Computer Science 75
Spring 2009
Scribe Notes

Lecture 2: February 9, 2009
Andrew Sellergren

Contents

Computer Science 75
Spring 2009
Scribe Notes

Lecture 2: February 9, 2009
Andrew Sellergren

1 Announcements (0:00–3:00)

- Sections will begin this evening! There will be one immediately following lecture held in Harvard Hall 103. Wednesday's section, held in 1 Story Street by Jesse Cohen, will be taped and also streamed live online. The opportunity to interact with Jesse via instant messaging is being investigated. Finally, Thursday's section, held in the Virtual Terminal Room (VTR) by the Pride of Philadelphia, Chris Power, who will use Voice over IP (VoIP) technology to showcase his talents (which will also be recorded).
- Section notes, like lecture handouts, will always be posted online.
- We'll be holding ad hoc office hours throughout the week, especially when projects are due, to offer the opportunity to ask questions and solicit help with debugging.
- The Coop will be stocking the recommended course books, if only so that you may window shop and buy them elsewhere for cheaper!

2 More with PHP (3:00–37:00)

- You might've noticed on the course website the box which allows you to search any one of four online manuals for Apache, PHP, YUI, and MySQL. Clicking Enter will, by default, search the PHP manual.
- How is this implemented? HTML forms, of course. Let's review two of the attributes of forms. The `action` attribute specifies the URL that data is submitted to. The `method` attribute specifies either the `GET` or `POST` method of HTTP requests. Recall that `GET` has the advantage of maintaining state in the URL—that is, you are able to link to it directly and bookmark it. On the other hand, `POST` has the advantage of not having length restrictions and also not being plainly visible.
- If we examine the source code of the course website, we can see that the `type` of each of these search buttons is `button`. Ultimately, we're leveraging this type in order to take advantage of its clickable property. In fact, we're not actually using them as part of PHP form submission, but rather we're using JavaScript. What we're doing is registering an event handler called `onClick`. As the name implies, it specifies what happens when the button is clicked. Notice that in each case, we're appending the following to the end of a URL

  ```
  document.search.q.value
  ```

  Now you might understand why we set the `name` attribute of the form to be `search`. We can then access this form as an element of the entire `document` object.

- So which is the default search? In other words, if we enter a query and hit Enter, which manual's results are returned. A simple experiment will show that it's PHP. But why? Or rather, how? As it turns out, it's because we specified the `action` attribute of the form to refer to the URL of the PHP manual.
- Why did we choose the `GET` method? If we go to the online PHP manual and search it ourselves, we see that the query is passed via the `GET` method. If we're aiming to replicate this, then we don't so much care whether the PHP manual supports the `POST` method or not (although we could figure out by trial and error) because we already have a method that works.
- What about the JavaScript for the other buttons? Well, since we're explicitly redirecting the user to a different URL which we've constructed by appending the search string, we're in fact still using the `GET` method.
- There are other types of methods, among them `HEAD`, which means just give back the headers. Whether or not a method is supported depends on how the web server is configured. For example, on an Apache server, this would be specified in the `httpd.conf` file, as we examined in Lecture 0.
- If you take a look at the actual URL you're redirected to when you visit `php.net`, you'll notice that it might actually be either `us2.php.net` or `us3.php.net`. This is an example of load balancing. It appears that the makers of PHP have at least 3 web servers across which they distribute their traffic. What's the downside of this approach? Well, if one web server goes down, it defeats the purpose of maintaining high availability.
- Your unofficial homework from last week was to figure out why Harvard's website can't be accessed without typing in the `www`. How might we do this? We can begin by using the command `nslookup`. We can notice, then, that providing the parameter `www.harvard.edu` gives a so-called "Non-authoritative answer" which points us to a canonical name. The non-authoritative refers to the fact that this lookup has been cached and is not the true source of this address. In contrast, providing the parameter `harvard.edu` to `nslookup` returns no answer whatsoever.
- If we go one step farther and use the `dig` command, we get the lower-level DNS configuration information in the form of a zone file. The command `dig www.harvard.edu` reveals that an A record exists. In contrast, `dig harvard.edu` shows no such A record.
- However, even if we were to fix this by adding an A record for `harvard.edu`, this still might not work. If the web server is configured to only serve up content when the host is `www.harvard.edu`, then updating the DNS records won't be enough to fix the `www` problem.
- Your next unofficial homework assignment (you have two weeks!): figure out how to "trick" your browser into finding the correct IP address for

`harvard.edu` even though you don't have the ability to update the DNS records.

- One of the powerful features of PHP (among other languages) is *regular expressions*. A regular expression is a series of characters which allows for pattern matching. Regular expressions are especially useful for validating user input (e.g. passwords, e-mail addresses). For example, if you wanted to check that a FAS username was valid—hypothetically, if they could only contain exactly 8 alphabetic characters—you might use the regular expression `[a-zA-Z]{8}`. This will include all letters, both uppercase and lowercase and require that there be exactly 8 in a row.

- If we take a look under the hood of the course website, we can see that it's implemented using XML files. These allow us to separate content from presentation and to exert minimal effort in updating the website when necessary. For example, when we need to add a new link to the Lectures section of the website, we simply edit the `lectures.xml` file rather than manually editing the actual HTML.

- The homepage of the course website actually has very little code and yet a considerable amount of content. This is due to the fact that we've gone to great lengths to factor out common code. We can see that none of the requisite tags (e.g. `<html>`, `<body>`, `<head>`) are actually in the `index.php` file. How then, is this valid XHTML? You'll notice that at the top and bottom of the file, there are several function calls. These produce the XHTML code which is common to almost all of the pages of the website, each of which is simply an `index.php` file that lives in its own subdirectory, for simplicity's sake.

- More specifically, the course pages all contain the following lines at their tops:

```
<? require_once(''lib/course/Course.php''); ?>
<? course()->header(); ?>
```

  As you might predict, this first line means "include the file `Course.php`" when you run this code. The second line is what creates the common content. If we take this second line out, the website looks dramatically different (and worse).

- Take a look at the code of the `footer()` function:

```
public function footer()
{
    //use template
    require_once("templates/footer.php");

    //close database connection
```

```
    if (is_resource($this->link))
    {
        mysql_close($this->link);
        unset($this->link);
    }

}
```

Obviously, this code is doing two things: including a file called `footer.php` and closing the connections to a database.

- Following the breadcrumbs and opening `footer.php` reveals a whole bunch of XHTML and PHP, as we expected. The Google Analytics script, for example, is in this file.
- So let's do some coding. Yippee!
- When you begin a programming project, you'll usually be faced with some problem that you're asked to solve.[1] Let's consider the case of David's freshman year[2] when the Intramural (IM) sports program required that you go to a certain door in a certain dorm and slide a registration slip underneath it. How old-fashioned! David wanted to improve upon this system, so he quickly wrote up a registration page which we now have the benefit of mocking. Err, studying.[3]

```
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Frosh IMs</title>
  </head>
  <body>
    <div align="center">
      <h1>Register for Frosh IMs</h1>

      <br /><br />
      <form action="register.php" method="post">
        <table border="0" style="text-align: left;">
          <tr>
            <td>Name:</td>
            <td><input name="name" type="text" /></td>
```

[1] Duh, Andrew.

[2] Scary, I know, to go that far back in time, but just bear with me.

[3] Truth be told, David wrote the page in Perl, so this PHP version is merely a recreation of it. But feel free to mock him, anyway.

```
            </tr>
            <tr>

              <td>Captain:</td>
              <td><input name="captain" type="checkbox" /></td>
            </tr>
            <tr>
              <td>Gender:</td>
              <td><input name="gender" type="radio" value="F" /> F
                  <input name="gender" type="radio" value="M" /> M
              </td>

            </tr>
            <tr>
              <td>Dorm:</td>
              <td>
                <select name="dorm" size="1">
                  <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>
              </td>
            </tr>
          </table>
          <br /><br />
          <input type="submit" value="Register!" />

        </form>
      </div>
    </body>
  </html>
```

Notice first that the entire web page is laid out with invisible tables which is frowned upon in certain circles. It's a useful trick, however, because it aligns content well, even in multiple browsers.

- Where does the data go when we click the Register button? It might go to a database or a file—for example, we could write it to a CSV file or a plain ASCII file or an XML file—but for our purposes, handling it directly in a PHP file is a better option.
- With the `action` attribute, we've specified a file called `register.php`. What does this file contain? Well, the short answer is: whatever we want. Let's start off simple:

```
<?php

  print("hello, world");

?>
```

  Now, if we go to `froshims.html` and click Register!, we see that "hello, world" is printed in the browser.

- Let's try to display what we're actually submitting:

```
<?php

  print("<PRE>");
  print_r($_POST);

?>
```

  Know that this isn't a valid web page, but just a testbed. If we go to `froshims.html` and enter David, check the captain checkbox, choose the Male radio button, and select Matthews as the dorm, we get the following output to our browser:

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

  Okay, good, so at least we know it's working. It's worth noting that even this minimal functionality would have been much more difficult to implement in the old days[4] using Perl.

[4] Think prehistoric since it was David's freshman year.

3 Something Completely Different (37:00–39:00)

- Say hello to our newest staff additions, Jennifer Rogers and Sid Chandrasekaran! They're both freshmen who took CS 50 with David.
- We're now at 156 students and 9 TFs! w00t.
- Do make use of the bulletin board for questions which other students might benefit from. If you haven't already, you'll get a username and password to login to the bulletin board within 72 hours of submitting the survey for Project 0.

4 Even More PHP (39:00–50:00)

- So by convention, the captain checkbox returned a value of `on`.
- Let's look at some other tricks in PHP:

  ```
  <select name="dorm" size="10">
  ```

  If we change the `size` attribute of the `select` box to 10, we can see that not 1, but now 10 dorm names will be displayed at a time in the scroll menu.

- What if we wanted to select multiple dorms? Turns out there's a `multiple` attribute:

  ```
  <select multiple="multiple" name="dorm" size="10">
  ```

  Oops, but there's a problem. When we select multiple dorms and click Register!, not all of the dorm names get transmitted. How can we fix this? We need to tell PHP that there will multiple values for the `dorm` variable:

  ```
  <select multiple="multiple" name="dorm[]" size="10">
  ```

  These square brackets imply that the `dorm` variable is going to be its own array.

- Now, let's see if we can go about validating the user's input. We'll start simple:

  ```
  <?php

      if($_POST["name"] == "")
        print("You must provide your name!");
      else if($_POST["gender"] == "")
        print("You must provide your gender!");
      else
  ```

```
    print("Welcome to the team!");

?>
```

Theoretically, this will ensure that a user has provided both his name and his gender. If we enter in appropriate values for all the fields and click Register!, we see that the "Welcome to the team!" message is printed, implying that all is well.

- Let's go one step further and start using regular expressions:

```
if(preg_match("", $_POST["name"]))
```

When we change the first if condition to the above, we've actually not changed anything at all in terms of functionality. The `preg_match` function takes two arguments: the pattern to be matched and the string to be matched in. In our case, the first regular expression again checks if no name has been provided.

- Now we'll fill in the gaps so that the regular expression is matching a more specific phrase rather than blank space:

```
if(!preg_match("/David/", $_POST["name"]))
  print("You must provide a David name!");
```

The convention for regular expressions is to place two forward slashes on either side of the expression (or any other such token which doesn't appear within the expression itself) in order to designate the beginning and end. Now, if we type Joe into the name field and click Register!, the program will yell at us to provide David as the name.

- As you might expect, we'll get the all-is-well message if we provide David as a name, but in fact also if we provide David Malan. If we want to match *only* David, we need to add the following so-called anchors to the regular expression:

```
if(!preg_match("/^David$/, $_POST["name"]))
```

What this requires is that the name provided begins and ends with David. In other words, only the string David will satisfy these constraints.

- Let's leverage these anchors in order to utilize more of the power of regular expressions:

```
if(!preg_match("/^[a-zA-Z]$/, $_POST["name"]))
  print("You must provide an alphabetical name!");
```

As you might've surmised, this specifies that the name provided consists only of alphabetical characters. If we want other characters in there, too, (such as an exclamation point, for example) we simply start enumerating them between the square brackets.

- Validating e-mail addresses is one of the first and foremost uses of regular expressions, but it's actually been somewhat abused recently. That is, there are websites which are too stringent in their validation of e-mail addresses. According to the RFC guidelines, the "+" character is allowed in e-mail addresses. Yet, some sites block this. The "+" character is useful for filtering e-mails:

  `malan+foo@harvard.edu`

  All mail sent to this e-mail address will still be directed to `malan@harvard.edu`, but it will also retain the entire string in the address field meaning that if you wanted to make sure that spam from 1-800-FLOWERS was filtered out, you could sign up for their website using the address `malan+flowers@harvard.edu` and then match for that address in your spam filter. Also, this little trick will circumvent the policy of a lot of websites to forbid multiple accounts for the same e-mail address. Shhh.

5 More TFs! (50:00–52:00)

- Please welcome Peter Lifland and Alex Chang, both of whom were on staff for CS 50 in the fall!

6 Seriously, More PHP? (52:00–61:00)

- Up till now, we haven't been outputting valid XHTML when we submit our form, but only simple text to check that it was working. Let's see if we can remedy this by first copying the source of `froshims.html` to a new file called `register2.php` so that we can retain as much of the XHTML as possible:

```
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Frosh IMs</title>
  </head>
  <body>
    <b>Name</b>:
    <br />
```

```
    <b>Captain?</b>:
    <br />
    <b>Gender</b>:
    <br />
    <b>Dorm</b>:
    <br />
  </body>
</html>
```

As you can see, we're able to hold onto the DTD as well as some of the tags. We've stripped out all of the `body` content because we want to completely redefine the page.

- We can change the Name line to be the following so that it will actually display the name provided by the user:

  ```
  <b>Name: <?php print($_POST["name"]); ?> </b>:
  ```

- We could do the same for the Captain field, but we don't really want to display just "on." So we'll add a bit of logic:

  ```
  <b>Captin?</b>:
  <?php

    if ($_POST["captain"] == "on")
        print("YES");
    else
        print("NO");

  ?>
  ```

  Needless to say, this is a much more aesthetically pleasing way to display the Captain data.

- Following suit, we can display the Gender data pretty simply:

  ```
  <b>Gender</b>: <?php print $_POST['gender'] ?>
  ```

  Note that for most purposes, single and double quotes are equivalent in PHP. The only difference is that double quotes are *magic*. That is, expressions between them are interpolated such that if you write `''$var''`, the value of `$var` will be printed rather than the variable name including the dollar sign.

- What if we try the same code with the Dorm data? Instead of the actual data, we'll get "Array" printed to the browser. This is, of course, because we specified that `dorm` be an array. If we want to display the actual dorm data, we'll need to iterate over each of the values in the `dorm` array like so:

```
<b>Dorm</b>:
<?php

    foreach ($_POST["dorm"] as $dorm)
        print("$dorm<br />");
?>
```

This construct will iterate over each value in the `dorm` array and assign each value to a variable called `$dorm` while doing so.

- Aaaaand, that's it! Check out the entire source code here. Note that PHP files actually get executed on the web server and, in general, you won't want users to be able to see the source of PHP files. However, for teaching purposes, we can use the file extension `.phps`, meaning PHP source, which will allow you to view the nicely formatted source code (with pretty syntax highlighting!).

7 Solving More Complex Problems with PHP (61:00–95:00)

- Consider the following problem: you want to password protect not only the main page but several other pages on your website. One simple solution to this problem is to include some file at the top of all such pages and in this file, execute code which redirects a user to the login page if he is not already logged in.
- Take a look at the incomplete source of `login1.php`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Log In</title>
  </head>
  <body>
    <form action="/lectures/src/2/login/login1.php" method="post">
      <table>
        <tr>
          <td>Username:</td>
          <td><input name="user" type="text" value="" />
          </td>
        </tr>
        <tr>
          <td>Password:</td>
          <td><input name="pass" type="password" /></td>
        </tr>
```

```
        <tr>
          <td></td>
          <td><input type="submit" value="Log In" /></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

A few things to notice: the `action` attribute specifies `login1.php`, the same file we're submitting from; the method is `POST`, for security reasons; the type of the password field is `password`, which simply prevents prying eyes from seeing the password as we type it; finally, after we submit, the URL does not change to `https:`, so there's room for security improvements.

- One paradigm that involves more separation of form and function is to have the login form submit to a different PHP file altogether, but for simplicity's sake, we're examining the case where a form submits to itself. One advantage to this is that it enables us to retain some information in the forms if there is some error while submitting.

- But if we're submitting a form to itself, then we're going to need some logic at the top that will actually handle the information. This logic can be seen in the pre-prepared version of `login1.php`:

```
<?
    /**
     * login1.php
     *
     * A simple login module.
     *
     * Computer Science E-75
     * David J. Malan
     */

    // enable sessions
    session_start();

    // were this not a demo, these would be in some database
    define("USER", "jharvard");
    define("PASS", "crimson");

    // if username and password were submitted, check them
    if (isset($_POST["user"]) && isset($_POST["pass"]))
    {
        // if username and password are valid, log user in
```

```
            if ($_POST["user"] == USER && $_POST["pass"] == PASS)
            {
                // remember that user's logged in
                $_SESSION["authenticated"] = TRUE;

                // redirect user to home page, using absolute path
                // http://us2.php.net/manual/en/function.header.php
                $host = $_SERVER["HTTP_HOST"];
                $path = rtrim(dirname($_SERVER["PHP_SELF"]), "/\\");
                header("Location: http://$host$path/home.php");
                exit;
            }
        }
    ?>
```

The `define()` function allows us to specify constants and their value. By convention, constants are named with all uppercase letters.

- The first conditional check in the above file is to determine if the user arrived at this page after submitting a form or for the first time. If they've arrived for the first time, then we want to simply display the XHTML content. If they've submitted the form already, we want to check their login information. We accomplish this using the function `isset()`, which in this case checks whether the username and password have been defined, as by a form submission.
- Subsequently, we want to check that they're equal to the hardcoded username and password we specified as constants earlier. If they *are* equal, then we want to somehow remember that the user is logged in.
- The fact that HTTP is a stateless protocol poses a problem toward remembering that users are logged in. When your browser has finished fetching content from the server, it severs the connection. How then do facebook and online banking systems, for example, remember that a user is logged in?
- More importantly, how do we uniquely identify a user? IP address is not enough because users connected through the same router, for example, will all share an IP address. Instead of using IP address to identify you, then, the server will plant on your computer, either on the hard drive or in RAM, a file called a *cookie*. What does the cookie contain? In short, anything the server wants. That means, if they wanted to be insecure and store your username and password, they can. Bad practice!
- A web server typically stores in the cookie not your initial login credentials but rather one large, random number which then identifies you in the server's database. The database stores more securely a whole slew of information about you. This improves performance, as well, because it

minimizes the amount of information that must be transmitted between client and server each time authentication is required.

- As a developer, we store information about a user via the superglobal `$_SESSION` variable. Cookies are used in the following way: when a cookie is transmitted to the server, it is checked against the database and all information which has been stored about the user will be retrieved from the database and inserted back into the `$_SESSION` variable where it was originally stored.

- For our purposes, the amount of information we want to store is trivial: we can remember whether a user is logged in or not using a single boolean.

  ```
  $_SESSION["authenticated"] = TRUE;
  ```

  In the homepage, we'll use this in an if condition:

  ```
  <? if ($_SESSION["authenticated"]) { ?>
    You are logged in!
    <br />
    <a href="logout.php">log out</a>
  <? } else { ?>
    You are not logged in!
  <? } ?>
  ```

  That's all there is to the first part of the login validation process.

- The second part of the login validation process is to redirect the user to the homepage if he is logged in. We do this using these lines of code:

  ```
  // redirect user to home page, using absolute path
  // http://us2.php.net/manual/en/function.header.php
  $host = $_SERVER["HTTP_HOST"];
  $path = rtrim(dirname($_SERVER["PHP_SELF"]), "/\\");
  header("Location: http://$host$path/home.php");
  exit;
  ```

  We might think about doing this the "easy" way by simply redirecting to `home.php`. However, the specification for HTTP headers indicates that the URL should be fully qualified, meaning that it includes the full domain name and the path. These lines of code are simply using the superglobal `$_SERVER` to retrieve the domain name and the path and append them to the redirect URL.

- Take a look at this line of code from `login1.php` and see if you can figure out what it's doing:

  ```
  <? if (count($_POST) > 0) echo "INVALID LOGIN"; ?>
  ```

At this point in the login page, after the initial validation, we've reached the primarily XHTML content. If we've gotten to this point in the page and there is information stored in the `$_POST` variable (i.e. `count` will return greater than 0), then the user must have already submitted the login form and yet *not* been redirected because his login was not validated. Thus, logically, we can assume that the login was invalid.

- If we take a look at `login2.php`, we see that we've made the effort to retain the username if the user's login fails. How do we do this? It's very simple, in fact:

```
<td><input name="user" type="text"
    value="<? echo $_POST["user"]; ?>" />
</td>
```

  As you can see, we've simply printed out the username into the proper field. Why didn't we do this with the password field? Well, if we had, it would've been cached. Someone with too much free time on their hands might be able to poke around on the hard drive and come up with this password if they were so inclined.

- So far, we've overlooked one line of code that's been in all of the login files:

```
//enable sessions
session_start();
```

  Anytime you want to enable sessions—that is, store information such as login status across multiple pages—you must call this function in PHP. Otherwise, PHP will not create cookies to uniquely identify users.

- In `login3.php`, we're setting a cookie explicitly like so:

```
// save username in cookie for a week
setcookie("user", $_POST["user"],
          time() + 7 * 24 * 60 * 60);
```

  Here, we're telling PHP to store the username in the cookie for a week's time. Later, we use this to pre-populate the username field:

```
<td>Username:</td>
<td><input name="user" type="text"
     value="<? echo ($_POST["user"])
               ? $_POST["user"] : $_COOKIE["user"]; ?>" />
</td>
```

These lines make use of a ternary operator. That is, it has three operands. The first is the condition, in parentheses, which checks whether the `user` variable is set. If it is, then it will be echoed to the screen. If not, then the username stored in the `$_COOKIE` superglobal will be outputted.

- What's the difference between session cookies and cookies created by calling `setcookie()`? Session cookies, by default, only exist in RAM and thus are destroyed when the user navigates away from the page or perhaps not until he closes his browser. When `setcookie()` is called with some positive value specified as the second argument, the cookie will persist on the user's hard drive until that length of time has run out.
- Question: what does the call to `exit()` do at the end of the redirect code? Although most browsers pay attention to the specification for HTTP headers and will redirect the user to a different page, if we don't have this `exit()` call, many of them will still output the remaining XHTML content on our `login1.php` page. Frankly, we don't want this because they *shouldn't* see this content if we mean for them to be redirected, but we also don't want this because it's a waste of bandwidth.
- Question: who specifies the names of cookies? The developer has this power. They are stored on a per-domain basis, though, so that there won't be name collisions across multiple domains.
- Question: what if the browser doesn't support cookies? Some browsers will compensate for this by appending the very large random number to the end of the URL. The Extension School's website does this to a certain extent. If you poke around long enough, you might notice that the string `?jsessid` will appear in the URL followed by a very long string of digits.
- In `login4.php`, we do a bad thing by storing the user's password as well as username. Your second unofficial homework assignment for next class is to navigate to login4.php and login as user `jharvard` with password `crimson`. Once you've done this, poke around on your hard drive and see if you can find where the cookie has been stored and, within that cookie, where the username and password are stored.
- One other difference between `login4.php` and `login3.php` is that we're now authenticating using the `$_COOKIE` superglobal rather than the `$_POST` superglobal. This is not recommended!
- What is the big danger of cookies? If someone is sniffing packets and grabs one which is transmitting your cookie to the server without SSL encryption, then certainly this malicious user can hijack your session.
- PHP version 5 supports object-oriented programming (OOP). Recall that our own `course.php` file uses classes, a key feature of OOP.
- A good example of the user of sessions is to implement the shopping cart functionality of e-commerce websites.

8 Transition to SSL (95:00–100:00)

- Last week, we used the `.htaccess` file to accomplish the rebranding of our website by adding a `www` in front of it. This week, we've added a few more lines to it:

```
RewriteEngine On

RewriteCond %{HTTP_HOST} !^www\.cs75\.net [NC]
RewriteRule (.*) http://www.cs75.net/$1 [R=301,L]

RewriteCond %{REQUEST_URI} ^/login/
RewriteCond %{HTTPS} != on
RewriteRule (.*) https://www.cs75.net/$1 [R=301,L]
```

  The three lines at the bottom tell the server to bounce the user back to SSL anytime he tries to access files in the `login` directory. SSL websites run not on port 80 but on port 443. This means that in your `httpd.conf` file, there needs to be some mention of listening on this port. The web server also needs to know what key file to use. This is not something we'll be asking you to set up because not only would we need 156 unique IP addresses, *you* would also need to fork over a whole lot more money. Even if you buy an SSL certificate from someone like GoDaddy or VeriSign, there's a chance that it won't be recognized by certain browsers (if the vendor hasn't made the right partnerships).

- What's with this key file? Once you've purchased an SSL certificate, you'll have to upload a very small text file to your web server which contains a very large number. After you've told the web server where to find this file, every HTTP request which uses SSL will begin by matching this key file against a key file on the client and generating from the two a shared very large, secret number with which all subsequent traffic will be encrypted, including the headers. This is why you need a unique IP address for SSL with virtual hosting because otherwise the server can't know where to direct the traffic if it can't decrypt the domain name.

- Setting up SSL is thus a matter of adding a few lines of code to the `httpd.conf` file:

```
SSLCertificateFile /path/to/certficate
SSLCertificateKeyFile /path/to/key
SSLCertificateChainFile /path/to/chain
```