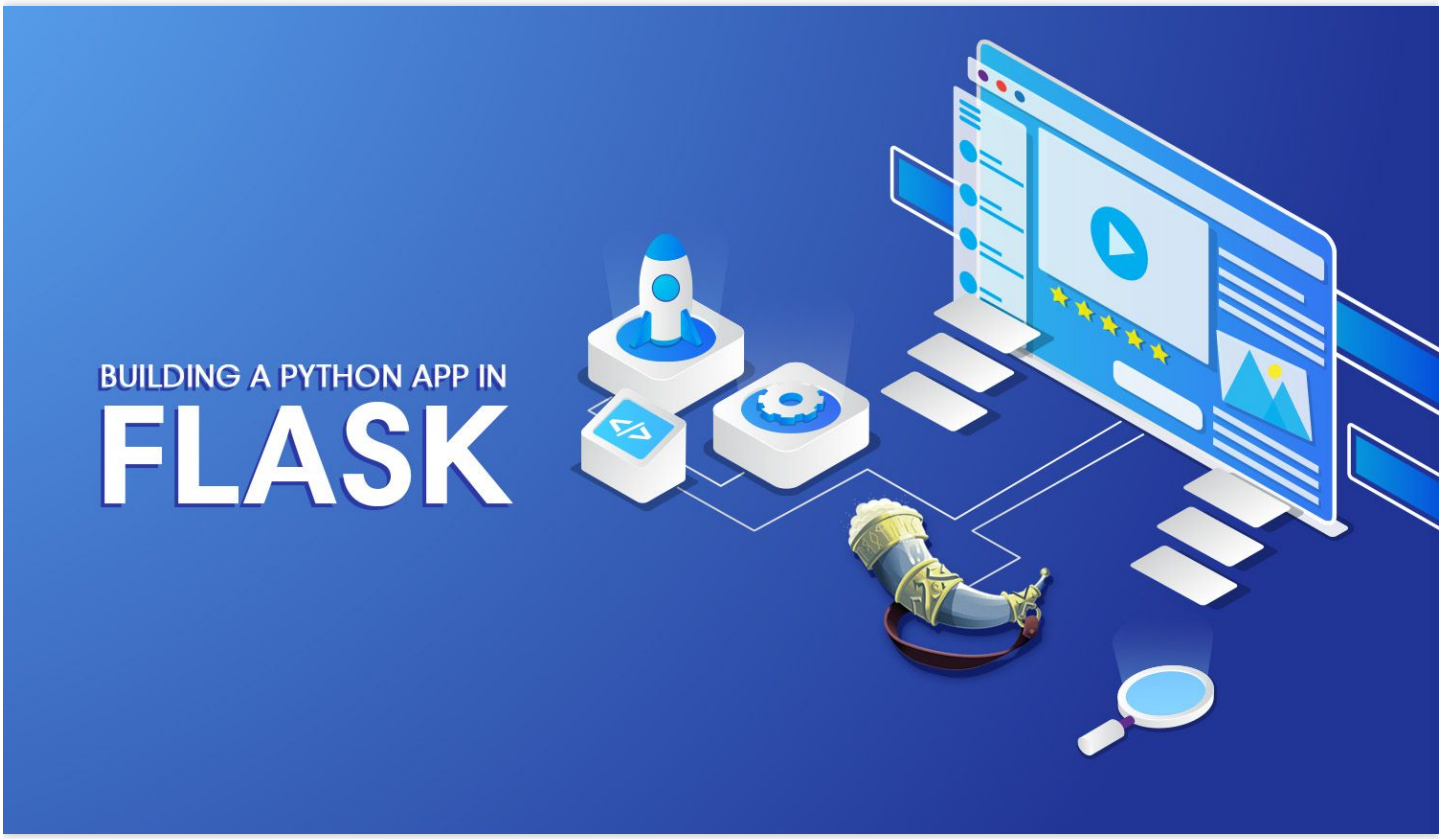
undefined Logo

# Creating Your First Flask Application

Todd Flask 5 Min Read

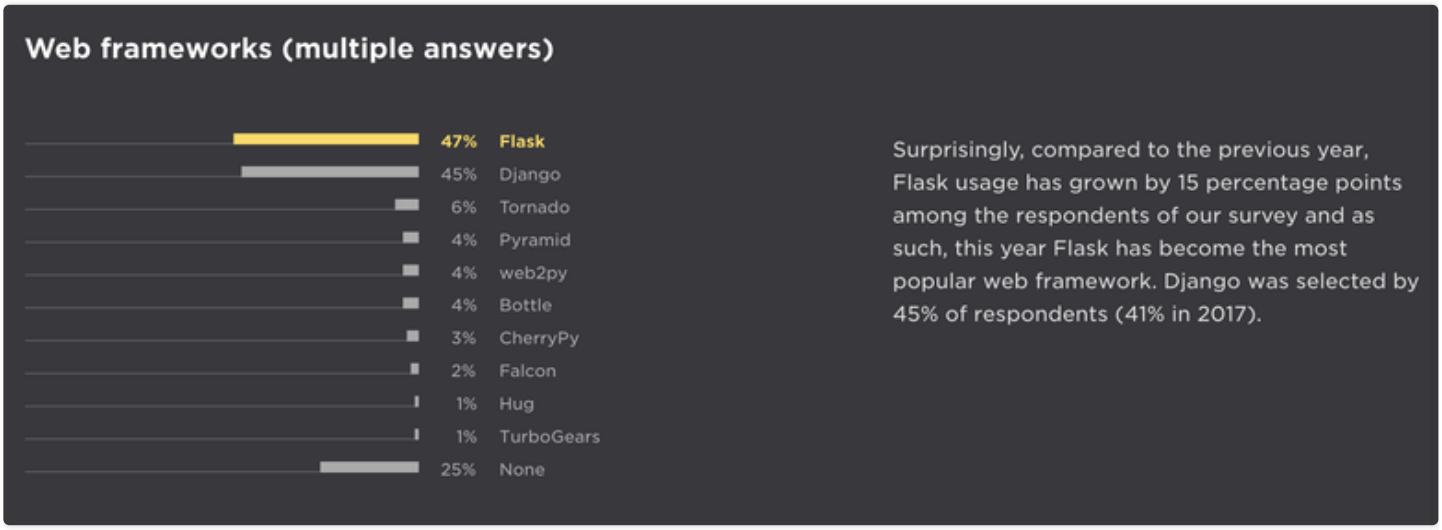



XI. Managing Session Data with Flask-Session & Redis
X. Handle User Accounts & Authentication in Flask with Flask-Login
IX. Connect Flask to a Database with Flask-SQLAlchemy
VIII. Compiling and Serving Frontend Assets in Flask
VII. Organizing Flask Apps with Blueprints
VI. Demystifying Flask's Application Factory
V. Configuring Your Flask App
IV. The Art of Routing in Flask
III. Handling Forms in Flask with Flask-WTF
II. Rendering Pages in Flask Using Jinja
I. **Creating Your First Flask Application**

Evidence of Flask's rise to power has been all around us for a couple of years now. Anybody paying close attention to the technology stacks chosen by startups has

undoubtedly noticed a flip: at some point, the industry standard flipped away from Django entirely.

Huge bets are being placed on Flask across the industry. **Plotly's** famous **Dash** product is an extension of Flask which has seen significant success. Even major cloud providers, namely **Google Cloud**, are choosing to default to Flask for Python implementations of serverless apps. Google Cloud Functions and Google App Engine both ship with Flask running at their core. JetBrains finally put an official number to this trend with their 2018 Python survey: **47%** of devs report using Flask to Django's **45%**. Game: Blouses.




Flask wins market dominence in 2018.

Put down the pitchforks: this isn't a Flask vs. Django post, nor are we implying that one framework is superior to the other. Both frameworks have their place, with that "place" being in the realm of preference.

## Why Flask?

Developing apps in Flask has a much different narrative than when developing in more traditional MVC Frameworks. In the past, the setup of a framework would easily take hours: with the assumption that our app needed all the bells and whistles, it was impossible to get a "Hello world!" off the ground without a full understanding of database configurations, static assets, templates, and other things our app may not even need. This is especially a concern for the Python ecosystem. Few people turn to Python for the sole purpose of building a web app: the vast majority of Python developers are in the field of data analysis without a traditional background in application development. Asking data analysts (who have mostly become accustomed to Jupyter notebooks) to pick up all the fundamentals of web development before even getting started is just unrealistic.

Flask's setup is merely a copy+paste of the following five lines:

app.py

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

@app.route("/")
def hello():
    return "Hello World!"
```

Those five lines create a live Flask application. Without any other knowledge about the framework, we can immediately plug in any Python logic we already have to change “Hello world!” to match any output imaginable. While it's possible to create an entire Flask application as a single tiny file, Flask can be as extended to be just as powerful and complex as its predecessors. When the author of a Flask application deems it necessary, various Flask plugins can be pulled in to give us robust application logic. Examples include:

- **Flask-SQLAlchemy** for database interaction.
- **Flask-Sessions** for user session management.
- **Flask-Login** to manage user logins.
- Literally hundreds of other libraries.

This plug-and-play structure makes Flask projects feel more expressive while simultaneously providing simplicity to developers starting from 0. Not only that, but reading somebody else’s source suddenly becomes simple: I know this app must do XYZ, because this person has imported XYZ.

## Dissecting Flask’s “Hello World!”

Let's go back to our 5-line application to pick apart the specifics:

app.py

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

The most important part of the Flask Python library is **Flask** with a capital “F” (as in: `from flask import Flask`). This five-letter word creates an object which refers to the entirety of the app itself: when we state `app = Flask(__name__)`, we are creating the variable **app** which represents our application. Therefore, when we configure the variable

app, we're configuring the way our entire application works. For example, setting `app = Flask()` can accept a few attributes:

app.py

```
from flask import Flask

app = Flask(__name__,
            instance_relative_config=False,
            template_folder="templates",
            static_folder="static")
```

This is an example of creating a Flask app with a few specifics: the location of our config file, the folder in which we'll store pages templates, and the folder in which we'll store frontend assets (JS, CSS, images, etc.).

## A Basic Flask Route

The primary function of our app is called `hello()`, which is importantly wrapped by Flask's most important decorator: `.route()`. If you aren't familiar with **decorators** in Python, a decorator is a function for us to wrap other functions with. It isn't critically important to know all the details, other than that Flask comes with a route decorator which allows us to serve up functions based on which page of the app the user is loading. By setting `@app.route("/")`, we are specifying that the function `hello()` should fire whenever somebody uses our app.

Of course, we can return any value besides "Hello world!" if we wanted. Let's say you've already a script which returns the square of a number, plus 9. We could save that logic in a function called `squareOfNumberPlusNine()`, in a file called `logic.py`. Now, our script can look like this:

app.py

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

@app.route("/")
def hello():
    value = squareOfNumberPlusNine(5)
    return value
```

This would return **34** as opposed to "**Hello world!**". Without any prior knowledge of Python web development, we can already use Flask to plug into logic we've written and serve up a result.

## Other Parts of Flask's Core Library

We can import other things `from flask` besides `Flask`. Here are some examples:

### Serving Raw HTML

`Markup` allows us to return an HTML page by rendering a string as HTML:

app.py

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

@app.route("/")
def hello():
    return Markup("<h1>Hello World!</h1>")
```

### Serving an HTML Page Template

`return_template` will return an HTML page by finding the page in our `/templates` folder:

app.py

```
from flask import Flask, render_template
app = Flask(__name__, template_folder="templates")

@app.route("/")
def hello():
    return render_template("index.html")
```

### Serving a Response

`make_response` has a number of uses, the most notable of which is to serve a response in the form of a JSON object. If our application is an API, we'd return a response objects instead of pages:

app.py

```
from flask import Flask, make_response
```

```
app = Flask(__name__)

@app.route("/")
def hello():
    headers = {"Content-Type": "application/json"}
    return make_response('it worked!', 200, headers)
```

On the topic of creating APIs with Flask, we can also specify whether the route at hand is a POST, GET, or some other method. This is handled easily within the route decorator:

app.py

```
from flask import Flask, make_response, request
app = Flask(__name__)

@app.route("/", methods=['GET'])
def hello():
    if request.method != 'GET':
        return make_response('Malformed request', 400)
    headers = {"Content-Type": "application/json"}
    return make_response('it worked!', 200, headers)
```

The above function checks to make sure the user is accessing the endpoint with the correct method first. If they've used the incorrect method, we return an error.

To create JSON responses, check out Flask's built-in `jsonify()` function. `jsonify()` outputs a Python dict as JSON inline:

app.py

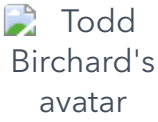
```
from flask import Flask, make_response, request, jsonify
app = Flask(__name__)

@app.route("/", methods=['GET'])
def hello():
    if request.method != 'GET':
        return make_response('Malformed request', 400)
    my_dict = {'key': 'dictionary value'}
    headers = {"Content-Type": "application/json"}
    return make_response(jsonify(my_dict), 200, headers)
```

If you're looking for an in-depth guide to serving routes in Flask, check out our post The Art of Routing in Flask. */selfpromotion*

## Succumb to Flask

Even if you chose to stick to your large Frameworks, it's easy to see why Flask is useful as a drop-in solution for many scenarios. There are undoubtedly plenty of useful Python scripts which go wasted because the final step of making them easily consumable by other people was never completed. Flask is an excellent way to achieve this last step, and the best part is: you already know how to use it.

Flask Python Architecture Software Development




### Todd Birchard

118 Posts Website Twitter

Engineer with an ongoing identity crisis. Breaks everything before learning best practices. Completely normal and emotionally stable.

Daveaneo
1 point
40 days ago

Your link, The Art of Routing in Flask, is to a password-protected page. Let us the Flask in!

Daveaneo
0 points
40 days ago

"return*template will return an HTML page by finding the page in our /templates folder:" I think you meant, "render*template".

ADD A COMMENT