

Design Decisions in Flask

If you are curious why Flask does certain things the way it does and not differently, this section is for you. This should give you an idea about some of the design decisions that may appear arbitrary and surprising at first, especially in direct comparison with other frameworks.

The Explicit Application Object

A Python web application based on WSGI has to have one central callable object that implements the actual application. In Flask this is an instance of the **Flask** class. Each Flask application has to create an instance of this class itself and pass it the name of the module, but why can't Flask do that itself?

Without such an explicit application object the following code:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello World!'
```

Would look like this instead:

```
from hypothetical_flask import route

@route('/')
def index():
    return 'Hello World!'
```

There are three major reasons for this. The most important one is that implicit application objects require that there may only be one instance at the time. There are ways to fake multiple applications with a single application object, like maintaining a stack of applications, but this causes some problems I won't outline here in detail. Now the question is: when does a microframework need more than one application at the same time? A good example for this is unit testing. When you want to test something it can be very helpful to create a minimal application to test specific behavior. When the application object is deleted everything it allocated will be freed again.

Another thing that becomes possible when you have an explicit object lying around in your code is that you can subclass the base class (**Flask**) to alter specific behavior. This would

not be possible without hacks if the object were created ahead of time for you based on a class that is not exposed to you.

But there is another very important reason why Flask depends on an explicit instantiation of that class: the package name. Whenever you create a Flask instance you usually pass it `__name__` as package name. Flask depends on that information to properly load resources relative to your module. With Python's outstanding support for reflection it can then access the package to figure out where the templates and static files are stored (see [`open_resource\(\)`](#)). Now obviously there are frameworks around that do not need any configuration and will still be able to load templates relative to your application module. But they have to use the current working directory for that, which is a very unreliable way to determine where the application is. The current working directory is process-wide and if you are running multiple applications in one process (which could happen in a webserver without you knowing) the paths will be off. Worse: many web servers do not set the working directory to the directory of your application but to the document root which does not have to be the same folder.

The third reason is “explicit is better than implicit”. That object is your WSGI application, you don't have to remember anything else. If you want to apply a WSGI middleware, just wrap it and you're done (though there are better ways to do that so that you do not lose the reference to the application object [`wsgi_app\(\)`](#)).

Furthermore this design makes it possible to use a factory function to create the application which is very helpful for unit testing and similar things ([Application Factories](#)).

The Routing System

Flask uses the Werkzeug routing system which was designed to automatically order routes by complexity. This means that you can declare routes in arbitrary order and they will still work as expected. This is a requirement if you want to properly implement decorator based routing since decorators could be fired in undefined order when the application is split into multiple modules.

Another design decision with the Werkzeug routing system is that routes in Werkzeug try to ensure that URLs are unique. Werkzeug will go quite far with that in that it will automatically redirect to a canonical URL if a route is ambiguous.

One Template Engine

Flask decides on one template engine: Jinja2. Why doesn't Flask have a pluggable engine interface? You can obviously use a different template engine, but Flask will still

configure Jinja2 for you. While that limitation that Jinja2 is *always* configured will probably go away, the decision to bundle one template engine and use that will not.

Template engines are like programming languages and each of those engines has a certain understanding about how things work. On the surface they all work the same: you tell the engine to evaluate a template with a set of variables and take the return value as string.

But that's about where similarities end. Jinja2 for example has an extensive filter system, a certain way to do template inheritance, support for reusable blocks (macros) that can be used from inside templates and also from Python code, uses Unicode for all operations, supports iterative template rendering, configurable syntax and more. On the other hand an engine like Genshi is based on XML stream evaluation, template inheritance by taking the availability of XPath into account and more. Mako on the other hand treats templates similar to Python modules.

When it comes to connecting a template engine with an application or framework there is more than just rendering templates. For instance, Flask uses Jinja2's extensive autoescaping support. Also it provides ways to access macros from Jinja2 templates.

A template abstraction layer that would not take the unique features of the template engines away is a science on its own and a too large undertaking for a microframework like Flask.

Furthermore extensions can then easily depend on one template language being present. You can easily use your own templating language, but an extension could still depend on Jinja itself.

Micro with Dependencies

Why does Flask call itself a microframework and yet it depends on two libraries (namely Werkzeug and Jinja2). Why shouldn't it? If we look over to the Ruby side of web development there we have a protocol very similar to WSGI. Just that it's called Rack there, but besides that it looks very much like a WSGI rendition for Ruby. But nearly all applications in Ruby land do not work with Rack directly, but on top of a library with the same name. This Rack library has two equivalents in Python: WebOb (formerly Paste) and Werkzeug. Paste is still around but from my understanding it's sort of deprecated in favour of WebOb. The development of WebOb and Werkzeug started side by side with similar ideas in mind: be a good implementation of WSGI for other applications to take advantage.

Flask is a framework that takes advantage of the work already done by Werkzeug to properly interface WSGI (which can be a complex task at times). Thanks to recent developments in the Python package infrastructure, packages with dependencies are no longer an issue and there are very few reasons against having libraries that depend on others.

Thread Locals

Flask uses thread local objects (context local objects in fact, they support greenlet contexts as well) for request, session and an extra object you can put your own things on ([g](#)). Why is that and isn't that a bad idea?

Yes it is usually not such a bright idea to use thread locals. They cause troubles for servers that are not based on the concept of threads and make large applications harder to maintain. However Flask is just not designed for large applications or asynchronous servers. Flask wants to make it quick and easy to write a traditional web application.

Also see the [Becoming Big](#) section of the documentation for some inspiration for larger applications based on Flask.

What Flask is, What Flask is Not

Flask will never have a database layer. It will not have a form library or anything else in that direction. Flask itself just bridges to Werkzeug to implement a proper WSGI application and to Jinja2 to handle templating. It also binds to a few common standard library packages such as logging. Everything else is up for extensions.

Why is this the case? Because people have different preferences and requirements and Flask could not meet those if it would force any of this into the core. The majority of web applications will need a template engine in some sort. However not every application needs a SQL database.

The idea of Flask is to build a good foundation for all applications. Everything else is up to you or extensions.