DEPARTMENT OF COMPUTER SCIENCE

University of Reading

CS2PP: Programming in Python

Web Programming with Flask

Recall: Why I don't use Python

In the early 2000s, I invested heavily in learning **C**, **Java** and **Perl**:

- **C** — dominant systems/applications langauge
- **Java** — dominant client-side Web language
- **Perl** — major server-side Web language
- **PHP** — major server-side Web language, but mostly bad code written by beginners
- **Python** — toy language no-one took seriously

Who uses Python for the Web?

- **C** — for Web servers only, not websites!
- **Java** — gigantic hulking corporate enterprise monoliths
- **Perl** — Booking.com, DuckDuckGo, formerly BBC News, Amazon
- **PHP** — Wordpress, phpBB, anything notorious for being hacked
- **Python** — NPR news, Pinterest, LinkedIn, Instagram

Trends in Web Development

- Early Web — static HTML files, images if lucky
- Server-side CGI and client-side Java applets — limited interactivity
- Perl and PHP — specialised programming languages, faster plugins
- HTML 4 + CSS — separation of markup and layout
- JavaScript and Flash — faster client-side interactivity
- "Web 2.0" — user-generated content, AJAX UI
- NodeJS — same language (JavaScript) on client and server
- Application frameworks — modular development
- WebAssembly and WebGL — high-performance on client, browser as OS
- "Web 3.0" — ???

Where does Python fit in?

Demo: Hello World in Flask

- `hello1.py` — first attempt at a Flask application
- `hello2.py` — use static content instead
- `hello3.py` — return HTML instead of plain text

Client side vs Server side

In the early Web, static files were stored on the server and sent over the network to the client (browser).

To get interactivity, we need support for some kind of computation.

On the client:

- we can only use browser-supported languages (today, mainly JavaScript and WebAssembly);
- sending computation results to the user is low-latency;
- website doesn't have to pay for computation, but also can't ensure it's available;
- any resources needed must be sent over the network;
- persistent results must be sent back to the server (over an unreliable network).

Client side vs Server side

In the early Web, static files were stored on the server and sent over the network to the client (browser).

To get interactivity, we need support for some kind of computation.

On the server:

- we can use any language we like;
- sending computation results to the user is high-latency;
- Web server needs enough capacity to serve all users' requests;
- resources (code, database) are available locally;
- persistent results can be immediately stored in a database.

LAMP: Linux + Apache + MySQL + PHP?

In the early 2000s, "LAMP" emerged as the most common Web development **stack**:

- **Linux** — highly configurable OS, modest hardware requirements;
- **Apache** — highly configurable Web server;
- **MySQL** — acceptable relational database (PostgreSQL is "better"?);
- **PHP** — easy to add interactivity onto an existing static site

Today:

- Linux still leads, but less visible in **cloud** configurations;
- Apache still popular, but **Nginx** more popular;
- MySQL forked and renamed **MariaDB**, PostgreSQL still "better", NoSQL popular;
- PHP thankfully dying, **Python** and **JavaScript** lead.

CGI — Common Gateway Interface

If your server side computation can be in any language, how does it communicate with the Web server?

Simple answer: all Web applications receive HTTP request data (including URL) on standard input (and through environment variables) and print page on standard output.

Good enough for smaller sites, but spawning a process has relatively high overhead.

Solutions that scale better:

- Run code in Web server (Apache's **mod_php**, **mod_perl**) — very fast, but must be reimplemented for each Web server/language pair; also increased security risks.
- CGI variants (**FastCGI**, **WSGI**) — persistent process communicates with server over pipe, local TCP socket or similar.

HTTP GET vs POST

HTTP is the main network protocol of the Web.

Originally: client sends textual request to **GET** an URL; server sends back text of page.

More recent versions include binary protocol with compression and reuse of network connection for multiple requests.

Web pages can also be requested by **POST**ed.

Conventionally, GET is used for changes that don't change state on the server, such as searching or viewing data. Parameters from GET show as part of the URL.

Meanwhile, POST is used for changes that persist, such as posting a comment, buying a product. Data is sent separately after the URL.

Idempotence: POST if making a request twice should be different from making it once.

Static vs Dynamic Content

- Early 1990s
  - Web server has directory of static files.
  - URLs mostly get mapped to directory paths.
- Late 1990s
  - A couple of CGI executables do dynamic pages; everything else static.
- 2000s
  - Everything is a PHP page generated dynamically on every page reload.
  - Apache's mod_rewrite breaks link between URL and directory path.
- 2010s
  - Make as much static as possible: better performance and security.
  - Push dynamic generation into client where possible; use server when needed.

Demo: Hello World in Flask

- `hello4.py` — 2 pages: request name and show name
- `hello5.py` — url_for() to avoid hard-coded URLs

Flask as a Micro Framework

Flask is a **micro** framework. It provides everything you need to write a Web application quickly, and nothing more. You can add more extensions, choosing what to add.

Flask has few dependencies, so you can install it easily and run it on low-powered hardware (such as a cheap VM).

Flask grew out of a joke web page. The developer wrote some low-quality software and advertised it with a punchy list of features. It became very popular!

He considered reimplementing it properly, and it became Flask.

Other "jokes":

- **Unix** — contrast to bloated Multics system
- **Gmail** — 1 GB free storage announced on 1st April

Web development is hard to learn

To write a Web application, you need to understand so many languages!

- **HTML** — markup for Web pages;
- **CSS** — layout and styling for Web pages;
- **SQL** — database querying;
- **JavaScript** — client-side computation/interactivity;
- **Python** — server-side computation/interactivity.
- **Linux shell/Bash?** — server administration;

NodeJS lets you use JavaScript on server side too. Less to learn, and can more easily switch computation between client and server.

Web development can be fiddly and boring

90% of software development (by number of lines of code written) is probably: **database + plumbing + UI**

- MariaDB/PostgreSQL/Sqlite3 — database
- Flask — plumbing
- HTML + CSS + Jinja — UI

A database is a sensible way to store persistent information used by your Web application.

Used properly, it ensures consistency if a request crashes or hangs.

To backup/restore the website, just backup/restore the database.

Flask does not mandate a database, but you can use Sqlite3 when starting out.

Demo: Hello World in Flask

- `hello6.py` — Jinja templates to separate layout/logic
- `hello7.py` — multiple names, control flow in Jinja

Login and sessions

Often, we want the Web server to "remember" a user for a while, at least until he has finished using the website. This sequence of interactions is called a **session**.

Many websites support login. Then user state can be stored in a database alongside other account information, but...

You still need a way of tracking which user has logged in.

There are various ways of tracking this. One option is for the server to send the browser a **session cookie**: a random ID it sends back with every subsequent request.

There are security risks! Session cookies could be intercepted by malware.

Flask's default solution: Encrypt the session cookie with a key secret to the Web server. Remembering data can be delegatd to the browser this way too. **But always validate!**

Deploying Flask applications

Start from the Flask config in the tutorial, not my "Hello World": it will make it easier to configure.

Don't use the development server on a public network.

If you have your own server, you can set up a Web server (for example, Lighttpd) to use your Flask application to process requests.

Many hosting companies support Python Flask applications, for example:

https://www.mythic-beasts.com/support/hosting/python

Testing

Testing Web applications is hard!

Flask has an integrated **debug mode**, which reloads your application when the code changes and shows errors when they occur.

Do not use it on a public network!

Flask also supports logging.

Flask has an API for writing unit tests with dummy requests.

Flask Documentation

The official documentation is good:

https://flask.palletsprojects.com/en/stable/

Start with Tutorial and/or Quickstart. If you want suggestions about how to handle something complex, look at "Patterns for Flask".

Example: Uploading files:

https://flask.palletsprojects.com/en/stable/patterns/fileuploads/

Very popular online "Flask Mega-Tutorial":

https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

DEPARTMENT OF COMPUTER SCIENCE

University of Reading

Next time: Multithreading and Networking