# Week 1, Lecture 2

COMP6[48]43

Hamish Cox

Feb 19th, 2025

# Lecture 1 Recap

Yesterday we built a basic server for HTTP, a text-based protocol for fetching content from and interacting with a server. The protocol looks like this:

```
GET /hello.txt HTTP/1.1
Host: localhost
```

Request from Client to Server

→

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 12

Hello world!
```

Response from Server to Client

# Formatting Content with HTML

This is boring! I want

## Headings

and **bold text** and *italics*

(which this slide theme seems to show as orange???)

# Formatting Content with HTML

```html
<html>
    <body>
        <h1>My Cool Webpage</h1>
        <h2>Introduction</h2>
        <p>Interesting content ... </p>
        <h2>Second Section</h2>
        <p>
            Even <b>more</b>
            interesting content ...
        </p>
    </body>
</html>
```

→

# My Cool Webpage

## Introduction

Interesting content...

## Second Section

Even **more** interesting content...

# HTML and Forms

We have a way to GET a document. We can format it and make it pretty using HTML.

What if I want to send information back to the server and **interact** with it?

# Forms

```
<form>
    <label for="search">
        Search
    </label>
    <input name="search">
    <br><br>
    <input type="submit"
value="Search">
</form>
```

→


What happens when you click on the submit button?

# Receiving Form Data on the Server

What gets sent to the server if we enter `test` as our search term?

```
GET /search.html?search=test HTTP/1.1
Host: localhost
```

What you see after the `?` is called the query string. It encodes “query parameters” (sometimes called search parameters, even when you aren’t using them for a search feature). Here’s another example:

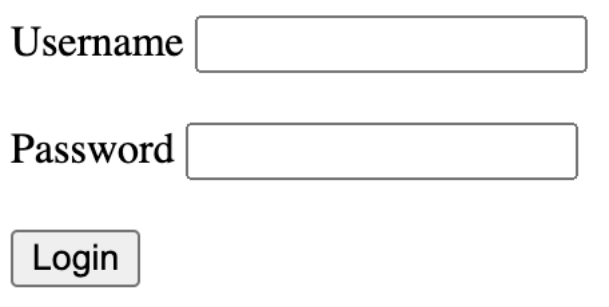
```
?search=test&sort_by=created_at&sort_dir=desc
```

# But now everything is stuck in the path/URL?

Ok, what if we allow body content in the request as well? We need our GET requests to work as-is, so we'll change it from "GET" to "POST" to indicate we want to send content.

```html
<form method="POST">
    <label for="username">
        Username
    </label>
    <input name="username">
    <br><br>
    <label for="password">
        Password
    </label>
    <input name="password" type="password">
    <br><br>
    <input type="submit" value="Login">
</form>
```

→



# POSTing Form Data

When we send form data, what does that look like in a request?

```
POST /login HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded

username=HamishWHC&password=Password123
```

If we change the content type, we can send even more stuff (e.g. JSON, files, etc.) but this is (usually) what forms send, so we'll stick to this for now.

# Encoding...

Now we have communication with the server, but let's quickly test some inputs and see what happens...

| | | |
|---|---|---|
| test search | → | `?search=test+search` |
| Smith & Co. | | `?search=Smith+%26+Co.` |
| O'Reilly | | `?search=O%27Reilly` |
| someone@example.com | | `?search=someone%40example.com` |

# URL Encoding

This encoding is simply converting the character to ASCII in hex (e.g. `/` -> `2F`) and throwing a percent sign in front.

Shall we go implement this?

No I'd really rather not... thankfully many other people already have!

# Flask

So what is Flask actually doing?

1. It parses the HTTP request just like we did (except it does it safer and more efficiently).
2. It uses the path and method to pick one of our handler functions to run.
3. It runs our code and sends back the response we return.

# What else?

Just about every language has frameworks for running HTTP servers.

Often, they just tell you what's running in the `Server` header:

```
HTTP/1.1 204 No Content
Server: nginx
```

Otherwise, they may have distinctive tells, like all the paths in the applications looking like `/something.php`

Certain languages/frameworks/libraries tend towards certain types of vulnerability, so this kind of information gathering is very helpful.

# Webservers

In addition to frameworks/libraries for particular languages, you'll also see more generic 'webserver' software.

Some examples:

- nginx (pronounced "engine x")
- Apache
- Caddy (❤️)

These are often used to simply host files or for setups such as redirections, caching and load balancing.

We'll discuss more of these misconfigurations in Topic 6, but you'll encounter this software throughout the term.