EECS 485: Introduction to Web Systems Notes

Noah Peters

May 8, 2023

Abstract

Notes for Dr. Andrew DeOrio's EECS 485: Introduction to Web Systems. Note template by Pingbang Hu.

Contents

Chapter 1

Introduction to Web Systems

Lecture 1: Introduction

1.1 The Request Response Cycle

The **request response cycle** is how two computers communicate with each other on the web.

> **Definition 1.1.1** (Request Response Cycle)**.** The request response cycle is a basic web communication protocol consisting of the following basic steps:
>
> 1. A client requests some data
> 2. A server responds to the request

A server may respond with different kinds of files. Common examples include:

- HTML
- CSS
- JavaScript

1.2 Static Pages

A **static page** is only HTML/CSS. There is no programming language on the server, and so you receive the same content upon every load of the webpage. It will send only files that do not change, such as HTML, CSS, or images.

When running a static file server, the server does the following:

1. Waits for connection from client
2. Receives a request
3. Looks in content directory, computes files name
4. Loads file from disk
5. Writes response to client: 200 OK, followed by the bytes of the file

Lecture 2: Static Pages

1.2.1 Document Object Model (DOM)

HTML tags form a tree called the **document object model (DOM)**:

```
<html>
        <head></head>
        <body>
        ...
        </body>
</html>
```

The DOM is a data structure built from the HTML. In the DOM, everything is a **node**. All HTML elements are element nodes, with text inside HTML being text nodes.

1.2.2 HTML5

HTML5 merges the advancements made during HTML's lifespan, specifically those from XHTML and browser-specific extensions of HTML.

1.3 URLs and HTTP

We can download a webpage using **curl**:

```
$ curl --verbose http://cse.eecs.umich.edu/ > index.html
```

or more generally:

```
$ curl --verbose protocol://server:port/path?query#fragment > out.html
```

1.3.1 URLs

We can observe each part of the URL in more detail:

- `protocol` - tells the server what protocol to use, i.e. what "language" to speak
- `server` - locates the machine we want to talk to
  - DNS lookup translates server name into an IP address (e.g. localhost and 127.0.0.1)
- `port` - used to identify a specific service
  - we can check what ports are open using `$ nmap cse.eecs.umich.edu`
- `path` - a file name relative to the server root
  - default is `/index.html`
- `query` - a general-purpose string of parameters that the server can use
- `fragment` - identified at the client, ignored by server; think page navigation

1.3.2 HTTP

Hypertext Transfer Protocol essentially reflects the request/response protocol outlined prior with the additional constraints that a **server cannot open connection to the client** and the protocol is completely **stateless**.

1.4 Requests

Definition 1.4.1 (`GET` Request)**.** A `GET` request is simply the prompting of a server for data. Requests using `GET` should only retrieve data.

1.4.1 General Request-Response Structure

So far we've seen what a `GET` request looks like. Now we will look at the other two types of requests: `HEAD` and `POST`. Take the following example server request-response:

```
$ curl --verbose http://cse.eecs.umich.edu/ > index.html
* Connected to cse.eecs.umich.edu
(141.212.113.143) port 80 (#0)
> GET / HTTP/1.1
> Host: cse.eecs.umich.edu
> User-Agent: curl/7.54.0
> Accept: */*
```

The above request to `cse.eecs.umich.edu` contains the following headers:

- `Host`: distinguishes between DNS names sharing a single IP address
- `User-Agent`: which browser is making the request
- `Accept`: which content (i.e. file) types the client will accept

The response from the server will start with a status code:

- 1XX: informational
- 2XX: successful
- 3XX: redirection error
- 4XX: server error

Further, the response will have accompanying headers, most of which are optional. Take the following example:

```
$ curl --verbose http://cse.eecs.umich.edu/
* Connected to cse.eecs.umich.edu
> GET / HTTP/1.1
...
< HTTP/1.1 200 OK
< Date: Tue, 12 Sep 2017 20:04:20 GMT
< Server: Apache/2.2.15 (Red Hat)
< Accept-Ranges: bytes
< Connection: close
< Transfer-Encoding: chunked
< Content-Type: text/html; charset=UTF-8
```

Where the `Content-Type` describes the "file" type and encoding (e.g. `text/html` or `image/png`). The type and encoding are known as the **MIME type**. Next is the character encoding, which is often just `UTF-8`, which corresponds to Unicode, which covers almost all of the characters and symbols in the world. It shares the first 128 values with ASCII, although unlike ASCII, Unicode is expandable.

1.4.2 `HEAD` Requests

`HEAD` requests are useful for checking if a page has changed or for caching stuff that doesn't change much and updating it. Along with other typical headers, it returns a `Last-Modified` header that may look like the following:

```
< Last-Modified: Thu, 28 May 2015 13:40:55 GMT
```

Definition 1.4.2 (`HEAD` Request). A `HEAD` request acts just like a `GET` request, but doesn't return the body, only the headers.

1.4.3 `POST` Request

Definition 1.4.3 (`POST` Request). A `POST` request sends data from the client to the server.

1.5 HTTP/1.0 vs HTTP/1.1 vs HTTP/2

The largest difference between HTTP/1.0 and HTTP/1.1 is the number of transmission control protocols that can occur simultaneously. Take the following HTML, for example:

```
<html>
  <body>
    <p>Block M<p>
    <img src="http://cse.eecs.umich.edu/eecs/images/image1.png">
    <img src="http://cse.eecs.umich.edu/eecs/images/image2.png">
  </body>
</html>
```

In the case of:

- HTTP/1.0: we will need to (1) set up a connection to load HTML, (2) set up another connection and load image1.png, and then (3) set up another connection to load image2.png; *this uses 3 connections*
- HTTP/1.1: we can use one connection to load the HTML, then load image1, then load image2 in a sequence; *this uses 1 connection*

HTTP/2 has the same methods, status codes, etc. as HTTP/1.1. One new feature is the idea of **server push**, where the server supplies data it knows a web browser will likely need to render a web page *without waiting* for the browser to examine the first response.

1.6 Server-Side Dynamic Pages

Lecture 3: Server-Side Dynamic Pages

So far we've seen static pages. For static pages, static content is the same every time. Pages rarely change via a manual process. You must *manually* update data, templates, and everyone accessing the website will get the same content until the next manual update. In other words, *generation of content is not specific to each request.* Instead, we can use **server-side dynamic pages**.

Definition 1.6.1 (Server-Side Dynamic Page). Webpages that are generated on the fly from a database. For server-side dynamic pages, the response is the output of a function.

1. Client makes a request
2. Server executes a function (output is usually HTML)
3. Server response is the output of the function

This highlights the principle of **data/computation duality**, which essentially notes that we can substitute *data* for *deterministic computation* and visa versa.

The process of server-side dynamic page generation occurs as follows. Note that the *generation of content is specific to each request.*

i. Client specifies a URL

ii. Server runs a function based on specified URL

iii. Function issues SQL queries to database to get relevant state

iv. Python object (e.g. a `dict`) is populated

v. Template is rendered using the object

vi. Rendered template is returned

1.6.1 URL Routing

The server utilizes **URL routing** to decide which function to call (and with what inputs) given a URL. This leads to a key distinction between static and dynamic pages:

- Static pages: *1 URL* maps to *1 file*
- Dynamic pages: *many URLs* map to *1 function*

There are a variety of ways to implement URL routing, although a common approach is to simply create a table that maps URLs to function-references. In Flask, a Python decorator is used to describe routing.

Definition 1.6.2 (Register Pattern). Creating a table of functions by "registering" them using some technique (e.g. with decorators in Flask).

1.7 Sessions

Lecture 4: Sessions

We now shift our goal towards maintaining state with stateless HTTP. We do this through the use of **sessions**.

Definition 1.7.1 (Session). A group of user interactions with a website that take place within a given time frame.

Note that we *could* have built state into HTTP, however this would clash with the principle of the web (and of computing) to build in layers, with each layer as simple as possible. Instead, we build state *on top of* HTTP. This "session protocol" is implemented at the application layer (i.e. Flask, React, etc. will handle sessions).

1.7.1 Server Session Model

Sessions are explicitly opened and closed **by the server**. We now address the details of sessions:

- Session data storage: best practice is to store a small amount of data to identify the session (e.g. a username, session ID, etc.)
- Session length: we can use timeouts to close a session

We can then link a session to a user by observing their session data (e.g. a username/password combination). Take a user trying to access `https://mail.google.com`, for example, we can link the user to their session in the following process:

1. Client requests `https://mail.google.com`
2. Server responds with redirect to `https://acccounts.google.com/signin/v2/identifier?continue=https%3A%2F%2Fmail.google.com%2Fmail%2F`
3. Client sends request with username and password
4. Server tests and responds with redirect to `https://mail.google.com`

1.8 Cookies

Cookies are simply the implementation of sessions.

Definition 1.8.1 (Cookies). Cookies are small files on client machines that carry state between HTTP requests. They're composed of key/value pairs.

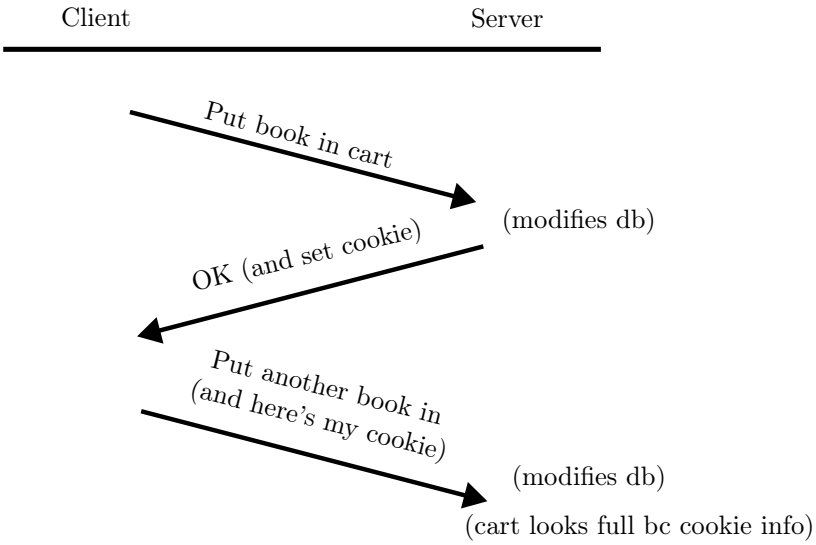

Figure 1.1: Basic cookie transmission

Cookies often contain the following:

- Name
- Value
- Domain (used by browser)
- Path
- Expiration
- Secure

The browser ensures that cookies are only over-writable by the same domain and path.

1.8.1 Cookie Transfer

Cookie transfer simply involves the server setting the cookie and the client sending the cookie, but never editing it. Either side can delete/ignore cookies. Because cookies are transferred as part of HTTP headers, cookies add to HTTP overhead.

Cookie leaks can be prevented by transmitting them only over HTTPS. However, if the client has a copy of cookies set by the server, then the client can manipulate the server. This can be prevented by encrypting cookie content such that *only the server can decrypt*.

1.8.2 Third-Party Cookies

A page may contain objects from many sources. These 3rd-party objects set and get cookies.

Definition 1.8.2 (First-Party Cookie)**.** A cookie where the domain is the same as the domain of the page you are on.

Definition 1.8.3 (Third-Part Cookie)**.** A cookie where the domain is different from the domain of the page you are on.

For an example, consider accessing nytimes.com via browser:

i. Browser issues `GET` request to nytimes.com; this includes nytimes.com cookies

ii. Browser receives HTML for nytimes.com; HTML contains some JavaScript

iii. Browser executes JavaScript included by nytimes.com; JS code initiates request to doubleclick.net

iv. Browser issues `GET` request to doubleclick.net; includes doubleclick.net cookie, appends current location (nytimes.com) to the URL

v. Now doubleclick.net knows you visited nytimes.com

Alternative methods exist to uniquely identify users, such as **browser fingerprinting**.