Section 3: SQL

About MySQL

SQL is a standard to which you can issue queries to retrieve or update data in its database. There exist several implementations, but the one we are using in this course is MySQL, an open source SQL implementation. MySQL tends to be a popular choice because, among other things, it is free and it is quite easy to retrieve or update the data with PHP.

http://www.mysql.com/

SQL is an example of a relational database, which means that the data is essentially stored as a table with rows and columns. As mentioned in lecture, a good analogy would be that of an Excel file.

Table Structure

Data is stored in a relational database in rows and columns within tables. Table structure is defined by creating a series of columns (also called *fields*) that define the type of data to be stored. The data itself is stored in rows.

Here is an example of a table, but note that this is not necessarily how SQL stores the data in its database but it is easiest to think about the data being stored in this manner.

HUID	First_name	Last_name
12345678	John	Harvard
87654321	Jane	Harvard

There are a number of data types that can be used to define a column. Many popular ones include:

INT (Integer): any negative or positive integer from -2147483648 to 2147483647 (for signed integers) or positive integers from 0 to 4294967295 (for an unsigned integer).

DECIMAL: Any decimal number up to 65 digits long (with a maximum of 30 digits after the decimal point). Unless you are working with extremely precise decimals, this data type will work with any decimal value.

DATETIME: Any date and time in the format "`YYYY-MM-DD HH:MM:SS`".

VARCHAR: a string up to 65,535 characters long.

There are many more data types available, but most are variations of the above (such as BIGINT, which would allow you to store integers that are considerably larger than just an integer). Please see the MySQL documentation for more information:

http://dev.mysql.com/doc/refman/5.0/en/data-types.html

Now that you have a better idea of some of the available data types, let's revisit that table above but declare data types for each column:

HUID VARCHAR(8)	First_name VARCHAR(25)	Last_name VARCHAR(50)
12345678	John	Harvard
87654321	Jane	Harvard

We have established that the data type for the First_name column will be a VARCHAR of length 25. This means that the string in the second column can be no longer than 25 characters long. Similarly, we've defined Last_name to be a VARCHAR with a maximum length of 50 characters. HUIDs are 8 digit integers but they can have leading zeroes (which might cause problems since leading zeroes are omitted in the INT data type), so we've defined the HUID to be a VARCHAR of length 8.

You do not have to explicitly define lengths for some data types. However, it's safest and is generally good practice to do so whenever possible to ensure that the table is optimized.

Creating a MySQL Database

You are allowed to create 6 databases (and an infinite number of tables within each of those databases) as part of your cs75.net account. You will use one database for each of the remaining projects and you may create others for testing or for fun.

Realize that MySQL is a separate service that runs on our server much like Apache or suPHP. In order for you to use the service, you must establish a connection to it, login with a username and password, and issue commands to it. Please note that the username and password are separate from your cs75.net login information, and you will create a username and password when you create your first database.

To do that, log in to the panel at:

http://panel.cs75.net/

And click on the “MySQL Management” link. On the page that appears, click the “Create new Database” link. Fill out the fields including the new database name and the username and password you will use to login to the MySQL server and click “create”. Feel free to create a test database to follow along with these notes.

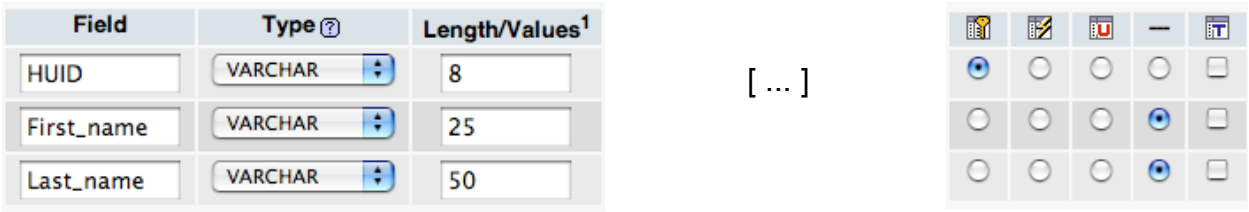
phpMyAdmin

phpMyAdmin is a PHP program that allows for MySQL database administration, table creation, verifying and updating the information in existing tables, and many other things. Login to phpMyAdmin by going to this site:

http://cs75.net/phpmyadmin/

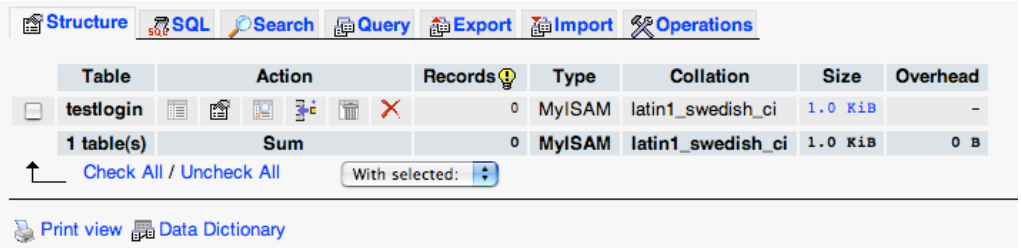
And enter in the username and password you created from MySQL Management from cs75.net's Panel. Once you are logged in, notice that the databases associated with the username are along the left navigation column. Click on one, and if you have not created a table in that database you will be prompted to do so. Enter a name for the table (we'll name the one above “login”), and set how many fields

(or columns) you would like. In the case of the table we're using as an example above, you can name the table “testlogin” with 3 fields. The next screen allows you to name and define the data types for each field. From the above table, the first field is “HUID” with a data type of “VARCHAR” and a length of 8. Click the bubble in the HUID row under the key icon. For First_name, fill in the field name, set the data type to “VARCHAR” and the length to 25. Finally, for Last_name, fill in the field name and set the data type to “VARCHAR” with a length of 50. The image below shows how it should look:



The key icon defines a “Primary key”. This allows the data in the table to be sorted and indexed to maximize efficiency when searching. You don't have to set a primary key, but phpMyAdmin will give you warnings if you don't. By convention, the primary key is the first field in a table. We will talk more about keys and indexes in future lectures and sections.

The remainder of the fields you can leave at the defaults. Once you have finished defining all of the fields, click the “Save” button to create the table. You may create additional tables in a database as necessary, but for now you should see something similar to the below:



Do not be alarmed that the collation says “Latin1_swedish_ci”! This is the default collation for a MySQL install and it includes all English characters as well. The “Records” field indicates how many rows of data your table contains. Since it's a fresh table, 0 records makes sense.

From this screen you can perform a variety of tasks on your databases and tables. The icons under the “Action” column are, from left to right:

Browse: This allows you to browse the records in your table and is only active when you have some records to browse.
Structure: Modify the structure of your table by adding, deleting, or modifying fields.
Search: Search records in the table. This is only active when there are records to search.
Insert: Manually add records to your table.
Empty: Which erases all records from the table.
Drop: Which erases all records from the table and the table itself (*ie*, the structure will be destroyed).

Notice the tabs along the top of the page. The current one is “Structure” but there is also a “SQL” tab. From here you can issue SQL statements that allow you to perform all of the functions phpMyAdmin offers, or it allows you to test a SQL statement before implementing it in your PHP. But let's not get ahead of ourselves!

Issuing SQL Statements

Data stored in tables within a database is not very useful unless we are able to retrieve and update it as necessary. SQL provides a robust language to interface with the database software so that you will be able to do just that. You can issue SQL statements via a program from the command line, from phpMyAdmin, or, as we will soon see, from your own PHP scripts.

Let's look at a very simple SQL statement:

```
SELECT * FROM testlogin;
```

This statement would do just as it implies: It would select all rows and fields from the testlogin table. If you are following along and attempt to issue this SQL statement after you had first created a table, you would get 0 rows returned because it has no records. You may want to insert some fields using phpMyAdmin before continuing. The two records from the sample table above should suffice!

Moving on, we can select certain fields like so:

```
SELECT HUID FROM testlogin;
```

This would return all HUIDs in the table. You can even select multiple fields:

```
SELECT HUID,First_name FROM testlogin;
```

To return all First_names and HUIDs in the table. However, we frequently don't want ALL of the records in the table and want to narrow it down. Let's say you are provided an HUID of 12345678 and you want to look up all of the fields that match that record. The following statement would accomplish this:

```
SELECT * FROM testlogin WHERE HUID='12345678';
```

Realize that by adding the WHERE condition, MySQL returns the row (or rows) that match the condition. Executing the above statement against a table with the rows from the table example above would return 1 row: John Harvard's. If the table did not contain a record that matches the condition, it would not return an error. Instead, it will return 0 rows. This will be especially useful for us when we perform queries from PHP!

What if, instead, we ran this statement:

```
SELECT * FROM testlogin WHERE Last_name='Harvard';
```

Note that this would return two rows! This is because the last name matches both records, so both are returned. If you'd like, you can run more complex logic in a SQL query:

```
SELECT * FROM testlogin WHERE Last_name='Harvard' AND HUID='12345678';
```

This would return, again, one row containing John Harvard's information. This is somewhat unnecessary in this example because we can return John's record without needing his last name, but you can imagine situations where more complex logic might be useful. If you instead had a table of students enrolled in a course, you might want to find all students that are enrolled in a course and have the last name of “Harvard”.

As one more example from our sample table a few pages ago, let's say we would like to return only the first name of all students whose HUID is NOT 12345678:

```
SELECT First_name FROM testlogin WHERE HUID<>'12345678';
```

Note that you can also use != (instead of <>) to test for inequalities. Frequently you will need to add records using a SQL statement. The syntax for insertions are:

```
INSERT INTO testlogin (HUID, First_name, Last_name) VALUES ('12345679', 'Joe', 'Harvard');
```

This would, as you can imagine, insert Joe Harvard with an HUID of 12345679 into the testlogin table. More generally, the syntax is the below:

```
INSERT INTO tablename (Field1, Field2, ...) VALUES ('Value1', 'Value2', ...);
```

To avoid errors it is a good idea to list all of the fields in the first parenthesis and assign some value for each in the second parenthesis.

SQL in PHP

Of course, the whole point of this exercise is to be able to access and manipulate the data contained within a SQL database from your PHP scripts. The wonderful thing about PHP is that it already includes well-documented functions specifically designed to communicate with MySQL servers. Be sure to check out the PHP site on MySQL functions:

```
http://us2.php.net/mysql
```

Since MySQL is a service, the very first thing you must do is connect to it from your PHP script:

```
if (($connection = mysql_connect("localhost", "username", "password")) === FALSE)
       die("Could not connect to MySQL");
```

Note that we use "localhost" because the MySQL service is on the same server as the PHP script. Replace username and password with your MySQL username and password. We use the if statement to verify that the connection was completed, otherwise it will cause the script to fail with an error message.

Next, you must tell SQL which database you would like to access. That can be accomplished this way:

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

Change "database_name" to the name of your own database. Again, we force the script to exit if we cannot select the database. If you get this failure, be sure that the database name is correct!

Finally, we can issue a SQL statement to the SQL database. We accomplish this with the `mysql_query()` function:

```
$query = "SELECT * FROM testlogin WHERE HUID='12345678'";
$result = mysql_query($query);
if ($result === FALSE)
       die("Could not query database");
```

You can build queries with any of the string tricks that you've learned: you may use concatenation, sprintf, or double quotes to insert a variable into the string. Remember from the "Issuing SQL Statements" section that SQL does NOT return false or an error if the SQL statement executed and it found no matching rows. It will simply return 0 rows as a result (which is not an error). This is why we test to see if the result is equal to false, this implies that there was an actual error performing the query.

Here is an example of inserting a variable into a SQL statement:

```
$huid = 12345678;
$query = sprintf("SELECT * FROM testlogin WHERE HUID='%s'",
             mysql_real_escape_string($huid));
[...]
```

Using `mysql_real_escape_string()` is a good idea so that a malicious user cannot hijack the SQL query and do something bad to your tables!

Now that the query has been performed, we need to be able to obtain the data from it. First, you may want to check to see how many rows of data were returned:

```
$number_of_rows = mysql_num_rows($result);
```

From the SQL query we performed above against our example table, we would expect to receive one row back. Therefore, if `$number_of_rows` is equal to 1, the user is found in our database. If equal to 0, the user is not found. It's possible in other implementations to have more than one row returned but it doesn't make sense in this case where we expect everyone to have unique HUIDs.

We can also obtain the data that was returned in the row of fields from the query:

```
$row = mysql_fetch_assoc($result);
echo 'First name: ' . $row['First_name'] . "<br />";
echo 'Last name: ' . $row['Last_name'] . "<br />";
echo 'HUID: ' . $row['HUID'];
```

The mysql_fetch_assoc() function returns an associative array from which you can reference the field names to retrieve the data. The above code would therefore return:

```
First name: John
Last name: Harvard
HUID: 12345678
```

If you expect multiple rows from a query, you must iterate through them with a loop:

```
$query = "SELECT HUID FROM testlogin";
$result = mysql_query($query);
if ($result === FALSE)
       die("Could not query database");

echo "List of HUIDs in database: ";
while ($row = mysql_fetch_array($result)) {
       echo $row['HUID'] . " ";
}
```

Assuming the table is populated just like the example table above, this code would return:

```
List of HUIDs in database: 12345678 87654321
```

When you are done with the database, you should free the result and close the MySQL connection:

```
mysql_free_result($result);
mysql_close($connection);
```

So, to put it all together, some PHP code to output a list of HUIDs from our test table would look like this:

```
if (($connection = mysql_connect("localhost", "username", "password")) === FALSE)
	die("Could not connect to MySQL");

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

$query = "SELECT HUID FROM testlogin";
$result = mysql_query($query);
if ($result === FALSE)
	die("Could not query database");

$number_of_rows = mysql_num_rows($result);

if($number_of_rows == 0)
	echo "Hm, the table is empty? No results found.";
else {
	echo "List of HUIDs in database: ";
	while ($row = mysql_fetch_array($result)) {
		echo $row['HUID'] . " ";
	}
}

mysql_free_result($result);
mysql_close($connection);
```

And this code would output (with a complete disregard for proper XHTML):

```
List of HUIDs in database: 12345678 87654321
```