Introduction to HTTP

Jerry Cain
CS 106AX
November 30, 2022

The Client-Server Model

1. The user launches a web browser.
2. The user requests a web page.
3. The browser sends a request for the page.
4. The server sends back the requested HTML.
5. The browser interprets the HTML and renders it in the browser.

The Client-Server Model, Take II

1. The user launches a web browser.
2. The user requests a document via some URL.
3. The browser sends a request for the document at that location.
4. The server synthesizes the requested document and replies with it.
5. The browser receives the document and renders it.

Establishing the Connection

• Whenever a web browser needs some resource, it opens a network connection to the server where that resource lives.
• Opening a connection to a server like www.stanford.edu or maps.google.com is akin to making a phone call, where the IP address of the host serves functions as the phone number.
• A port number—almost always the number 80 for web servers—is used to identify the server process that’s listening for incoming web requests. Other services (e.g., email) are managed by applications listening to different port numbers.
• Typically, web servers listen to port 80, secure web servers listen to port 443, email servers listen to port 993, etc.
• Most port numbers between 1 and 1024 have been assigned to well known services. Those that haven’t are reserved for services that haven’t been invented yet.

The Hypertext Transfer Protocol

• Once the connection has been established—that is, the client has initiated the connection and the server has accepted it—the two endpoints are free to exchange data, provided the exchange respects the Hypertext Transfer Protocol, or HTTP.
• In a standard exchange, the client sends over an HTTP-compliant request. The server ingests the request, processes it, and sends back an HTTP-compliant response.
• In some cases, the response is little more than the contents of a static file, like index.html or JSGraphics.js.
• Other times, the server programmatically synthesizes a response and sends that back as if the response payload were locally stored (e.g., your Google search result).

HTTP GET Request Structure

• When we enter http://numbersapi.com/156?json=true, the browser opens a connection to numbersapi.com, port 80, and sends a request that looks like this:

GET /156?json=true HTTP/1.1
Host: numbersapi.com
Cache-Control: max-age=0
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,fr;q=0.8
other similarly structured request headers

• The first line of the request line contains three tokens: the method, the request path, and the protocol version.
• The remaining lines are response headers—think of them as key/value pairs of a dictionary—the further inform the server how to respond.
• All GET requests are terminated by a single blank line.
HTTP POST Request Structure

- Whenever the browser needs to upload new data to the server, the method will be POST instead of GET, as with:
  
  ```
  POST http://localhost:8000/factor.py?number=156
  ```

- The Content-Length request header must be present, so the server knows exactly how many bytes follow the blank line.
- The material after that blank line is referred to as the request payload, and it can only be present for POST requests.
- The most common actions that result in POST requests are ones like secure logins, payment submissions, photo and video uploads, and so forth.

Implementing HTTP Servers

- Python provides a generic HTTP server implementation that allows us to serve static resources (images, JavaScript files, etc.) and run Python scripts to dynamically generate responses.
- The `SimpleHTTPServer` and `CGIHTTPRequestHandler` classes are built-in, and the above program can be run as is. Doing so creates a server that listens for incoming requests on port 8000.
- Any request path beginning with `/scripts/` invokes a specific Python program that knows to programatically synthesize a response and send it back to the client.

Example: Prime Factorization Service

- Let’s implement a server endpoint called `factor.py` that assumes a single query string parameter (often called a `query parameter`) called `number` and responds with an HTTP response payload that looks like this:

  ```
  Number: 156
  Factors: [2, 3, 3, 5, 107]
  ```

  The implementation can assume the existence of a function called `extractRequestParameter` that returns the string value associated with the provided key in the query string. Assume the response is formatted as JSON so the client can easily parse the response.

Example: Prime Factorization Service

- The script assumes the full HTTP request has already been ingested. In fact, that’s what the `HTTPServer` class does.
- The only thing we need from the request is the value attached to `number` in the query string. We rely on a function we wrote for you—`extractRequestParameter`—to do that.
- Provided the `number` parameter is well-formed—that is, it’s truly a number and it’s positive (so it can be factored)—we tap a standard Python function that computes the prime factorization, places all factors in a list, and returns it.
- Once the response dictionary has been assembled, we invoke `json.dumps` to generate its JSON serialization. The script publishes two response headers (Content-Length is mandatory, the other is optional if the client knows to expect JSON), followed by a blank line, followed by the payload.