Computer Science 75 — Lecture 5: October 19, 2009
Fall 2009 — Andrew Sellergren
Scribe Notes

Contents

1 Announcements (13:00–17:00, 53:00–56:00) **2**
1.1 Final Projects 2

2 HarvardTweets (0:00–13:00) **3**

3 Project 2 (17:00–23:00) **3**

4 SQL (23:00–53:00, 53:00–86:00) **4**
4.1 Interacting with Databases 4
4.1.1 Via the Command Line 5
4.1.2 Via DirectAdmin and phpMyAdmin 6
4.1.3 Via PHP 9

Computer Science 75 | Lecture 5: October 19, 2009
Fall 2009 | Andrew Sellergren
Scribe Notes

1 Announcements (13:00–17:00, 53:00–56:00)

- Do yourself a favor and get to work early on Project 2! These projects are designed to take 30+ hours spread out over the course of three weeks. We noticed a definitive spike in bulletin board posts, e-mails, and phone calls in the past few nights before Project 1 was due. It's no fun to be working on a project at the very last minute.
- In terms of grading, we have three axes: Correctness, Design, and Style. Our focus, however, is on qualitative rather than quantitative feedback because we feel it is more useful to you in learning. Correctness is a a matter of fulfilling all the project requirements: is your code mostly free of bugs? Did you correctly complete all the tasks described in the specification? Design is a little more subjective. Whereas you may find yourself getting 3's, 4's and 5's on Correctness at the beginning of the term, it's perfectly reasonable to find yourself getting 3's and lower on Design. As we, the staff, push you toward better Design with our guidance, you'll see these scores improve. Ultimately, improvement is the goal of the course; 3 out of 5 is **not** 60%! Finally, Style measures whether your code is well-commented and pretty-printed with meaningful variable names. Realize that Correctness is weighted more than Design which is weighted more than Style.
- Question: is there a pre-determined scale or curve? No, not at all. At the end of the course, we consider each student's work individually and look for improvement over the course of the semester.

1.1 Final Projects

- The final project specification will be posted this week—apologies that it wasn't up sooner considering that the syllabus mentioned it would be up in August! Not to worry, though, it's still weeks away.
- The CS 75 Fair is the culmination of the course and will give you an (optional) opportunity to display your final project and interact with staff and other students. It will be held on Monday, December 21 from 6:30 to 8:30 PM in one of Harvard's computer science buildings. You'll be displaying your projects alongside students from Computer Science E-7: Digital Photography, taught by our own Dan Armendariz. If for no other reason, come for the cake!
- To get you started on final project ideas, check out this list of APIs we're borrowing from CS 50. HarvardEvents, HarvardMaps, and HarvardTweets will all have their own APIs as well, with documentation forthcoming.
- The final project is meant to be a chance for you to showcase the skills you've learned in this course, but realize that you don't need to use *every* single language and concept we've discussed.

2 HarvardTweets (0:00–13:00)

- To motivate today's discussion of SQL, take a look at HarvardTweets. HarvardTweets is a PHP- and MySQL-driven site that aggregates posts from Harvard Twitter users. David managed to write this while sitting in his hotel room in Virginia and by the end of the course, you will have all the skills necessary to do the same. Even the tag cloud in the bottom right is just an open-source Flash animation which is fed with XML. Lots of fancy-looking content but nothing very complicated underneath the hood!

- Thankfully, Twitter has an API (application programming interface), which is a set of functions and documentation that allow a programmer to interact with Twitter's back end. If not for the API, David would've had to resort to screen-scraping, meaning he would have to parse the messy HTML on Twitter's actual website in order to gather the data he wanted. Twitter has a REST (representational state transfer) API, which is to say that you can query its server for data by hitting a URL that includes a `GET` string with certain parameters defined. For example, you can specify that you want JSON or Atom output returned by the server.

- Once we've queried Twitter's servers for Harvard users' posts, we can iterate over them and insert them into our page. But we can also store them in our own database so that they can be cached locally and we won't have to hit Twitter's server every time a user searches our page. This, of course, involved some forethought as to what data would need to be stored and *how* it would be stored (i.e. database design). Frankly, this can be a fun exercise!

- Question: if the posts are cached locally, then won't posts that have been deleted continue to show up on HarvardTweets? Yes, but this is intentional. Twitter's servers have a limit to how frequently they can be hit and gathering this past data would risk surpassing this limit.

- Question: how is the site updated? Every 5 minutes or so, a `cron` job executes on the server. `cron` is a Linux utility which allows commands to be scheduled to run automatically. This particular `cron` job executes `update.php` which then queries the database for a list of usernames, hits Twitter's servers to retrieve those users' Tweets, and then populates the page with them.

3 Project 2 (17:00–23:00)

- Project 2 will task you with implementing a database-driven stock-trading website which empowers users to register as well as buy and sell (or at least simulate buying and selling) stocks. Databases aren't a new concept in this course, however. We've already interacted with CSV and XML databases.

- On the course's homepage is The Big Board, which is a game based on the staff's implementation of C$75 Finance. If you click play the BIG BOARD, you'll be able to test your financial savvy against the other students in the course. How much money can you make given $10,000 (fake) dollars?

- The C$75 Finance data is retrieved from Yahoo in CSV format. You need only make a request to Yahoo's servers with certain parameters defined in a `GET` string and you'll be handed back data which you can then easily parse.

- If we take a look at the CSV file that's returned if we look up GOOG, we see that it's just a single row of data. This file can actually be opened in Excel and each piece of data will display in its own column. We can, in fact, query for multiple rows at once if we use the correct parameters in our `GET` string. Let's take a look at an example of this `GET` string:

  ```
  quotes.csv?s=GOOG&f=sl1d1t1c1ohgv&e=.csv
  ```

  Although the file is named `quotes.csv`, we know that there's actually a script running in the background. The `s` parameter is almost certainly the stock symbol. The `f` parameter is a little more cryptic and specifies what data we want to get back from Yahoo (see here for the unofficial definition of tag). Finally, the `e` parameter specifies the file format of the data being returned.

4 SQL (23:00–53:00, 53:00–86:00)

4.1 Interacting with Databases

- For Project 1, our database was implemented in XML and for all practical purposes, it was static. Theoretically, the menu could've been changed with a simple text editor, but if we had taken a little more time, we might've been able to design a front-end interface that could've outputted XML using SimpleXML. As for customers' orders, we were content to "store" them as e-mail. This is reasonable when there isn't time or desire to implement more persistent, machine-readable data storage.

- In terms of complexity, an intermediate between XML and SQL proper is SQLite, which allows you to interact with data via SQL statements but which is ultimately implemented in the current working directory as a binary file. This is useful when a full-fledged SQL database would really be overkill or when there's a desire to package an entire application in a single directory without the need for a database username, password, etc.

- CSV can also be used to store data persistently, although representing complex objects and data as a series of columns and rows can often be a difficult task. Even relational databases have this shortcoming to a certain extent.

- The relational database engine we will work with is MySQL. Although there are certainly valid arguments for other engines, we will opt for MySQL because it's high-performing, well-documented, popular (i.e. well-supported), and, perhaps most importantly, free. Other database engines include PostgreSQL, SQL Server, and Oracle. For the most part, all these database engines use the same language, SQL (structured query language), albeit with slight variations, to interact with the underlying data.

- What is a relational database? You can think of it like an Excel spreadsheet which has multiple worksheets, each of which represents a table in our database. A table is simply data stored as rows and columns. Relational database design boils down to deciding what columns should be named and what data types those columns should store as well as how to divide data among tables. You might think it would be simplest to store all your data in a single table, but these will inevitably lead to redundancy. For example, consider an address book stored in a relational database. If you have a lot of friends who live in the Cambridge area, the city and state information is going to be repeated row after row after row. The process of *normalization* is meant to reduce this redundancy.

- The advantage of a SQL database over a CSV or XML database is that the SQL database has an engine backing it up. This engine is actually a program running and listening for connections on port 3306. Thus, data can be retrieved more quickly because it's not necessarily in files on the hard drive, but may in fact already be loaded up in RAM.

- For Project 2, you are welcome and encouraged to develop on your home computer, but realize that you won't be able to connect remotely to `cs75.net`'s database server. By default, MySQL traffic is unencrypted, so for security purposes, we block connections on port 3306. You'll need to create your database on your home computer and then export it to `cs75.net`.

4.1.1 Via the Command Line

- To interact with the MySQL database engine, we can run the `mysql` command from the command line. We'll need to use the `-u` and `-p` to specify our username and password in order to connect. We'll then be presented with a command prompt at which we can type SQL commands. If we type `SHOW DATABASES;`, being careful to end with a semicolon, the command terminator, we'll get a list of our databases printed to stdout. Two databases which will appear by default are `information_schema`, which is best to leave alone, and `test`, which you can play around with. Everything else is a database which was created with the `CREATE DATABASE` command.

- If we type `USE malan_lecture`[1] followed by `SHOW TABLES`, we can see all the tables for our `malan_lecture` database. Finally, we can type `DESCRIBE users` to see a description of the `users` table. We'll see that we have two columns or fields, `user` and `pass`, both of which have a type of `VARCHAR(255)`, meaning they consist of a variable number of characters up to 255. Neither of them can be `NULL` and the `user` field is a primary key for the table, which we'll discuss later.

- Now we can type `SELECT user FROM users` to retrieve a list of all the usernames in our `users` table. We see that there's only one, `jharvard`. If we want to add another username `malan`, we could execute the following command:

  ```
  INSERT INTO users (user,pass) VALUES ('malan', '12345');
  ```

  To see what effect this had on our table, we can retrieve all the rows from our `users` table using the `*` wildcard syntax:

  ```
  SELECT * FROM users;
  ```

  We see that another row has been added with `malan` as `user` and `12345` as `pass`. We can select only this row using the following syntax:

  ```
  SELECT * FROM users WHERE user='malan';
  ```

  As you can see, SQL is fairly accessible. If we want to filter our results, we generally just add a predicate to our query using the `WHERE` keyword.

4.1.2 Via DirectAdmin and phpMyAdmin

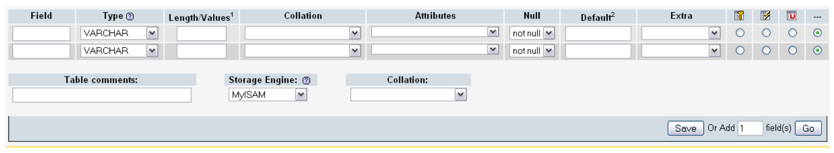
- The command line, of course, is not the only way to interact with MySQL. We'll make use of a GUI named phpMyAdmin as well as a PHP interface. We can also create new databases via the panel. Know that while it's partly just semantics, the database engine is the actual software running in the background which manages connections and actually retrieves data whereas a database itself is a piece of memory in which the data is stored. In our previous analogy, an Excel spreadsheet represents a database whereas Excel itself might represent the database engine.

- To create a database on `cs75.net`, we'll need to login to the panel and click on MySQL Management followed by Create new Database. Once there, we'll be given the option to name a database with the prefix `username_`. This is a DirectAdmin convention in order to segregate users' databases. On your own database server, you can name your databases whatever you want.

[1]This would be a good type to mention that SQL commands are case-insensitive (though table names aren't necessarily), but by convention, SQL keywords are written in all-caps.

- Once we've created a database, we can choose to interact with it via phpMyAdmin. Realize that the "php" in phpMyAdmin is purely coincidental: it just happens to be implemented in a language that we're studying extensively in this course. phpMyAdmin is one of the most useful, well-designed open-source applications out there. It takes the hassle out of interacting with MySQL. If you forget what data types MySQL supports, for example, you'll find a convenient dropdown menu to remind you. That being said, the one annoying aspect of phpMyAdmin is its cookie handling. If you leave an instance of it running for an hour or so, the session will time out but it won't be clear that it did and you might observe some strange behavior. Just clear your cache or restart your browser and it should take care of it.

- Let's say we want to recreate the `users` table we saw in our `malan_lecture` database. Once we click the create table link and give our table a name, specifying that it will have two fields, we'll be directed to a screen that looks something like this:



Field	Type	Length/Values[1]	Collation	Attributes	Null	Default[2]	Extra				---
	VARCHAR				not null			○	○	○	●
	VARCHAR				not null			○	○	○	●

Table comments:	Storage Engine:	Collation:
	MyISAM	

We'll name the fields `user` and `pass`, just like in our previous example. And, likewise, we'll specify the default data type, `VARCHAR` with a maximum length of 255. The `VARCHAR` data type only stores as many bytes as are necessary, so as not to waste space. You pay a slight performance penalty, as a result, however. The Collation menu has to do with character encoding—by default, it is Swedish because the original designers of MySQL were Swedish, but for the most part you don't need to worry about this. Attributes is not applicable for string data types, but if we were using an `INT` or a `TIMESTAMP`, we could tweak them here. Null specifies whether a field can have the value of `NULL` or not—in other words, do we want this field to be allowed to be blank? If you specify `NOT NULL`, then `INSERT` operations will fail if this field isn't specified. That can be a useful way of doing error-checking at the database layer. Default allows us to specify a default value for this field. Under Extras, we find `AUTO_INCREMENT` which automatically assigns the next available smallest number—in this way, we can maintain a unique ID column without manually keeping track of the number. Next, there are a series of radio buttons and checkboxes which allow us to specify primary keys, indexes, unique keys, and fulltext search columns. More about those next time. Finally, below these two field definitions, there's a dropdown menu for Storage Engine. The two we'll discuss in this course are MyISAM and InnoDB. For now, just know that MyISAM is higher-performing, but doesn't support transactions. We might also specify MEMORY as our Storage Engine if we want all of our database to be stored in RAM. As you can guess, this will be very fast!

- When we click Save, our table will be created and, conveniently, the SQL syntax that was used to create it will be displayed for us. In this way, we can use phpMyAdmin as a learning tool for SQL.
- Question: why 255? Historical reasons and nothing more. This used to be the default value for `VARCHAR`, so it's still pretty common. Something like 8 or 16 would probably work just as well.
- The basic list of SQL statements which can be executed are as follows:
  - `CREATE`
  - `ALTER`
  - `DROP`
  - `SELECT`
  - `INSERT`
  - `UPDATE`
  - `DELETE`
- The MySQL Documentation is not nearly as user-friendly as that of PHP, but it is quite thorough. Quite frankly, Google and the Resources page will be your friend if you ever have questions on MySQL usage. Note that we use version 5.0 of MySQL, although versions 5.1 and 5.4 are now available. Mostly, this is because of DirectAdmin, which doesn't allow installation utilities like `yum` to have free reign. Also, there's not really any features of 5.1 that you'll need to complete your coursework. Likewise, we are using PHP 5.1 rather than the newest version, 5.2.6.
- Database engines can speed up access to data not only by loading data into RAM but also by optimizing tables for common operations. For example, if a particular field in a table is one that you are likely to search on over and over again, you can add an index to it so that the database engine will implement something like a hash table in order to achieve constant time lookup on that field. If you decide later that a field is one that you will search on frequently, you can always add an index on it after the table is created.
- Just to make clear the connection between the command line and phpMyAdmin, the `SELECT *` statement is equivalent to the Browse tab and the `DESCRIBE` statement is equivalent to the Structure tab for a given table.

4.1.3 Via PHP

- Taking a look at `login5.php`, we see our first example of interacting with MySQL via PHP:

```
<?
    /**
     * login5.php
     *
     * A simple login module that checks a username and password
     * against a MySQL table with no encryption.
     *
     * David J. Malan
     * Computer Science E-75
     * Harvard Extension School
     */

    // enable sessions
    session_start();

    // connect to database
    if (($connection =
        mysql_connect("localhost", "malan_fall2009", "12345")) === FALSE)
        die("Could not connect to database");

    // select database
    if (mysql_select_db("malan_fall2009", $connection) === FALSE)
        die("Could not select database");

    // if username and password were submitted, check them
    if (isset($_POST["user"]) && isset($_POST["pass"]))
    {
        // prepare SQL
        $sql = sprintf("SELECT * FROM users WHERE user='%s'",
                       mysql_real_escape_string($_POST["user"]));

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

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

                    // redirect user to home page, using absolute path, per
                    // 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;
                }
            }
        }
    ?>

    <!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="<? echo $_SERVER["PHP_SELF"]; ?>" method="post">
          <table>
            <tr>
              <td>Username:</td>
              <td>
                <input name="user" type="text" /></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>
```

Here, we're hardcoding in our database login credentials, but better practice would be to abstract them away into a separate config file. Note that `die()` is not a very elegant way of informing users that an error has occurred, but in this context, we're focusing on the interaction with MySQL. Once we've connected to MySQL using the `mysql_connect()` function, we can select a database using the `mysql_select_db()` function. Pretty straightforward. This API is actually rather old, but the newer ones, namely `MySQLi` and `PDO`, have a steeper learning curve. For your final projects, feel free to use whichever API you are most comfortable with.

- Recall the logic from weeks ago whereby if the `POST` variables aren't set, then the login page is being visited for the first time, so the form will be displayed. If the variables *are* set, then we need to check them against what's stored in our database. To do this, we'll prepare a SQL string using `sprintf()` to fill in placeholders with values. In this case, we'll be plugging in the escaped username into the SQL string where the placeholder `%s` is. Escaping data that is plugged into SQL queries is important to protect against SQL injection attacks. We don't want users to be able to insert characters like single quotes or keywords like `DELETE` into our query. More on this next time.

- Once we've prepared our SQL string by plugging in escaped user input, we pass it to `mysql_query()`. `mysql_query()` will then return a result set or MySQL resource, an object containing all the rows of data that we asked for. We'll then need to ask for one row at a time from this object until it returns false, at which point we know there are no more rows left.

- First, we do a sanity check: if `mysql_query()` returns false, it's probably because of a syntax error in our SQL query. If the return value is not false, then we're checking if the number of rows equals 1. If the number of rows equals 0, then the user isn't registered. If it's more than one, we also have a problem. This won't ever be the case if we've established username as a primary key on our table. In the future, however, we'll be using an `INT` as the primary key on our users table.

- If the number of rows is 1 (i.e. we've found the user), we need to check the password. We call `mysql_fetch_assoc()` to retrieve the single row from our result set. That row will be an associative array, so we can access the password field using bracket notation. If the password field is equal to what the user typed in, then we log the user in by setting an index in the `$_SESSION` variable. Finally, we redirect the user.

- Question: what does `mysql_real_escape_string()` actually do? It's going to escape single quotes which will prevent a malicious user from attempting to end our SQL string and then tack on his own predicate. More on this next week.

- Even here, though, we're relying on PHP to check the password for us. Why not ask SQL to do this for us? We do just that in `login6.php`:

```
<?
    /**
     * login6.php
     *
     * A simple login module that checks a username and password
     * against a MySQL table with no encryption by asking for a binary answer.
     *
     * David J. Malan
     * Computer Science E-75
     * Harvard Extension School
     */

    // enable sessions
    session_start();

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

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

    // if username and password were submitted, check them
    if (isset($_POST["user"]) && isset($_POST["pass"]))
    {
        // 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)
        {
            // remember that user's logged in
            $_SESSION["authenticated"] = TRUE;

            // redirect user to home page, using absolute path, per
            // 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;
        }
    }
?>

<!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="<? echo $_SERVER["PHP_SELF"]; ?>" method="post">
      <table>
        <tr>
          <td>Username:</td>
          <td>
            <input name="user" type="text" /></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>
```

Here, we're filtering our result set based on the user-provided username and password. We know that if any results are returned at all (in this case, the result will simply be the number 1), it's because both the username and password were matched in the database.