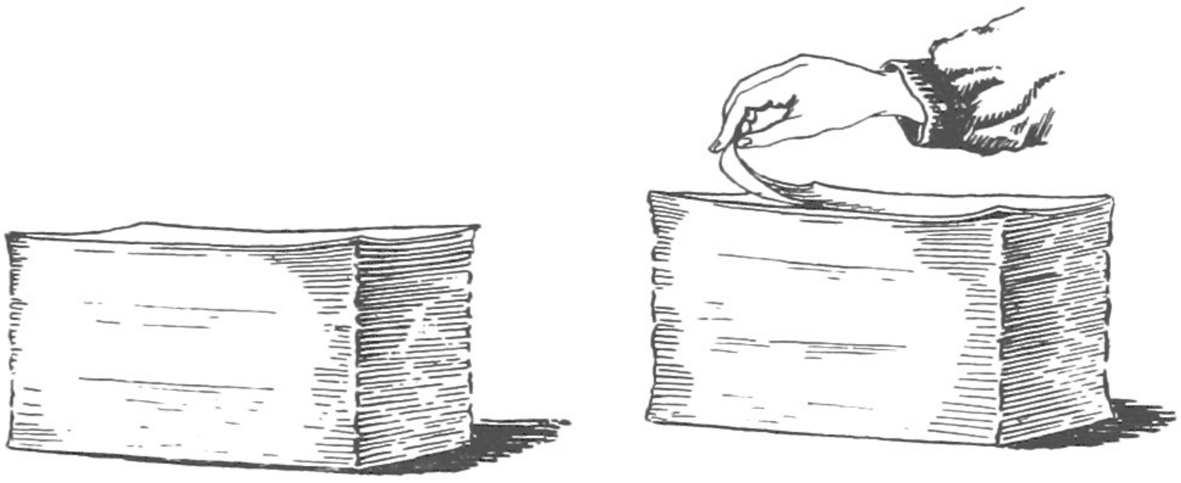
Django versus Flask with Single File Applications

2019-04-03



A lot of people pick Flask over Django because they believe it is simpler to start with. Indeed, the Flask front page includes an 8 line “hello world” application, whilst the Django default project has 172 lines in 5 files, and still doesn’t say “hello world”! (It *does* show you a welcome rocket and have a full admin interface though, both are pretty fun).

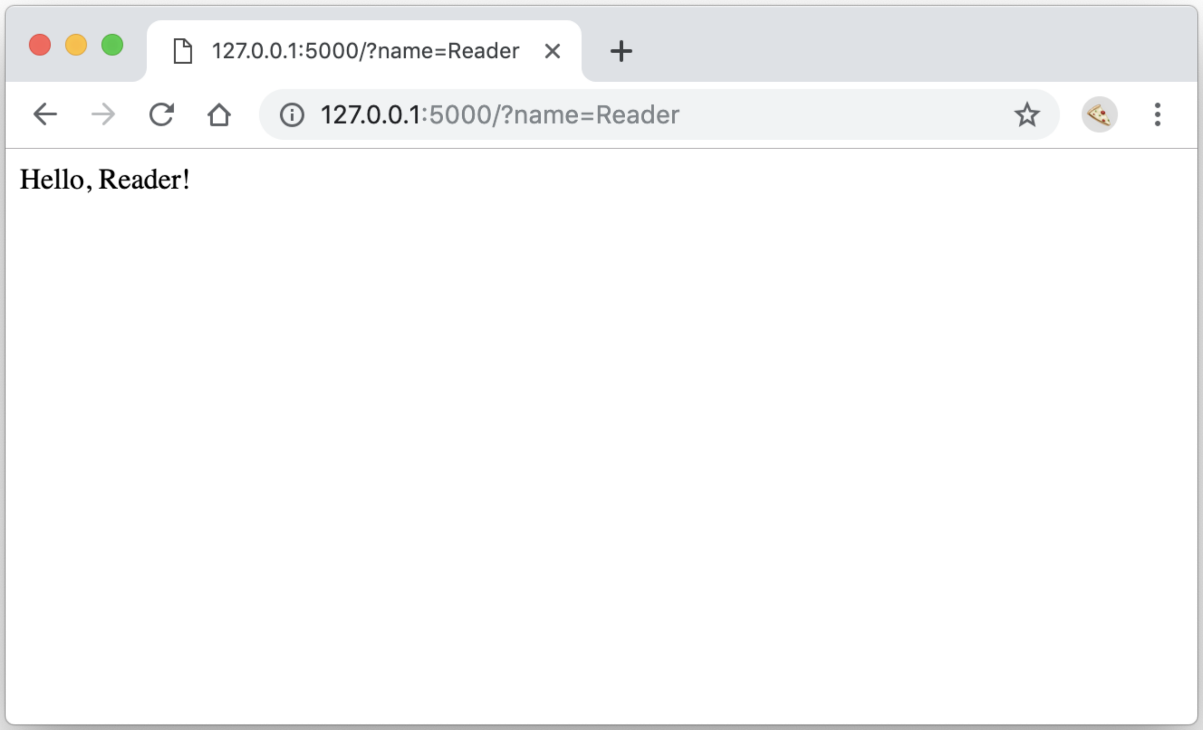
It *is* also possible to make a single file “hello world” application in Django, although it’s probably something most Django developers haven’t seen done. It takes a bit more code than Flask, but it’s still quite understandable.

Let’s take a look at the Flask and Django implementations and use them to compare the two frameworks.

Application

We'll be using the simple "hello world" application from the Pallets page on Flask (plus a security fix). It has one page which gets a name from the URL or the default "world", and returns that with a "hello" message.

For example:



Hello, Reader!

In Flask

Here is the code, 10 lines long, tested on Flask 1.0.2 and Python 3.7.2:

(I added `html.escape` to the Flask example, to fix a security problem.)

```
import html

from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def hello():
    name = request.args.get("name", "World")
    return f'Hello, {html.escape(name)}!'
```

We run it with:

```
$ FLASK_APP=app.py FLASK_ENV=development flask run
```

Here's what's happening:

1. An application object is created with the name equal to the module's import name, the special variable `__name__`.
2. The decorator `@app.route('/')` is applied to the function `hello()`. This means that when the URL `/` is visited, that function will be run to generate a response.
3. The `hello()` function does two things. First it gets the value of the `name` parameter from the URL arguments, or the default value "World". Then it returns a string which contains that value, which Flask turns into an HTTP response.

Not too much to understand!

In Django

This is a bit longer at 31 lines, but stay with me, all will be explained! Here is the code, tested on Django 2.1.7 and Python 3.7.2:

```
import html
import os
import sys

from django.conf import settings
from django.core.wsgi import get_wsgi_application
from django.http import HttpResponse
from django.urls import path
from django.utils.crypto import get_random_string

settings.configure(
    DEBUG=(os.environ.get("DEBUG", "") == "1"),
    ALLOWED_HOSTS=["*"],  # Disable host header validation
    ROOT_URLCONF=__name__,  # Make this module the urlconf
    SECRET_KEY=get_random_string(50),  # We aren't using any secu
)


def index(request):
    name = request.GET.get("name", "World")
    return HttpResponse(f"Hello, {html.escape(name)}!")


urlpatterns = [
    path("", index),
]

app = get_wsgi_application()

if __name__ == "__main__":
    from django.core.management import execute_from_command_line
    execute_from_command_line(sys.argv)
```

We run this with:

```
$ DEBUG=1 python app.py runserver
```

Here's what's happening:

1. We configure Django's settings. Normally Django's settings get defined in a separate file, but here we're using `settings.configure()` to

set them directly (which has a small caveat).

2. We set `DEBUG` to be `True` when the `DEBUG` environment variable is set to `"1"` . This is not strictly necessary but it's very useful to switch on debug mode when developing.
3. We set `ALLOWED_HOSTS` to accept any host in URL's. This lazily disables Django's Host header validation, a security feature which protects against types of attack. In a valuable application we'd set this correctly to contain only the application's domain name.
4. We set `ROOT_URLCONF` so the current module (again the `__name__` magic variable) will be used by Django for the URL configuration. This is normally a separate file too. Django imports the named module and accesses its `urlpatterns` attribute to get the URL mapping in the application, which is why we have it later in this file.
5. We set `SECRET_KEY` to a random string. As the comment says, it's needed by Django but here we aren't using any of the features that depend on it. Currently the random key changes every time the application starts, which would cause some security features to not work properly, such as logins being invalidated. Normally we would be using the security features, so we'd set this to a single random string that we'd store securely, not in the source code, but for example, in an environment variable.
6. We create a view function `index()` that implements the functionality to say "hello" to the name in the URL. This is passed an instance of Django's `HttpRequest` class, from which the URL argument "name" is retrieved, and returns an `HttpResponse` with the content we want.
7. We define the `urlpatterns` list which maps the empty path, equivalent to `/` , to the `index` function.
8. We create the WSGI application object. Django can create this for us in development mode but not for deployment in production.
9. We add the Python idiom `if __name__ == "__main__"` to call Django's commandline entrypoint to run when the file is executed

directly. This allows us to run `python app.py runserver` , as well as `python app.py --help` and any of Django's other commands.

There's a bit more code doing a few more things, but it's still less than 10 steps.

Comparison

This is a limited example, so it doesn't show every way the frameworks differ. However I do think it shows some of the most important ways, especially as we consider ways to extend the application into doing something a bit more useful.

Here are seven different parts of both frameworks that I'd like to take a closer look at:

1. Application Objects

In Flask, we create an application object as our first step, and later attach our route to it. The application is the top-level "thing" we configure, and also the WSGI application that we can use in production.

If we were extending the application, we'd also attach other things like error handlers or settings to this object.

This also lets us have more than one application in the same Python process, if we ever need to.

Django, by contrast, only allows us to have a single application per process, and it's implicitly created for us. The top-level "thing" we are configuring is Django itself, which we do when we configure `django.conf.settings` to let Django discover the other parts of the application such as the URL's.

The WSGI application object comes across as a bit of an afterthought. Django can automatically create one for us in development, and it's only needed for production so that common WSGI servers can run the application.

Overall here I'd say the Flask method of explicitly creating your application object and attaching things to it *is* clearer. That said, once you've created an application in one module, in some situations Python makes it hard to import it in another file, and Flask has a "workaround" global object `flask.current_app` to fix this.

Also whilst Flask allows multiple applications in a single process, I've not seen the need (except it could make certain test situations easier).

> **Sidebar: Django Apps**
> Django also has the concept of apps. These are individual sections of your application, containing related files that together implement some functionality. For example, you might have a "profile" app containing everything related to user profiles. In order to reduce this confusion, and counter to the Django documentation, I normally say "application" to mean the whole shebang, and "app" to mean those individual sections.

Flask calls its related feature blueprints which is a whole lot easier to talk about, unless you're making an application for city planners.

2. View Functions

This is the key code you write web applications, taking a request and returning a response. Everything a web framework does for you is, simply, either wrapping around your views or providing some kind of help for you to write those views quicker.

Our example views are short and similar, but differ in two key ways.

First, in Flask we import and use `flask.request` to get the headers, whilst in Django we are directly passed the request.

Flask uses some magic to maintain this global variable `flask.request` (a proxy object), which changes each time the function is run (and in each thread). The first time you read the code this might surprise you, as normally in Python imported objects don’t change. Django uses normal Python semantics, which reduces the confusion for beginners.

Global variables have been considered harmful by many programmers for a long time. Whilst they are convenient, they’re an easy place for bugs to lurk as any part of the program could change them at any point.

Second, in Flask we return a string, whilst in Django we return an `HttpResponse` . Flask actually supports multiple return types, including its own response class, but Django always requires you to explicitly return a response.

This is a philosophical difference between the two. Django tends to follow Python’s TOOWTDI slogan (There’s Only One Way To Do It) (pronounced “toody”), whilst Flask often provides several extra shortcuts. Django does provide some shortcuts but you have to explicitly import and use them.

3. URL Configuration

Flask lets us set up URL's with the `route()` decorator, which also takes other arguments such as restricting which HTTP methods the route accepts. Again though, Flask has more than one way to do it, with three ways to add URL's, the last being a URL map similar to Django's URL conf.

Django only supports a URL conf, which can add URL's from other URL confs with `include()`. This made us write a few more lines in our single file application, but it is a single way of doing URL's and a technique that can scale to big sites.

Our single application uses the most basic URL, so we didn't get to see URL syntax, but I want to briefly point out both support a syntax like `/users/<int:user_id>/profile/`. Originally Django supported only regular expression based URL's, but this can get really complicated and make beginners run for the hills. Django version 2.0 copied Flask's simpler URL syntax as a new, default option, so now they both work similarly.

4. Development Mode

Both frameworks support a development mode, to use when developing the application locally. Each provides a web server, automatic reloading the code changes, and debugging information in the browser when things break. Flask even includes an in-browser debugger which can be added to a Django project as well.

Flask asks us to set `FLASK_ENV=development` when we run our application with `flask run` in order to activate its development mode. Another option is to set `app.debug` directly.

Django only checks the settings `DEBUG` flag to activate development mode, and it's up to us how that is set. Some projects use separate settings files for development and production, whilst others use in-file mechanisms, like checking an environment variable. I prefer the latter, and that's what we've done in our single file example. This follows the well-known 12 factor methodology.

Besides this, Django also needs some boilerplate code under `if __name__ == "__main__"` to support running the application. This boilerplate still exists in larger projects in a file called `manage.py` but it is generated for us when we run the `startproject` command. Flask avoids the boilerplate by having `flask run` always work.

Going beyond basic development mode, it's popular to extend Django's debug mode with the django-debug-toolbar package. It displays several extra types of debug information and is well maintained by the community. There's a similar package for Flask called flask-debugtoolbar but it seems to have fewer features and at time of writing hasn't been updated in a year.

5. Security

Both applications do some work to avoid HTML injection, by using `html.escape` to safely output the URL input from the user. This is pretty standard, and in larger applications we'd probably use a template system which would do this for us automatically (more later).

Another security feature we saw is host validation. Django has it built-in by default and we have to configure it to work, but in this example we deliberately disabled it. Flask, as far as I know, has no such feature, so whilst we saved doing work to configure it, it may be less secure here (it depends on how the application is deployed).

Another security feature which Django makes us think of up-front is adding a secret key. Flask also has this concept in `app.secret_key`. Both frameworks use their keys for cryptographic functions, like encrypting user sessions, it's just that Django *requires* one to start. It's just a rare case that our single file application doesn't use it in any way, and whilst in theory Django could be changed to not *require* a key, it's probably not worth it.

Both frameworks also have security pages in their documentation: Django / Flask. We can see the philosophical differences again on these pages. Django follows Python's "batteries included" philosophy, whilst Flask either refers to other libraries, such as Google's Flask-Talisman, or gives you a code snippet and asks you to finish the implementation yourself.

Django has a dedicated Security Team and security policies that detail such things as how to report security vulnerabilities without making them public. As far as I can tell, Flask has no equivalent.

Sidebar: Governance
Django is maintained by the Django Software Foundation, a US non-profit. The donations it collects fund Django education, two fellows to maintain the project, and a number of other things.

Flask's is maintained by Pallets, whose Structure Page lists Armin Ronacher as the head. It is entirely volunteer-lead with no registered organization.

6. More Batteries

Both frameworks do include some "batteries" which can help you build your application faster, although Flask only provides a minimum. Django also tends to provide its extras as pluggable backends so you can swap them out for those from third party packages, for example using an unofficial database backend.

If we had larger pages in our example, we'd probably want to use a templating system to keep the content in multiple files, and manage inserting the data without doing HTML escaping each time. Both frameworks provide template system integrations. Django has a pluggable backend system which includes support for its own Django Template Language, whilst Flask only supports Jinja2. Jinja2 is essentially Django Template Language rebuilt for performance by Armin Ronacher, but with enough differences that it's not fully compatible. However Django also provides a backend for using Jinja2, so you can use it there.

Nearly every web application needs some kind of data storage. Django includes a very fully featured database layer, including migrations. Flask doesn't include anything but it's very common to use it with SQLAlchemy + Alembic.

Besides these examples, Django includes many extra tools beyond Flask, for things like translations, site maps, and geographic data (many are in its contrib packages).

7. Ecosystem

Both frameworks have ecosystems providing extra integrated tools in third party packages.

Django has the site djangopackages which lists and compares several thousand packages.

The Flask website has a moderated list of just 68 packages.

Overall it seems there are many more Django-specific packages, although this is in keeping with Flask's minimalist/DIY philosophy.

Conclusion

Overall, I prefer Django. This is probably not so very surprising, given I'm a core contributor, but I hope this has shown I have some reasoning behind this viewpoint! I must admit though I'm not a Flask expert, I've only worked on a few projects using it for short-term engagements, and I learned a lot about it whilst composing this blog post. If I've misrepresented anything please let me know.

According to the Python Developers Survey 2018, Flask and Django have about even usage at the moment, at 47% and 45% respectively. I think part of the preference for Flask is that it's easy to start with just a single file, so I hope my example shows this isn't necessarily the case. Using Django shouldn't be hard for small applications, though there's probably some work we can do to make it easier.

Enjoy creating web applications whichever framework you use,

—Adam

Thanks to Jazeps Basko, Markus Holtermann, and Tom Grainger for reviewing this post.

Related posts: