Chapter 4: Message Board App

djangoforbeginners.com/message-board

Create your first database-driven app and use the Django admin

In this chapter we will use a database for the first time to build a basic *Message Board* application where users can post and read short messages. We'll explore Django's powerful built-in admin interface which provides a visual way to make changes to our data. And after adding tests we will push our code to GitHub and deploy the app on Heroku.

Thanks to the powerful Django ORM (Object-Relational Mapper), there is built-in support for multiple database backends: PostgreSQL, MySQL, MariaDB, Oracle, and SQLite. This means that we, as developers, can write the same Python code in a `models.py` file and it will automatically be translated into the correct SQL for each database. The only configuration required is to update the DATABASES section of our `django_project/settings.py` file. This is truly an impressive feature!

For local development, Django defaults to using SQLite because it is file-based and therefore far simpler to use than the other database options that require a dedicated server to be running separate from Django itself.

Initial Set Up

Since we've already set up several Django projects at this point in the book, we can quickly run through the standard commands to begin a new one. We need to do the following:

- make a new directory for our code called `message-board`
- install Django in a new virtual environment
- create a new project called `django_project`
- create a new app call `posts`
- update `django_project/settings.py`

In a new command line console, enter the following commands:

```
# Windows
> cd onedrive\desktop\code
> mkdir message-board
> cd message-board
> python -m venv .venv
> .venv\Scripts\Activate.ps1
(.venv) > python -m pip install django~=4.0.0
(.venv) > django-admin startproject django_project .
(.venv) > python manage.py startapp posts

# macOS
% cd ~/desktop/code
% mkdir message-board
% cd message-board
% python3 -m venv .venv
% source .venv/bin/activate
(.venv) % python3 -m pip install django~=4.0.0
(.venv) % django-admin startproject django_project .
(.venv) % python3 manage.py startapp posts
```

Next we must alert Django to the new app, `posts`, by adding it to the top of the `INSTALLED_APPS` section of our `django_project/settings.py` file.

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "posts.apps.PostsConfig",  # new
]
```

Then execute the `migrate` command to create an initial database based on Django's default settings.

```
(.venv) > python manage.py migrate
```

If you look inside our directory with the `ls` command, you'll see the `db.sqlite3` file representing our SQLite database.

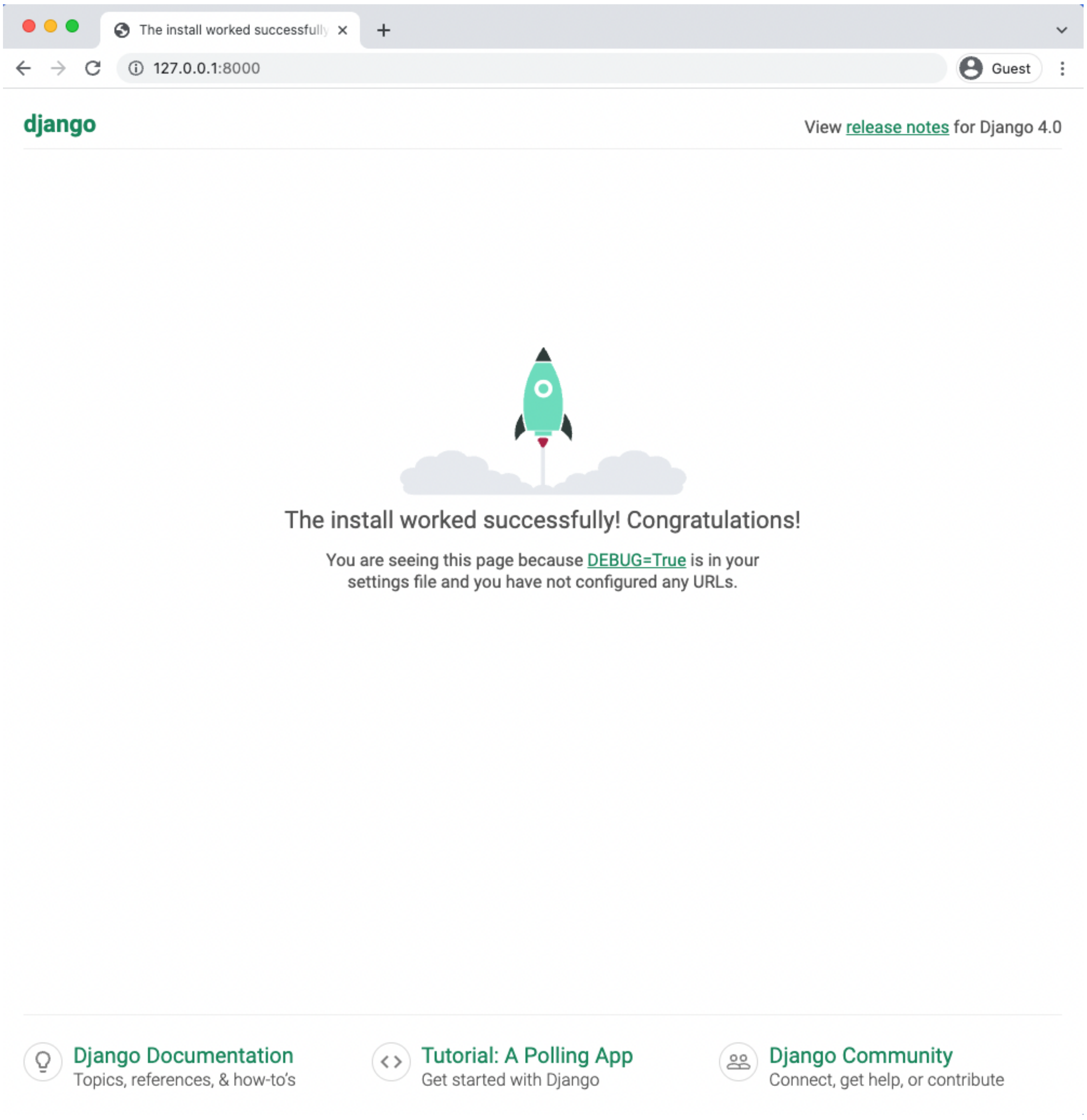
```
(.venv) > ls
.venv     db.sqlite3     django_project     manage.py     posts
```

Technically, a `db.sqlite3` file is created the first time you run *either* `migrate` or `runserver`, however `migrate` will sync the database with the current state of any database models contained in the project and listed in `INSTALLED_APPS`. In other words, to make sure the database reflects the current state of your project you'll need to run `migrate` (and also `makemigrations`) each time you update a model. More on this shortly.

To confirm everything works correctly, spin up our local server.

```
(.venv) > python manage.py runserver
```

In your web browser, navigate to `http://127.0.0.1:8000/` to see the familiar Django welcome page.


The install worked successfully
127.0.0.1:8000
Guest
django
View release notes for Django 4.0

The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Django Documentation
Topics, references, & how-to's
Tutorial: A Polling App
Get started with Django
Django Community
Connect, get help, or contribute

Create a Database Model

Our first task is to create a database model where we can store and display posts from our users. Django's ORM will automatically turn this model into a database table for us. In a real-world Django project, there are often many complex, interconnected database models but in our simple message board app we only need one.

Open the `posts/models.py` file and look at the default code which Django provides:

```
# posts/models.py
from django.db import models

# Create your models here
```

Django imports a module, `models`, to help us build new database models which will "model" the characteristics of the data in our database. We want to create a model to store the textual content of a message board post, which we can do as follows:

```
# posts/models.py
from django.db import models


class Post(models.Model):  # new
    text = models.TextField()
```

Note that we've created a new database model called `Post` which has the database field `text`. We've also specified the *type of content* it will hold, `TextField()`. Django provides many model fields supporting common types of content such as characters, dates, integers, emails, and so on.

Activating models

Now that our new model is created we need to activate it. Going forward, whenever we create or modify an existing model we'll need to update Django in a two-step process:

1. First, we create a migrations file with the `makemigrations` command. Migration files create a reference of any changes to the database models which means we can track changes–and debug errors as necessary–over time.

2. Second, we build the actual database with the `migrate` command which executes the instructions in our migrations file.

Make sure the local server is stopped by typing `Control+c` on the command line and then run the commands `python manage.py makemigrations posts` and `python manage.py migrate`.

```
(.venv) > python manage.py makemigrations posts
Migrations for 'posts':
  posts/migrations/0001_initial.py
    - Create model Post

(.venv) > python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, posts, sessions
Running migrations:
  Applying posts.0001_initial... OK
```

Note that you don't *have* to include a name after `makemigrations`. If you simply run `python manage.py makemigrations`, a migrations file will be created for *all* available changes throughout the Django project. That is fine in a small project such as ours with only a single app, but most Django projects have more than one app! Therefore ,if you made model changes in multiple apps the resulting migrations file would include *all* those changes! This is not ideal. Migrations file should be as small and concise as possible as this makes it easier to debug in the future or even roll back changes as needed. Therefore, as a best practice, adopt the habit of always including the name of an app when executing the `makemigrations` command!

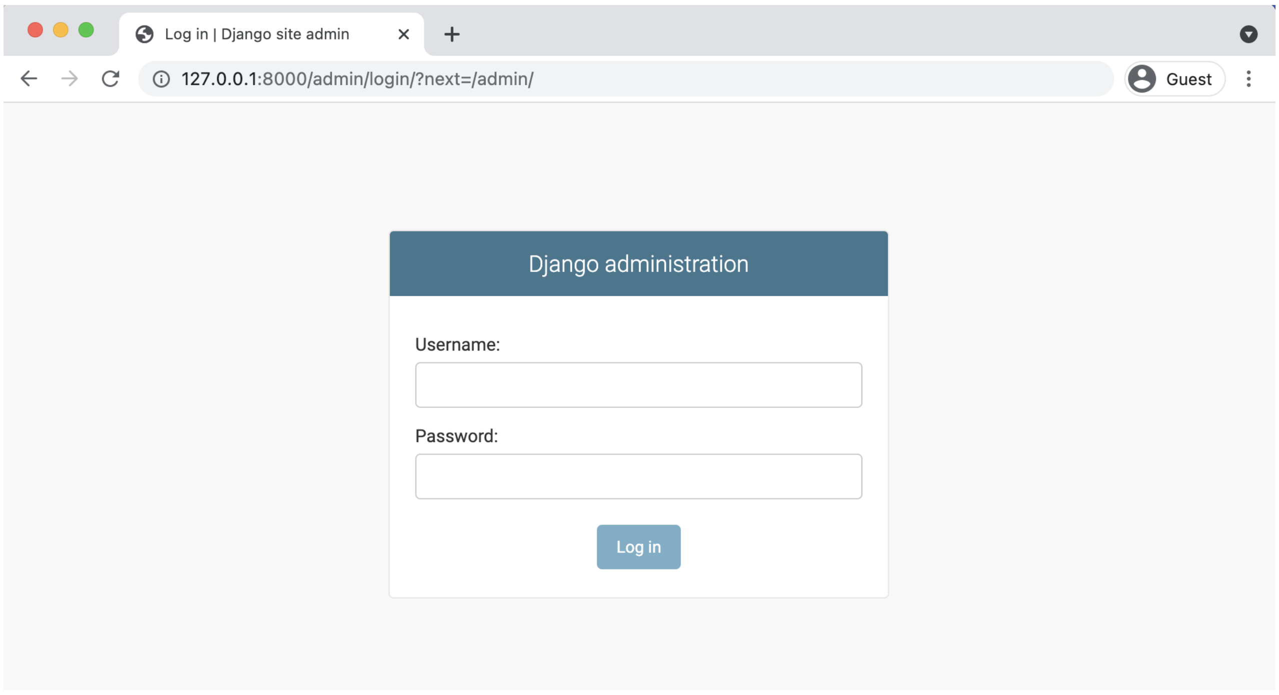
Django Admin

One of Django's killer features is its robust admin interface that provides a visual way to interact with data. It came about because Django was originally built as a newspaper CMS (Content Management System). The idea was that journalists could write and edit their stories in the admin without needing to touch "code." Over time, the built-in admin app has evolved into a fantastic, out-of-the-box tool for managing all aspects of a Django project.

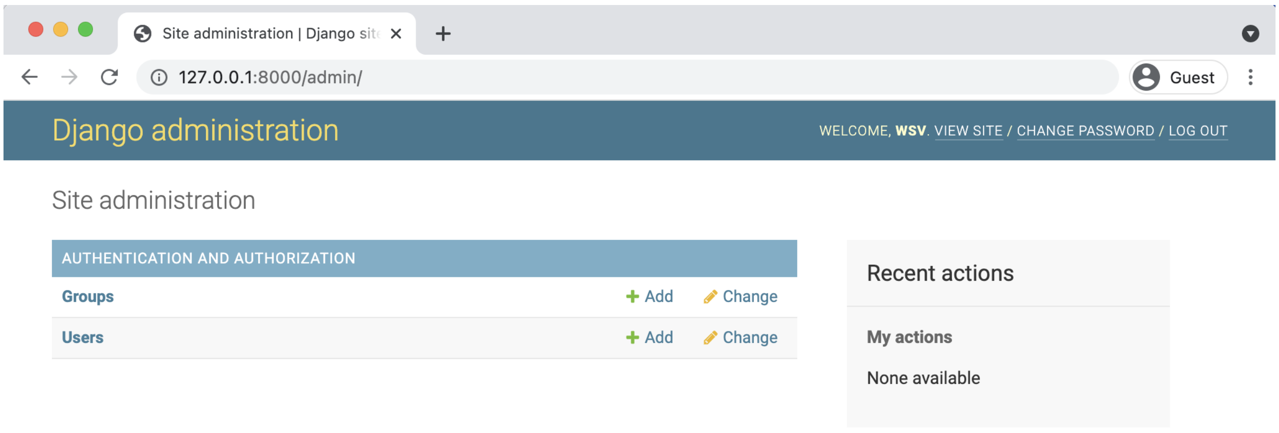
To use the Django admin, we first need to create a `superuser` who can log in. In your command line console, type `python manage.py createsuperuser` and respond to the prompts for a username, email, and password:

```
(.venv) > python manage.py createsuperuser
Username (leave blank to use 'wsv'): wsv
Email: will@wsvincent.com
Password:
Password (again):
Superuser created successfully.
```

When you type your password, it will not appear visible in the command line console for security reasons. Restart the Django server with `python manage.py runserver` and in your web browser go to `http://127.0.0.1:8000/admin/`. You should see the log in screen for the admin:


Log in | Django site admin
127.0.0.1:8000/admin/login/?next=/admin/
Guest
Django administration
Username:
Password:
Log in

Log in by entering the username and password you just created. You will see the Django admin homepage next:


Site administration | Django site
127.0.0.1:8000/admin/
Guest

Django administration

WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

Recent actions

My actions

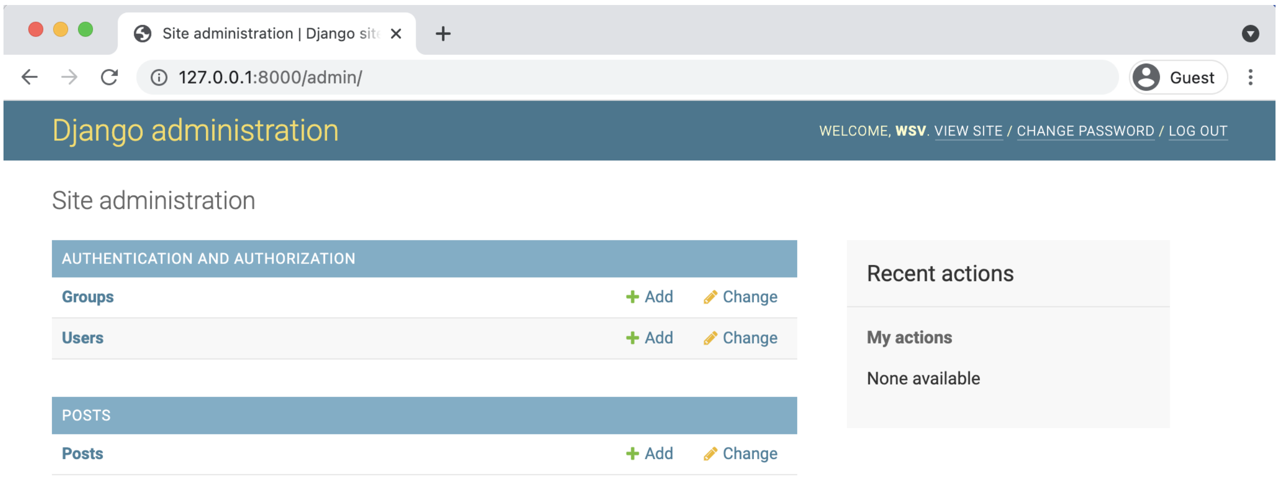
None available

But where is our `posts` app? It's not displayed on the main admin page! Just as we must explicitly add new apps to the `INSTALLED_APPS` config, so, too, must we update an app's `admin.py` file for it to appear in the admin.

In your text editor open up `posts/admin.py` and add the following code so that the Post model is displayed.

```
# posts/admin.py
from django.contrib import admin

from .models import Post

admin.site.register(Post)
```

Django now knows that it should display our `posts` app and its database model `Post` on the admin page. If you refresh your browser you'll see that it appears:


Site administration | Django site
127.0.0.1:8000/admin/
Guest

Django administration

WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

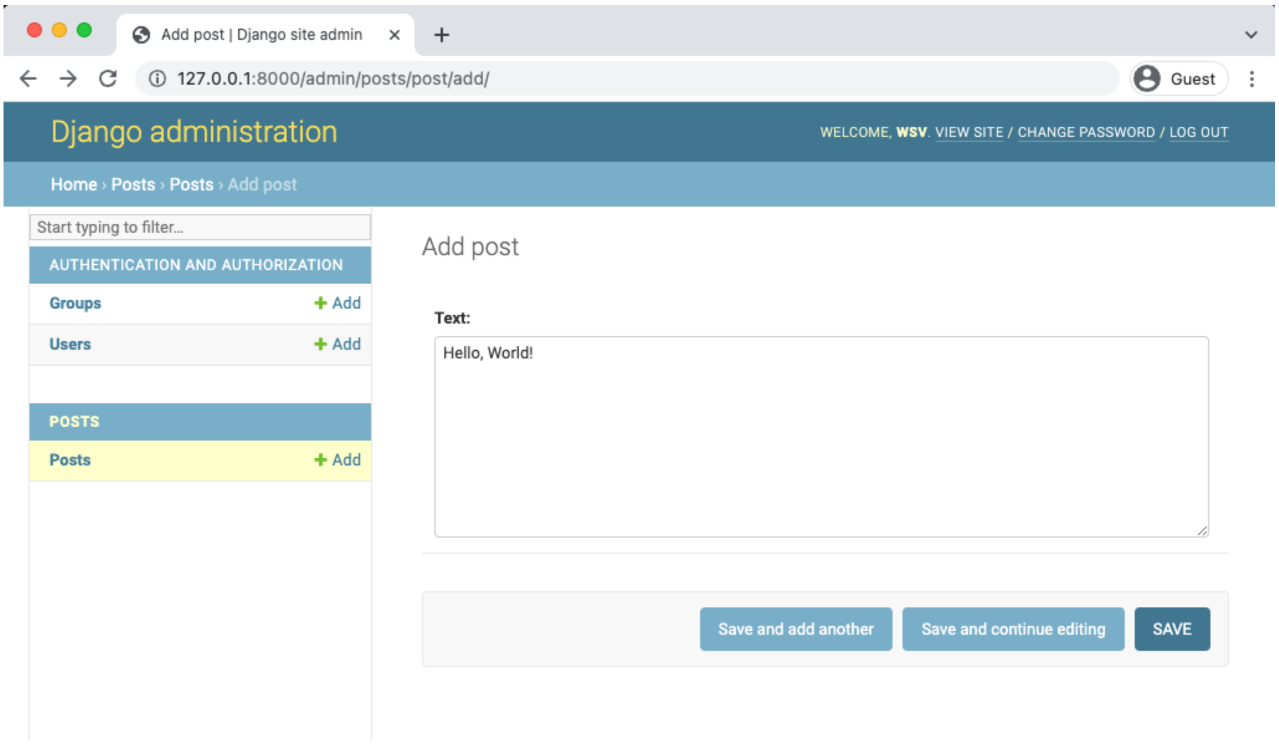
POSTS		
Posts	+ Add	Change

Recent actions

My actions

None available

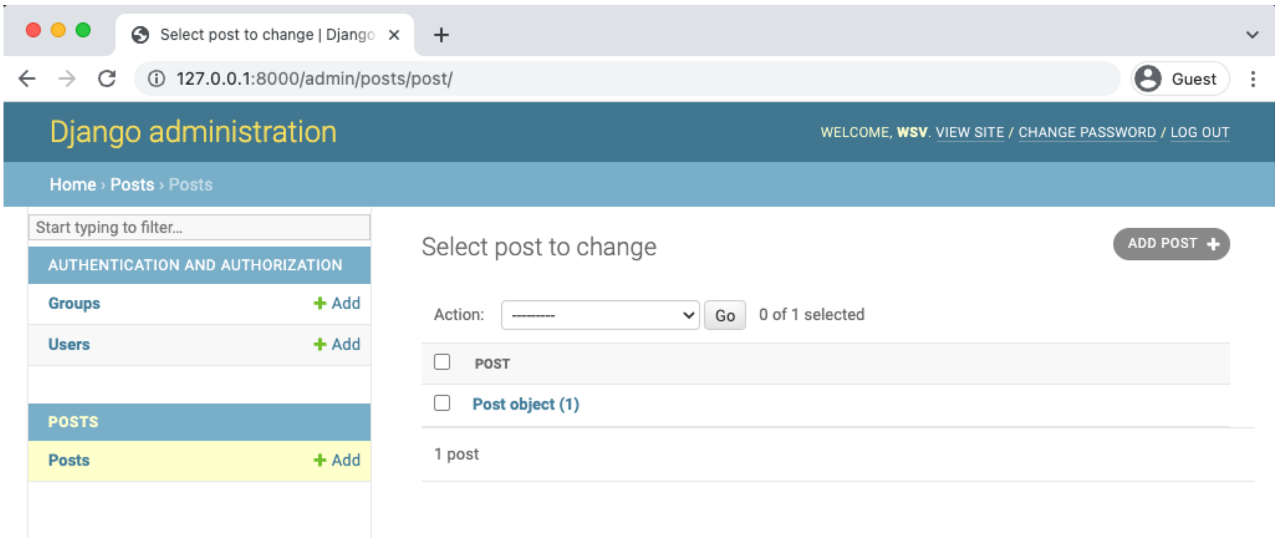
Let's create our first message board post for our database. Click on the `+ Add` button opposite `Posts` and enter your own content in the `Text` form field.


Add post | Django site admin
127.0.0.1:8000/admin/posts/post/add/
Guest
Django administration
WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Posts › Posts › Add post
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
POSTS
Posts + Add
Add post
Save and add another
Save and continue editing
SAVE

Text:

Hello, World!

Then click the "Save" button, which will redirect you to the main Post page. However if you look closely, there's a problem: our new entry is called "Post object (1)" which isn't very descriptive!


Select post to change | Django
127.0.0.1:8000/admin/posts/post/
Guest
Django administration
WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Posts › Posts
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
POSTS
Posts + Add
Select post to change
ADD POST +
Action:

Go
0 of 1 selected

POST

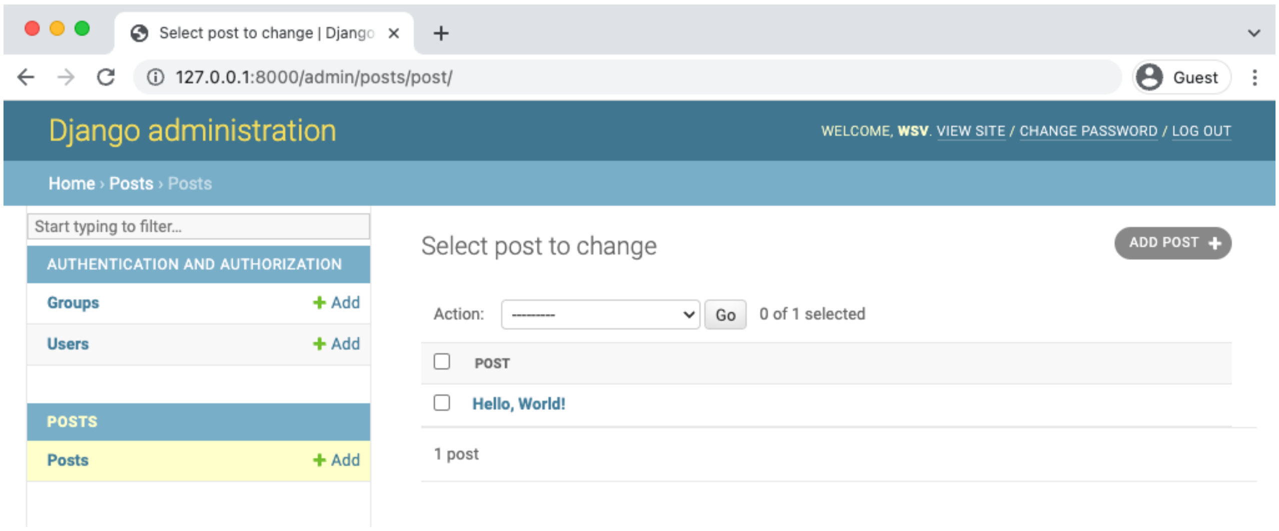
Post object (1)

1 post

Let's change that. Within the `posts/models.py` file, add a new function `__str__` as follows:

```
# posts/models.py
from django.db import models


class Post(models.Model):
    text = models.TextField()

    def __str__(self):  # new
        return self.text[:50]
```

This will display the first 50 characters of the `text` field. If you refresh your Admin page in the browser, you'll see it's changed to a much more descriptive and helpful representation of our database entry.


Select post to change | Django
127.0.0.1:8000/admin/posts/post/
Guest
Django administration
WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Posts › Posts
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
POSTS
Posts + Add
Select post to change
ADD POST +
Action: --------- Go 0 of 1 selected
POST
Hello, World!
1 post

Much better! It's a best practice to add `str()` methods to all of your models to improve their readability.

Views/Templates/URLs

In order to display our database content on our homepage, we have to wire up our views, templates, and URLs. This pattern should start to feel familiar now.

Let's begin with the view. Earlier in the book we used the built-in generic TemplateView to display a template file on our homepage. Now we want to list the contents of our database model. Fortunately this is also a common task in web development and Django comes equipped with the generic class-based ListView.

In the `posts/views.py` file enter the Python code below:

```
# posts/views.py
from django.views.generic import ListView
from .models import Post


class HomePageView(ListView):
    model = Post
    template_name = "home.html"
```

On the first line we're importing `ListView` and in the second line we import the `Post` model. In the view, `HomePageView`, we subclass `ListView` and specify the correct model and template.

Our view is complete which means we still need to configure our URLs and make our template. Let's start with the template. Create a new directory called `templates`.

```
(.venv) > mkdir templates
```

Then update the `DIRS` field in our `django_project/settings.py` file so that Django knows to look in this new templates directory.

```
# django_project/settings.py
TEMPLATES = [
    {
        ...
        "DIRS": [BASE_DIR / "templates"],  # new
        ...
    },
]
```

In your text editor, create a new file called `templates/home.html`. ListView automatically returns to us a context variable called `<model>_list`, where `<model>` is our model name, that we can loop over via the built-in for template tag. We'll create our own variable called `post` and can then access the desired field we want displayed, `text`, as `post.text`.

```
<!-- templates/home.html -->
<h1>Message board homepage</h1>
<ul>
  {% for post in post_list %}
    <li>{{ post.text }}</li>
  {% endfor %}
</ul>
```

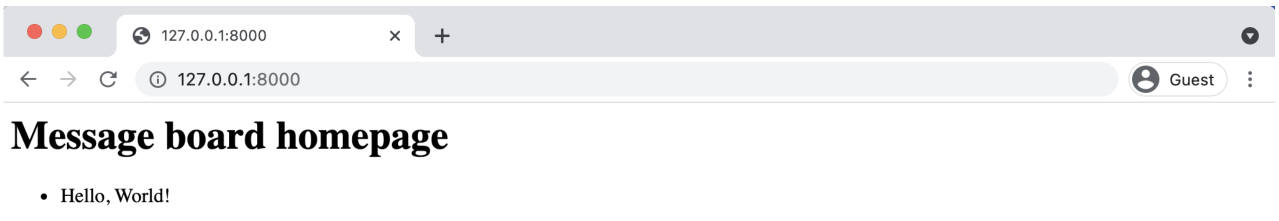
The last step is to set up our URLs. Let's start with the `django_project/urls.py` file where we include our `posts` app and add `include` on the second line.

```
# django_project/urls.py
from django.contrib import admin
from django.urls import path, include  # new

urlpatterns = [
    path("admin/", admin.site.urls),
    path("", include("posts.urls")),  # new
]
```

Then in your text editor create a new `urls.py` file within the `posts` app and update it like so:

```
# posts/urls.py
from django.urls import path
from .views import HomePageView

urlpatterns = [
    path("", HomePageView.as_view(), name="home"),
]
```

Restart the server with `python manage.py runserver` and navigate to our homepage, which now lists out our message board post.



Message board homepage

- Hello, World!

We're basically done at this point, but let's create a few more message board posts in the Django admin to confirm that they will display correctly on the homepage.

Adding New Posts

To add new posts to our message board, go back into the Admin and create two more posts. If you then return to the homepage you'll see it automatically displays the formatted posts. Woohoo!

Everything works so it is a good time to initialize our directory and create a `.gitignore file`. In your text editor create a new `.gitignore` file and add one line:

```
.venv/
```

Then run `git status` again to confirm the `.venv` directory is being ignored, use `git add -A` to add the intended files/directories, and add an initial commit message.

```
(.venv) > git status
(.venv) > git add -A
(.venv) > git commit -m "initial commit"

(.venv) > git init
(.venv) > git add -A
(.venv) > git commit -m "initial commit"
```

Tests

In the previous chapter we were only testing static pages so we used SimpleTestCase. Now that our project works with a database, we need to use TestCase, which will let us create a test database we can check against. In other words, we don't need to run tests on our *actual* database but instead can make a separate test database, fill it with sample data, and then test against it which is a much safer and more performant approach.

We will use the hook setUpTestData() to create our test data. This was added to Django 1.8 and is much faster than using the `setUp()` hook from Python's unittest because it creates the test data only once per test case rather than per test. It is still common, however, to see Django projects that rely on `setUp()` instead. Converting any such tests over to `setUpTestData` is a reliable way to speed up a test suite and should be done!

Our `Post` model contains only one field, `text`, so let's set up our data and then check that it is stored correctly in the database. All test methods must start with the phrase `test*` so that Django knows to test them! The code will look like this:

```
# posts/tests.py
from django.test import TestCase

from .models import Post


class PostTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.post = Post.objects.create(text="This is a test!")

    def test_model_content(self):
        self.assertEqual(self.post.text, "This is a test!")
```

At the top we import `TestCase` and our `Post` model. Then we create a test class, `PostTests`, that extends `TestCase` and uses the built-in method `setUpTestData` to create initial data. In this instance, we only have one item stored as `cls.post` that can then be referred to in any subsequent tests within the class as `self.post`. Our first test, `test_model_content`, uses `assertEqual` to check that the content of the `text` field matches what we expect.

Go ahead and run the test on the command line with command `python manage.py test`.

```
(.venv) > python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

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

It passed! Why does the output say only one test was run when we have two functions? Again, only functions that start with the name `test*` will be run! So while we can use set up functions and classes to help with our tests, unless a function is so named it won't be run on its own as a standalone test when we execute the `python manage.py test` command.

Moving along it is time to check our URLs, views, and templates in a manner similar to the previous chapter. We will want to check the following four things for our message board page:

- URL exists at `/` and returns a 200 HTTP status code
- URL is available by its name of "home"
- Correct template is used called "home.html"
- Homepage content matches what we expect in the database

We can include all of these tests in our existing `PostTests` class since there is only one webpage in this project. Make sure to import `reverse` at the top of the page and add the four tests as follows:

```
# posts/tests.py
from django.test import TestCase
from django.urls import reverse  # new

from .models import Post


class PostTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.post = Post.objects.create(text="This is a test!")

    def test_model_content(self):
        self.assertEqual(self.post.text, "This is a test!")

    def test_url_exists_at_correct_location(self):  # new
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_url_available_by_name(self):  # new
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)

    def test_template_name_correct(self):  # new
        response = self.client.get(reverse("home"))
        self.assertTemplateUsed(response, "home.html")

    def test_template_content(self):  # new
        response = self.client.get(reverse("home"))
        self.assertContains(response, "This is a test!")
```

If you run our tests again you should see that they all pass.

```
(.venv) > python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 5 tests in 0.009s

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

In the previous chapter we discussed how unit tests work best when they are self-contained and extremely verbose. However there is an argument to be made here that the bottom three tests are really just testing that the homepage works as expected: it uses the correct URL name, the intended template name, and contains expected content. We can combine these three tests into one single unit test, `test_homepage`.

```
# posts/tests.py
from django.test import TestCase
from django.urls import reverse  # new
from .models import Post


class PostTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        cls.post = Post.objects.create(text="This is a test!")

    def test_model_content(self):
        self.assertEqual(self.post.text, "This is a test!")

    def test_url_exists_at_correct_location(self):
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)

    def test_homepage(self):  # new
        response = self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "home.html")
        self.assertContains(response, "This is a test!")
```

Run the tests one last time to confirm that they all pass.

```
(.venv) > python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 3 tests in 0.008s

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

Ultimately, we want our test suite to cover as much code functionality as possible while also being easy for us to reason about, both the error messages *and* the testing code itself. In my view, this update is easier to read and understand.

We're done adding code for our testing so it's time to commit the changes to git.

```
(.venv) > git add -A
(.venv) > git commit -m "added tests"
```

GitHub

We also need to store our code on GitHub. You should already have a GitHub account from previous chapters so go ahead and create a new repo called `message-board`. Select the "Private" radio button.

On the next page scroll down to where it says "or push an existing repository from the command line." Copy and paste the two commands there into your terminal, which should look like like the below after replacing `wsvincent` (my username) with your GitHub username:

```
(.venv) > git remote add origin https://github.com/wsvincent/message-board.git
(.venv) > git push -u origin main
```

Heroku Configuration

You should already have a Heroku account set up and installed from Chapter 3. Our deployment checklist contains the same five items:

- install `Gunicorn`
- create a `requirements.txt` file
- update `ALLOWED_HOSTS` in `django_project/settings.py`
- create a `Procfile`
- create a `runtime.txt` file

First, install `Gunicorn` with Pip.

```
(.venv) > python -m pip install gunicorn==20.1.0
```

Then output the contents of our virtual environment to a `requirements.txt` file.

```
(.venv) > python -m pip freeze > requirements.txt
```

Previously, we set `ALLOWED_HOSTS` to `*`, meaning accept all hosts, but that was a dangerous shortcut. We can–and should–be more specific. The two local hosts Django runs on are `localhost:8000` and `127.0.0.1:8000`. We also know, having deployed on Heroku previously, that any Heroku site will end with `.herokuapp.com`. We can add all three hosts to our `ALLOWED_HOSTS` configuration.

```
# django_project/settings.py
ALLOWED_HOSTS = [".herokuapp.com", "localhost", "127.0.0.1"]
```

Fourth in your text editor create a new `Procfile` in the project root directory, next to the `manage.py` file.

```
web: gunicorn django_project.wsgi --log-file -
```

And the fifth and final step is to create a `runtime.txt` file, also in the base directory, that specifies what version of Python to run within Heroku.

```
python-3.10.2
```

We're all done! Add and commit our new changes to git and then push them up to GitHub.

```
(.venv) > git status
(.venv) > git add -A
(.venv) > git commit -m "New updates for Heroku deployment"
(.venv) > git push -u origin main
```

Heroku Deployment

Make sure you're logged into your correct Heroku account.

```
(.venv) > heroku login
```

Then run the `create` command and Heroku will randomly generate an app name.

```
(.venv) > heroku create
Creating app... done, ⬢ sleepy-brook-64719
https://sleepy-brook-64719.herokuapp.com/ |
https://git.heroku.com/sleepy-brook-64719.git
```

For now, continue to instruct Heroku to ignore static files. We'll cover them in the next section while deploying our *Blog* app.

```
(.venv) > heroku config:set DISABLE_COLLECTSTATIC=1
```

Push the code to Heroku and add free scaling so it's actually running online, otherwise the code is just sitting there!

```
(.venv) > git push heroku main
(.venv) > heroku ps:scale web=1
```

You can open the URL of the new project from the command line by typing `heroku open` which will launch a new browser window. To finish up, deactivate the current virtual environment by typing `deactivate` on the command line.

Conclusion

We've now built, tested, and deployed our first database-driven app. While it's deliberately quite basic, we learned how to create a database model, update it with the admin panel, and then display the contents on a web page. That's the good news. The bad news is that if you check your Heroku site in a few days, it's likely whatever posts you've added will be deleted! This is related to how Heroku handles SQLite, but really it's an indication that we should be using a production database like PostgreSQL for deployments, even if we still want to stick with SQLite locally. This is covered in Chapter 16!

In the next section, we'll learn how to deploy with PostgreSQL and build a *Blog* application so that users can create, edit, and delete posts on their own. No admin access required! We'll also add styling via CSS so the site looks better.

This concludes the free sample chapters of the book.

The complete book features 16 chapters and goes in-depth on how to build both a blog app with user authentication and a newspaper site with articles and comments. Additional concepts covered include custom user models, Bootstrap for styling, password change and reset, sending email, permissions, foreign keys, and more.

4.0

DJANGO *for* BEGINNERS

Build websites with Python & Django

WILLIAM S. VINCENT