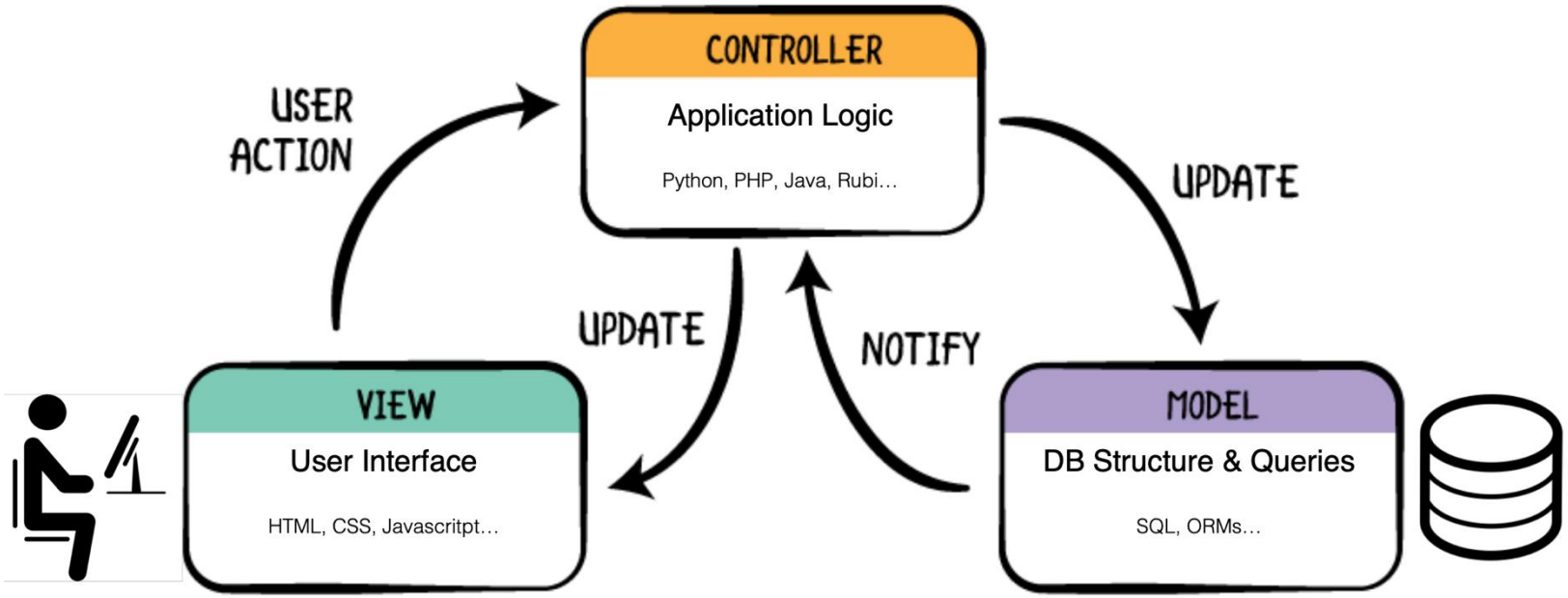
Flask

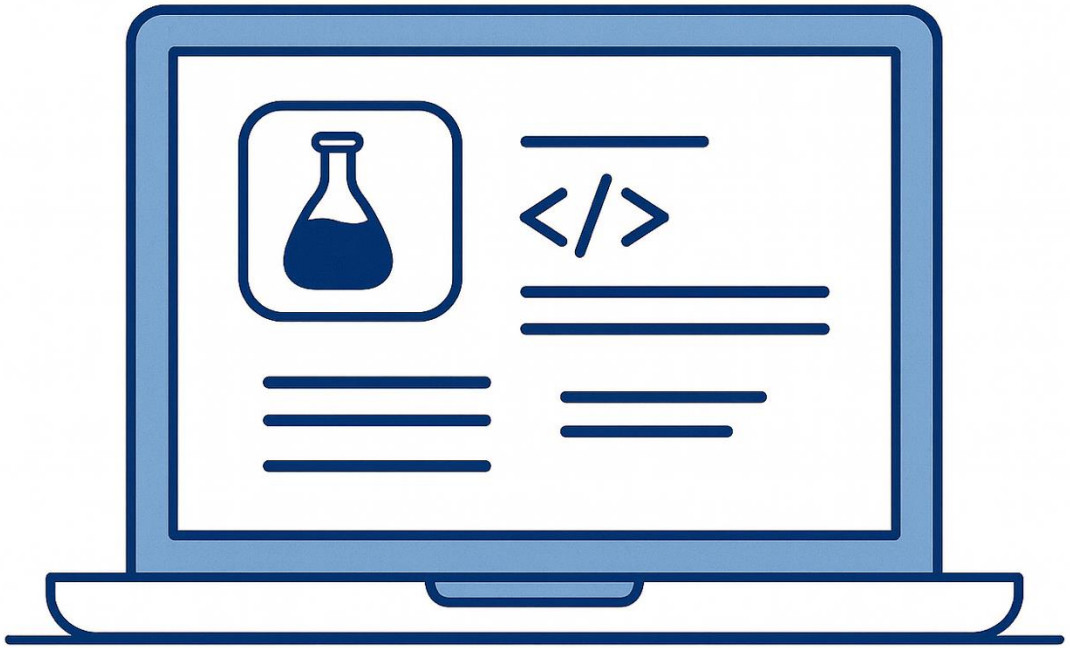
web development,

one drop at a time.

Model View Controller (MVC)

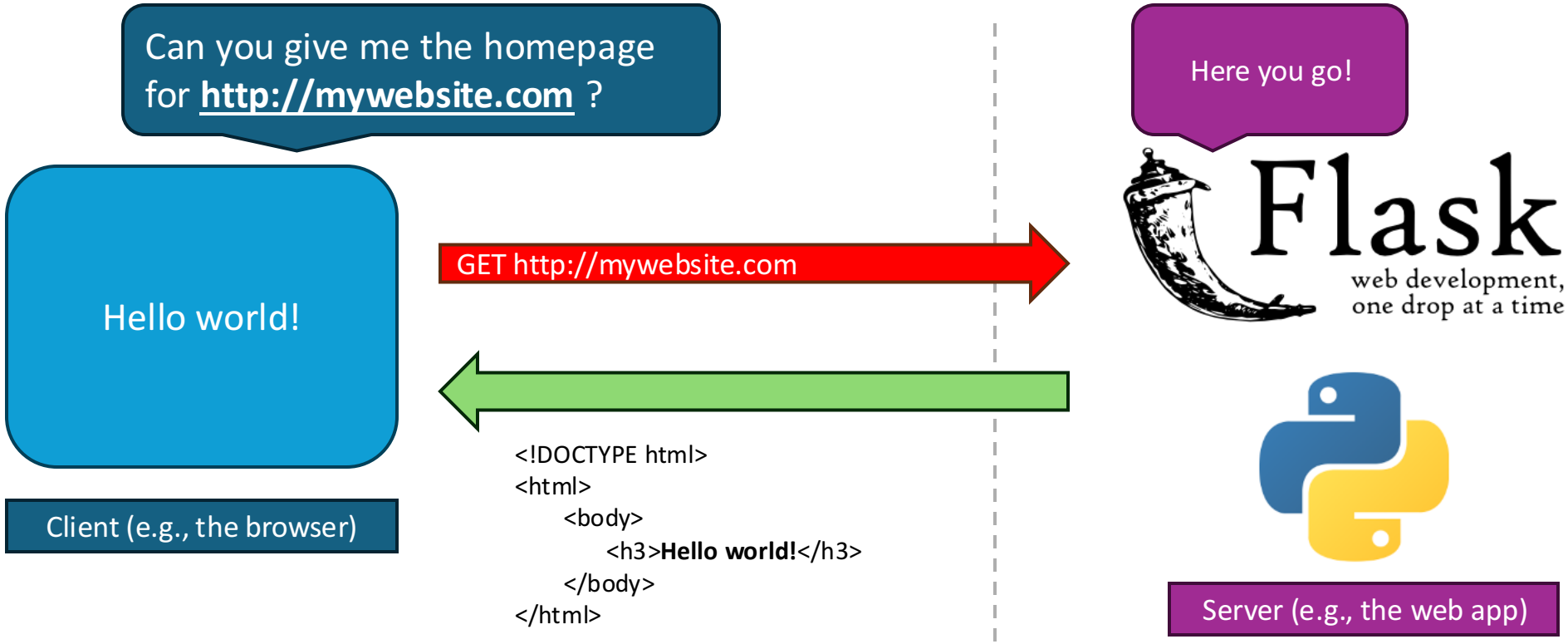


Introduction to Flask



What is Flask? A micro web framework written in Python, developed by Armin Ronacher and part of the Pallets project. It is a micro-framework because it does not provide a database layer or form validation and requires few external libraries.

Framework for building web apps in Python



Flask installation

Requirement: Flask supports Python ≥ 3.9

```
(Coding) marcocascio@Marcos-MacBook-Air coding % pip install Flask
Collecting Flask
  Downloading flask-3.1.2-py3-none-any.whl.metadata (3.2 kB)
Collecting blinker>=1.9.0 (from Flask)
  Using cached blinker-1.9.0-py3-none-any.whl.metadata (1.6 kB)
Collecting click>=8.1.3 (from Flask)
  Using cached click-8.3.0-py3-none-any.whl.metadata (2.6 kB)
Collecting itsdangerous>=2.2.0 (from Flask)
  Downloading itsdangerous-2.2.0-py3-none-any.whl.metadata (1.9 kB)
Collecting jinja2>=3.1.2 (from Flask)
  Using cached jinja2-3.1.6-py3-none-any.whl.metadata (2.9 kB)
Collecting markupsafe>=2.1.1 (from Flask)
  Downloading markupsafe-3.0.3-cp313-cp313-macosx_11_0_arm64.whl.metadata (2.7 kB)
Collecting werkzeug>=3.1.0 (from Flask)
  Downloading werkzeug-3.1.3-py3-none-any.whl.metadata (3.7 kB)
Downloading flask-3.1.2-py3-none-any.whl (103 kB)
Using cached blinker-1.9.0-py3-none-any.whl (8.5 kB)
Using cached click-8.3.0-py3-none-any.whl (107 kB)
Downloading itsdangerous-2.2.0-py3-none-any.whl (16 kB)
Using cached jinja2-3.1.6-py3-none-any.whl (134 kB)
Downloading markupsafe-3.0.3-cp313-cp313-macosx_11_0_arm64.whl (12 kB)
Downloading werkzeug-3.1.3-py3-none-any.whl (224 kB)
Installing collected packages: markupsafe, itsdangerous, click, blinker, werkzeug, jinja2, Flask
Successfully installed Flask-3.1.2 blinker-1.9.0 click-8.3.0 itsdangerous-2.2.0 jinja2-3.1.6 markupsafe-3.0.3 werkzeug-3.1.3
```

Writing your first Flask application

```
from flask import Flask

app = Flask(__name__)
app.run(debug=True, host='127.0.0.1', port=5000)
# In production, set debug to False
```

1. First, we import the **Flask** class.
2. Next, we **create an instance** of this class. The first argument is the name of the application's module or package. **__name__** is a convenient shortcut for this that is appropriate for most cases. This is needed so that Flask knows where to look for resources such as templates and static files.
3. We activate the **debug mode**:
   - interactive debugger;
   - automatic reloader;
   - logging and development config.

Running your first Flask application

```
(Coding) marcocascio@Marcos-MacBook-Air Flask % python3 main.py
 * Serving Flask app 'main'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
 Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000/
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 994-516-589
```

Since `app.run()` is present, we can **run the application** with:

> *python3 filename.py*

Note the **/**. That's the **route**.

The web server is running at **http://127.0.0.1:500**

but **we didn't specify what to return** when someone visits the **/** route.

Defining static routes

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home() -> str:
    return "<h3>Hello, world!</h3>"
```

Flask uses python **decorators** to **define the routes**. There can be any number of routes in your application.

To **run the application** you can also use the **flask** command. You need to tell Flask **where your application is** by using the `--app` option, followed by the file name (without the extension) that contains the `app` object. For example, if your python file is **main.py**, you would run:

> *flask --app **main** run* OR *flask --app **main** run --debug --host 127.0.0.1 --port 5000*

Notice that if the file is named **app.py** or **wsgi.py**, you don't need to use `--app` option.

In this case, you can simply run:

> *flask run* OR *flask run --debug --host 127.0.0.1 --port 5000*

Defining dynamic routes

```python
@app.route('/users/<string:username>')
def show_user_profile(username: str) -> str:
    return f"Profilo di {username}"

@app.route('/posts/<int:post_id>')
def show_post(post_id: int) -> str:
    return f"Post {post_id}"
```

It is possible to **include variable sections to a route** by marking sections with <*converter:variable_name*>.

Flask passes these **variables as arguments**. You can **specify converters** to control the type:

`string`	(default) accepts any text without a slash
`int`	accepts positive integers
`float`	accepts positive floating point values
`path`	like `string` but also accepts slashes
`uuid`	accepts UUID strings

URL building with url_for()

```
from flask import Flask, url_for

app = Flask(__name__)
…
with app.test_request_context():
    print(url_for('home'))
    print(url_for('show_user_profile', username='John Doe'))
    print(url_for('show_post', post_id=42))
```

The *url_for()* function allows you **to reconstruct the URL (i.e., the route)** for a specific function. It is useful because:

1. It lets Flask build the URL from the function name, passing variable parts as arguments, instead of having to remember and write it manually.
2. Changing the route in one place is automatically reflected everywhere.
3. The generated paths are always absolute, which helps avoid errors caused by using

relative paths.

4. It automatically handles special characters (e.g., %20 for spaces).