Computer Science 75
Spring 2009
Scribe Notes

Lecture 6: March 16, 2009
Andrew Sellergren

Contents

Computer Science 75 Lecture 6: March 16, 2009
Spring 2009 Andrew Sellergren
Scribe Notes

1 Announcements (0:00–2:00)

- Grab yourself a Happy Cat!
- Bug of the Week: Bank of America showcased what was hopefully just an aesthetic bug presumably resulting from Daylight Savings Time. Can you spot the bug here?

2 Why SQL? (2:00–8:00)

- Why use a database engine? Recall several reasons we discussed: optimizing data processing, enabling access for multiple users, and enhancing security practices.
- The magic missing slide from last week! Don't worry, nothing special, just MySQL data types, which you can read more about here. Pay particular attention to the notes on storage requirements, which detail how much data can be stored in each type.
- Last week we used the `VARCHAR` data type to store a name in our database. In contrast to the `CHAR` data type, the amount of space required to store a `VARCHAR` is variable. The downside of this, as we discussed, is longer lookup time.
- What data type might you think to use to store the amount of a cash in a user's account for Project 2? Perhaps `DECIMAL` or `FLOAT`. With `DECIMAL`, you can specify the precision of the variable so that your implementation isn't prone to rounding errors.[1]

3 More on Sessions (8:00–43:00)

- Recall from Lecture 2 that login functionality can be implemented purely using PHP. We did this using sessions, as you now know. Sessions are server-side storage of arbitrary key-value pairs that are associated with users by way of cookies. When you visit our dinky little login page, the function `session_start()` is called and a very large pseudorandom number is generated to be stored in a cookie that will be sent to the user in the HTTP headers. The user's browser will then send that cookie back to the server for each request it makes. The cookie itself is stored on the server in a text file usually named `sess_*`, where * is a very large number, and usually located in the `/tmp` directory.
- Session hijacking is a security concern you should be aware of. It can occur anytime a user's session ID is compromised. Using only that session ID, a malicious user can impersonate a legitimate user.

[1] It's called salami slicing. No joke.

- What exactly is PHP doing when you store a value in the `$_SESSION` superglobal? It's *serializing* the value. That is, it's converting it to a string such that when it gets read back in by the interpreter, all of the information that was previously in memory will be restored. For example, if we were to write the following line of code:

  ```
  $_SESSION["foo"] = "bar";
  ```

  the data might be stored in the following format:

  ```
  {s: 3: "bar"}
  ```

  Basically, this says that the data is a string of length 3 containing the letters "bar." Note that this is probably not perfectly representative of how PHP serializes the data, but it's a good, instructive example. If you're ever curious how PHP stores some of your data in a cookie, feed it into the `serialize()` function and pay attention to its output. This function could even enable you to store objects as strings in your database.

- There's a special kind of cookie known as a *session cookie* that has a liftime of 0, meaning that it won't be stored at all on the server, but rather will be destroyed as soon as the browser is closed. Generally, the lifetime of a cookie is more on the order of a week.

- Question: what happens if you write an infinite loop in your PHP script? Most likely, it will be killed by the browser or the server, probably after 30 seconds or so.

- Question: what happens if you write an infinite redirect loop, either in your PHP file or in a `.htaccess` file? This one tends to get by servers more often than not, but browsers such as Firefox are now trained to catch this error.

- Question: generally, it's a good assumption that nothing in PHP will persist in RAM. This is in contrast to Java, which has the concept of shared memory space.

- You'll need to call `session_start()` at the top of every PHP file for which you want a cookie to be sent to the user via the "Set-cookie:" HTTP header. As you might've already read on the Bulletin Board, you'll get a "Headers already sent" error if you have *any* output (including whitespace and output in helper files!) before this function call. This is, of course, because by the time you're beginning to output anything to the browser, the delimiter between headers and content, a newline character, for example, has already been sent. This can create a problem even if you are executing pure PHP code before calling `session_start()`. If there's an error in your code, this error might be output to the browser and will interrupt your establishment of a session.

4 More SQL! (43:00–10:00)

- Just to get our semantics straight, be sure you recognize the difference between a database server and a database, although we might use the term "database" to refer to either. Basically, the database server is a server like any other but one which hosts multiple databases. For example, the database server you'll be using for Project 2 is `cs75.net`, but the actual database(s) will be of the naming scheme `username_*`.
- Generally, it's good practice to have separate databases for separate applications so as to explicitly control who has access to what information.
- As we did in lecture once previously, we can use DirectAdmin to create a new database and assign a new user to that database.
- When we're specifying arguments for `mysql_connect()`, there are two ways we can refer to the database server. Since `cs75.net` is both a web server and a database server, we can refer to it as `localhost` or `127.0.0.1`. This is probably best since if you're developing on your home computer using XAMPP, you'll also be specifying the hostname as `localhost` or `127.0.0.1`. If you're so inclined, however, you can refer to the course's database server as `cs75.net`.
- Developing on your home computer is a great option because you'll be able to export your entire database to a single `.sql` file and import it on the course's database server using phpMyAdmin.
- So if we've created a database called `malan_monday` and assigned a username `malan_monday` with password `passwrd` to it, our connection info will look like so:

```
// connect to database
if (($connection = mysql_connect("localhost",
                                 "malan_monday",
                                 "passwrd")) === FALSE)
    die("Could not connect to database");

// select database
if (mysql_select_db("malan_monday", $connection) === FALSE)
    die("Could not select database");
```

- Whereas previously, we were hardcoding in the values `jharvard` and `crimson` for the username and password check, now we're actually going to be doing some lookup. We need to know whether the username and password provided by the user via the `$_POST` variable are actually in our database. We'll do this by creating a SQL query like so:

```
// prepare SQL
$sql = sprintf("SELECT * FROM users WHERE user='%s'",
               mysql_real_escape_string($_POST["user"]));
```

The `sprintf()` function is way of dynamically generating strings in PHP. It takes as its first argument the base string from which the output string will be generated, including placeholders like `%s`, which specify what type of variable will be input, along with one or more other arguments specifying what variable is to be input into the string. In this case, we're filling in the username with the contents of the `$_POST` superglobal. Note that we're also escaping the user input to prevent SQL injection attacks.

- We could've used the concatenation operator instead of the `sprintf()` function. The latter tends to be a little cleaner even if it entails additional overhead from the function call.
- Question: because there's no concept of namespaces in PHP, the convention is to name related functions with a prefix that identifies them. For example, all the MySQL-related functions begin with the prefix `mysql_`.
- Why bother creating a separate variable that contains our SQL query as a string? It makes for easy debugging if we need to print it out to make sure it's well-formed.
- What are we selecting from our database? We're using the `*` operator, which means *all*, but what's the actual number we're expecting? In this case, 0 or 1 because we expect usernames, if they exist in the database, to be unique. So once we query the database, we'll be checking that an actual result was returned:

```
// execute query
$result = mysql_query($sql);
if ($result === FALSE)
    die("Could not query database");

// check whether we found a row
if (mysql_num_rows($result) == 1) {...}
```

Once we execute the SQL statement using the `mysql_query()` function, which by default will use the last database handle created by `mysql_connect()`, we can test the number of rows in our result set using the `mysql_num_rows()` function. If the number of results is exactly 1, then we have a match in our database and we can proceed to check it against the user input for password.

- We access the result set using the `mysql_fetch_assoc()` function, which returns an associative array containing the next row in our result set. Here, we've assigned that row to the `$row` variable, which we then access like any other array in PHP:

```
// fetch row
$row = mysql_fetch_assoc($result);

// check password
if ($row["pass"] == $_POST["pass"]) {...}
```

As you can see, the fields in the `$row` array will be indexed by column name since we've specified that we want an *associative* array.

- The rest of the code is actually identical to what it was before in that it redirects the user appropriately.
- What happens, though, when we try to execute our login script? We get the error "Could not query database." It should be pretty obvious at this point that we're running into problems because our newly created database doesn't actually contain any data.
- Now we'll add a `users` table to our `malan_monday` database. We'll do this as we did last week for Project 1 except we'll specify two fields named `user` and `pass`. We'll specify them as `CHAR(8)` and `NOT NULL`.
- Next to the "Extra" column for fields, you'll see a series of radio buttons with pictures above them. The one with the key symbol specifies that the field will be a `PRIMARY KEY` for the table. This means that this field will be unique for every row in the table. This has positive performance implications since we will be able to quickly lookup any row in the table using this field. For our purposes, the `user` field can act as this primary key.
- What problems might we run into if we specify the `user` field as a primary key, however? Consider that we've indicated that usernames can be up to 8 characters long. That means in order to do lookup in the `users` table, we must pattern-match 8 characters. A better solution might be to use a numeric ID as our primary key. For now, we'll keep it simple.
- Now when we try to login using `login5.php`, we no longer get the "Could not query" error. But nothing happens. Well, we obviously need to add an entry into the `users` table. Let's add `jharvard` with password `crimson` using phpMyAdmin's Insert tab. Just for reference, the form of the SQL statement that's actually executed is as follows:

  ```
  INSERT INTO users VALUES(‘jharvard‘, ‘crimson‘);
  ```

  Note that those single quotes are actually backticks (however they might appear in these notes).
- Let's leverage the SQL database engine to do the comparison of username and password. Take a look at the following lines of code from `login6.php`:

```
// prepare SQL
$sql = sprintf("SELECT 1 FROM users WHERE user='%s' AND pass='%s'",
               mysql_real_escape_string($_POST["user"]),
               mysql_real_escape_string($_POST["pass"]));

// execute query
$result = mysql_query($sql);
if ($result === FALSE)
     die("Could not query database");

// check whether we found a row
if (mysql_num_rows($result) == 1) {...}
```

Notice that now we're comparing both username and password to the fields in the database. Now, we know that if we get any results back at all from our query, it must be because both username and password matched.

- Question: what if two users have the same username? This actually isn't possible given our database schema, since the `user` field has a primary key constraint.
- What other problems arise from our database schema? As you might've guessed, the fact that we're not encrypting any of the data in our database presents a security risk. If someone gets into our database, they have a whole slew of sensitive data at their fingertips.
- So let's add some encryption. In `login7.php`, we're using the built-in SQL function called `PASSWORD`. This is a form of one-way encryption which will take the user's password and store an encrypted version of it rather than the password itself. Then, when we go to look it up in our database, we will encrypt the user's input and compare it against the stored, encrypted value. In this way, if the database is compromised, the user's passwords are not immediately compromised because the malicious user has no way of knowing what values were fed to the encryption function in the first place.
- That being said, the `PASSWORD` function is not very secure. It can be pretty easily cracked, so anyone with malicious intent and too much time on his hands might still abscond with your users' information.
- One other problem we'll run into now that we're using the `PASSWORD` function is with our 8-character length limit. What the `PASSWORD` function actually does is to convert a short password into a much longer pseudo-random sequence of characters, so in order to accommodate that length, we're going to have to change the data type of our `pass` field. To do this, simply click on the pencil icon next to the field on the Structure tab and select an appropriate data type from the dropdown menu. Perhaps `PASSWORD` would be appropriate, with a length longer than 8.

- Slightly more secure than the `PASSWORD` function is the `AES_ENCRYPT` function, which also native to MySQL. This function takes two arguments, the string to encrypt and a secret key by which the string will be encrypted. This secret key might be hardcoded, as it is in the case of `login8.php`, or it might be user-provided. For example, you might choose to use the user's password as a key to encrypt the very same password. In that way, a malicious user must know the password in order to decrypt the password. A Catch-22 of sorts. Other methods include using the username, for the secret key. Take care, however, that the method you use is ultimately deterministic. That is, it must be reproducible. If you used the current time as a secret key, for example, you wouldn't be able to match against the database the next time the user logged in.

- Note that it's hard to mount a mathematical argument which would support using the user's password as its own key, but it's a useful heuristic nonetheless. Realize, also, that you'll never be able to *recover* passwords if you use this method. Rather, you'll simply have to overwrite them and let users change them the next time they log in. DUCY?

- Question: why do banks and other sites require non-standard characters in passwords to strengthen them? If users only rely on alphabetical characters to create passwords, a brute-force method of cracking them will take less time given that there are only 26 possible characters to choose from.

- Question: how are password strength functions implemented? Most likely in a scripting language like PHP—probably not in SQL.

- `PASSWORD` and `AES_ENCRYPT` are the first of many built-in MySQL functions which we'll make use of. Others include `AVERAGE` and `SUM`, along with multiple functions that format dates and times in human-readable fashion.

- In terms of database design, how might we begin to implement the stock portfolio for Project 2? We might think to have one single table which has a user's username and password as fields along with multiple fields for the stocks they own, but what would be the problem with this approach? Obviously, we would have to decide ahead of time what the maximum number of stocks a user could own would be. In addition, any user who *didn't* own the maximum number of stocks would have a lot of wasted space associated with them in the database.

- A better solution would be to have a separate table, perhaps named `portfolio`, to hold a stock symbol and the quantity of a stock which a user owns. One of the fields in that table, then, should be a user ID or one which uniquely identifies the user in our `users` table. Conceptually, you can see that this is how we lookup data. The primary key in the

`users` table will become the means by which we identify users in all other tables. In SQL, this operation is called a `JOIN`.

- Any time you find yourself repeating data in a table, you've come across an opportunity to *normalize* your database. This is simply the process of splitting a single table into multiple tables as we've already done with the `users` and `portfolio` tables. In most cases, this will optimize your database for lookup time as well as storage space.
- What's arguably inefficient about our database design for Project 2? For each user who owns a given stock, we're repeating the stock symbol in our `portfolio` table. However, consider that the stock symbol is quite short and we wouldn't be saving much time or space by factoring stocks out into another table and identifying them by an integer. At some point, programmatic convenience comes into play when you're making database design decisions.
- Imagine, for example, that we have a company database wherein employees are identified by an ID number and employee orders are being tracked. We might maintain two different tables for employees and orders because we don't want to repeat certain information like an employee's name for every single order. For efficiency's sake, we split up this information, but when we want to access that information, we need to join it back up. The query below will accomplish just this:

```
SELECT Employees.Name, Orders.Product
FROM Employees, Orders
WHERE Employees.Employee_ID=Orders.Employee_ID
```

- The query above is what's known as an implicit join. We can also rewrite it using an explicit join:

```
SELECT Employees.Name, Orders.Product
FROM Employees
JOIN Orders ON Employees.Employee_ID=Orders.Employee_ID
```

- Let's talk about race conditions again. As an analogy, consider the situation where you and your roommate both very much enjoy the taste of milk.[2] On a given day when you've run out of milk, you leave to go to the store and buy more. While you're gone, your roommate also discovers that you're out of milk and leaves to go to a separate store to buy more milk. Now when you meet up upon returning, you have more milk than you wanted or needed.
- In this case, checking the contents of the refrigerator and refilling the refrigerator are not *atomic*. That is, they are two separate operations. The state of our variable changed in the middle of our updating it.

[2] What a mad, mad world this would be.

- If a malicious user wanted to exploit this problem in your database design, he might open two instances of your website. At almost the exact same time, he asks to buy 10 shares of Google. On the first instance, the balance check returns an amount sufficient to buy those ten shares, say, $10,000 dollars. In the middle of updating the balance to reflect the deduction from the purchase, the server switches back over to the other instance which then checks the balance of the user's account. Since it hasn't been updated by the first instance yet, this second balance check returns $10,000 dollars as well. Now, the user might get 20 stocks for the price of 10. You could imagine this same scenario occurring with two ATM's that were side by side if the bank weren't checking for this kind of race condition.

- One way to avoid this race condition in SQL is the `INSERT... ON DUPLICATE KEY UPDATE` syntax. What this says, basically, is "try to insert a row, but if there is already one there with the same primary key, then simply update the row that's already there." Because this is one SQL statement, it is, by definition, atomic.

- The other solution to the race condition problem is *transactions*. A transaction allows you to execute several SQL statements as if they were a single SQL statement. That is, atomically. Note that the default MySQL database engine, MyISAM, doesn't support transactions, so you'll need to change your engine type to something like InnoDB. The syntax for a transaction is as follows:

  ```
  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;
  # suppose account #2 has a negative balance!
  ROLLBACK;
  ```

  The `ROLLBACK` keyword, as opposed to `COMMIT` allows you to undo a transaction even after you've completed it. You would also need to specify some `CASE` where account #2 had a negative balance in the example above.

- To be clear, InnoDB implements row-level locking, which makes transactions possible. To accomplish somethign similar in MyISAM, you must use locks explicitly. The syntax for this is as follows:

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

  With InnoDB, multiple users can touch the same table at the same time assuming they're altering different rows. With MyISAM, however, the

entire table must be locked. MyISAM tends to be faster, but it is at a disadvantage for not having transactions.

- For Project 2, check out the PHP function `fgetscsv()` function!