Week 8

This is CS50. Harvard University. Fall 2014.

Cheng Gong

Table of Contents

PHP Basics 1
    Hash tables 6
PHP for the Web 10
    Forms, sessions, and emails 10
    MVC with require and functions 24

PHP Basics

- Last time we left off with a new language, HTML, that we used to structure and make webpages.
- We also used Cascading Style Sheets (CSS) to change the colors and positions of elements.
- Today we'll start to use a more powerful programming language for the web, PHP.
- One feature that a language needs, in order to dynamically generate webpages, is simply a `print` function.
- We've had this function in C and Scratch that can generate strings of text, but HTML is made up of manually written, static text.
- To dynamically generate HTML, we'll use PHP (and later JavaScript).
- Remember that C is a compiled language, which means we need to assemble it into object code before we run it. PHP is an interpreted languages, meaning our source code is provided as input to some program (an interpreter) that understands it and runs it as-is.
- Let's open `hello`[1]:

[1] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/hello

```
#!/usr/bin/env php ❶
<?php ❷

    printf("hello, world\n"); ❸

?> ❹
```

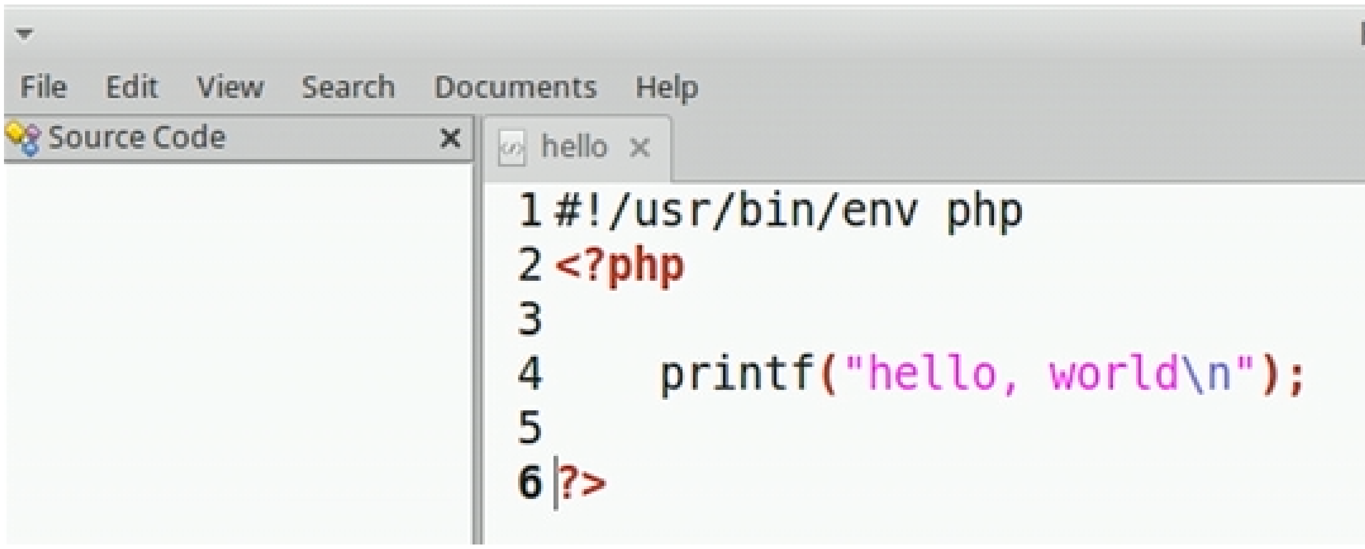
# In line 1, we are telling the operating system to find the interpreter for PHP, wherever it is.

# In line 2, we are writing a PHP start tag that means "this is the beginning of PHP code."

# In line 4, we can write a `printf` statement that looks suspiciously like C.

# And in line 6, we're simply telling the interpreter, "this is the end of our PHP code."

# When we save this file in `gedit`, it's smart enough to realize that we're writing a PHP file, based on that first line, and syntax-highlights (color-codes) the text accordingly:



```
1 #!/usr/bin/env php
2 <?php
3
4     printf("hello, world\n");
5
6 ?>
```

- Remember that we saved this to our home directory, so in our Terminal window we can try to run it:

```
jharvard@appliance (~): ./hello
-bash: ./hello: Permission denied
```

# We get this error, which we've gotten before, and know how to fix with `chmod`, that changes the mode of a file.

- Let's allow everyone to execute this file:

```
jharvard@appliance (~): chmod a+x hello
jharvard@appliance (~): ./hello
hello, world
```

And now it runs as we expected.

- But notice that we skipped the step of compiling it!
- Let's open `conditions-1`[2]:

```
#!/usr/bin/env php
<?php

    // ask user for an integer
    $n = readline("I'd like an integer please: "); ❶

    // analyze user's input
    if ($n > 0)
    {
        printf("You picked a positive number!\n");
    }
    else if ($n == 0)
    {
        printf("You picked zero!\n");
    }
    else
    {
        printf("You picked a negative number!\n");
    }

?>
```

# Notice that we have a few comments, but this looks oddly familiar like a program we wrote in Week 1, `conditions.c`.

[2] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/conditions-1

  # We ask the user for an integer, and tell them if it's positive, negative, or zero.

  # There are two differences with C. One is the `$` that is in front of all variables in PHP. The second is the `readline` function that we haven't seen in C, but is basically the PHP version of `GetString`.

- Let's go into our source code directory where we have `conditions-1` (which we saved in our `~/vhosts/` directory for convenience later):

```
jharvard@appliance (~/vhosts/localhost/public/src8m): ./conditions-1
-bash: ./conditions-1: Permission denied
jharvard@appliance (~/vhosts/localhost/public/src8m): chmod a+x
 conditions-1
jharvard@appliance (~/vhosts/localhost/public/src8m): ./conditions-1
I'd like an integer please: 50
You picked a positive number!
```

  # So first we had the same problem of not being able to run it, but we can fix that quickly with `chmod`, and now we see `conditions-1` runs like we'd expect.

- But notice that the top of the source code again had:

```
#!/usr/bin/env php
<?php
...
```

  which allows us to create what looks like a C program, but is executed not as compiled zeroes and ones. It's passed in to an interpreter that happens to be called `php`, the same as the language.

- We can open `return`[3]:

[3] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/return

```
#!/usr/bin/env php
<?php

    $x = 2;
    printf("x is now %d\n", $x);
    printf("Cubing...\n");
    $x = cube($x); ❶
    printf("Cubed!\n");
    printf("x is now %d\n", $x);

    /**
     * Cubes argument.
     */
    function cube($n) ❷
    {
        return $n * $n * $n;
    }

?>
```

# Notice that we've defined `cube`, a function referenced in line 7, in line 14, with no prototype. PHP (and a lot of modern languages) has a smarter interpreter than C's compiler in that it reads in the entire file that's passed in as input, before deciding that a function doesn't exist. So we can call a function and declare it later.

# Another difference compared to C is that there's no need to declare the types of variables, even though types do exist (as we'll see later).

# There's also no need to include as many libraries, since PHP includes many more functions by default.

# A final big difference is that there's no `main` function, and we can just start writing code to be executed. This will come in useful shortly when we want many entry points into our code, spread among multiple URLs and files, not just a single `main` function.

Hash tables

- We glance through the source code of `speller`[4], which is too long to include here, but the takeaway is that the program is a **porting**, a manual conversion, or "translation," if you will, from C to PHP.
  - David's taken care to convert line by line as similarly as possible, if you're interested in opening both side-by-side and comparing them to notice differences. (For your convenience, here's `speller.c`[5].)
- But let's see what we can do now that `speller` is in PHP. We can start a new file, `dictionary.php`, and save it to our `Desktop` folder for now:

```
<?php

    function check($word)
    {

    }

    function load($dictionary)
    {

    }

    function size()
    {

    }

    function unload()
    {

    }

?>
```

[4] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mispellings/speller
[5] http://cdn.cs50.net/2014/fall/psets/5/pset5/pset5/speller.c

  # These four functions were what we wrote in `dictionary.c` for Problem Set 5, but now we can rewrite them in PHP. Notice how we declare functions with `function`, and pass in variables not with a type but just the variable name with a `$` in front.

- We can declare a hash table in PHP by just adding this:

```
<?php

    $size = 0; ❶

    $table = []; ❷

    function check($word)
    {
...
```

  # Notice that line 5 was all it took to declare a hash table. In line 3, we create a variable to store the size of the hash table, just like we would in C.

- Now let's look at what we can do with `load`:

```
...
    function load($dictionary)
    {
        global $size;
        gloabl $table;

        foreach(file($dictionary) as $word)
        {
            $table[$word] = true;
        }
    }
...
```

  # Notice that we have to start by mentioning the global variables we want to use in our function (which we didn't have to in C).

  # Then we use a new `foreach` construct. `file($dictionary)` returns the dictionary file as an array of strings, which we can iterate over, and refer to each string as `$word`.

  - With each `$word`, we set the value in the hash table for that word to be `true`, signifying that it is a valid word.

- Let's add a few more features to our `load` function:

```
...
    function load($dictionary)
    {
        global $size;
        gloabl $table;

        foreach(file($dictionary) as $word)
        {
            $table[chop($word)] = true;
            $size++;
        }

        return true;
    }
...
```

  - `chop` removes the `\n` in each word (or trailing whitespace more generally), since each line that is passed in from `$dictionary` will have a `\n` that we don't want to keep.
  - We'll also keep track of the size, and then `return true`.

- The `size` function is pretty straightforward:

```
...
    function size()
    {
        global $size;
        return size;
    }
...
```

  - All we have to do is refer to the global variable `$size` and return it.

- How about `check`? We can do this:

```
...
    function check($word)
    {
        global $table;
        if (isset($table[strtolower($word)]))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
...
```

  - First, we want to have access to the global `$table` variable. Then, we `return true` if our hash table has a value set for `$word` (after we convert it to lowercase).
  - Notice that we can simplify this even more by returning the value of our condition, instead of returning `true` if our condition is `true` and `false` if our condition is `false`:

    ```
    ...
        function check($word)
        {
            global $table;
            return isset($table[strtolower($word)]);
        }
    ...
    ```

    - And this applies in C, too!

- And `unload` is already done, since we didn't allocate any memory ourselves! In C, we've had to `malloc` (and `free`) memory manually, but PHP manages it for us. One tradeoff here, however, is the speed of execution. David's implementation of `speller` in C takes 0.38 seconds to run in total, but the PHP version takes 0.93 seconds. This difference would surely add up, even though the PHP version was faster to implement.
- Another price of PHP is the amount of memory used. If we wanted the most performance, we'd probably want to avoid PHP. Lots of webservers are actually written

in C (like the one we'll make in Problem Set 6) in order to get the most performance and control over our server.

- Moreover, the hash table in PHP (actually called an **associative array**) is meant to be a generic tool, since whoever wrote that doesn't know what you might want to do with it, and thus probably has extra features to account for a wide range of situations.
- The conclusion is that there are many tools to use, and the important thing is deciding on the correct one to use!

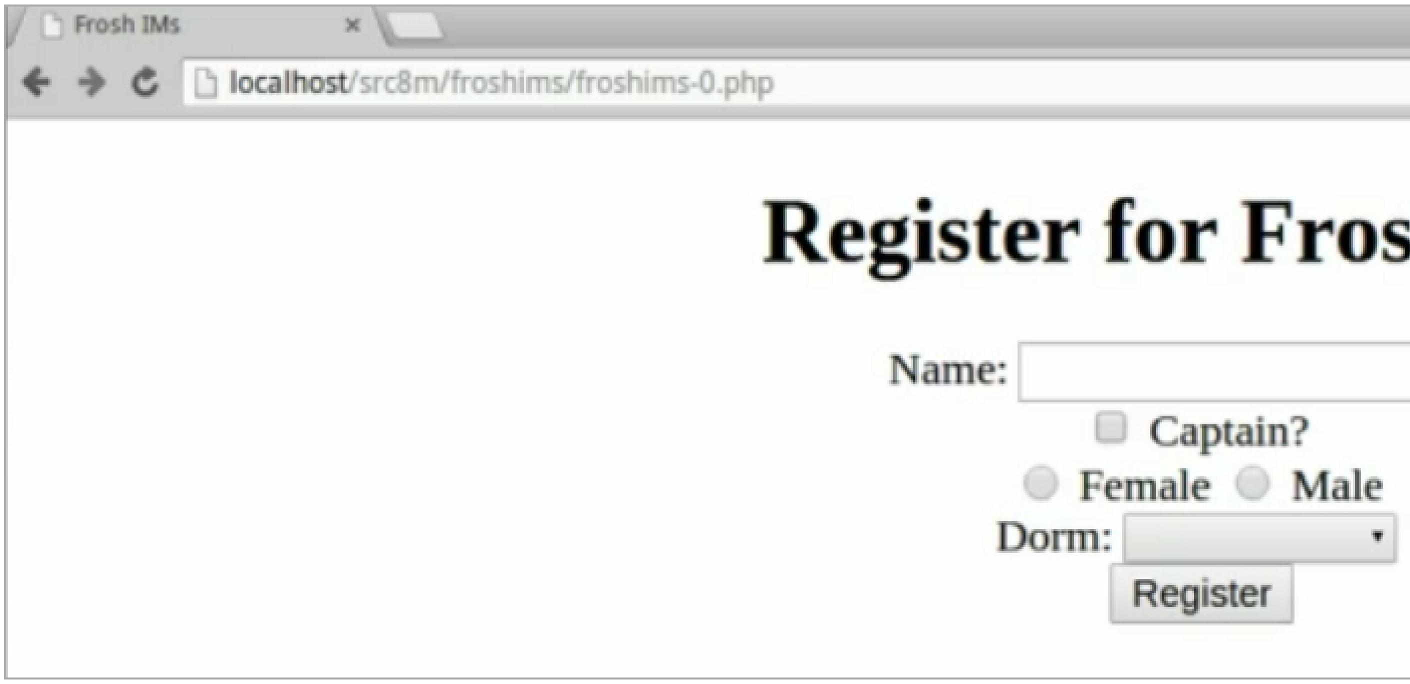
PHP for the Web

- So far, we've executed PHP files in the command line. But it's much more common to use PHP as files ending in `.php` to generate web content.

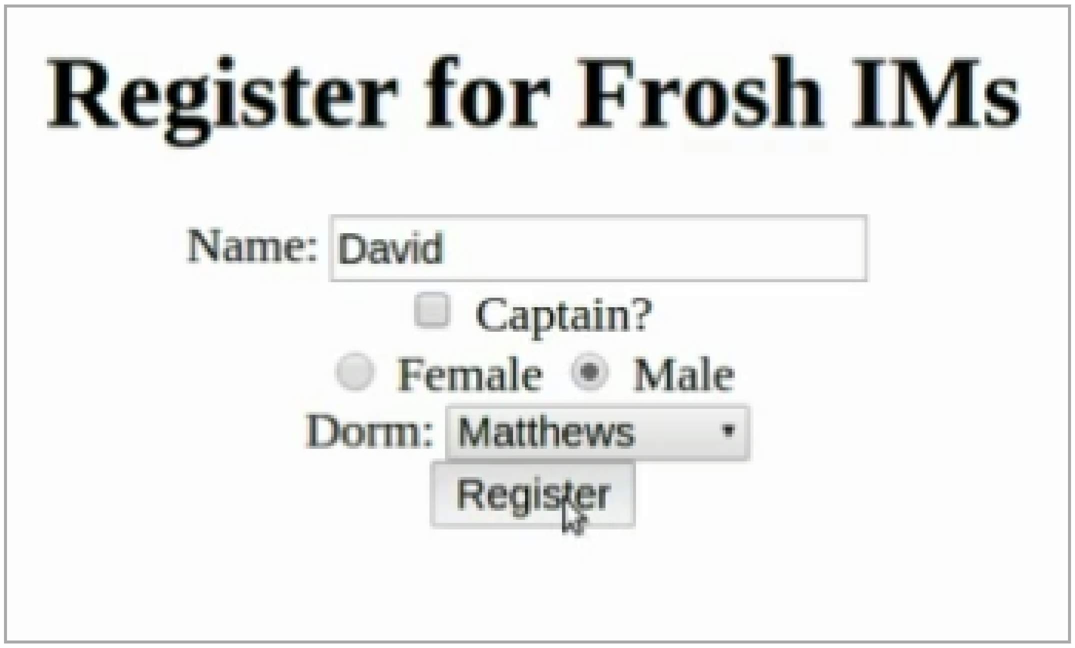
Forms, sessions, and emails

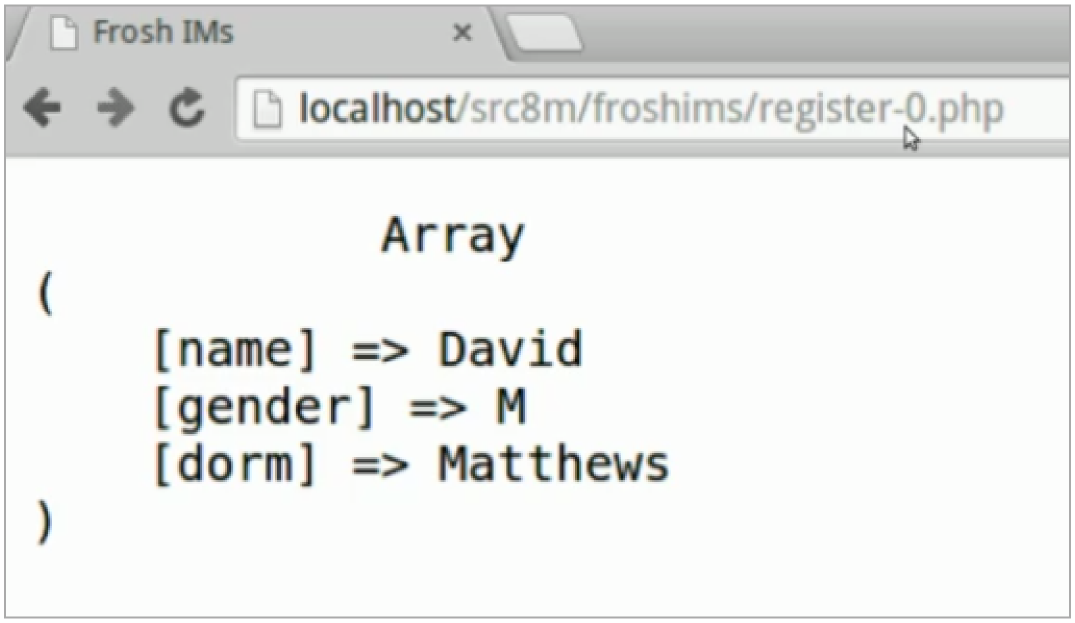
- Back in the day, one of David's first projects was helping improve the freshman intramurals program. Students had to fill out paper forms (gasp!) and drop them off with some proctor, who had to keep track manually. These days, we might just use a Google Form, but in the Olden Days when that didn't exist, David and his roommate used a language called Perl to create online forms. And now, we have the glory of recreating that idea in PHP.
- Expository story aside, let's take a look at `froshims-0.php`[6]:

[6] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/froshims/froshims-0.php



- Notice that it has a form that allows us to take input from the browser. Last time we submitted our query parameter to Google, but now we’ll do something else. First, let’s see what happens when we fill out the form:





```
           Array
(
    [name] => David
    [gender] => M
    [dorm] => Matthews
)
```

  - # Notice that the URL has changed to `register-0.php`, and even though the formatting isn't very pretty, we see that the values were passed in.
  - # The page has printed what looks like an `Array` in which these three values were stored.
  - # But also notice that the URL doesn't include the queries as strings, like we saw with Google and `/search?q=cats`.

- Let's investigate by opening the source code for `froshims-0.php`:

```
<!DOCTYPE html>

<html>
    <head>
        <title>Frosh IMs</title>
    </head>
    <body style="text-align: center;">
        <h1>Register for Frosh IMs</h1>
        <form action="register-0.php" method="post"> ❶
            Name: <input name="name" type="text"/>
            <br/>
            <input name="captain" type="checkbox"/> Captain?
            <br/>
            <input name="gender" type="radio" value="F"/> Female
            <input name="gender" type="radio" value="M"/> Male
            <br/>
            Dorm:
            <select name="dorm">
                <option value=""></option>
                <option value="Apley Court">Apley Court</option>
                <option value="Canaday">Canaday</option>
                <option value="Grays">Grays</option>
                <option value="Greenough">Greenough</option>
                <option value="Hollis">Hollis</option>
                <option value="Holworthy">Holworthy</option>
                <option value="Hurlbut">Hurlbut</option>
                <option value="Lionel">Lionel</option>
                <option value="Matthews">Matthews</option>
                <option value="Mower">Mower</option>
                <option value="Pennypacker">Pennypacker</option>
                <option value="Stoughton">Stoughton</option>
                <option value="Straus">Straus</option>
                <option value="Thayer">Thayer</option>
                <option value="Weld">Weld</option>
                <option value="Wigglesworth">Wigglesworth</option>
            </select>
            <br/>
            <input type="submit" value="Register"/>
        </form>
    </body>
</html>
```

- This page is all HTML, with the most interesting part on line 9 that sends the form information to `register-0.php`, with the `post` method. `get` puts it in the URL, and `post` sends it another way. `post`, then, should be used for information we want to keep secure, like passwords or credit card information. Photos, too, should probably be sent with `post` since it's not easy to put them in a URL.
- So if we were to look now at `register-0.php`[7]:

```
<!DOCTYPE html>

<html>
    <head>
        <title>Frosh IMs</title>
    </head>
    <body>
        <pre>
            <?php print_r($_POST); ?> ❶
        </pre>
    </body>
</html>
```

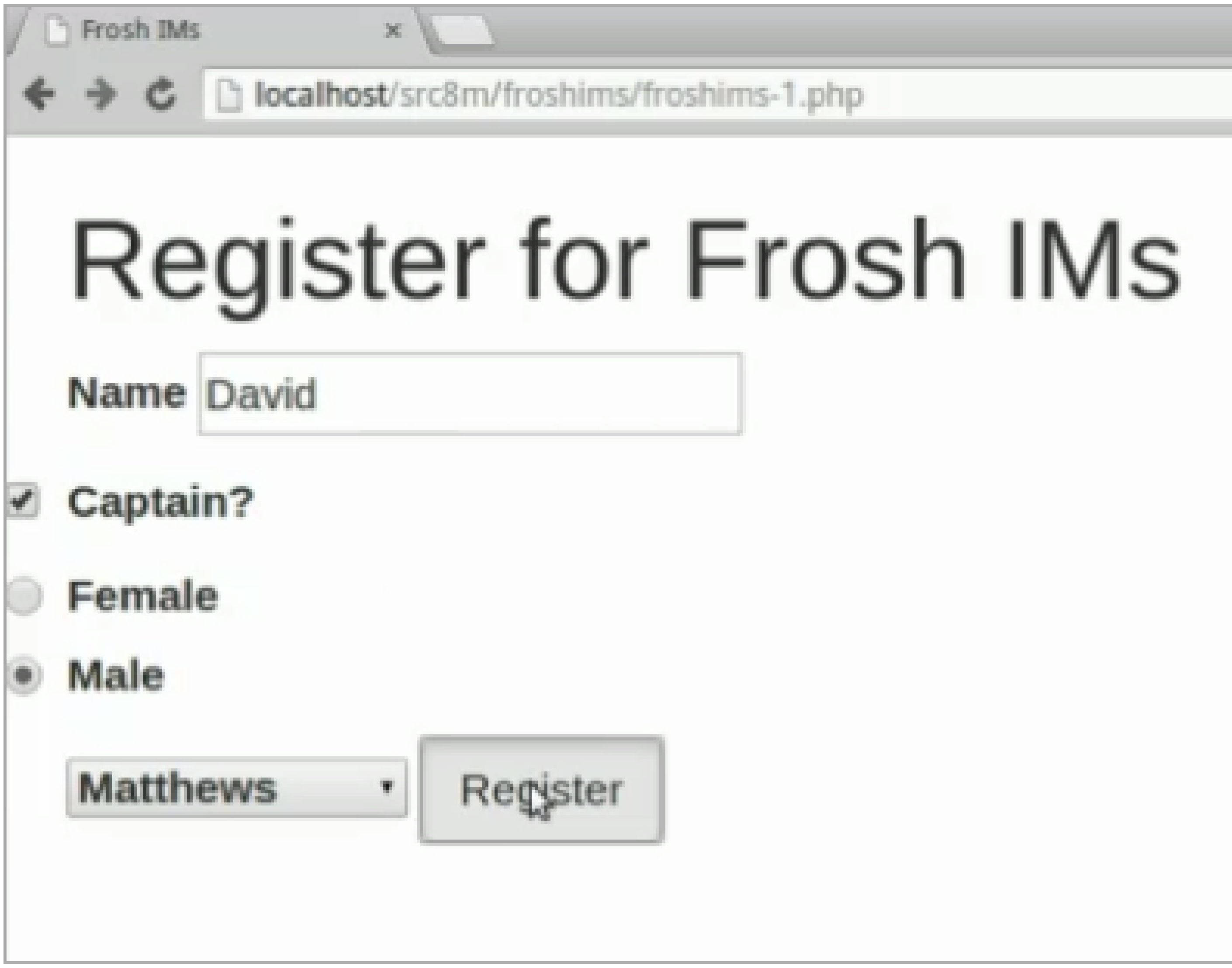
  # all it does is print out the `$_POST` variable. We see that line 9 has a `<?php` tag, nested inside a `<pre>` HTML tag that means "preformatted text," or text we want monospaced like a typewriter.

  # `print_r` is the "print recursive" function that prints everything in its argument.

  # And inside, we pass in `$_POST`, which contains everything the browser passed in, stored for us by PHP.

- Let's open another example, `froshims-1.php`[8], which looks a little better but is otherwise the same:

[7] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/froshims/register-0.php
[8] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/froshims/froshims-1.php



Register for Frosh IMs

Name David

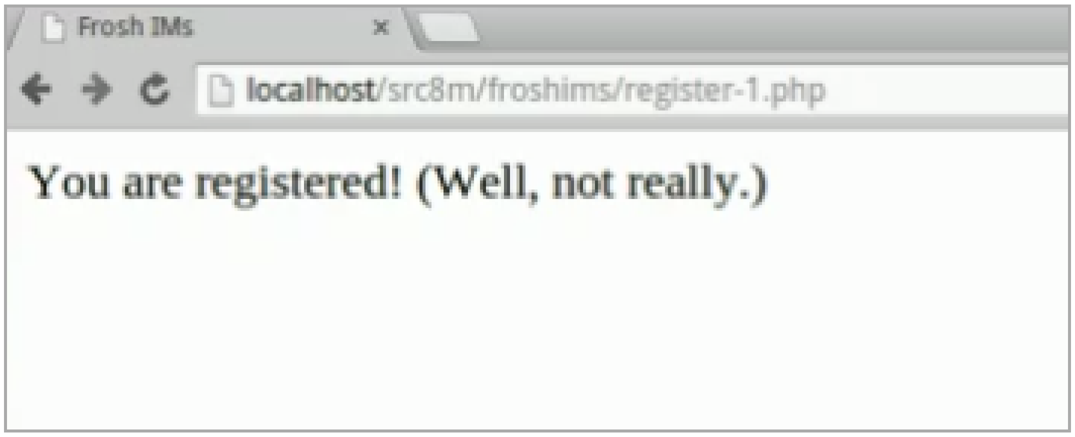
[✓] Captain?

Female

[● Male]

Matthews

# which brings us to:



You are registered! (Well, not really.)

- Mysterious. Let's investigate `register-1.php`[9]:

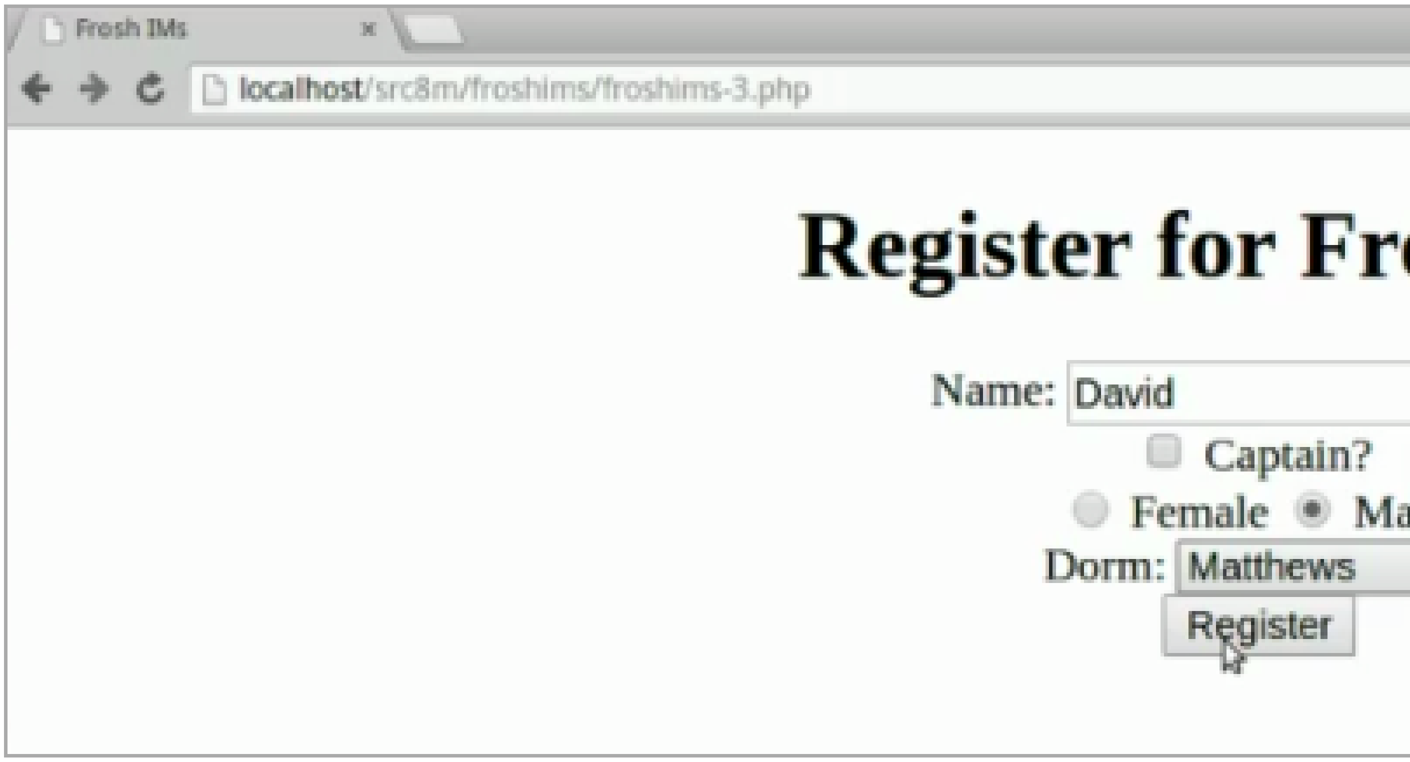
```
<?php

    // validate submission
    if (empty($_POST["name"]) || empty($_POST["gender"])
 || empty($_POST["dorm"])) ❶
    {
        header("Location: http://localhost/src8m/froshims/
froshims-1.php"); ❷
        exit;
    }

?>

<!DOCTYPE html>

<html>
    <head>
        <title>Frosh IMs</title>
    </head>
    <body>
        You are registered!  (Well, not really.)
    </body>
</html>
```

  - So there's more interesting code here, particularly lines 4-8. The comment says that this chunk of code is validating the submission.
    - Some variables, including `$_POST`, are **superglobals**, variables that are always accessible in your program.
    - In this case, we're indexing into them with words (as though `$_POST` were a hash table), passing in a **key**, or lookup word, in the square bracket, and it'll provide us with a **value**. In line 4, we first try to get the value of `$_POST["name"]`, and `empty` is just a function in PHP that tells us if its argument is empty or not. The `||` means "or" just like in C, and so the line as a whole is just checking that the user gave a `name`, `gender`, and `dorm`. If any of those are empty, then we send them to `froshims-1.php` with line 6.

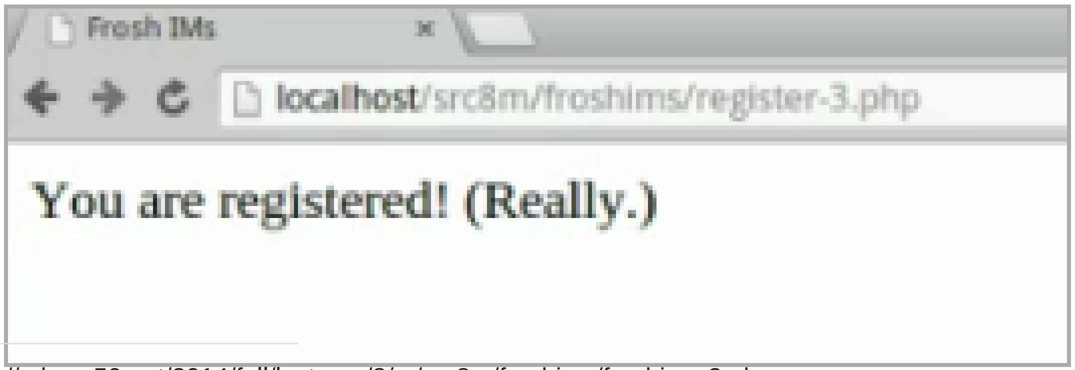
[9] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/froshims/register-1.php

# If that `if` condition didn't evaluate to true (every field has a value), then we leave PHP mode and return to basic HTML. And this is an important feature of PHP, where we can open and close the PHP tag to put PHP code anywhere. After we close the tag, the server will just return whatever text is there, so we get this ability to comingle code and markup language (for better or worse).

- Let's check out `froshims-3.php`[10]:



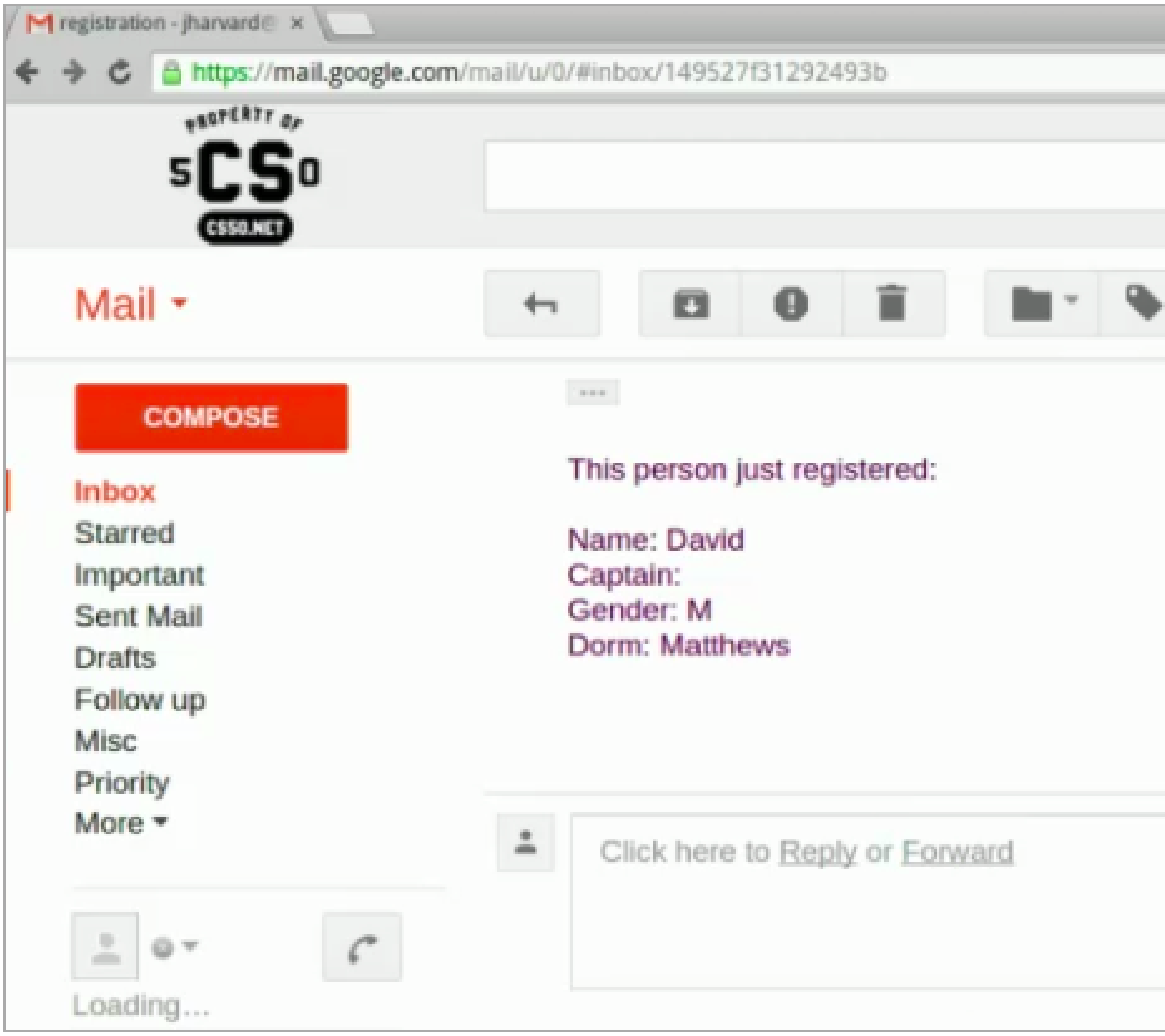
# The form looks the same, but when we submit it, we see this:



You are registered! (Really.)

[10] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/froshims/froshims-3.php

- And when we check John Harvard's email, we see this:


registration - jharvard
https://mail.google.com/mail/u/0/#inbox/149527f31292493b
PROPERTY OF CS50 CS50.NET
Mail
COMPOSE
Inbox
Starred
Important
Sent Mail
Drafts
Follow up
Misc
Priority
More
Loading...
Click here to Reply or Forward

This person just registered:

Name: David
Captain:
Gender: M
Dorm: Matthews

# The real frosh IMs program would just email the proctor in charge the form information, but here we've programmed it to email `jharvard@cs50.harvard.edu` so we can see what happens.

- Let's look at the source for `register-3.php` [11]:

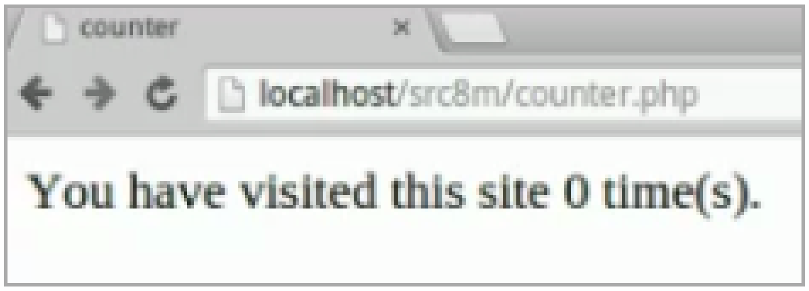
[11] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/froshims/register-3.php

```
<?php

    // require PHPMailer
    require("libphp-phpmailer/class.phpmailer.php"); ❶

    // validate submission
    if (!empty($_POST["name"]) && !empty($_POST["gender"])
 && !empty($_POST["dorm"])) ❷
    {
        // instantiate mailer ❸
        $mail = new PHPMailer();

        // use SMTP
        $mail->IsSMTP(); ❹
        $mail->Host = "smtp.fas.harvard.edu"; ❺
        $mail->Port = 587; ❻
        $mail->SMTPSecure = "tls"; ❼

        // set From:
        $mail->SetFrom("jharvard@cs50.harvard.edu"); ❽

        // set To:
        $mail->AddAddress("jharvard@cs50.harvard.edu"); ❾

        // set Subject:
        $mail->Subject = "registration";

        // set body
        $mail->Body =
            "This person just registered:\n\n" .
            "Name: " . $_POST["name"] . "\n" .
            "Captain: " . $_POST["captain"] . "\n" .
            "Gender: " . $_POST["gender"] . "\n" .
            "Dorm: " . $_POST["dorm"];

        // send mail
        if ($mail->Send() == false) ❿
        {
            die($mail->ErrInfo); 11
        }
    }
    else
    {
        header("Location: http://localhost/src8m/froshims/
froshims-3.php"); 12
```

  - We start by opening the `<?php` tag, and then requiring a special library called `phpmailer` in line 4, like how we might include one in C.
  - Then we validate the submission, and if none of the fields are empty, we "instantiate mailer" in line 9, which is just asking for a new `PHPMailer`, which is an **object**, that we put into `$mail`.
  - In lines 13-16, we set the protocol for sending our email to SMTP, the host to Harvard’s servers, the port to 587, and adding security with TLS, a protocol for securing communication with those servers. (We figured all this out by asking HUIT’s help desk.)
  - But then in line 19, it looks like we can send email from anyone to anyone, followed by a subject and a body of the email.
  - In PHP, the dot (`.`) operator allows us to concatenate strings (put strings together), as we do in lines 29-33, since we’re mixing variables with text.
  - Finally, in line 36, we try to send the email by calling the send function with `$mail->Send()`, and, if that doesn’t work for some reason, die with an error (no, seriously) in line 38.
  - Otherwise, if we didn’t meet the condition that all input was not empty (looking back up to line 7), then we send the user back to `froshims-3.php` in line 43, which is our original registration form.
  - If everything went successfully, though, we’d get to line 55, "You are registered! (Really.)", which is what we saw at the beginning of this example.
- This would be how, with real websites or final projects, you might email users with password reminders or new messages.
- And we see that this allows us to forge emails fairly easily, but at the same time those emails can still be traced back to us.
- In PHP, there are many **superglobal** variables, including:
  - $_COOKIE
  - $_GET
  - $_POST
  - $_SERVER
  - $_SESSION

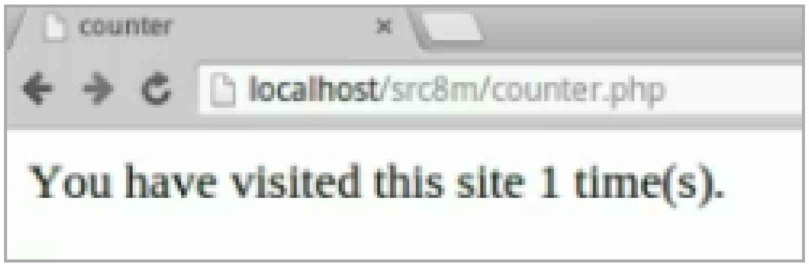
```
# …
```

- We've seen one, $_POST, and similarly $_GET is where the stuff from the URL is stored for us.
- Let's look at $_SESSION, which is interesting because we can save the state of our browsing session. So far, we've accessed webpages and clicked around, but HTTP is stateless in that once that we finish loading the page, the connection closes. Other applications, like Skype, FaceTime, and Gchat maintain a constant connection, and we'll see how we can do that too eventually.
- Let's look at `counter.php`[12]:



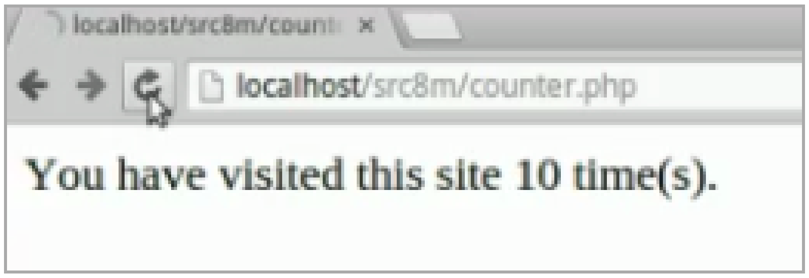
You have visited this site 0 time(s).

- If we click refresh, we get:



You have visited this site 1 time(s).

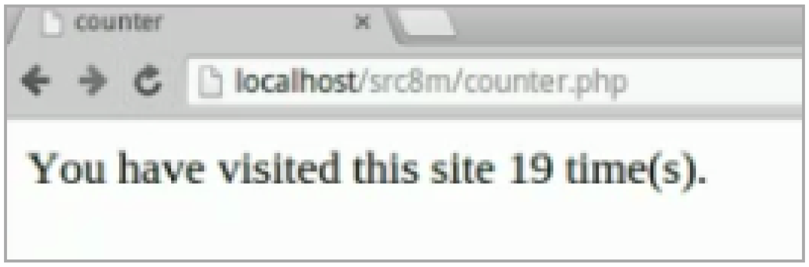
- And we can keep going:
  …



You have visited this site 10 time(s).

[12] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/counter.php

…



You have visited this site 19 time(s).

# So somehow the page knows how many times we’ve been there, even though the page finishes loading and the connection is closed.

- Let’s go into the source code of `counter.php`[13] to see how this works:

[13] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/counter.php

```
<?php

    // enable sessions
    session_start(); ❶

    // check counter
    if (isset($_SESSION["counter"])) ❷
    {
        $counter = $_SESSION["counter"];
    }
    else
    {
        $counter = 0;
    }

    // increment counter
    $_SESSION["counter"] = $counter + 1; ❸

?>

<!DOCTYPE html>

<html>
    <head>
        <title>counter</title>
    </head>
    <body>
        You have visited this site <?= $counter ?> time(s).
    </body>
</html>
```

# It looks like we're calling a function, `session_start()`, in line 4, which does what it says, start our session, which really means that we're creating a shopping cart, or bucket, that we can store values in and get again later when the user comes back. (Technically, this "bucket" is stored as a cookie in the user's browser, so if they clear their cookies, we won't have the values anymore.)

# So in line 7, we check if the value for `counter` is set in `$_SESSION`, and if it is, set `$counter`, our local variable, to it. Otherwise, we set it to 0.

  # Then we increment the value of `$counter`, and put it back in our shopping cart (i.e., $_SESSION superglobal). (Remember that `$_SESSION`, like `$_POST`, is like a hash table — an associative array that we can index into using words.)

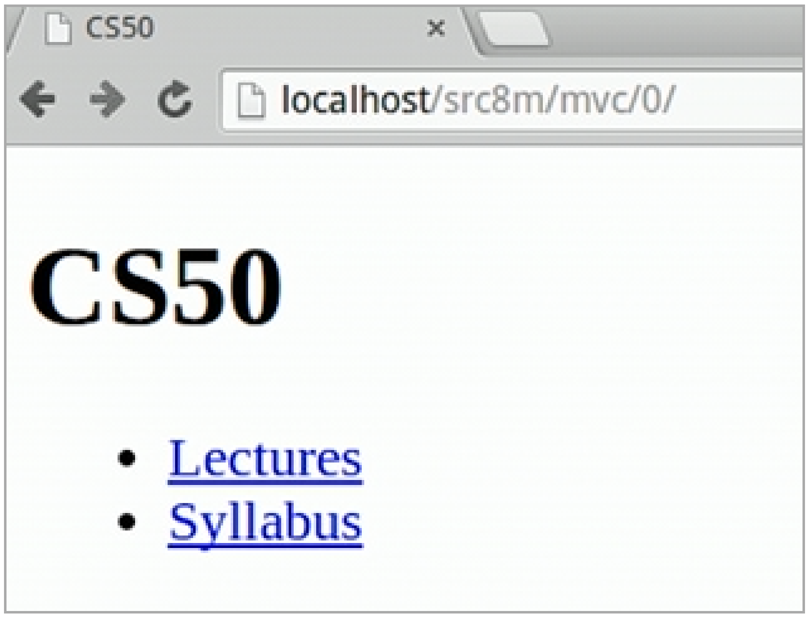
- So `$_SESSION` is going to be important in any website that has features like remembering if you're logged in, or if you've put something in your literal shopping cart to buy, or remember other information that's pending.
- Even though HTTP doesn't maintain a constant connection or state, PHP allows us this illusion.
- As an aside, `froshims-1.php` looked a bit less ugly because the source code includes Bootstrap, one of many CSS libraries on the Internet, that contains lots of styles prewritten by someone (or likely someones) that we can use to make our page pretty:

```
<!DOCTYPE html>

<html>
    <head>
        <link href="bootstrap/css/bootstrap.min.css" rel="stylesheet"/> ❶
        <title>Frosh IMs</title>
    </head>
...
```

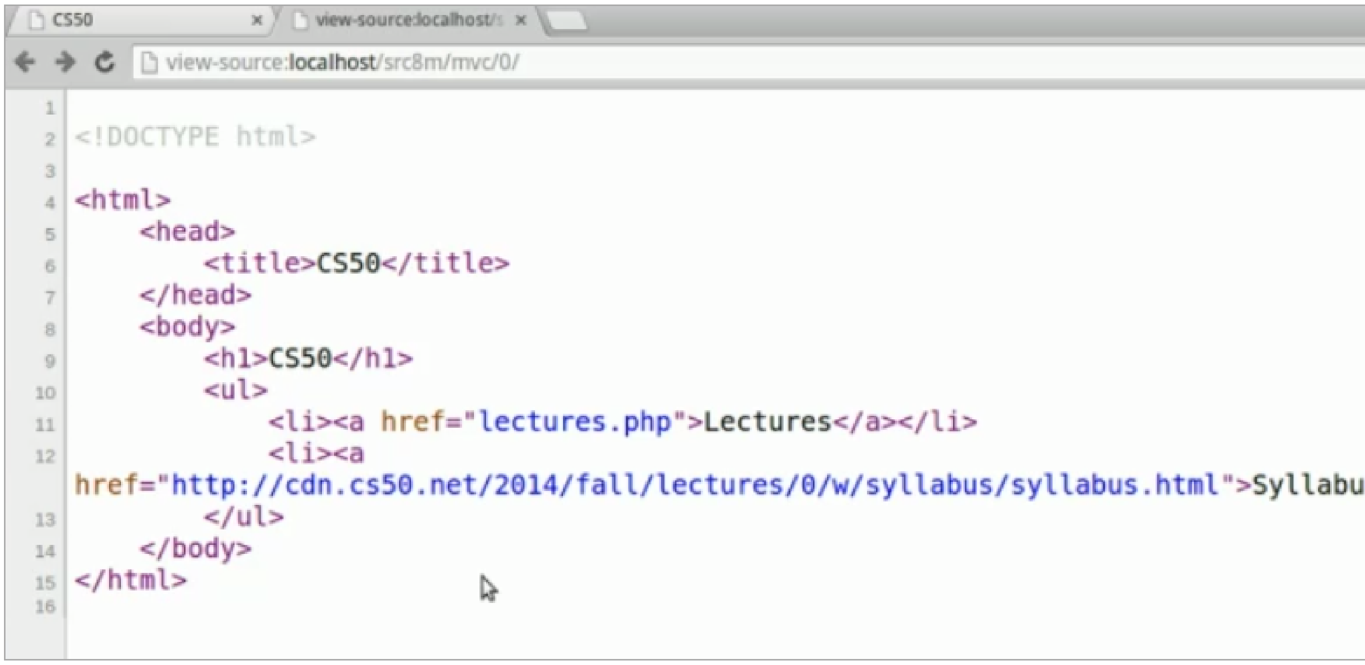
MVC with require and functions

- Let's do a little better in design. We can open `mvc/0`[14]:

[14] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/0/

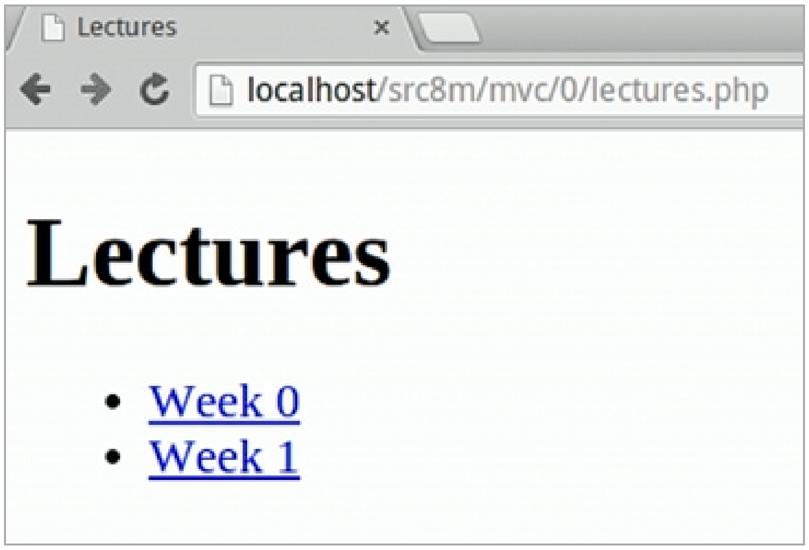


# This looks like version 0 of CS50's website, with links to Lectures and the Syllabus in what looks like an unordered list (`<ul>`). We can right click the page and **View Source** to confirm such:



```

<!DOCTYPE html>

<html>
    <head>
        <title>CS50</title>
    </head>
    <body>
        <h1>CS50</h1>
        <ul>
            <li><a href="lectures.php">Lectures</a></li>
            <li><a
href="http://cdn.cs50.net/2014/fall/lectures/0/w/syllabus/syllabus.html">Syllabu
        </ul>
    </body>
</html>

```

- And if we go back and click on a few links, this is what happens:

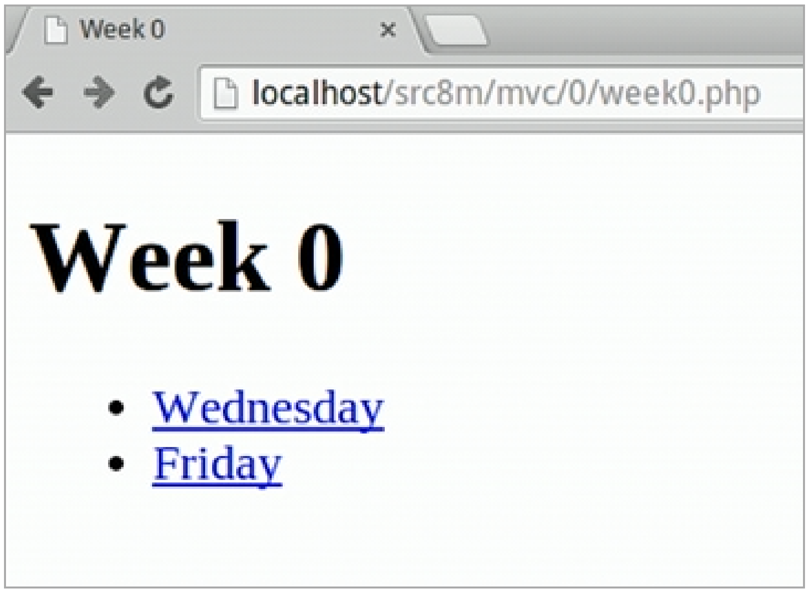


Lectures

localhost/src8m/mvc/0/lectures.php

Lectures

- Week 0
- Week 1



Week 0

localhost/src8m/mvc/0/week0.php

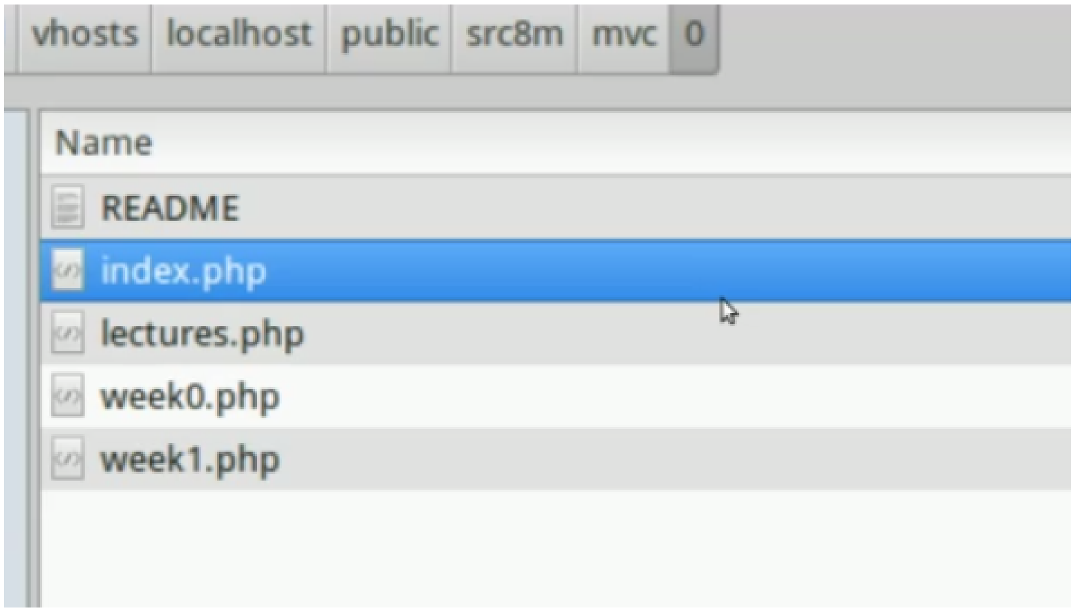
Week 0

- Wednesday
- Friday

d2o9nyf4hwsci4.cloudf
d2o9nyf4hwsci4.cloudfront.net/2014/fall/lectures/0/w/week0w.pdf
wee

  # So it looks like the Lectures page links to each week (although we only have 0 and 1 for now) and each week has a link for the slides for each day.

- Let's look at its implementation. We can open the directory `mvc/0` in `gedit`, and before we choose a file to edit, notice this:



# There's a `README` file that probably has an explanation of the code, `index.php`, which is what webservers usually use as the default page if no URL is specified. Earlier, we just visited `http://localhost/src8m/mvc/0/`, and our server just knew to return `index.php` by convention.

- If we look at the source code of `index.php`[15], we see that it's all hardcoded HTML:

```
<!DOCTYPE html>

<html>
    <head>
        <title>CS50</title>
    </head>
    <body>
        <h1>CS50</h1>
        <ul>
            <li><a href="lectures.php">Lectures</a></li>
            <li><a href="http://cdn.cs50.net/2014/fall/lectures/0/w/
syllabus/syllabus.html">Syllabus</a></li>
        </ul>
    </body>
</html>
```

[15] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/0/index.php

- The other files, like `week1.php`[16], are also just HTML:

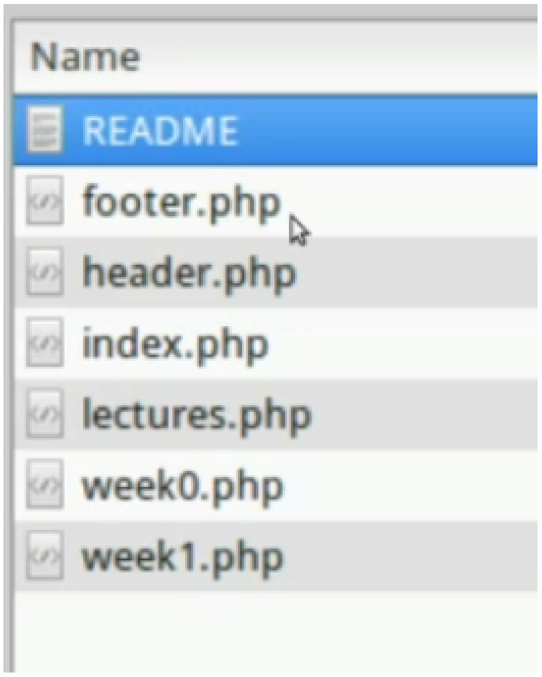
```
<!DOCTYPE html>

<html>
    <head>
        <title>Week 1</title>
    </head>
    <body>
        <h1>Week 1</h1>
        <ul>
            <li><a href="http://cdn.cs50.net/2014/fall/lectures/1/m/
week1m.pdf">Monday</a></li>
            <li><a href="http://cdn.cs50.net/2014/fall/lectures/1/w/
week1w.pdf">Wednesday</a></li>
        </ul>
    </body>
</html>
```

- Notice how redundant the code for `week0.php`[17] is:

```
<!DOCTYPE html>

<html>
    <head>
        <title>Week 0</title>
    </head>
    <body>
        <h1>Week 0</h1>
        <ul>
            <li><a href="http://cdn.cs50.net/2014/fall/lectures/0/w/
week0w.pdf">Wednesday</a></li>
            <li><a href="http://cdn.cs50.net/2014/fall/lectures/0/f/
week0f.pdf">Friday</a></li>
        </ul>
    </body>
</html>
```

- These files are basically the same, so we can try do better. `mvc/1` has a few more files:

[16] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/0/week1.php
[17] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/0/week0.php



# If we look back at `index.php` from `mvc/0`, we notice that there are many parts that are reused in all our files:

```
<!DOCTYPE html>

<html>
    <head>
        <title>CS50</title>
    </head>
    <body>
        <h1>CS50</h1>
        <ul>
            <li><a href="lectures.php">Lectures</a></li>
            <li><a href="http://cdn.cs50.net/2014/fall/lectures/0/w/
syllabus/syllabus.html">Syllabus</a></li>
        </ul>
    </body>
</html>
```

# For example, lines 1-9 would probably be almost identical, apart from the parts that say `CS50`.

# And lines 12-14 are probably also the same, with lines 10 and 11 being the only main differences.

- In version 1 of this example, `index.php`[18]:

```
<?php require("header.php"); ?> ❶

<ul>
    <li><a href="lectures.php">Lectures</a></li>
    <li><a href="http://cdn.cs50.net/2014/fall/lectures/0/w/syllabus/
syllabus.html">Syllabus</a></li>
</ul>

<?php require("footer.php"); ?> ❷
```

  # It looks a bit more complicated before, but all it does is replace some of the repeated lines earlier with line 1 and line 8. (We left the lines with `<ul>` there in case some pages didn't have a list.)

  # `require` is like `#include` in C, where the files are effectively copied and pasted in that location. So we can see that `header.php`[19] contains the header of our page:

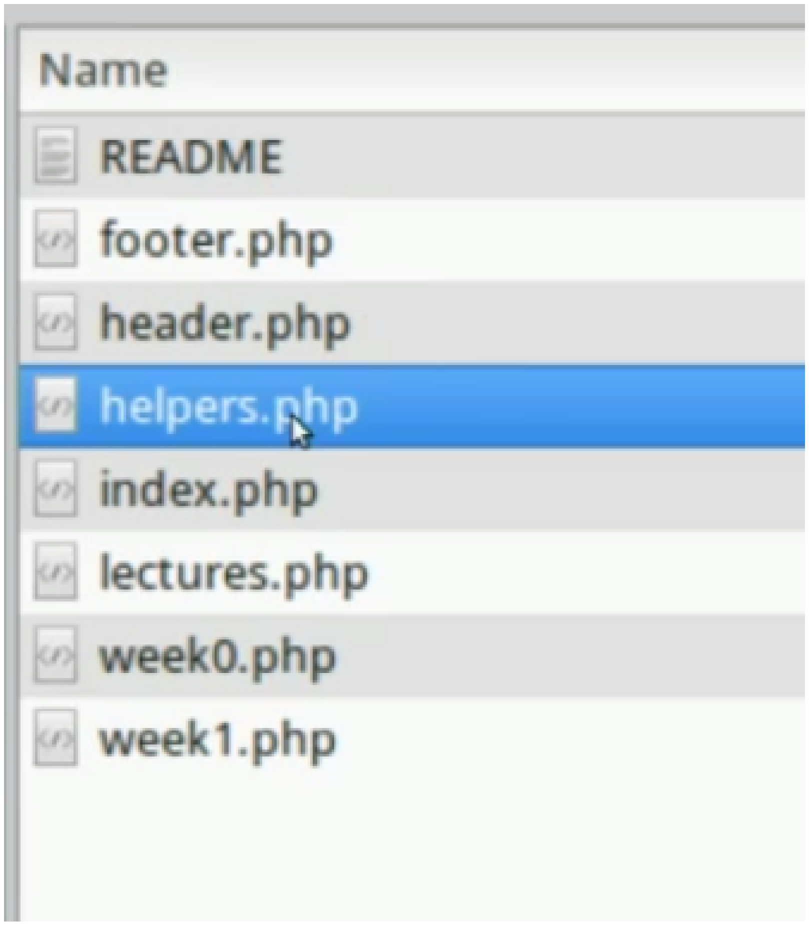
```
<!DOCTYPE html>

<html>
    <head>
        <title>CS50</title>
    </head>
    <body>
        <h1>CS50</h1>
```

  # And `footer.php`[20] just contains the footer, the bottom, of our page:

```
    </body>
</html>
```

- Since `index.php` is the file that's changing, we took those common lines out. But we can do even better (!) with version 2, whose directory looks like this:

[18] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/1/index.php
[19] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/1/header.php
[20] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/1/footer.php



- There's a new file called `helpers.php`, but let's look at `index.php`[21] first:

```
<?php require("helpers.php"); ?>

<?php renderHeader(["title" => "CS50"]); ?> ❶

<ul>
    <li><a href="lectures.php">Lectures</a></li>
    <li><a href="http://cdn.cs50.net/2014/fall/lectures/0/w/syllabus/
syllabus.html">Syllabus</a></li>
</ul>

<?php renderFooter(); ?>
```

[21] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/2/index.php

# Now we're requiring some `helpers.php` file, which, like `helpers.c` from Problem 3, is where we place helper functions.

# Line 3 looks a little different, which looks like it's calling a function called `renderHeader` and passing, inside its parentheses, square brackets that represent an associative array. In C, we had to pass in the same number of arguments in the same order to functions every time, but now we can pass in any number of key-value pairs in any order. `title` is the name of an argument and `CS50` is its value. We can go into `helpers.php` and see how this is handled.

- `helpers.php`[22]:

```
<?php

    /**
     * Renders footer.
     */
    function renderFooter($data = []) ❶
    {
        extract($data); ❷
        require("footer.php");
    }

    /**
     * Renders header.
     */
    function renderHeader($data = [])
    {
        extract($data);
        require("header.php");
    }

?>
```

# In line 6, we define a function by simply stating `function`, with no need to declare a return type, and `($data = [])` means that our function `renderFooter` takes an associative array as an argument named `$data`, and if one isn't passed in, we assume it's `= []`, an empty array (a feature that C doesn't have).

[22] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/2/helpers.php

- In line 8, `extract` just takes the keys in `$data` and creates variables from them.
- So in `index.php` we passed in `["title" # "CS50"]`, and in `helpers.php`, `renderHeader` extracts that data and requires `header.php`, which now looks like:

```
<!DOCTYPE html>

<html>
    <head>
        <title><?= htmlspecialchars($title) ?></title>
    </head>
    <body>
        <h1><?= htmlspecialchars($title) ?></h1>
```

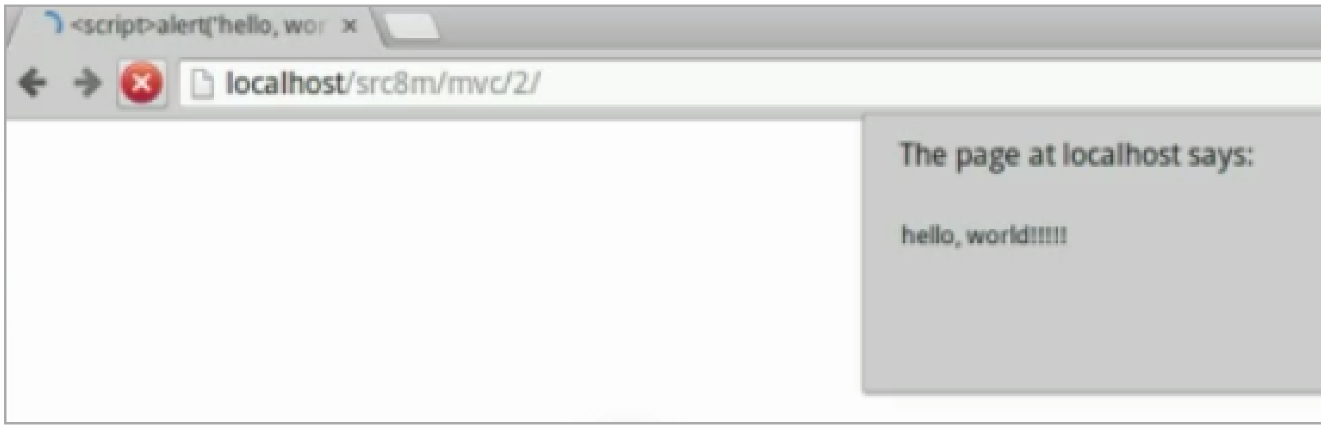
- Now we don't have hardcoded titles, but rather `htmlspecialchars($title)`, which takes whatever we passed in with the key `title` and cleans away characters like `<` and `>` that might break the site by closing or opening tags, if they were printed as-is. (Or even worse, run scripts that alert or redirect the user.)
- Notice too that this time our code is starting with `<?=` instead of `<?php`, which is shorthand for `<?php echo`, or print with PHP, whatever is inside those tags.
  - As an aside, scripts (which we'll learn more about later) can be run like this. Set `index.php` to pass in something like this:

```
<?php require("helpers.php"); ?>

<?php renderHeader(["title" => "<script>alert('hello, world!!!!!');</
script>"]); ?>
...
```

  - And if `helpers.php` doesn't call `htmlspecialchars` to clean up `$title`:

```
<!DOCTYPE html>

<html>
    <head>
        <title><?= $title ?></title>
    </head>
    <body>
        <h1><?= $title ?></h1>
```

# Then we are literally printing the script into our page, which does this:



The page at localhost says:

hello, world!!!!!

# So our best bet is to always remember to call `htmlspecialchars`.

- Let's open `index.php`[23] in `mvc/3` where we simplified the `render` function even more:

```
<?php require("helpers.php"); ?>

<?php render("header", ["title" => "CS50"]); ?>

<ul>
    <li><a href="lectures.php">Lectures</a></li>
    <li><a href="http://cdn.cs50.net/2014/fall/lectures/0/w/syllabus/
syllabus.html">Syllabus</a></li>
</ul>

<?php render("footer"); ?>
```

[23] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/3/index.php

  # We realized that `renderHeader` and `renderFooter` were almost identical functions, so we combined them into one function, `render`, that takes another argument, the name of the template, or file to render, and optionally some key-value pairs (which we want for the header but not the footer).

- To make this work, `helpers.php`[24] looks like this now:

```php
<?php

    /**
     * Renders template.
     */
    function render($template, $data = [])
    {
        $path = $template . ".php";
        if (file_exists($path))
        {
            extract($data);
            require($path);
        }
    }

?>
```

  # It's a bit more complex, since we take the first argument, `$template`, and look for the file we want to use, instead of hardcoding it, but otherwise works the same way.

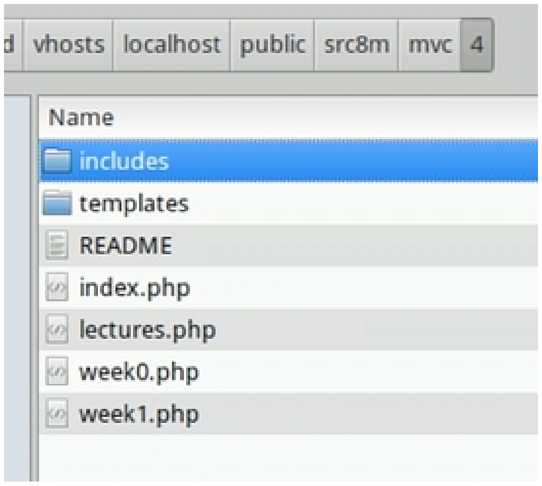
- In `index.php`[25] in `mvc/4`, we get even fancier with this:

[24] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/3/helpers.php
[25] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/4/index.php

```
<?php require("includes/helpers.php"); ?> ❶

<?php render("header", ["title" => "CS50"]); ?>

<ul>
    <li><a href="lectures.php">Lectures</a></li>
    <li><a href="http://cdn.cs50.net/2014/fall/lectures/0/w/syllabus/
syllabus.html">Syllabus</a></li>
</ul>

<?php render("footer"); ?>
```

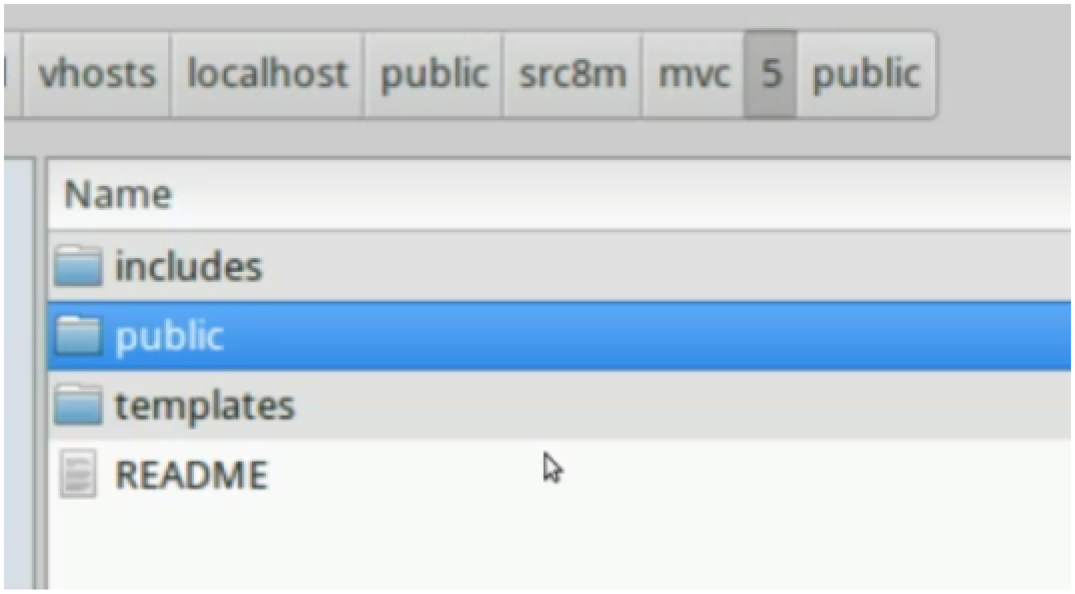
# We've kept everything else the same, but moved `helpers.php` to a folder called `includes`, and updated line 1 to call the file in that directory now.

- If we open that folder in `gedit`, we see that we've gotten rid of the extra files and moved them to `includes` and `templates`:



# `header.php` and `footer.php` live in `templates` now, another step toward better design. Instead of mixing together HTML and CSS and PHP, which is hard to maintain, we want to factor out as much as we can, just like how we had separate functions and even files in C.

- In `mvc/5`, we've moved the rest of the files to a `public` folder:



- And the first line of `index.php` [26] in the `public` folder looks like:

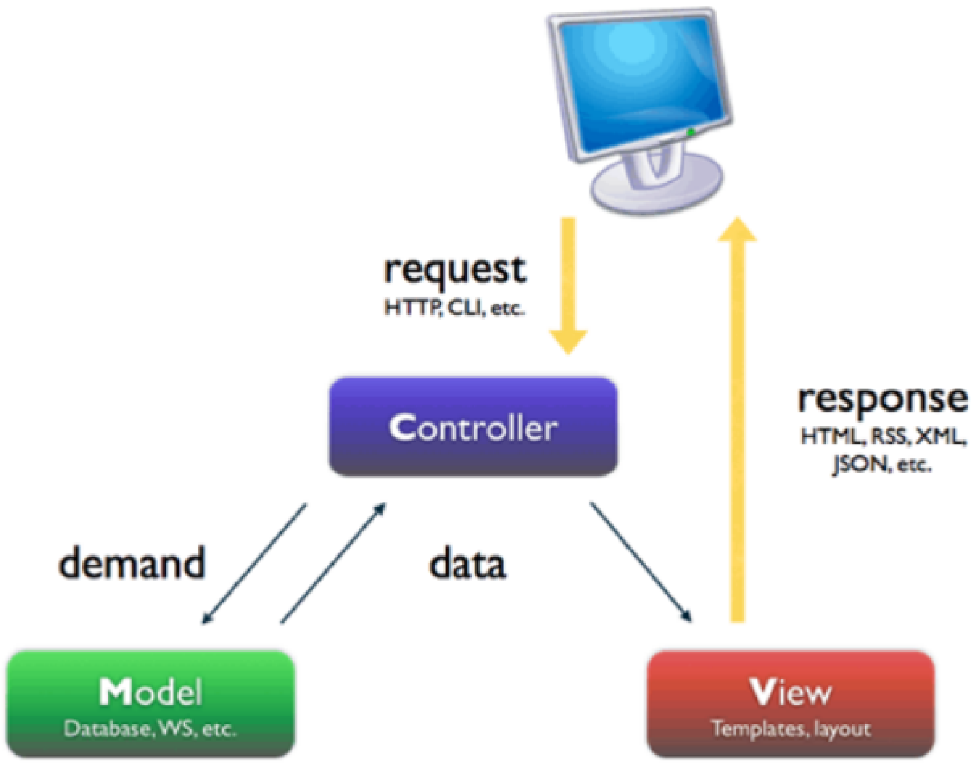
```
<?php require("../includes/helpers.php"); ?>
```

  # We can use `..` to access the parent directory, since `index.php` is in `public` and we need to go up one level.

- We've structured the site like this to organize them better, and will also help us increase security more easily.

- Let's look at this picture from last time:

[26] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/mvc/5/index.php



  # We'll put our programming logic, the brains of our website, into files that we'll start calling **controllers**. We'll have **views**, like templates, that has the aesthetics - taking the data and formatting them with color and layout. Finally, we'll eventually get to the **model**, databases that we can store and retrieve data from.

- We'll do that next time with yet another new language, SQL (read as "sequel").