Problem Set 6: Web Server

This is CS50. Harvard University.

Table of Contents

Objectives

- Become familiar with HTTP.
- Apply familiar techniques in unfamiliar contexts.
- Transition from C to web programming.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with

classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on quizzes is not permitted at all. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from the course's heads. Acts considered not reasonable by the course are handled harshly. If the course refers some matter for disciplinary action and the outcome is punitive, the course reserves the right to impose local sanctions on top of that outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

If you commit some act that is not reasonable but bring it to the attention of the course's heads within 72 hours, the course may impose local sanctions that may include an unsatisfactory or failing grade for work submitted, but the course will not refer the matter for further disciplinary action except in cases of repeated acts.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).
- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code at office hours, elsewhere, or even online, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past semesters' quizzes and solutions thereto.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.

- Sharing snippets of your own code online so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate a solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.

- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Getting Ready

First, join David (hey, that's me) for a tour of HTTP, the "protocol" via which web browsers and web servers communicate.

Next, consider reviewing some of these examples from Week 7, via which we introduced HTML, the language in which web pages are written.

And also some of these examples, via which we introduced CSS, the language with which web pages can be stylized.

Next, consider reviewing some of these examples, via which we introduced HTML forms, which we used to submit GET queries to Google.

For another perspective altogether, join Daven for a tour of HTML too. Don’t miss the bloopers at the end!

Finally, join Joseph (and Rob) for a closer look at CSS.

Getting Started

Log into CS50 IDE (https://cs50.io/) and, in a terminal window, execute

```
update50
```

to ensure that your workspace is up-to-date!

Like Problem Set 5, this problem set comes with some distribution code that you'll need to download before getting started. Go ahead and execute

```
cd ~/workspace
```

in order to navigate to your `~/workspace` directory. Then execute

```
wget http://cdn.cs50.net/2015/fall/psets/6/pset6/pset6.zip
```

in order to download a ZIP (i.e., compressed version) of this problem set's distro. If you then execute

```
ls
```

you should see that you now have a file called `pset6.zip` in your `~/workspace` directory. Unzip it by executing the below.

```
unzip pset6.zip
```

If you again execute

```
ls
```

you should see that you now also have a `pset6` directory. You're now welcome to delete the ZIP file with the below.

```
rm -f pset6.zip
```

Now dive into that `pset6` directory by executing the below.

```
cd pset6
```

Now execute

```
tree
```

(which is a hierarchical, recursive variant of `ls`), and you should see that the directory contains the below.

```
.
├── Makefile
├── public
│   ├── cat.html
│   ├── cat.jpg
│   ├── favicon.ico
│   ├── hello.html
│   ├── hello.php
│   └── test
│       └── index.html
└── server.c
```

Dang it, still C. But some other stuff too!

Go ahead and take a look at `cat.html`. Pretty simple, right? Looks like it has an `img` tag, the value of whose `src` attribute is `cat.jpg`.

Next, take a look at `hello.html`. Notice how it has a `form` that's configured to submit via GET a `text` field called `name` to `hello.php`. Make sense? If not, try taking another look at the walkthrough for `search-0.html` (https://youtu.be/RQ2_TlXBo00) from Week 7!

Now take a look at `hello.php`. Notice how it's mostly HTML but inside its `body` is a bit of PHP code:

```
<?= htmlspecialchars($_GET["name"]) ?>
```

The `<?=` notation just means "echo the following value here". `htmlspecialchars`, meanwhile, is just an attrociously named function whose purpose in life is to ensure that special (even dangerous!) characters like `<` are properly "escaped" as HTML "entities." See http://php.net/manual/en/function.htmlspecialchars.php (http://php.net/manual/en/function.htmlspecialchars.php) for more details if curious. Anyhow, `$_GET` is a "superglobal" variable inside of which are any HTTP parameters that were passed via GET to `hello.php`. More specifically, it's an "associative array" (i.e., hash table) with keys and values. Per that HTML form in `hello.html`, one such key should be `name`! But more on all that in a bit.

Now the fun part. Open up `server.c`.

Yup. You guessed it. The challenge ahead is to implement your own web server that knows how to serve static content (i.e., files ending in `.html`, `.jpg`, et al.) and dynamic content (i.e., files ending in `.php`).

Want to try out the staff's solution before we dive into the distribution code? Execute the below to download the latest version of the staff's solution, as the version in CS50 IDE by default is outdated. Note that the `O` in `-O` is a capitalized letter `O`, not a zero.

```
sudo wget -O ~cs50/pset6/server http://cdn.cs50.net/2015/fall/psets/6/pset6/server
sudo chmod a+x ~cs50/pset6/server
```

Then execute the below to run the staff's implementation of `server`.

```
~cs50/pset6/server
```

You should see these instructions:

```
Usage: server [-p port] /path/to/root
```

Looks a bit complex, but that's just a conventional way of saying:

- This program's name is `server`.
- To specify a (TCP) port number on which `server` should listen for HTTP requests, include `-p` as a command-line argument, followed by (presumably) a number. The brackets imply that specifying a port is optional. (If you don't specify, the program will default to port 8080, which is required by CS50 IDE.)
- The last command-line argument to `server` should be the path to your server's "root" (the directory from which files will be served).

Let's try it out. Execute the below from within your own `~/workspace/pset6` directory so that the staff's solution uses your own copy of `public` as its root.

```
~cs50/pset6/server public
```

You should see output like the below.

```
Using /home/ubuntu/workspace/pset6/public for server's root
Listening on port 8080
```

Toward the top-right corner of CS50 IDE, meanwhile, you should see your workspace's "fully qualified domain name," an address of the form `ide50-username.cs50.io`, where `username` is your own username. Visit `https://ide50-username.cs50.io/` (where `username` is your own username) in another tab. You should see a "directory listing" (i.e., an unordered list) of everything that's in

`public`, yes? And if you click **cat.jpg**, you should see a happy cat?? If not, do just reach out to classmates or staff for a hand!

Incidentally, even though `server` is running on port 8080, CS50 IDE is "port-forwarding" port 80 (which, recall, is browsers' default) to 8080 for you. That's why you don't need to specify **8080** in the URL you just visited.

Anyhow, assuming you indeed saw a happy cat in that tab, you should also see

```
GET /cat.jpg HTTP/1.1
```

in your terminal window, which is the "request line" that your browser sent to the server (which is being outputted by `server` via `printf` for diagnostics' sake). Below that you should see all of the headers that your browser sent to `server` followed by

```
HTTP/1.1 200 OK
```

which is the server's response to the browser (which is also being outputted by `server` via `printf` for diagnostics' sake).

Next, just like I did in that short on HTTP, open up Chrome's developer tools, per the instructions at https://developer.chrome.com/devtools (https://developer.chrome.com/devtools). Then, once open, click the tools' **Network** tab, and then, while holding down Shift, reload the page.

Not only should you see Happy Cat again. You should also see the below in your terminal window.

```
GET /cat.jpg HTTP/1.1
HTTP/1.1 200 OK
```

You might also see the below.

```
GET /favicon.ico HTTP/1.1
HTTP/1.1 200 OK
```

What's going on if so? Well, by convention, a lot of websites have in their root directory a `favicon.ico` file, which is a tiny icon that's meant to be displayed a browser's address bar or tab. If you do see those lines in your terminal window, that just means Chrome is guessing that your server, too, might have `favicon.ico` file, which it does!

Here's a quick walkthrough if a demo might help.

Alright, now try visiting `https://ide50-username.cs50.io/cat.html`. (Note the `.html` instead of `.jpg` this time.) You should see Happy Cat again, possibly with a bit of a margin around him (simply because of Chrome's default CSS properties). If you look at the developer tools' **Network** tab (possibly after reloading, if they weren't still open), you should see that Chrome first requested `cat.html` followed by `cat.jpg`, since the latter, recall, was specified as the value of that `img` element's `src` attribute that we saw earlier in `cat.html`. To confirm as much, take a look at the developer tools' **Elements** tab, wherein you'll see a pretty-printed version of the HTML

in `cat.html`. You can even change it but only Chrome's in-memory copy thereof. To change the actual file, you'd need to do so in the usual way within CS50 IDE. Incidentally, you might find it interesting to tinker with the developer tools' **Styles** tab, too. Even though this page doesn't have any CSS of its own, you can see and change (temporarily) Chrome's default CSS properties via that tab.

Okay, one last test. Try visiting `https://ide50-username.cs50.io/hello.html`. Go ahead and input your name into the form and then submit it, as by clicking the button or hitting Enter. You should find yourself at a URL like `https://ide50-username.cs50.io/hello.php?name=Alice` (albeit with your name, not Alice's, unless your name is also Alice), where a personalized hello awaits! That's what we mean by "dynamic" content. By submitting that form, you provided input (i.e., your name) to the server, which then generated output just for you. (That input was in the form of an "HTTP parameter" called `name`, the value of which was your name.) Indeed, if you look at the page's source code (as via the developer tools' **Elements** tab), you'll see your name embedded within the HTML! By contrast, files like `cat.jpg` and `cat.html` (and even `hello.html`) are "static" content, since they're not dynamically generated.

Neat, eh?? Though odds are you'll find it easier to test your own code via a command line than with a browser. So let's show you one other technique.

Open up a second terminal window and position it alongside your first. In the first terminal window, execute

```
~cs50/pset6/server public
```

from within your own `~/workspace/pset6` directory, if the server isn't already running. Then, in the second terminal window, execute the below. (Note the `http://` this time instead of `https://`.)

```
curl -i http://localhost:8080/
```

If you haven't used `curl` before, it's a command-line program with which you can send HTTP requests (and more) to a server in order to see its responses. The `-i` flag tells `curl` to include responses' HTTP headers in the output. Odds are, whilst debugging your server, you'll find it more convenient (and revealing!) to see all of that via `curl` than by poking around Chrome's developer tools.

Incidentally, take care not to request `cat.jpg` (or any binary file) via `curl`, else you'll see quite a mess! (You're about to try, aren't you.)

Unfortunately, your own copy of `server.c` isn't quite so featureful as the staff's solution… yet! Let's dive into that distribution code. Let's start with a high-level overview.

And now a lower-level tour through the code.

server.c

Open up `server.c`, if not open already. Let's take a tour.

- Atop the file are a bunch of "feature test macro requirements" that allow us to use certain functions that are declared (conditionally) in the header files further below.
- Defined next are a few constants that specify limits on HTTP requests sizes. We've (arbitrarily) based their values on defaults used by Apache, a popular web server. See http://httpd.apache.org/docs/2.2/mod/core.html (http://httpd.apache.org/docs/2.2/mod/core.html) if curious.
- Defined next is `BYTES`, a constant the specifies how many bytes we'll eventually be reading into buffers at a time.
- Next are a bunch of header files, followed by a definition of `BYTE`, which we've indeed defined as an 8-bit `char`, followed by a bunch of prototypes.
- Finally, just above `main` are a just a few global variables.

main

Let's now walk through `main`.

- Atop `main` is an initialization of what appears to be a global variable called `errno`. In fact, `errno` is defined in `errno.h` and is used by quite a few functions to indicate (via an `int`), in cases of error, precisely which error has occurred. See `man errno` for more details.
- Shortly thereafter is a call to `getopt`, which is a function declared in `unistd.h` that makes it easier to parse command-line arguments. See `man 3 getopt` if curious. Notice how we use `getopt` (and some Boolean expressions) to ensure that `server` is used properly.
- Next notice the call to `start` (for which you may have noticed a prototype earlier). More on that later.
- Below that is a declaration of a `struct sigaction` via which we'll listen for `SIGINT` (i.e., control-c), calling `handler` (a function defined by us elsewhere in `server.c`) if heard.

- And then, after declaring some variables, `main` enters an infinite `while` loop.
  - Atop that loop, we first free any memory that might have been allocated by a previous iteration of the loop.
  - We then check whether we've been "signalled" via control-c to stop the server.
  - Thereafter, within an `if` statment, is a call to `connected`, which returns `true` if a client (e.g., a browser or even `curl`) has connected to the server.
  - After that is a call to `parse`, which parses a browser's HTTP request, storing its "absolute path" and "query" inside of two arrays that are passed into it by reference.
  - Next is a bunch of code that decodes that path (decoding any URL-encoded characters like `%20`) and "resolves" the path to a local path, figuring out exactly what file was requested on the server itself.
  - Below that, we ascertain whether that path leads to a directory or to a file and handle the request accordingly, ultimately calling `list`, `interpret`, or `transfer`.
    - For directories (that don't have an `index.php` or `index.html` file inside them), we call `list` in order to display the directory's contents.
    - For files ending in `.php` (whose "MIME type" is `text/x-php`), we call `interpret`.
    - For other (supported) files, we call `transfer`.

And that's it for `main`! Notice, though, that throughout `main` are a few uses of `continue`, the effect of which is to jump back to the start of that infinite loop. Just before `continue` in some cases, too, is a call to `error` (another function we wrote) with an HTTP status code. Together, those lines allow the server to handle and respond to errors just before returning its attention to new requests.

connected

Take a quick peek at `connected` below `main`. Don't fret if unsure how this function works, but do try to infer from the `man` pages for `memset` and `accept`!

error

Spend a bit more time looking through `error`, which is that function via which we respond to browsers with errors (e.g., 404). This function's a bit longer but perhaps has some more familiar constructs. Before forging ahead, be sure you're reasonably comfortable with how this function works. (If curious, we're using `log10` simply to figure out how many digits, and thus `char`s, `code` is.)

freedir

This function exists simply to facilitate freeing memory that's allocated by a function called `scandir` that we call in `list`.

handler

Thankfully, a short one! This function (called whenever a user hits control-c) essentially tells `main` to call `stop` by setting `signaled`, a global variable, to `true`.

htmlspecialchars

This function, named identically to that PHP function we saw earlier, escapes characters (e.g., `<` as `<`) that might otherwise "break" an HTML page. We call it from `list`, lest some file or directory we're listing have a "dangerous" character in its name.

indexes

Uh oh, forgot to implement this one. About that…

interpret

This function enables the server to interpret PHP files. It's a bit cryptic at first glance, but in a nutshell, all we're doing, upon receiving a request for, say, `hello.php`, is executing a line like

```
QUERY_STRING="name=Alice" REDIRECT_STATUS=200 SCRIPT_FILENAME=/home/ubuntu/workspace/pset6/pul
```

the effect of which is to pass the contents of `hello.php` to PHP's interpreter (i.e., `php-cgi`), with any HTTP parameters supplied via an "environment variable" called `QUERY_STRING`. Via `load` (a function we wrote), we then read the interpreter's output into memory (via `load`). And then we respond to the browser with (dynamically generated) output like:

```
HTTP/1.1 200 OK
X-Powered-By: PHP/5.5.9-1ubuntu4.12
Content-type: text/html

<!DOCTYPE html>

<html>
    <head>
        <title>hello</title>
    </head>
    <body>
                    hello, Alice
            </body>
</html>
```

Even though the PHP code in `hello.php` is pretty-printed, it's output isn't quite as pretty. (Take a look at `hello.php`. Can you deduce why?)

Odds are you're unfamiliar with `popen`. That function opens a "pipe" to a process (`php-cgi` in our case), which provides us with a `FILE` pointer via which we can read that process's standard output (as though it were an actual file).

Notice how this function calls `load`, though, in order to read the PHP interpreter's output into memory.

list

Ah, here's that function that generates a directory listing. Notice how much code it takes to generate HTML using C, thanks to requisite memory management. (No more, come PHP in Problem Set 7!)

load

Phew, a short one. Oh, wait.

lookup

Dang, another.

parse

Aaaaand, another. But the last of our TODOs!

reason

This function simply maps HTTP "status codes" (e.g., `200`) to "reason phrases" (e.g., `OK`).

redirect

Ah, neat, this function redirects a client to another location (i.e., URL) by sending a status code of `301` plus a `Location` header.

request

Ah, this one's a biggie. But worth reading through. When the server receives a request from a client, the server doesn't know in advance how many characters the request will comprise. And so this function iteratively reads bytes from the client, one buffer's worth at a time, calling `realloc` as needed to store the entire message (i.e., request).

Notice this function's use of pointers, dynamic memory allocation, pointer arithmetic, and more. All somewhat familiar by now, but definitely a lot of it all in once place! Do try to understand each and every line, if only for the practice. Ultimately, it keeps reading bytes from the client until it encounters \r\n\r\n (aka CRLF CRLF), which, according to HTTP's spec, marks the end of a request's headers.

If curious, know that `read` is quite like `fread` except that it reads from a "file descriptor" (i.e., an `int`) instead of from a `FILE` pointer (i.e., `FILE*`). See its `man` page for more.

Phew.

respond

It's this function that actually sends to a client an HTTP response, given a status code, heads, a body, and that body's length. For instance, it's this function that sends a response like the below.

```
HTTP/1.1 200 OK
X-Powered-By: PHP/5.5.9-1ubuntu4.12
Content-type: text/html

<!DOCTYPE html>

<html>
    <head>
        <title>hello</title>
    </head>
    <body>
                    hello, Alice
            </body>
</html>
```

Know that `dprintf` is quite like `printf` (or, really, `fprintf`) except that the former, like `read`, writes to a "file descriptor" instead of to a `FILE*`.

start

Here's that function that started it all (pun intended). Don't worry if (even with `man`) you don't understand all of its lines, particularly the networking code. But do keep in mind that `start` is the function that configures the server to listen for connections on a particular TCP port!

stop

And `stop` does the opposite, freeing all memory and ultimately compelling the server to exit, without even returning control to `main`.

transfer

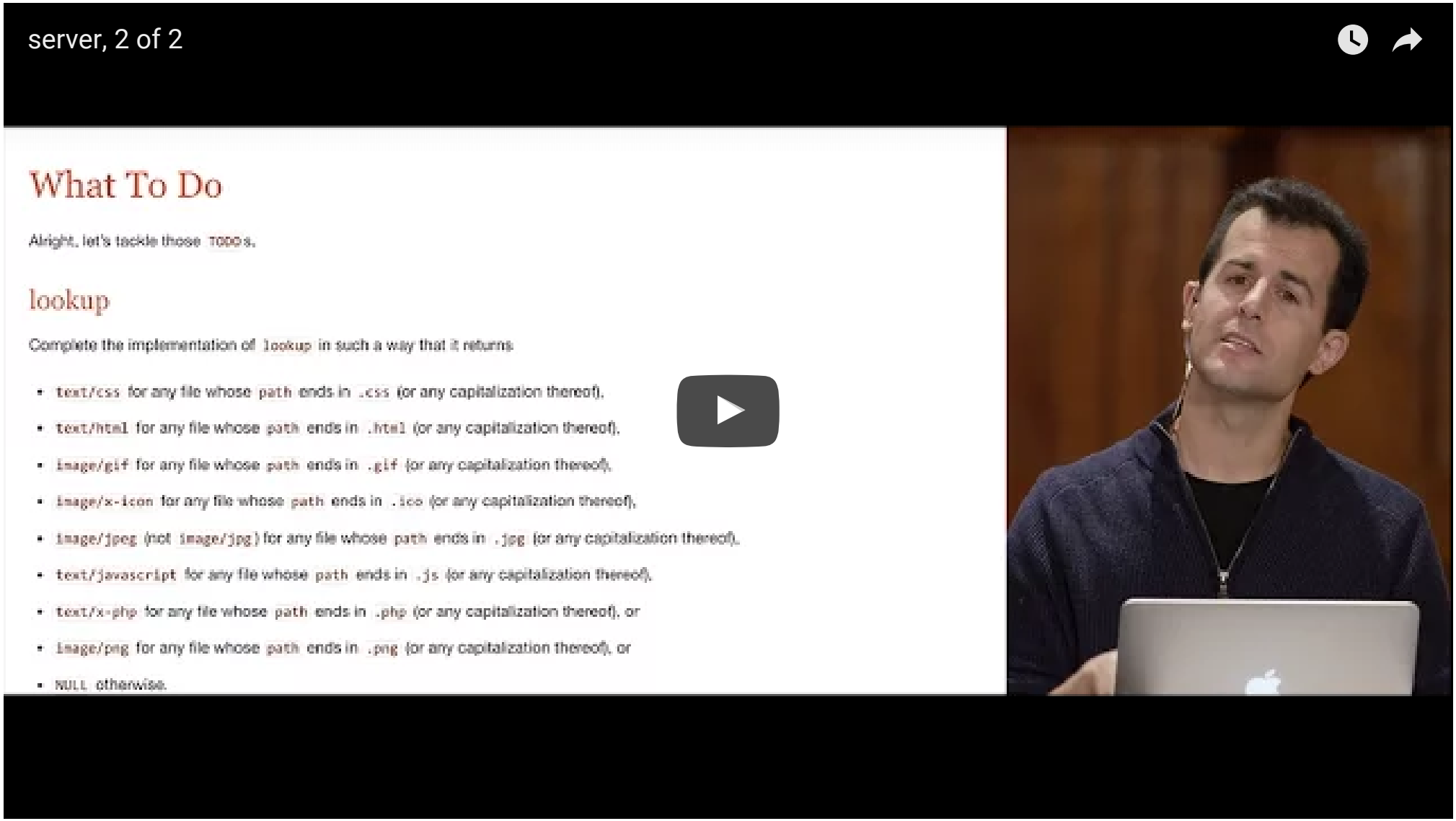
This function's purpose in life is to transfer a file from the server to a client. Whereas `interpret` handles dynamic content (generated by PHP scripts), `transfer` handles static content (e.g., JPEGs). Notice how this function calls `load` in order to read some file from disk.

urldecode

This function, also named after a PHP function, URL-decodes a string, converting special characters like `%20` back to their original values.

What To Do

Alright, let's tackle those `TODO`s.



lookup

Complete the implementation of `lookup` in such a way that it returns

- `text/css` for any file whose `path` ends in `.css` (or any capitalization thereof),
- `text/html` for any file whose `path` ends in `.html` (or any capitalization thereof),

- `image/gif` for any file whose `path` ends in `.gif` (or any capitalization thereof),
- `image/x-icon` for any file whose `path` ends in `.ico` (or any capitalization thereof),
- `image/jpeg` (not `image/jpg`) for any file whose `path` ends in `.jpg` (or any capitalization thereof),
- `text/javascript` for any file whose `path` ends in `.js` (or any capitalization thereof),
- `text/x-php` for any file whose `path` ends in `.php` (or any capitalization thereof), or
- `image/png` for any file whose `path` ends in `.png` (or any capitalization thereof), or
- `NULL` otherwise.

Odds are you'll find functions like `strcasecmp`, `strcpy`, and/or `strrchr` of help!

parse

Complete the implementation of `parse` in such a way that the function parses (i.e., iterates over) `line`, extracting its absolute-path and query and storing them at `abs_path` and `query`, respectively.

Here's how.

`abs_path`

Per 3.1.1 of http://tools.ietf.org/html/rfc7230 (http://tools.ietf.org/html/rfc7230), a `request-line` is defined as

```
method SP request-target SP HTTP-version CRLF
```

wherein `SP` represents a single space ( ) and `CRLF` represents `\r\n`. None of `method`, `request-target`, and `HTTP-version`, meanwhile, may contain `SP`.

Per 5.3 of the same RFC, `request-target`, meanwhile, can take several forms, the only one of which your server needs to support is

```
absolute-path [ "?" query ]
```

whereby `absolute-path` (which will not contain `?`) must start with `/` and might optionally be followed by a `?` followed by a `query`, which may not contain `"`.

Ensure that `request-line` (which is passed into `parse` as `line`) is consistent with these rules. If it is not, respond to the browser with **400 Bad Request** and return `false`.

Even if `request-line` is consistent with these rules,

- if `method` is not `GET`, respond to the browser with **405 Method Not Allowed** and return `false`;
- if `request-target` does not begin with `/`, respond to the browser with **501 Not Implemented** and return `false`;
- if `request-target` contains a `"`, respond to the browser with **400 Bad Request** and return `false`;
- if `HTTP-version` is not `HTTP/1.1`, respond to the browser with **505 HTTP Version Not Supported** and return `false`; or

Odds are you'll find functions like `strchr`, `strcpy`, `strncmp`, `strncpy`, and/or `strstr` of help!

If all is well, store `absolute-path` at the address in `abs_path` (which was also passed into `parse` as an argument). You may assume that the memory to which `abs_path` points will be at least of length `LimitRequestLine` + 1.

If you'd like to test your code with `check50`, you can execute the below.

```
check50 2015.fall.pset6.server1 server.c
```

`query`

Store at the address in `query` the `query` substring from `request-target`. If that substring is absent (even if a `?` is present), then `query` should be `""`, thereby consuming one byte, whereby `query[0]` is `'\0'`. You may assume that the memory to which `query` points will be at least of length `LimitRequestLine` + 1.

For instance, if `request-target` is `/hello.php` or `/hello.php?`, then `query` should have a value of `""`. And if `request-target` is `/hello.php?q=Alice`, then `query` should have a value of `q=Alice`.

Odds are you'll find functions like `strchr`, `strcpy`, `strncpy`, and/or `strstr` of help!

load

Complete the implementation of `load` in such a way that the function:

1. reads all available bytes from `file`,
2. stores those bytes contiguously in dynamically allocated memory on the heap,
3. stores the address of the first of those bytes in `*content`, and
4. stores the number of bytes in `*length`.

Note that `content` is a "pointer to a pointer" (i.e., `BYTE**`), which means that you can effectively "return" a `BYTE*` to whichever function calls `load` by dereferencing `content` and storing the address of a `BYTE` at `*content`. Meanwhile, `length` is a pointer (i.e., `size_t*`), which you can also dereference in order to "return" a `size_t` to whichever function calls `load` by dereferencing `length` and storing a number at `*length`.

indexes

Complete the implementation of `indexes` in such a way that the function, given a `/path/to/a/directory`, returns `/path/to/a/directory/index.php` if `index.php` actually exists therein, or `/path/to/a/directory/index.html` if `index.html` actually exists therein, or `NULL`. In the first of those cases, this function should dynamically allocate memory on the heap for the returned string.

If you'd like to test your code further with `check50` (once `parse`, `query`, `load`, and `indexes` are implemented), you can execute the below.

```
check50 2015.fall.pset6.server2 server.c
```

How to Submit

Step 1 of 2

1. When ready to submit, log into CS50 IDE (https://cs50.io/).
2. Toward CS50 IDE's top-left corner, within its "file browser" (not within a terminal window), control-click or right-click your `pset6` folder and then select **Download**. You should find that your browser has downloaded `pset6.tar.gz`, a "gzipped tarball" that's similar in spirit to a ZIP file.
3. In a separate tab or window, log into CS50 Submit (http://cs50.edx.org/submit), logging in if prompted.
4. Click **Submit** toward the window's top-left corner.
5. Under **Problem Set 6** on the screen that appears, click **Upload New Submission**.
6. On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
7. Navigate your way to `pset6.tar.gz`. Odds are it's in your **Downloads** folder or wherever your browser downloads files by default. Once you find `pset6.tar.gz`, click it once to select it, then click **Open** (or the like).
8. Click **Start upload** to upload all of your files at once to CS50's servers.
9. On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different

(or modified) files, simply return to CS50 Submit (http://cs50.edx.org/submit) and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

Head to http://cs50.edx.org/2016/psets/6/ (http://cs50.edx.org/2016/psets/6/) where a short form awaits. Once you have submitted that form (as well as your source code), you are done! If you end up resubmitting your files (per step 1 of 2), no need to resubmit the form.

This was Problem Set 6.