id does not exist. Our tests should look like this:

```
def test_misdated_url(self):
    entry = Entry.objects.create(
        title="title", body="body", author=self.user)
    url = "/0000/00/00/{0}-misdated/".format(entry.id)
    response = self.client.get(url)
    self.assertEqual(response.status_code, 200)
    self.assertTemplateUsed(
        response, template_name='blog/entry_detail.html')

def test_invalid_url(self):
    entry = Entry.objects.create(
        title="title", body="body", author=self.user)
    response = self.client.get("/0000/00/00/0-invalid/")
    self.assertEqual(response.status_code, 404)
```

Now let's run our tests and make sure they still pass.

Tip: If you try to add an entry in the admin, you will notice that you must write a slug (it isn't optional) but then whatever you write is overwritten in the `Entry.save()` method. There are a couple ways to resolve this but one way is to set the `SlugField` in our `Entry` model to be `editable=False` which will hide it in the admin or other forms:

```
slug = SlugField(editable=False)
```

See the Django docs on [editable](#) for details.

Adding Gravatars

Wouldn't it be cool if we could show user avatars next to comments? Let's use the free [Gravatar](#) service for this. As usual, we'll start with a test.

According to the [Gravatar documentation](#) a Gravatar profile image can be requested like this:

```
http://www.gravatar.com/avatar/HASH
```

Where `HASH` is an MD5 hash of the user's email address. We can use the `hashlib` package in the Python standard library to generate an MD5 hash.

Tip: There are lots of options for displaying gravatars such as setting the display size for the image and having a default image if there is no Gravatar for a specific email.

First, let's write a test for Gravatars. This test will be added to our already existing test `CommentModelTest` since the plan is to add a method to the `Comment` model to get the Gravatar URL.

```
def test_gravatar_url(self):
    comment = Comment(body="My comment body", email="email@example.com")
    expected = "http://www.gravatar.com/avatar/5658ffccee7f0ebfda2b226238b1eb6e"
    self.assertEqual(comment.gravatar_url(), expected)
```

Note: We didn't calculate that MD5 hashes in our head. You can use Python's `hashlib` library to calculate the hash or just type `md5 email@example.com` into Duck Duck Go.

When running our tests now, we'll see an error since we have not yet written a `gravatar_url()` method to the `Comment` model:

```
$ python manage.py test blog
Creating test database for alias 'default'...
....E.....
=====
ERROR: test_gravatar_url (blog.tests.CommentModelTest)
-----
Traceback (most recent call last):
...
AttributeError: 'Comment' object has no attribute 'gravatar_url'
-----
Ran 26 tests in 0.175s

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

Adding comment gravatars

Let's add a `gravatar_url()` method to `Comment` so our tests pass. This involves editing `models.py`:

```
def gravatar_url(self):
    # Get the md5 hash of the email address
    md5 = hashlib.md5(self.email.encode())
    digest = md5.hexdigest()

    return 'http://www.gravatar.com/avatar/{}'.format(digest)
```

Note: Remember to import the `hashlib` library at the top of our `models.py` file.

Tip: If you've never used `hashlib` before, this may look a little daunting. **MD5** is a cryptographic hash function that takes a string of any size and creates a 128-bit binary string. When rendered as hexadecimal, it is a 32 character string.

`self.email` is always converted to unicode because it is possible that it is `None` since it is not required. If you're feeling up to it, write a test for this case and see what happens.

If you run the tests at this point, you should see that our test case passes.

Displaying gravatars on the site

Now, let's display the Gravatars with the comments.

In our content block in `templates/blog/entry_detail.html`, let's add the Gravatar images:

```
{% for comment in entry.comment_set.all %}
    <p>
        <em>Posted by {{ comment.name }}</em>
        
    </p>
```

```
    {{ comment|linebreaks }}
{% empty %}
    No comments yet.
{% endfor %}
```

If you fire up the development web server and look at a specific blog entry, you should see an image for each comment.

CHAPTER 5

Getting Help & Contributing

Markdown source files and working code examples for these tutorials can be found on [Github](#). If you found a bug or have a suggestion to improve or extend the tutorials, please open an issue or a pull request.

These tutorials are provided under a Creative Commons license (CC BY-SA 3.0).