Section 4: SQL and PHP

About MySQL, cont.

Computer Science E-75: Building Dynamic Websites
Harvard Extension School
Spring 2008

I. WHERE DO I START?

So, you've opened up the spec, and you've done everything that David told you to do therein. But now what? Here are a few quick tips to get you going.

a. As the spec says, a user shouldn't be able to do anything on your site unless they've logged in. So, the first thing you should do is give them a way to do that. That means whipping up an XHTML form, passing it to some code, and working with a…
b. Database! As you're designing your log-in scheme, think about (and ultimately, create) the table you'll need to do this. A simple example would be creating a table called 'Users' with three columns: id, username, and password.
c. Once you've got your forms, your code, and your databases all working together to allow a user to login (and you've made sure the browser remembers that the user has logged in as they travel around your site), you can start thinking about where to go from here. Design is important for this project, so try to be organized from the start; a good way to help you stay organized is to try using a template.

II. TEMPLATES AND INCLUDES

Ever find yourself copying chunks or entire files of xhtml so that you can use the same framework for each page? Why copy, when you can include it! Even the course website uses templates to keep things simple. Let's explore this topic.

First, what are important things that every webpage needs?

1. Doctype
2. Html tags
3. Head tags
4. Title tags
5. Style tags (or an external link to a css) ---- optional, but very common
6. Body tags

We need not copy this information at the top of every single one of our pages. For example, chances are the beginning of every one of your pages looks similar to this:

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

<html xmlns="www.w3.org—xhtml">
<head>

<link rel="stylesheet" type="text/css" href="css/styles.css"
/>

<title>My Webpage!</title>

</head>
<body>
```

Notice that we have not included the </body> and </html> tags. We'll get to those later. Your specific xhtml may vary quite greatly to this, but the main point is that we've isolated the code that occurs at the top of every page. The code that will close every page will then look something like this:

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

Keep in mind these are very simple examples. As your code gets more complicated and you want to include more things on every one of your pages, these snippets will become longer and harder to deal with. Using templates allows you to simplify this all, by place your code in one place, and using require_once(); to "paste" it into every page.

Let's begin working with templates!

A Simple Tutorial

The commands described here can be used if you SSH to ns1.cs75.net. If you're
working on your own machine, note that you can still do all of this by other methods.

1. If you don't have one already, create a directory in your "project2" directory called "templates". In this subdirectory will go all the code that you use to lay out
all of your pages.

```
> mkdir templates
```

2. If you don't have one already, create another directory in your "project2" directory called "includes". In this subdirectory will go all the main code that you will include (via require_once) in most if not all of your pages.

```
> mkdir includes
```

3. Move to your "templates" directory, and create a file called begin.php. This page will house all of your xhtml that starts every page (the same stuff that was shown above). Simply copy and paste all of that into begin.php and save it. No <? ?> are necessary. Do the same for a file called end.php (only paste in the code that comes at the end of your page.

4. Now move to your "includes" directory. Create a file called functions.php and put the following code into it:

```
<?
/* void
* website_begin()
*
* Begins a page.
*/
function website_begin()
{
        // require template
        require_once("templates/begin.php");
}
/* void
* website_begin()
*
* Begins a page.
*/
function website_end()
{
        // require template
        require_once("templates/end.php");
}
?>
```

This code defines two functions "website_begin()" and "website_end()" that include all the code that you pasted into "templates/begin.php" and "templates/end.php".

5. At the top of every one of your pages, put the following code:

```
<?
// Puts our function declarations into this page
require_once("includes/functions.php");
?>
```

6. Now, whenever you call “website_begin()” or “website_end()” on your pages, the code that you required will automatically be pasted into your page. You can now write a completely valid xhtml page that looks like the following:

```
<?
// Puts our function declarations into this page
require_once(“includes/functions.php”);
// Begin the website
website_begin();
?>
<p>This is all the code that I have in my webpage!</p>
<?
// End the website
website_end();
?>
```

By calling those functions, you have “pasted” the necessary beginning and ending code into your webpage in a simple and efficient way, letting you forget about what you’ve already implemented, and focus on the actual content of each
page.

The course website uses functions like this to define our title bar across the top and the left and right columns on the page. The only part that we have to put on each page (aside from the function calls and the require_once call) is the content
found in the middle (the stuff that changes from page to page).

Feel free to experiment with the kind of code that you require. For example, you can start a table at the beginning, and put the end table code at the end, and whatever you put in the middle will appear within a table.

If you have not already experimented with require_once(), the function can also be used to copy php code into a page as well (in much the same way), so you don’t have to copy any code by hand.

III. TABLES AND DATA TYPES (that are of particular use for Project 2, hints)

So last week, we learned how to use phpmyadmin. But let’s dig a bit deeper, and maybe get a head start on Project 2.

We already discussed the need for a users table; something else we might need though, as this is a financial site, is a way to keep track of each user’s portfolio. We could add a ‘stocks’ column to the “Users” table, but we might need 5, 10 or 100 of those, and we’d be limited. As this is a dynamic website, let’s make it as dynamic as we can. We should use a separate table to keep track of *who owns what and how much.* So, a table with a simple design might include ID SYMBOL, and SHARES. Remember, David alluded to the concept of 'JOINING' tables in lecture (more on that later), and so conceptually, we need to be able to 'JOIN', or cross-reference the users table with the portfolio table. UID sounds like a good bet in terms of 'joining' your databases conceptually. As UID is likely unique (a primary key) in your users table, it makes sense to use it as the common column on your other tables.. However you choose to implement the database, keep it simple, easy to remember, and **practical.** It should be foolproof to add new stocks, add new shares, delete stocks/shares, etc. Think of implementing those features as you design your tables, and make your coding easier later on.

## IV.	USING CSV FILES IN PHP

Throughout the course of Project 2, it's possible that you might run into a need to manipulate a CSV file. If you don't want to fail, it's even more likely. Just like we learned how to use and manipulate XML and SQL, so too can we work with CSV files. Let's look:

To read a CSV file in real life, you'd follow a series of steps, right? You'd have to find the file, open the file, look at the file, put it in your brain, and then you could use the information. Well, turns out php is a lot like real life:

```
<?
$row = 1;
//FIND AND OPEN THE FILE
$handle = fopen("test.csv", "r");
//PUT IT IN YOUR BRAIN
while (($data = fgetcsv($handle)) !== FALSE) {
//START USING THE DATA
$num = count($data);
$row++;
for ($c=0; $c < $num; $c++) {
   echo $data[$c] . "<br />\n";
      }
}
fclose($handle);
?>
```

This example from php.net illustrates how to use 'fgetcsv', and a few things you can do with it. In this case, the example uses 'fopen' to open a file stored on his own server. The "r" , the second argument of 'fopen', specifies that the file is opened only for reading.

Once the file is open, we get to the really interesting part; 'fgetcsv'. Here, it is within a while loop, but don't let that distract you:

```
($data = fgetcsv($handle, 1000, ","))))
```

fgetcsv turns a file into data that can be used by PHP. Here, that usable data is stored in $data, as an array. The argument of fgetcsv is the location of the csv.

The example above, as we determined, uses a while loop. What it's ultimately doing is printing the data in each row of the CSV file. So, if test.csv was actually

> http://download.finance.yahoo.com/d/quotes.csv?s=INFXE.OB&f=sl1d1t1c1ohgv&e=.csv ,

we would see something like:

```
INFXE.OB                 0   N/A          9:10pm        N/A           N/A           N/A
```

V. RACE CONDITONS

Race conditions, simply put, are:

WIKIPEDIA: "A **race condition** or **race hazard** is a flaw in a system or process whereby the output and/or result of the process is unexpectedly and critically dependent on the sequence or timing of other events. The term originates with the idea of two signals racing each other to influence the output first."

This is a flaw because it makes both unexpected errors and exploitation very possible. As David said in lecture, someone has probably tried debiting two different ATM machines simultaneously in hopes that they could get twice as much money; we want to avoid any and all similar situations in our databases.

MyISAM: Locks

If your database uses the MyISAM storage engine (ours does by default), then the primary way of avoiding these situations is a lock. A lock simply disallows access to a part of a database while a process is taking place. For a small website like CS75 finance, this probably isn't a problem; but on a large website where millions of people could be accessing a database at one time, locks pose a threat to efficiency, and could even cause errors under extreme stress. Regardless, you may decide that for your purposes, Locks and MyISAM are a-okay. Here's a simple example from lecture that presents a basic SQL query that has been locked.

```
LOCK TABLES account WRITE;
SELECT balance FROM account WHERE number = 2;
UPDATE account SET balance = 1500 WHERE number = 2;
UNLOCK TABLES;
```

This bit of code disallows all writing to the tables being queried, and allows it again once the query has finished.

InnoDB: Transactions

InnoDB is a MySQL storage engine that is slightly less efficient than MyISAM, but has other advantages; including the ability to employ transactions.

Ex.

(Note: though the examples below use semi-colons at the end of each query, realize that you need not do so when calling the php function mysql_query.)

```
START TRANSACTION;
UPDATE account SET balance = balance -1000 WHERE number = 2;
UPDATE account SET balance = balance + 1000 WHERE number = 1;
COMMIT;
```

What a transaction is effectively doing is treating a group of SQL statements as one large statement. So, if any part of the group fails, they all fail. So, for example, if you used SQL to move $1000 from someone's Savings account (number=2) to their checking account (number=1), that's two processes. Taking $1000 from Savings, and adding $1000 to checking. For simplicity's sake, lets look at the following:

```
START TRANSACTION;
UPDATE account SET balance = balance -1000 WHERE number = 2;
UPDATE account SET balance = balance + 1000 WHERE number = 1;
SELECT balance FROM account WHERE number = 2;
# select tells us that account #2 has a negative balance!
# we'd better abort
ROLLBACK;
```

Here, we rollback at the end because one or more things went wrong. Because we were in a transaction, none of the things leading up to the point ever actually happened; so in this case, the balance of either account never really changed.

VI. PROCEDURE

Just as a quick final recap, recall that David suggested a specific order in to follow in terms of implementing features; listen to him. Here's why:

1) Log In: users shouldn't be able to do anything unless logged in, so do this first!

2) Register is less important than logging in, because you can always just add new users from phpmyadmin.

3) Get Quotes: This makes sense because our eventual goal is to buy and sell stocks; but to do that, we have to know which symbols are really stocks, and how much they're worth. Once we get that info, we can continue.

4) Sell Stocks: Again, selling stocks is easy to do first because it's so simple to add stocks via phpmyadmin.

5) Buy Stocks: Once sell is implemented, this should be much easier.

6) History: can't make a history until users can do everything, right?

7) **. . .:** Okay, hotshot—worry about adding amenities when everything else works!

Good luck!