Computer Science 75
Fall 2009
Scribe Notes

Lecture 2: September 21, 2009
Andrew Sellergren

Contents

Computer Science 75
Fall 2009
Scribe Notes

Lecture 2: September 21, 2009
Andrew Sellergren

1 Announcements (0:00–3:00)

- Project 1 will task you with implementing an online ordering system for our beloved, but now defunct Three Aces Pizza. One of the challenging aspects of this project will be to develop a logical representation of the menu in XML format. The menu itself is, for lack of a better term, all over the place. This is fine for their purposes, of course, because they don't need it to be so well-organized in order to take orders in person. But you're going to revolutionize their business!
- Next week, we'll introduce you to XML and you'll have all the tools you need to tackle Project 1, which will be released concurrently. This week's section will be led by former TF Keito Uchiyama, who will do his best to plug any holes that David left!

2 More with PHP (3:00–107:00)

2.1 A Search Engine

- Although they seem uninteresting at first glance, forms are worth discussing because they are the basic building blocks of all user-driven websites. Later in the semester, we'll talk about ways to dress them up using CSS and JavaScript, but for now we'll focus on their functionality.
- If you take a look at the course website on the bottom of the lefthand menu, you'll notice a form which allows you to search the Apache, MySQL, PHP, and YUI manuals. Even if you've never used it, we can learn something from it as an example of a form. We implemented this form simply by examining how each website searches its own manual and then mimicking that behavior via `POST` or `GET`.
- Let's start by implementing the PHP search. If we go to php.net and type "count" in the search box, specifying "function list" as what to search, we will immediately be whisked away to the right answer—the manual page for `count`. From a user's standpoint, this is great, but from a developer's standpoint, this doesn't help us very much. We need to know *how* they found the right answer.
- If we go back and select "online documentation" instead of "function list" from the search dropdown menu, we get a page of results, which is more useful to us as programmers. How was this list generated? First, let's take note of the URL:

  ```
  http://www.php.net/results.php?q=count&p=manual&l=en
  ```

  Obviously, the `GET` method is being used here. Of the three parameters, the `p` and `l` can be hard-coded whereas the `q` represents our actual query. So it looks like we can pretty easily mimic this.

- Let's take a look at the actual form on the course website. We can do so either by right clicking the page and selecting View Source, which gives us a somewhat ugly and static interface, or by right clicking the actual search field and selecting Inspect Element, which brings up the Firebug interface. Of course, we're talking only about Firefox, here, but other browsers offer similar functionality.
- If we click Inspect and the run the mouse over the search field, we'll see that the form is actually highlighted in the source XHTML. Now we can see that the `action` attribute is specified as `http://us.php.net/results.php`, the same URL we saw on the PHP search results page but without the parameters.
- Inside of the `form` tag is a single `input` tag, a text area with its `name` specified as `q`. Farther down in the source, we see the PHP button which simply submits the form, albeit with a little bit of JavaScript mixed in to allow us to combine four different search fields into one.
- Notice that the search works even though we haven't specified the `p` or the `l` attributes. And, in fact, we've thrown in a parameter of our own, `php`, which seems to be ignored.
- Question: can we add parameters into the text of our query itself? No, because the ampersand and equal characters will be converted to `%26` and `%3d`, their HTML entity representations. This is a security measure to prevent URL tampering.
- You may have noticed that some forms have a default value. You might want to prepopulate a form with a username that has been previously typed, for example. To do this, we'll use the following syntax:

  ```
  <input type="text" value="<? echo $name; ?>">
  ```

  This assumes that the correct value is stored in the `$name` variable. What happens though if the user previously populated the form with a value that included double quotation marks? Those double quotation marks will be spit out along with the rest of the name and our XHTML code will be broken. So we need to escape user input before we use it to fill in the `value` attribute. We can do this by passing it to the `htmlspecialchars` function. Although this might seem like syntactic minutia, we'll see later in the semester when we work with databases that forgetting to escape user input can be a serious security concern.
- We resorted to JavaScript because we have four different buttons which we hope will lead to four different websites, but a form can only have a single `action` attribute. One issue with JavaScript is that it runs clientside so it can actually be disabled by a user. And some (really, really old) browsers don't even have it installed. So, unfortunately, those users simply won't be able to use this feature of the course website.

- If we type "date_format" into the search box and click the MySQL button, we can see that another `GET` string has been generated. Conveniently, it looks quite similar to the PHP URL:

  ```
  http://search.mysql.com/search?q=date_format&lr=lang_en
  ```

  The query is being specified by the `q` parameter and again we have some kind of language parameter, which we can quickly determine is optional. So what happens when we click the MySQL button that causes this URL to be generated? Let's take a look at the XHTML of the button:

  ```
  <input type="button" value="MySQL" onclick="window.location=
  'http://dev.mysql.com/doc/mysql/search.php?version=5.0&amp;q='
  + document.search.q.value;" class="button"/>
  ```

  The `class` we can ignore because it pertains to CSS. The `value` is simply what is displayed on the button and the `type` controls some of how it's displayed in XHTML. What about `onclick`? In the case of each 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. In addition, we named the text field of the entire form as `q`, which we're now accessing here. To get what the user has typed in the search box, we access the `value` property. Ultimately, when we click any of those buttons, we're altering the `window.location` property, which has the effect of whisking us away to the new URL value that we specify.

- If all of that went over your head, not to worry since we won't be diving into JavaScript for a few weeks yet!

- Question: could we implement the same functionality serverside? Certainly, let's take a stab at it. In a file called `search.html`, we'll write the following:

  ```
  <!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>Search</title>
    </head>
    <body>
      <form action="search.php" method="get">
  ```

```
      <input name="q" type="text" />

      <br />
      <input name="site" type="submit" value="MySQL" />
      <input name="site" type="submit" value="PHP" />
    </form>
  </body>
</html>
```

We're starting first with just two of the buttons to keep things simple. We specified the `method` of the form as `GET` if for no other reason than to keep things visible and we specified the `action` as a file called `search.php` which we haven't implemented yet. Our search field against has a `name` of `q` and we've made two buttons, although we've specified their `type` as `submit` rather than `button` so that when we click them, they'll actually submit the form.

- When we click either of the buttons now, we'll get a properly formed `GET` string, but an HTTP 404 error will be returned because `search.php` doesn't exist yet. If you were to get this error after implementing `search.php`, it might be a caching issue. Try reloading the page or clearing your cache.

- Now let's write `search.php`:

```
<?
    switch ($_GET["site"])
    {
        case "PHP":
            header("Location: http://us.php.net/results.php?q=" . $_GET["q"]);
            break;

        case "MySQL":
            header("Location: http://search.mysql.com/search?q=" . $_GET["q"]);
            break;
    }
?>
```

Since we're writing this code only for ourselves, the short open tag is okay. If we wanted to distribute this code, it would be better to use the full `<?php` instead. Specifying the Location header—in this case via a call to `header()`—has the effect of whisking the user away to a new page (assuming no output has already been spit out and the headers haven't yet been sent).

- We're switching on the value of `$_GET["site"]` which tells us which button the user clicked. In the case that it equals `PHP`, we'll set the Location header to be the URL of the PHP manual search. To make it dynamic,

however, we also need to append the value that the user has entered into the search box, which we access by referencing `$_GET["q"]`. Then we can tack it on using the concatenation or dot operator.

- Calls to the `header` function which alter the Location header are often followed immediately by calls to the `exit` function because their main purpose is to whisk the user away, in which case there shouldn't be any time left to execute any other code or otherwise generate output that the user will actually see.

- Question: what is the behavior when you don't click either button but simply hit Enter? Not sure, to be honest. It could be that it's non-deterministic or it could be that the first of the two is chosen or it could be some third option which we can't think of right now.

- We could've also accomplished the same functionality using an if-else statement (which David does to begin with in lecture) rather than a switch statement.

- Question: what considerations should you take into account when deciding whether to implement something client-side (as we did initially with JavaScript) or server-side (as we did ultimately with PHP)? One advantage of the JavaScript is that your server doesn't have to field an extra HTTP request simply to bounce the user to another site. One disadvantage of the JavaScript is that you have no way of logging how users are interacting with your search box because no information is being sent to your server.

- Which approach might we want to take if we were developing for a mobile device? Arguably, it would be good to leverage the 600 MHz of an iPhone to do some of our computation. But if the mobile device doesn't support JavaScript, then we've excluded a small subset of users. Ultimately, though, and perhaps most importantly, the PHP approach requires an extra request to the server, which, no matter how you look at it, is slow for a mobile device to undertake.

- Question: how do we know that a separate function doesn't exist in PHP that whisks the user away to a different URL? You don't, really. Of course, you have to learn the functions somewhere, whether it's by word of mouth or by reading the manual. It's worth looking through the manual at least briefly before implementing some functionality which you suspect may already be built in.

- Question: is it possible to alter content based on the user's browser? Certainly. You could choose not to show the search box at all if you knew that the user's browser wouldn't work with it. You could do this with a simple if-else statement although it might be ugly.

- Question: you could also program your site so that if JavaScript failed, the server would take over and take care of the search. YUI does an excellent job of this.

2.2 User Registration

- A quick sidebar about the notation of the PHP manual. Arguments to functions are optional if they are enclosed in square brackets. If those square brackets are nested in other square brackets, the argument is contingent on the previous argument existing as well. The `header` function, for example, takes one mandatory argument, the actual header string, and two optional arguments. The third argument is an HTTP response code, but we can't specify it without also passing the second argument, a boolean which determines whether a previous header will be replaced. Since the manual write "bool $replace = true," we know that the default value of this argument is "true," so if we want to specify an HTTP response code as the third argument, we'll simply hardcode the value "true" as the second argument.

- Let's consider the case of David's freshman year[1] 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.[2]

```
<!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] Scary, I know, to go that far back in time, but just bear with me.

[2] 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.

- 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 (and, for the moment, it contains nothing). Let's start off by displaying what we're submitting:

```
<pre>
<?php

    print_r($_GET);

?>
</pre>
```

This isn't well-formed XHTML but is rather just a temporary sanity check. 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
)
```

The "on" value is simply the way that checked checkboxes are denoted. The "M" comes from the `value` attribute of our radio button, which has two `input` tags that are mutually exclusive because they both have the same `name` attribute/.

- Now let's actually do something with the data we've received. We can start by sending an e-mail, presumably to the RA who was in charge of registering freshmen for intramurals. An inbox full of e-mails is still probably more manageable than a pile of sheets.

- To send an e-mail, we'll use the `mail` function of PHP. If we look at its manual page, we can see that it takes three mandatory arguments: the recipient, the subject, and the message. So let's try using it:

```
<pre>
<?php
```

```
    $to = "ra@harvard.edu";
    $subject = "Frosh IMs Registration";

    $message = "This person just registered: \n\n";
    $message .= $_GET["name"] . "\n";
    $message .= $_GET["captain"] . "\n";
    $message .= $_GET["gender"] . "\n";
    $message .= $_GET["dorm"] . "\n";

    $message .= "love, the website";

    mail($to, $subject, $message);

?>
</pre>
```

Note that instead of using the concatenation operator, we could've leveraged the fact that double quotes are "magic" in PHP, meaning that if we write variable names within them, the variables will automatically be replaced with their values through a process called interpolation. In this case, since our variable is an array, we'll need to surround it with curly braces like so:

```
$message .= "{$_GET["name"]}\n";
```

Realize also that `$message .= "foo"` is shorthand for `$message = $message . "foo"` —in other words, we're appending to the string.

- Once we've specified the recipient (`$to`), the subject (`$subject`) and the message (`$message`), we simply call the `mail` function and we're done. Of course, we haven't given the user himself any feedback about the data submission, so let's add some HTML to do just that:

```
<html>
  <head>
    <title>Thanks!</title>
  </head>
  <body>
      Thanks!  You're registered.
  </body>
</html>
```

To be a little more precise, we might want to actually check whether the `mail` function returned `true` to know whether the e-mail was actually sent successfully or not. In the weeks to come, we'll see that instead of simply e-mailing the data, we can write it persistently to a database.

- So if we want to change our message to the user based on whether the data was actuall submitted or not, we might write the following instead:

```
<html>
  <head>
    <title>Thanks!</title>
  </head>
  <body>
    <? if (mail($to, $subject, $message)): ?>
      Thanks!  You're registered.
    <? else: ?>
      Sorry, problem!  Call RA.
    <? endif ?>
  </body>
</html>
```

  Using this syntax, we can intersperse HTML and PHP quite cleanly. Instead of the colon and the `endif`, we could've also used open and close curly braces.

- Question: one of the best third-party PHP mail libraries is PHPMailer which is free and open-source.

- Question: what will the default sender be? It will probably come out as `username@cs75.net`. If you want to configure this, use the additional arguments to the `mail` function. One implication of this is that it's very easy to spoof or forge the sender of an e-mail.

2.3 Regular Expressions

- 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.

- Let's add a field for e-mail address to our registration page and begin using regular expressions to validate input:

```
<tr>
    <td>Email:</td>
    <td><input name="email" type="text" /></td>
</tr>
```

Now that we have an e-mail field we're collecting, let's turn to `register2.php` for our validation:

```
<?
    session_start();
    $correct = true;
    if ($_GET["name"] == "")
        $correct = false;
    if ($_GET["email"] == "")
        $correct = false;

?>

<html>
  <head>
    <title>Thanks!</title>
  </head>
  <body>
    <? if ($correct): ?>
      Thanks!  You're registered.
    <? else: ?>
      Sorry, problem!  Go back and fix your form.
    <? endif ?>
  </body>
</html>
```

Here we're doing the bare minimum of validation—checking if the user has provided a name and e-mail at all. If not, then we're announcing a problem. Once we've made sure to assign `register2.php` to the `action` attribute of `froshims.html`, we can test out our validation and see that an error is thrown if we leave the Name or Email fields blank.

- Of course, an enterprising student could get around these validations simply by entering a space into both fields. So we need to step it up a notch. We can perform the same check as before, but after passing the value to `trim()`, a function which removes whitespace. If after `trim()` returns, the field is empty, then we'll again throw an error.
- But now we can type in gibberish and get past the validations. Finally, we'll start using regular expressions:

```
if(!preg_match("/malan@post.harvard.edu/", $_POST["email"]))
```

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 checks if David is the

registrant. This isn't a very good use of regular expressions, but it's all we know how to do so far!

- We can see now that we'll get through the validation if we use David's e-mail address. Unfortunately, we'll also get through if the e-mail address we use *contains* David's e-mail address. For example:

  ```
  joemalan@post.harvard.education
  ```

  That's because the regular expression still matches the string. We haven't told it that the string must exclusively be David's e-mail address. We could rewrite the regular expression like so if we wanted to do that:

  ```
  if(!preg_match("/^malan@post.harvard.edu$/", $_POST["email"]))
  ```

  Now the string must begin and end with this regular expression. One other thing worth mentioning: the dot character actually is a placeholder for *any* character, so we're still not exclusively letting through David's e-mail. To do that, we'd have to write the following:

  ```
  if(!preg_match("/^malan@post\.harvard\.edu$/", $_POST["email"]))
  ```

  The backslash escapes the dot so that it stands for a literal dot. But now we really don't want to let through only David, so let's change it to let through anyone with a Harvard e-mail address:

  ```
  if(!preg_match("/^.*@.*harvard\.edu$/", $_POST["email"]))
  ```

  The `.*` signifies an unlimited number of any character. Of course, this is still going to let through some fake e-mail addresses, but realize there's a trade-off between how many invalid addresses you let through and how much time you spend cooking up this regular expression. If this were an enterprise program, it might be worth your while to find someone else's snippet which obeys all the standards of the RFC.

- If we wanted to be a little more specific, instead of the dot character we could define a character class which includes all characters except the `@` symbol. To do this, we'd write `[^@]`. The caret, in this context, stands for "not."

- Don't make the mistake of writing a regular expression which excludes e-mail addresses that have subdomains built in since David really hates that!

2.4 Cookies, Sessions, and SSL

- As a sidebar, know that PHP offers support for object-oriented programming (OOP) as of version 5, but in this course we won't delve too much into it since we realize that many students have no programming background and are starting with the basics. For Project 1, this might prove useful if you wanted to create an item class which would encapsulate multiple pieces of information about something on the menu.
- A cookie is temporary data stored on the clientside which is sent to the server to remind the server what data is associated with a particular user. As a programmer, you have control over the lifetime of a cookie, whether it be until the user closes his browser or even long after. The course website uses such persistent cookies to remember that you've logged in, assuming you checked the checkbox the first time you entered your password.
- We could, of course, store the username and password in a cookie on the clientside. However, this is a huge security concern because cookies are not encrypted and are stored locally on the file system. Any user who has access to that file system can read your cookies.
- Sessions are a way of temporarily storing data in memory that's associated with a particular user. PHP will hand you a variable called `$_SESSION` which is a bucket in which you can store whatever information you want. When a user that you've remembered accesses your site on separate occasions, PHP and Apache will work together to hand you the same `$_SESSION` variable.
- To use sessions, we need to call `session_start()` at the top of our PHP files.[3] When you call this function, PHP instructs the server to send an additional header named Set-Cookie. By default, the cookie's name is `PHPSESSID` and its value is a long random number. This random number is the user's identifier, which will be mapped to a file on the server that contains the contents of the `$_SESSION` variable. These files are stored in the `/tmp/` directory, by default, but can also be stored in a database or shared server space for scalability reasons.
- If a malicious user were to get a hold of the session identifier, the very long random number associated with a user, he could hijack this session, essentially impersonating another user. Anyone could do this to you while sitting in Starbuck's or any other public place that has un-encrypted WiFi. Your Facebook account is vulnerable, for instance, because most of Facebook's pages don't use SSL to encrypt traffic.
- On the course website, the login page, the bulletin board, and the grades page are all SSL-protected. Here's a snippet of code from our `.htaccess` file that helps ensure this security measure:

[3]Note, if you've defined custom classes and you want to store one such object in your session, then you need to load your class definitions before calling `session_start()`.

```
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 first few lines of code will be familiar from last week. The last chunk of code, however, asks if the user is accessing the login page on a non-SSL connection, then redirect to the SSL version of the login page.

- Sessions and cookies are obviously tremendously useful for login and authentication purposes, which we'll examine next week as we prepare for Project 2, C$75 Finance, which tasks you with implementing a stock-trading website.
- SSL certificates, unfortunately, aren't very user-friendly. The easiest to use are the ones that you can buy which instruct you to simply copy and paste a large text file into a certain directory. Then you'll need to change a few configuration options for Apache like so:

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

  Frankly, the entire SSL market is something of a scam. The fact that you have to pay for this kind of cryptographic protection is lamentable. But, for many companies, it's worth the extra money to provide paranoid users with a security blanket.
- Next week you'll be handed the specification for Project 1 along with a menu for Three Aces Pizza. You will be tasked with coming up with a data model to represent (a portion of) this menu in XML. Users must be empowered to browse this menu, add items to their shopping cart, and ultimately checkout and receive an e-mail receipt of their order. The setup for this project is that a full-fledged database would be heavy-handed for this purpose and perhaps too difficult for a layman to administer.