Last updated February 8th, 2022

Working with Static and Media Files in Django

Amal Shaji

Twitter Reddit Hacker News Facebook

This article looks at how to work with static and media files in a Django project, locally and in production.

Objectives

By the end of this article, you will be able to:

1. Describe the three different types of files that you'll usually find in a Django project
2. Explain the difference between static and media files
3. Work with both static and media files locally and in production

Types of Files

Django is an opinionated, full-stack web application framework. It comes with many batteries that you can use to build a fully functional web application, including static and media file management.

Before we look at the *how*, let's start with some definitions.

What are static and media files?

First off, you'll typically find these three types of files in a Django project:

1. **Source code**: These are your core Python modules and HTML files that make up every Django project, where you define your models, views, and templates.
2. **Static files**: These are your CSS stylesheets, JavaScript files, fonts, and images. Since there's no processing involved, these files are very energy efficient since they can just be served up as is. They are also much easier to cache.
3. **Media file**: These are files that a user uploads.

This article focuses on static and media files. Even though the names are different, both represent regular files. The significant difference is that static files are kept in version control and shipped with your source code files during deployment. On the other hand, media files are files that your end-users (internally and externally) upload or are dynamically created by your application (often as a side effect of some user action).

Why should you treat static and media files differently?

1. You can't trust files uploaded by end-users, so media files need to be treated differently.
2. You may need to perform processing on user uploaded, media files to be better served -- e.g., you could optimize uploaded images to support different devices.
3. You don't want a user uploaded file to replace a static file accidentally.

Feedback

Additional Notes:

1. Static and media files are sometimes referred to as static and media assets.
2. The Django admin comes with some static files, which are stored in version control on GitHub.
3. Adding to the confusion between static and media files is that the Django documentation itself doesn't do a great job differentiating between the two.

Static Files

Django provides a powerful battery for working with static files, aptly named staticfiles.

If you're new to the staticfiles app, take a quick look at the How to manage static files (e.g., images, JavaScript, CSS) guide from the Django documentation.

Django's staticfiles app provides the following core components:

1. Settings
2. Management Commands
3. Storage Classes
4. Template Tags

Settings

There are a number of settings that you *may* need to configure, depending on your environment:

1. STATIC_URL: URL where the user can access your static files from in the browser. The default is `/static/`, which means your files will be available at `http://127.0.0.1:8000/static/` in development mode -- e.g., `http://127.0.0.1:8000/static/css/main.css`.
2. STATIC_ROOT: The absolute path to the directory where your Django application will serve your static files from. When you run the collectstatic management command (more on this shortly), it will find all static files and copy them into this directory.
3. STATICFILES_DIRS: By default, static files are stored at the app-level at `<APP_NAME>/static/`. The collectstatic command will look for static files in those directories. You can also tell Django to look for static files in additional locations with `STATICFILES_DIRS`.
4. STATICFILES_STORAGE: The file storage class you'd like to use, which controls how the static files are stored and accessed. The files are stored in the file system via StaticFilesStorage.
5. STATICFILES_FINDERS: This setting defines the file finder backends to be used to automatically find static files. By default, the `FileSystemFinder` and `AppDirectoriesFinder` finders are used:
   - `FileSystemFinder` - uses the `STATICFILES_DIRS` setting to find files.
   - `AppDirectoriesFinder` - looks for files in a "static" folder in each Django app within the project.

Management Commands

The staticfiles apps provides the following management commands:

1. `collectstatic` is a management command that collects static files from the various locations -- i.e., `<APP_NAME>/static/` and the directories found in the `STATICFILES_DIRS` setting -- and copies them to the `STATIC_ROOT` directory.
2. `findstatic` is a really helpful command to use when debugging so you can see exactly where a specific file comes from
3. `runserver` starts a lightweight, development server to run your Django application in development.

Notes:

1. Don't put any static files in the `STATIC_ROOT` directory. That's where the static files get copied to automatically after you run `collectstatic`. Instead, always put them in the directories associated with the `STATICFILES_DIRS` setting or

`<APP_NAME>/static/`.

2. Do not use the development server in production. Use a production-grade WSGI application server instead. More on this shortly.

Quick example of the `findstatic` command:

Say you have two Django apps, `app1` and `app2`. Each app has a folder named "static", and within each of those folders, a file called *app.css*. Relevant settings from *settings.py*:

```
STATIC_ROOT = 'staticfiles'

INSTALLED_APPS = [
    ...
    'app1',
    'app2',
]
```

When `python manage.py collectstatic` is run, the "staticfiles" directory will be created and the appropriate static files will be copied into it:

```
$ ls staticfiles/

admin   app.css
```

There's only one *app.css* file because when multiple files with the same name are present, the staticfiles finder will use the first found file. To see which file is copied over, you can use the `findstatic` command:

```
$ python manage.py findstatic app.css

Found 'app.css' here:
  /app1/static/app.css
  /app2/static/app.css
```

Since only the first encountered file is collected, to check the source of the *app.css* that was copied over to the "staticfiles" directory, run:

```
$ python manage.py findstatic app.css --first

Found 'app.css' here:
  /app1/static/app.css
```

Storage Classes

When the `collectstatic` command is run, Django uses storage classes to determine how the static files are stored and accessed. Again, this is configured via the STATICFILES_STORAGE setting.

The default storage class is StaticFilesStorage. Behind the scenes, `StaticFilesStorage` uses the FileSystemStorage class to store files on the local filesystem.

You may want to deviate from the default in production. For example, django-storages provides a few custom storage classes for different cloud/CDN providers. You can also write your own storage class using the file storage API. Review Serving static files from a cloud service or CDN for more on this.

Storage classes can be used to perform post processing tasks like minification.

Template Tags

To load static files in your template files, you need to:

1. Add `{% load static %}` to the top of the template file
2. Then, for each file you'd like to link, add the `{% static %}` template tag

For example:

```
{% load static %}

<link rel="stylesheet" href="{% static 'base.css' %}">
```

Together, these tags generate a complete URL -- e.g, `/static/base.css` -- based on the static files configuration in the *settings.py* file.

You should always load static files in this manner rather than hard coding the URL directly so that you can change your static file configuration and point to a different `STATIC_URL` without having to manually update each template.

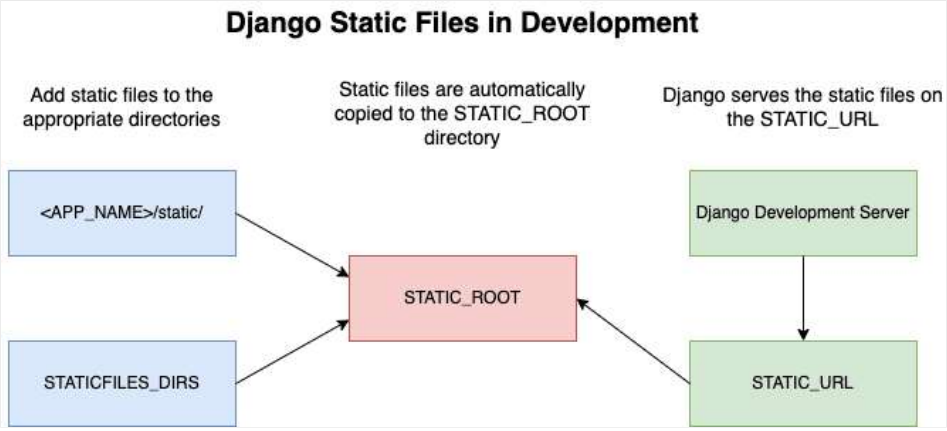
> For more on these template tags, review the static section from Built-in template tags and filters.

Static Files in Development Mode

During development, as long as you have `DEBUG` set to `TRUE` and you're using the staticfiles app, you can serve up static files using Django's development server. You don't even need to run the `collecstatic` command.

Typical development config:

```
# settings.py
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / 'staticfiles'
STATICFILES_DIRS = [BASE_DIR / 'static',]
STATICFILES_STORAGE = 'django.contrib.staticfiles.storage.StaticFilesStorage'
```



Static Files in Production

Handling static files in production isn't quite as easy as your development environment since you'll be using either a WSGI (like Gunicorn) or ASGI (like Uvicorn) compatible web application server, which are used to serve up the dynamic content -- i.e., your Django source code files.

There are a number of different ways to handle static files in production, but the two most popular options are:

1. Use a web server like Nginx to route traffic destined for your static files directly to the static root (configured via `STATIC_ROOT`)
2. Use WhiteNoise to serve up static files directly from the WSGI or ASGI web application server

Regardless of the option, you'll probably want to leverage a CDN.

> For more on these options, review How to deploy static files.

Nginx

Sample Nginx config:

```
upstream hello_django {
    server web:8000;
}

server {

    listen 80;

    location / {
        proxy_pass http://hello_django;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }

    location /static/ {
        alias /home/app/web/staticfiles/;
    }

}
```

In short, when a request is sent to `/static/` -- e.g, `/static/base.css` -- Nginx will attempt to serve the file from the "/home/app/web/staticfiles/" folder.

> Curious how the above Nginx config works? Check out the Dockerizing Django with Postgres, Gunicorn, and Nginx tutorial.

Additional resources:

1. Prefer to store your static files on Amazon S3? Check out Storing Django Static and Media Files on Amazon S3.
2. Prefer to store your static files on DigitalOcean Spaces? Check out Storing Django Static and Media Files on DigitalOcean Spaces.

WhiteNoise

You can use WhiteNoise to serve static files from a WSGI or ASGI web application server.

The most basic set up is simple. After you install the package, add WhiteNoise to the `MIDDLEWARE` list above all other middleware apart from `django.middleware.security.SecurityMiddleware`:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',  # <---- WhiteNoise!
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

Then, for compression and caching support, update `STATICFILES_STORAGE` like so:

```
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

That's it! Turn off debug mode, run the `collectstatic` command, and then run your WSGI or ASGI web application server.

> Refer to the Using WhiteNoise with Django guide for more on configuring WhiteNoise to work with Django.

Media Files

Again, media files are files that your end-users (internally and externally) upload or are dynamically created by your application (often as a side effect of some user action). They are not typically kept in version control.

Almost always, the files associated with the FileField or ImageField model fields should be treated as media files.

Just like with static files, the handling of media files is configured in the *settings.py* file.

Essential configuration settings for handling media files:

1. MEDIA_URL: Similar to the `STATIC_URL`, this is the URL where users can access media files.
2. MEDIA_ROOT: The absolute path to the directory where your Django application will serve your media files from.
3. DEFAULT_FILE_STORAGE: The file storage class you'd like to use, which controls how the media files are stored and accessed. The default is FileSystemStorage.

> Refer to the File uploads section from Settings for additional configuration settings.

Media Files in Development Mode

Typical development config:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'uploads'
```

Unfortunately, the Django development server doesn't serve media files by default. Fortunately, there's a very simple workaround: You can add the media root as a static path to the `ROOT_URLCONF` in your project-level URLs.

Example:

```
from django.conf import settings
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import path, include


urlpatterns = [
    path('admin/', admin.site.urls),
    # ... the rest of your URLconf goes here ...
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Media Files in Production

When it comes to handling media files in production, you have less options than you do for static files since you can't use WhiteNoise for serving media files. Thus, you'll typically want to use Nginx along with django-storages to store media files outside the local file system where your application is running in production.

Sample Nginx config:

```
upstream hello_django {
    server web:8000;
}

server {

    listen 80;

    location / {
        proxy_pass http://hello_django;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }

    location /media/ {
        alias /home/app/web/mediafiles/;
    }

}
```

So, when a request is sent to `/media/` -- e.g, `/media/upload.png` -- Nginx will attempt to serve the file from the "/home/app/web/mediafiles/" folder.

> Curious how the above Nginx config works? Check out the Dockerizing Django with Postgres, Gunicorn, and Nginx tutorial.

Additional resources:

1. Prefer to store your media files on Amazon S3? Check out Storing Django Static and Media Files on Amazon S3.
2. Prefer to store your media files on DigitalOcean Spaces? Check out Storing Django Static and Media Files on DigitalOcean Spaces.

Conclusion

Static and media files are different and must be treated differently for security purposes.

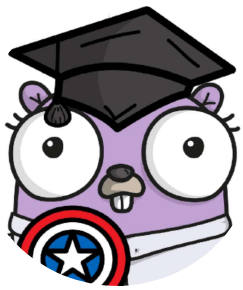
In this article, you saw examples of how to serve static and media files in development and production. In addition, the article also covered:

1. The different settings for both types of files
2. How Django handles them with minimal configuration

You can find a simple Django project with examples for serving static files in development and production and media files in development here.

This article only walks you through handling static and media files in Django. It does not discuss pre/post-processing for the static files such as minifying and bundling. For such tasks, you have to set up complex build processes with tools like Rollup, Parcel, or webpack.

django



Amal Shaji

Amal is a full-stack developer interested in deep learning for computer vision and autonomous vehicles. He enjoys working with Python, PyTorch, Go, FastAPI, and Docker. He writes to learn and is a professional introvert.

CONTRIBUTORS

 Michael Herman

SHARE THIS TUTORIAL

Twitter Reddit Hacker News Facebook

Featured Course

Test-Driven Development with Django, Django REST Framework, and Docker

Buy Now $30

View Course

Search all tutorials

TUTORIAL TOPICS

api architecture aws devops django django rest framework docker fastapi flask front-end heroku kubernetes machine learning python react task queue testing vue web scraping

RECOMMENDED TUTORIALS

Storing Django Static and Media Files on Amazon S3

 Michael Herman Jan 29th, 2023

Configure Django to load and serve up static and media files, public and private, via an Amazon S3 bucket.

aws devops django docker

Django Session-based Auth for Single Page Apps

Nik Tomazic Jul 31st, 2023

Add session-based authentication to a Single-Page Application (SPA) powered by Django and React.

architecture django front-end react

Dockerizing Django with Postgres, Gunicorn, and Nginx

Michael Herman Jul 27th, 2023

This tutorial details how to configure Django to run on Docker along with Postgres, Nginx, and Gunicorn.

django docker

Stay Sharp with Course Updates

Join our mailing list to be notified about updates and new releases.

Enter your email Subscribe

TestDriven.io is a proud supporter of open source

10% of profits from each of our FastAPI courses and our Flask Web Development course will be donated to the FastAPI and Flask teams, respectively.

Follow our contributions

© Copyright 2017 - 2023 TestDriven Labs.
Developed by Michael Herman.

Follow @testdrivenio