Problem Set 6: Web Server

This is CS50. Harvard University. Fall 2014.

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 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.

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 in CS50 Vault 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.

http://www.youtube.com/watch?v=hU4XuBe50K4
Next, consider reviewing some of these examples, via which we introduced HTML forms, which we used to submit GET queries to Google.

http://www.youtube.com/watch?v=RQ2_TIXBo00

Getting Started

Start up your appliance and, upon reaching John Harvard's desktop, open a terminal window (remember how?) and execute

```
update50
```

to ensure that your appliance 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 ~/Dropbox
```

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

```
wget http://cdn.cs50.net/2014/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 `~/Dropbox` 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.

```
.
|-- public
|   |-- cat.html
|   |-- cat.jpg
|   |-- hello.html
|   `-- hello.php
`-- server.c
```

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

Go ahead and take a look at `cat.html` with `gedit`. 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` with `gedit`. 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` 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 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` with `gedit`.

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`).

But let's try out the staff's solution before we dive into the distribution code.

Open up a big terminal window (outside of `gedit`), as via **Menu > Accessories > Terminal Emulator**. Then execute the below.

```
~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, a random port will be chosen for you.) As an aside, most any number between 1024 (or, even more properly, 49152) and 65536 should be fine.

- 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 `~/Dropbox/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

```
Using /home/jharvard/solutions/2014/fall/psets/6/public for server's root
Listening on port #####
```

where `#####` is a random port number. Take note of your appliance's IP address, which should be a number of the form `#.#.#.#` in the appliance's bottom-right corner. Then, open up Chrome (inside of the appliance or on your own computer) and visit `http://#.#.#.#:#####/cat.jpg`, where, again, `#.#.#.#` is your appliance's IP address (not `#.#.#.#` literally) and `#####` is the random port number. For instance, if your appliance's IP address is `1.2.3.4` and the random port number is 1337, you should visit `http://1.2.3.4:1337/cat.jpg`.

Anyhow, you should see a happy cat?? In your terminal window, meanwhile, you should see

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

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

```
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).

Go ahead and stop the server by hitting control-c. Then, re-run it on, say, port `8080` which is a popular choice when running a webserver without "superuser" privileges. Remember how? Like this:

```
~cs50/pset6/server -p 8080 public
```

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. 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
404 Not Found
```

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. It doesn't (unless you put one to `public`), so not to worry.

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

http://www.youtube.com/watch?v=mRSxes0gwfw

Now try visiting `http://#.#.#.#:8080/cat.html`. 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 with, say, `gedit` in the usual way. Incidentally, you might find it interesting to tinker with the developer tools' **Styles** tab at right, 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 `http://#.#.#.#:8080/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 `http://#.#.#.#:8080/hello.php?name=Alice` (albeit with your name, not Alice's), 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.

Take care not to omit the port number from your URLs. Else your browser will end up talking to the appliance's own web server (Apache), and you'll see 403s and 404s that you're probably not expecting!

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 -p 8080 public
```

from within your own `~/Dropbox/pset6` directory, if the server isn't already running on that port. Then, in the second terminal window, execute

```
telnet #.#.#.# 8080
```

where `#.#.#.#` is, as before, your appliance's IP address. `telnet` is a program via which you can "communicate with another host" via textual commands. (Back in my day, too, it was how we checked email!) Anyhow, note that `telnet` expects a space (not a colon) between an address and the port.

You should find that `telnet` is now waiting for input. Go ahead and type

```
GET /cat.html HTTP/1.1
```

followed by Enter twice, which `telnet` will interpret as CRLF CRLF (i.e., `\r\n\r\n`).

You should see that the server's responded with a request line, some headers, a blank line, and some HTML?? Nice! Odds are, whilst debugging your server, you'll find it more convenient (and revealing!) to see all of that via `telnet` than by poking around Chrome's developer tools.

Incidentally, take care not to request `cat.jpg` (or any binary file) via `telnet`, 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.

http://www.youtube.com/watch?v=L7Xr5b1WwCY
And now a lower-level tour through the code.

server.c

Open up `server.c` with `gedit`, 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 if curious.
- Defined next is `OCTETS`, a constant the specifies how many "octets" we'll eventually be reading into buffers at a time. An "octet" is just another word for what we know as a "byte" (i.e., 8 bits); it's common nomenclature in the world of networking.
- Next are a bunch of header files, followed by a definition of `octet`, which we've indeed defined as 8 bits (i.e., a `char`), followed by some prototypes.
- Finally, just above `main` are a bunch of global variables. In general, it's wise to avoid global variables, but sometimes you can't. Indeed, as you'll soon see, this web server has a "signal handler" that requires that some of the server's state be global so that it can thoroughly free memory when a user stops the server by hitting control-c.

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

  ```
  signal(SIGINT, handler);
  ```

  which tells the program to listen for `SIGINT` (i.e., control-c) and call `handler` (a function defined by us elsewhere in `server.c`) if heard.
- And then `main` enters an infinite `while` loop.

  # Inside of that loop is a call to `reset` (another function defined by us) that takes care of freeing some memory and resetting some state in between HTTP requests.
- Thereafter is a call to `connected` within an `if` statement. Even though `connected` does return `true` or `false`, it does so by "blocking," waiting until a browser actually connects to the server before returning a value.
- After that is a call to `parse`, which parses a browser's HTTP request and returns, as one big string (i.e., `char*`), its request line and any headers, with each line terminated by CRLF (i.e., `\r\n`) instead of just LF (i.e., `\n`). Per the spec, aka "request for comments" (RFC), for HTTP, lines in HTTP messages should indeed be terminated by CRLF (just like text files in Windows). See https://tools.ietf.org/html/rfc7230 if curious. Note that `parse` ultimately stores the address of the parsed request in `request`, one of the global variables declared above `main`.

- Next is a bunch of code that parses that request, extracting only its request line.
- And then there's a `TODO` for you! But more on that in a bit.
- Below the `TODO` is a bunch of code we wrote that adds support for `.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/
jharvard/Dropbox/pset6/public/hello.php php-cgi
```

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 (storing the address thereof in a global variable called `body`, which was also declared above `main`). And then we respond to the browser with (dynamically generated) output like:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 127
X-Powered-By: PHP/5.5.9-1ubuntu4.4
Content-type: text/html

<!DOCTYPE html>

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

Perhaps new is our use of `popen`, `memmem`, `dprintf`, and `write`, but see each's `man` page for details! And know that `popen` 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). Note, too, that functions like `dprintf` and `write` (and `open` and `read` and others) use a "file descriptor" (i.e., an `int`) instead of a `FILE` pointer (i.e., `FILE*`) to refer to files. That aside, though,

they're not all that different from `fprintf` and `fwrite` (and `fopen` and `fread` and others). Do just consult `man` pages for which functions use which.

- Below that code for handling `.php` files is another `TODO` for you! But more on that in a bit too.

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, where `reset` is called. Just before `continue` in some cases, too, is a call to `error` (another function we wrote) with an HTTP status code. Together, those lines (along with others you'll write!) allow the server to handle and respond to errors just before returning its attention to (i.e., blocking for) 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.

load

Recall that `load` is the function that reads a file into memory (using a global variable called `file` that's declared above `main`). Read through this one slowly, taking note how it uses one buffer (`buffer`) to read a fixed number of octets at a time that it then copies into a second buffer (the address of which is stored in `body`, another global variable that's declared above `main`), which it re-sizes as needed via `realloc`. Long story (nay, long function!) short, this function reads the entirety of a file (or a pipe) into memory.

handler

Thankfully, a short one! Essentially, this function (called whenever a user hits control-c) simply calls `stop`.

lookup

Awww, just a `TODO`.

parse

Another biggee, albeit pretty similar to `load`! This one, though, reads an HTTP request from a network connection, ultimately throwing away one CRLF and the request's "body", if any, preserving only its request line and headers, the address of which is stored in that global variable called `request` that's declared above `main`. Notice, too, that this function calls `error` in a couple of cases.

reset

And now that function called `reset`, which gets called within that infinite loop within `main`. Notice how this function not only frees memory, it also resets some of those global variables to known values.

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`.

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` if `extension` is `css` (or any capitalization thereof),

- `text/html` if `extension` is `html` (or any capitalization thereof),
- `image/gif` if `extension` is `gif` (or any capitalization thereof),
- `image/x-icon` if `extension` is `ico` (or any capitalization thereof),
- `image/jpeg` (not `image/jpg`) if `extension` is `jpg` (or any capitalization thereof),
- `text/javascript` if `extension` is `js` (or any capitalization thereof),
- `image/png` if `extension` is `png` (or any capitalization thereof), or
- `NULL` otherwise.

main

validate request-line

Per 3.1.1 of 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 already stored for you in a variable called `line`) is consistent with these rules. If it is not, respond to the browser with **400 Bad Request**.

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

- if `method` is not `GET`, respond to the browser with **405 Method Not Allowed**;
- if `request-target` does not begin with `/`, respond to the browser with **501 Not Implemented**;

- if `request-target` contains a `"`, respond to the browser with **400 Bad Request**;
- if `HTTP-version` is not `HTTP/1.1`, respond to the browser with **505 HTTP Version Not Supported**; or
- if `absolute-path` does not contain a `.` (and thus a file extension), respond to the browser with **501 Not Implemented**.

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

extract query from request-target

Re-define a string called `query` that contains the `query` substring from `request-target`. If the latter is absent (even if a `?` is present), then the string should be `""`, thereby consuming one byte, whereby `query[0]` is `'\0'`.

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`.

It's probably best to ensure that `query` references memory on the stack, not the heap, so that you don't need to worry about freeing it in `stop` if the user hits control-c.

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

concatenate root and absolute-path

Re-define a string called `path` that contains the concatenation of `root` (a global variable) and `absolute-path`.

It's probably best to ensure that `path` references memory on the stack, not the heap, so that you don't need to worry about freeing it in `stop` if the user hits control-c.

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

ensure path exists

Ensure that `path` actually exists. Respond to the browser with **404 Not Found** if not!

Odds are you'll find functions like `access` and/or `stat` of help! For the latter, be sure to consult `man 2 stat` (i.e., chapter 2) rather than `man stat` (i.e., chapter 1).

ensure path is readable

Ensure that `path` is readable. Respond to the browser with **403 Forbidden** if not!

Odds are you'll find functions like `access` and/or `stat` of help!

extract path's extension

Re-define a string called `extension` that contains the file extension of the file specified by `path` (i.e., the substring after the last `.` in path).

It's probably best to ensure that `extension` references memory on the stack, not the heap, so that you don't need to worry about freeing it in `stop` if the user hits control-c.

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

respond to client

Complete the implementation of support for static content in such a way that, after loading a file into memory via `load`, `main` responds to a browser with these lines

```
HTTP/1.1 200 OK
Connection: close
Content-Length: %i
Content-Type: %s
```

each of which is terminated with CRLF (i.e., `\r\n`), which are followed after a single blank line (i.e., an additional CRLF) by the bytes of the file itself, whereby `%i` represents the file's size in bytes and `%s` represents the file's MIME type.

How to Submit

Step 1 of 2

When ready to submit, open up a Terminal window and navigate your way to `~/Dropbox`. Create a ZIP (i.e., compressed) file containing your entire `pset6` directory by executing the below. Incidentally, `-r` means "recursive," which in this case means to ZIP up everything inside of `pset6`, including any subdirectories (or even subsubdirectories!).

```
zip -r pset6.zip pset6
```

If you type `ls` thereafter, you should see that you have a new file called `pset6.zip` in `~/Dropbox`. (If you realize later that you need to make a change to some file and re-ZIP everything, you can delete the ZIP file you already made with `rm pset6.zip`, then create a new ZIP file as before.) * Once done creating your ZIP file, open up Chrome *inside* of the appliance (not on your own computer) and visit cs50.edx.org/submit[1], logging in if prompted. * Click **Submit** toward the window's top-left corner. * Under **Problem Set 6** on the screen that appears, click **Upload New Submission**. * On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear. * Navigate your way to `pset6.zip`, as by clicking **jharvard**, then double-clicking **Dropbox**. Once you find `pset6.zip`, click it once to select it, then click **Open**. * Click **Start upload** to upload your ZIP file to CS50's servers. * 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.edx.org/submit[2] 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/2015/psets/6/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 6.

[1] http://cs50.edx.org/submit
[2] http://cs50.edx.org/submit