Week 8, continued

This is CS50. Harvard University. Fall 2014.

Cheng Gong

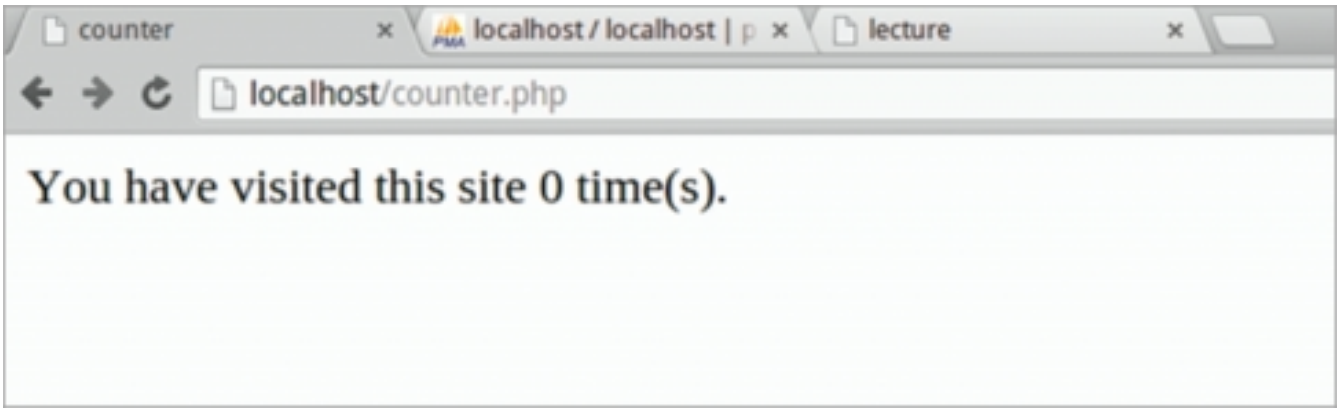
Table of Contents

Superglobals and Cookies

- The web server in Problem Set 6 demonstrates writing software that knows how to take HTTP requests from a browser (or a human with a program called `telnet`), and respond to those requests with an HTML, JPG, or PHP file.
- But a web server shouldn't return a raw PHP file, but interpret it first.
    # It notices that it ends in `.php`, so it interprets it line by line, using a pre-existing program, PHP (which happens to have the same name as the language it interprets).
- We took care of that part for you in Problem Set 6, so all you had to do was support static content, among other things.
- In Problem Set 7, we'll start to write the PHP code that gets interpreted, talking to the back-end database that stores our information.
- Remember from last time that we have these **superglobals**:
    # $_COOKIE
    # $_GET
    # $_POST
    # $_SERVER
    # $_SESSION

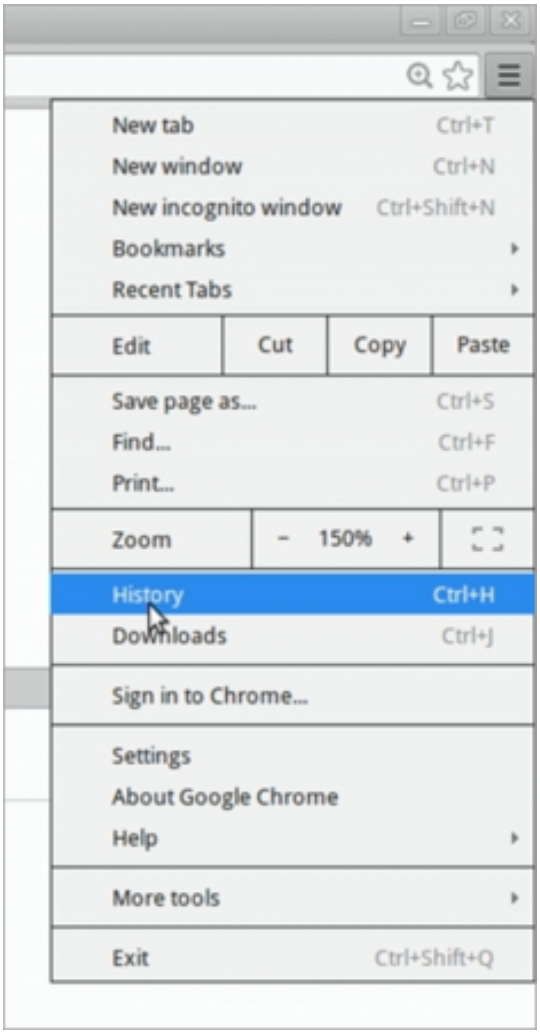
```
# …
```

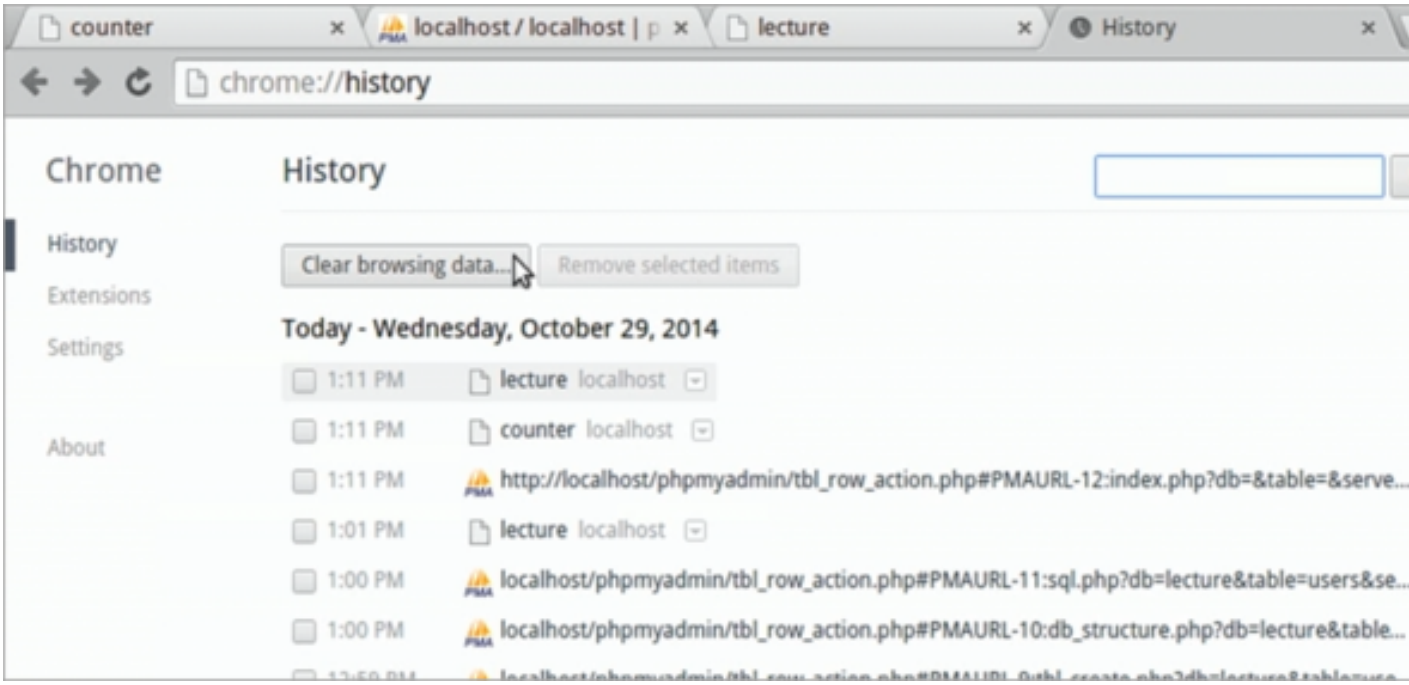
- On Monday we used `$_GET`, which had all the parameters we put in the **query string**. When we implemented our own version of Google, the URL had a question mark and then `q=` (like `https://www.google.com/search?q=cats`). The question mark indicated the start of the query string `q=cats`, and `$_GET` would automatically have a key named `q` with the value `cats`.
- All of these superglobals are associative arrays, or hash tables that store keys and values.
- In Problem Set 5, the hash table or trie you implemented was really an associative array where the keys were associated with values of true or false, whether the word was in the dictionary or not.
- But we can associate more interesting values with keys, and return arbitrary strings, which is what `$_GET` and these other variables allow us to do.
- `$_POST` stores forms sent by the POST method, and files are actually stored in a variable not listed, `$_FILES`[1].
- `$_SERVER`[2] gives you details about the server, which we won't go into detail about.
- `$_COOKIE` and `$_SESSION` are what we need to implement things like a simple shopping cart.
- Last time we had an example of a `counter.php` file that counted how many times we visited the page:

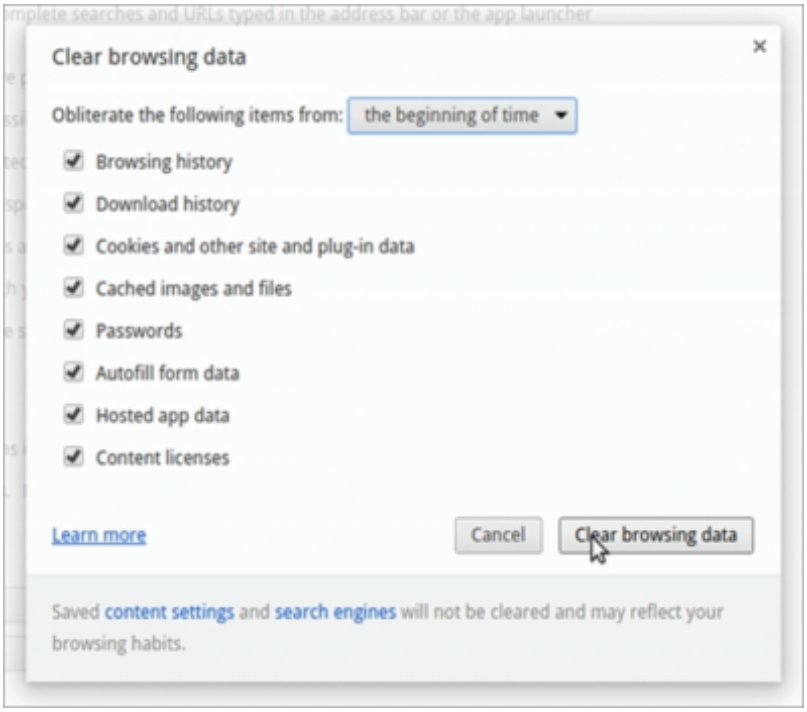


- We can open Chrome's **Developer Tools**, but first clear the cache:

[1] http://php.net/manual/en/reserved.variables.files.php
[2] http://php.net/manual/en/reserved.variables.server.php


New tab Ctrl+T
New window Ctrl+N
New incognito window Ctrl+Shift+N
Bookmarks
Recent Tabs
Edit
Cut
Copy
Paste
Save page as... Ctrl+S
Find... Ctrl+F
Print... Ctrl+P
Zoom
150%
History Ctrl+H
Downloads Ctrl+J
Sign in to Chrome...
Settings
About Google Chrome
Help
More tools
Exit Ctrl+Shift+Q

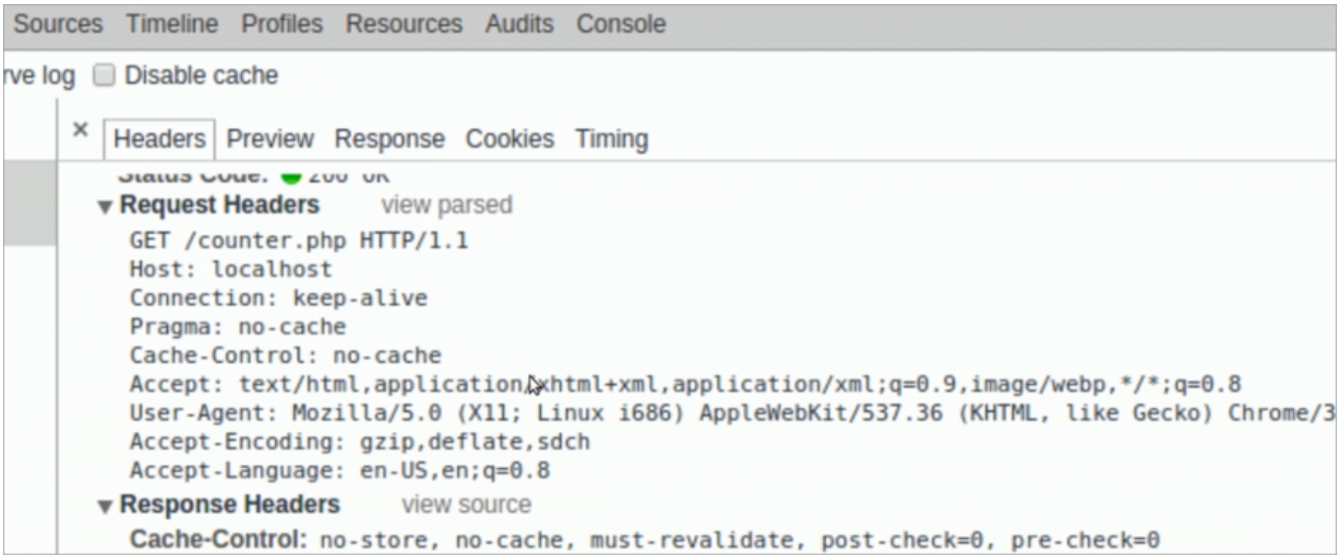




- We'll do this for debugging purposes, getting rid of **cookies**, a piece of data that a server asks to put on your computer, to remember who you are.
- After you log into a website like Gmail or Facebook, the server remembers that you are logged in, even though you're not typing in your username and password at every

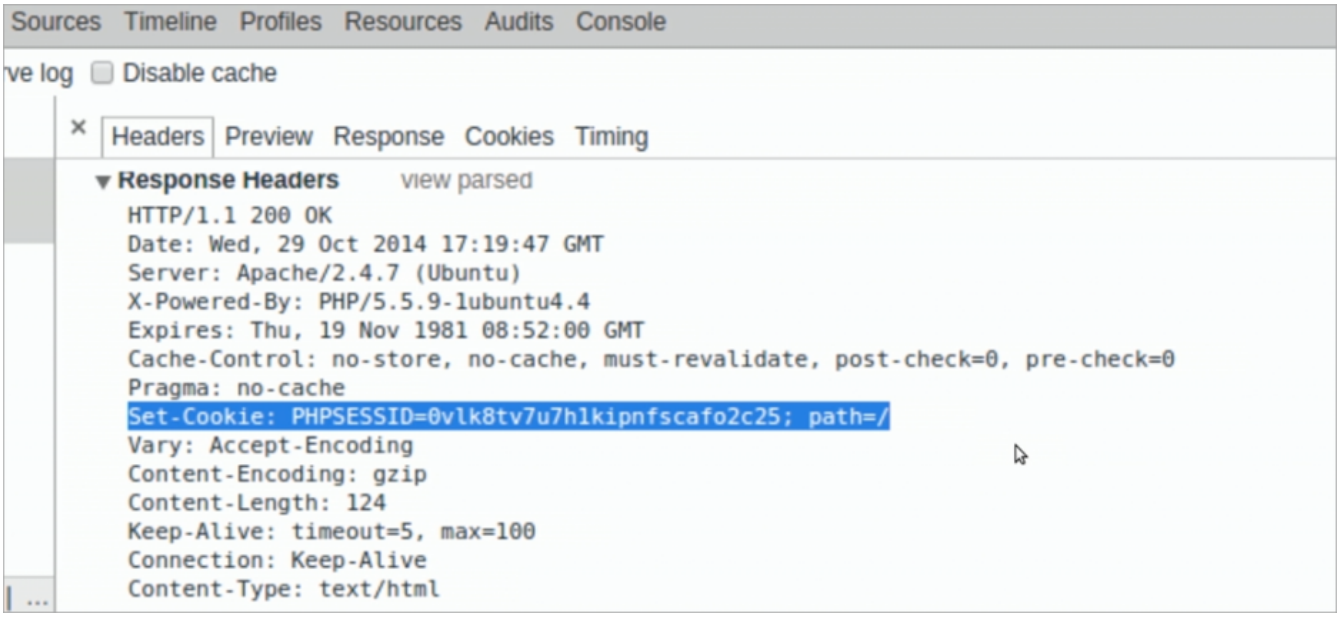
page. Cookies are the answer. They are like a digital handstamp that you might get at an amusement park or club, that indicates you've already showed your ID, and have been identified already.

- We can reload `counter.php` and view the request as we've done before (clicking **view source** next to Request Headers to see the raw request):

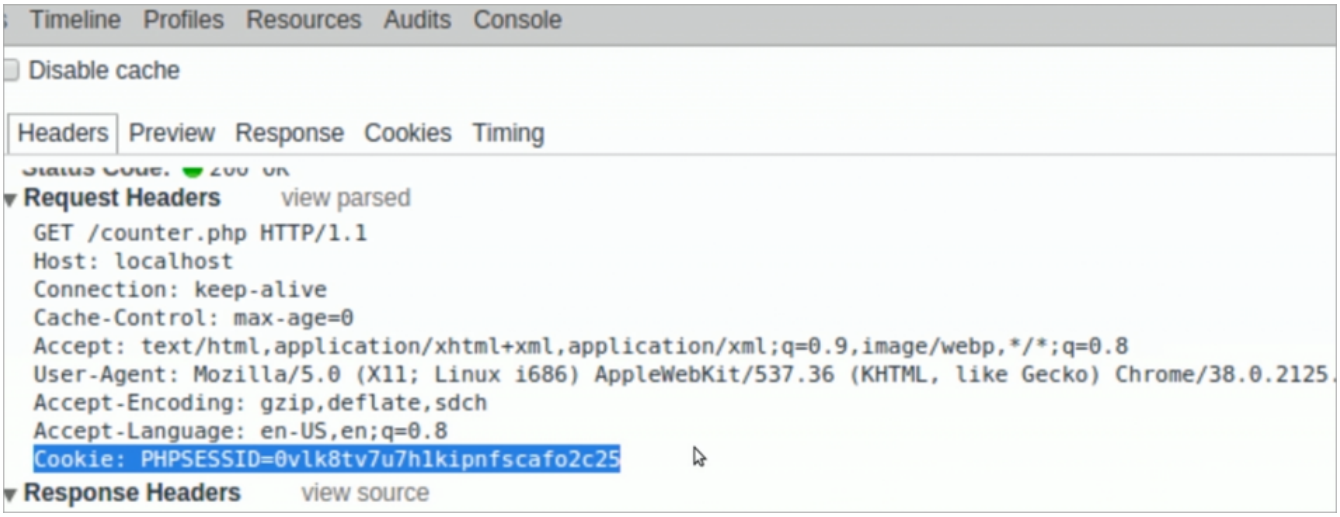


  # In Problem Set 6 we've familiarized ourselves with lines like `GET /counter.php HTTP/1.1`, so not much new here.

- But if we scroll down to Response Headers and click **view source**, we see:



    - `Set-Cookie` is what puts a piece of information on your computer. It's an HTTP header that tells your browser, IE or Chrome or Firefox, to store stuff on the user's hard drive or memory. In this case, we're storing a key called `PHPSESSID` with a value of `0vlk8t…`, a really long pseudorandom string (actually a number encoded with letters) that identifies the user's session. And `path=/` just means that this cookie should be associated with everything on this website, not just the current page.
        - This is just like the website server writing `0vlk8t…` on your hand at an amusement park or club, to identify you later.
- Notice that the server doesn't store our username or (god forbid) our password, but rather a pseudorandom piece of information so our login info isn't saved there, to prevent other people from finding it. The server, on the other hand, remembers any personal information that should be associated with `0vlk8t…`.
- So every time we reload the page, the server knows how many times we've visited. After we reload the page, we look at **Request Headers** again, and we see this time:



    - The browser presents our virtual handstamp with that line, `Cookie: PHPSESSID=0vlk8t…`, and the server realizes that we are that user, and brings up all the information that should be associated with that user.
- In the source code of `counter.php`[3], we indeed store a `$counter` variable in `$_SESSION`:

[3] http://cdn.cs50.net/2014/fall/lectures/8/m/src8m/counter.php

```
<?php

    // enable sessions
    session_start();

    // check counter
    if (isset($_SESSION["counter"]))
    {
        $counter = $_SESSION["counter"];
    }
    else
    {
        $counter = 0;
    }

    // increment counter
    $_SESSION["counter"] = $counter + 1; ❶

?>
```

In line 17 we store the previous value of the counter plus 1.

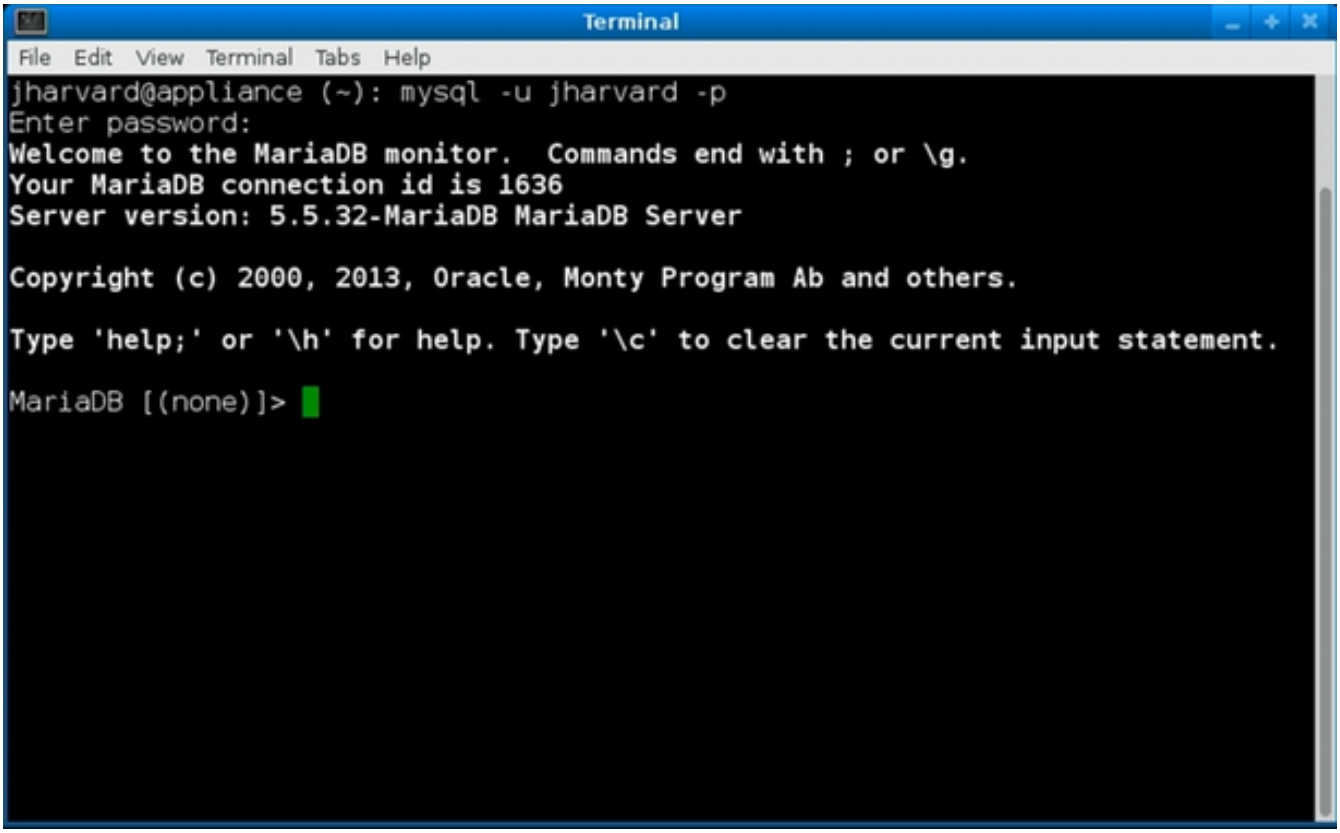
- So that's how `$_SESSION` works under the hood. Modern websites today might set half a dozen cookies or more, and troublingly, if a central party is serving advertisements embedded in multiple websites, then the cookies (which are per domain) will allow that central party to track who you are and what you're visiting.
- If you're super paranoid (like Cheng, who doesn't even have a Facebook) and turn off your cookies, lots of websites will stop working, because many websites require cookies for you to log in, and have no other way of remembering who you are.
- So let's take for granted that we can remember information like that.

Introduction to SQL

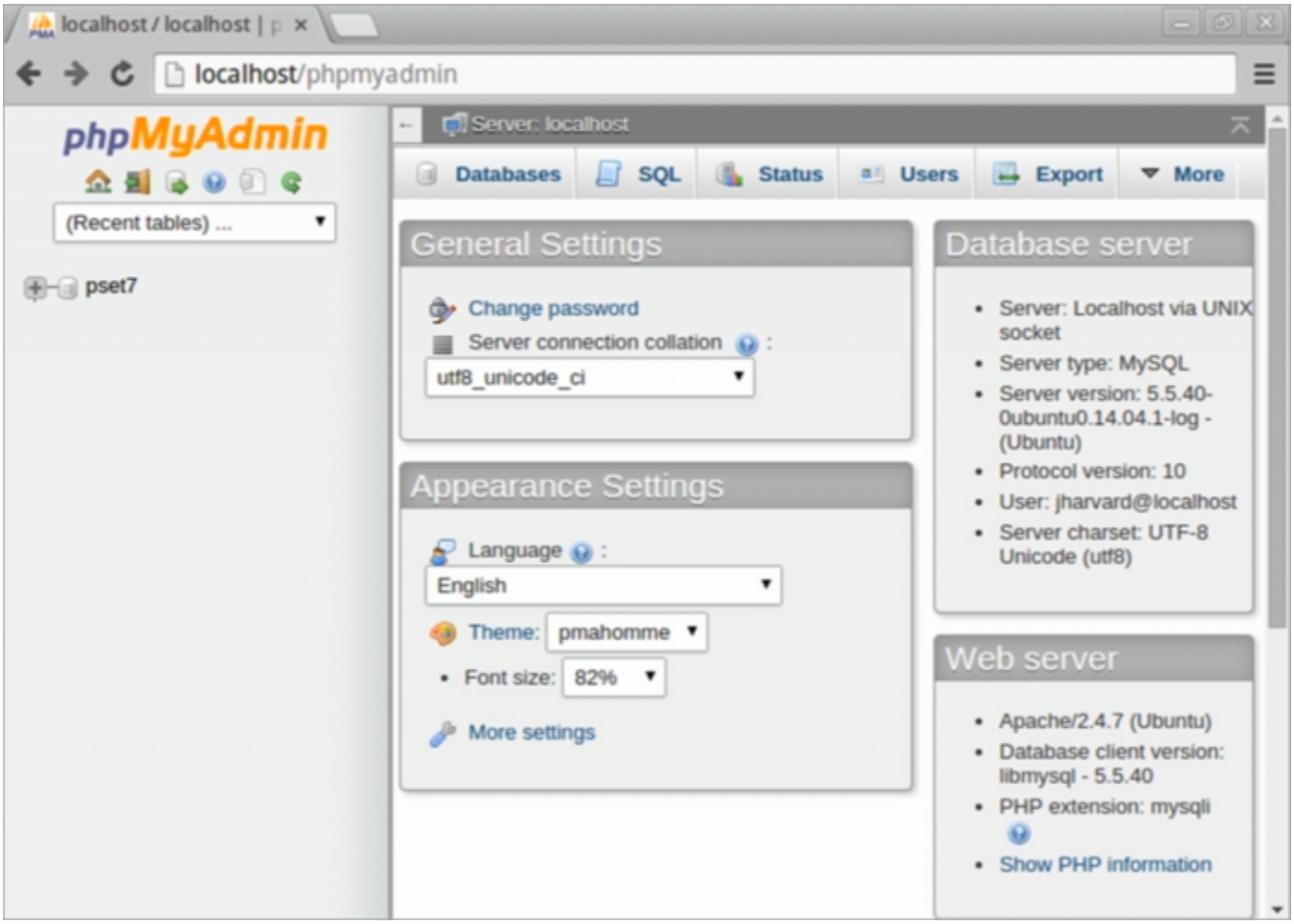
- In the Frosh IMs example from last time, all we did was email the information that a user submitted, like the name, gender, and dorm, to the proctor (or John Harvard, as we were testing).
- We can do that better, by using **SQL**, Structure Query Language, read as "sequel." This is a language we use to talk to a **database**, which is like an Excel (or Numbers or

Google Sheets) file. Those files are essentially **relational databases**, which have rows and columns that store information. SQL, however, is special because we can execute **queries**, commands to sort or remove or look for data, programmatically.

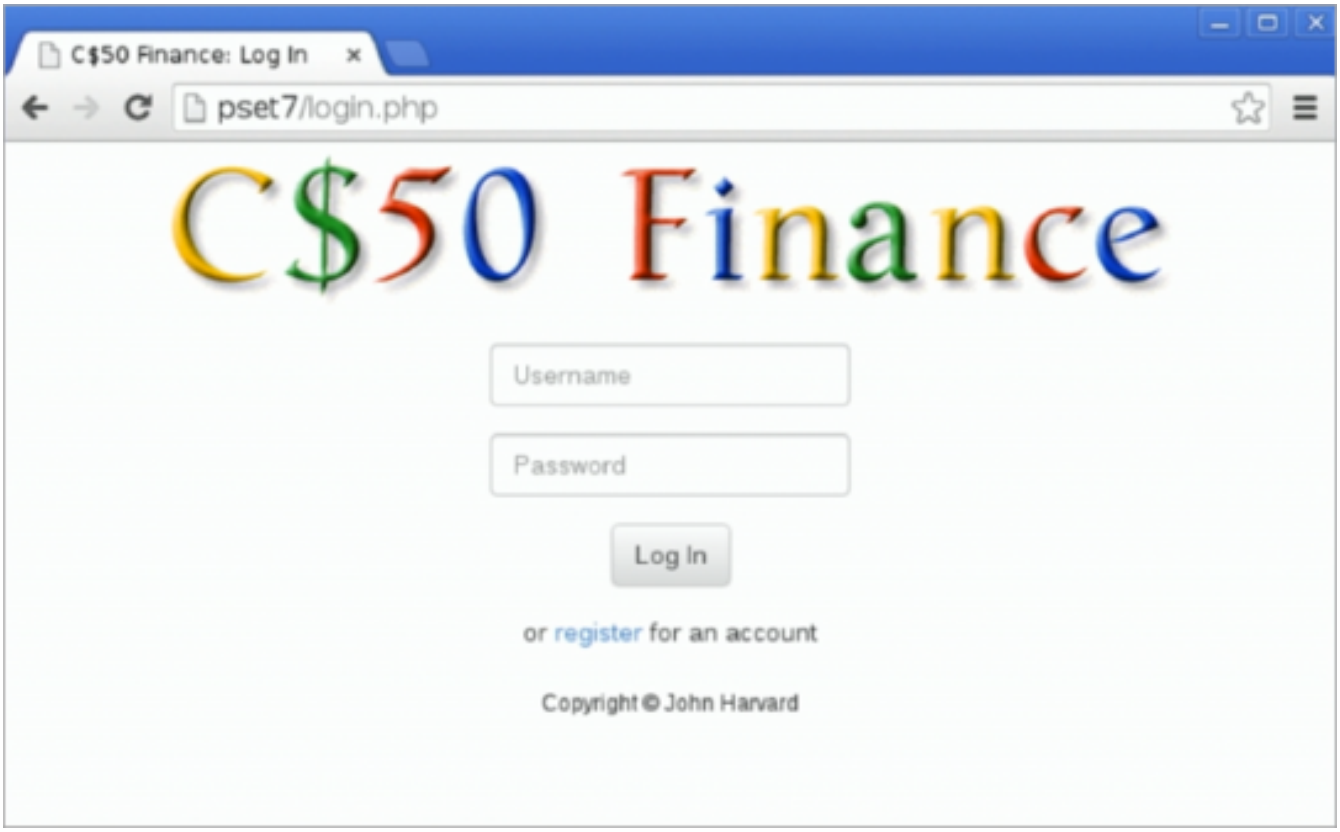
- SQL has fundamental statements like:
  - `DELETE`
  - `INSERT`
  - `UPDATE`
  - `SELECT`
  - …
  - `DELETE` removes data, `INSERT` adds rows, `UPDATE` changes rows, and `SELECT` gets rows.
- SQL operates entirely on rows, so `SELECT` for example would return a **result set**, or an array of rows.
- We can communicate with our database in a Terminal window:



- But that's not particularly fun. A graphical user interface, GUI, is much better:

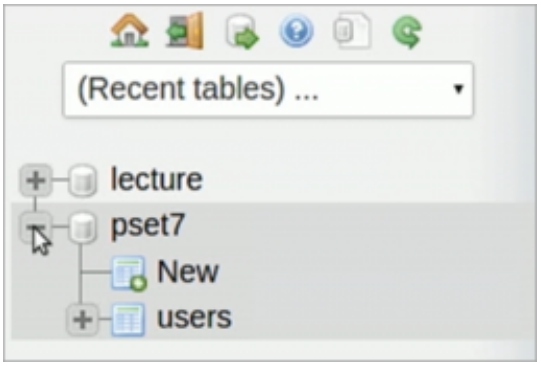


- The tool we recommend and have preinstalled in the appliance is phpMyAdmin. The "php" in the name refers to the language it was written in, and the tool *admin*isters a *My*SQL server.
- On the left we see a list of databases that we have in the appliance (or a server on the Internet, in the case of many final projects), and here we only have the **pset7** database.
- On the top there are a bunch of tabs, **Databases**, **SQL**, **Status**, **Users**, etc.
- In Problem Set 7 we'll be making a website called C$50 Finance, which lets you "buy" and "sell" stocks. It'll get prices from Yahoo! Finance which has a free service where we can pass in a stock ticker, like GOOG for Google, Inc., and it will give you back the current stock price for Google. So we'll use that to allow users to pretend to buy and sell stocks with virtual money.
- The first screen that users will see is this:



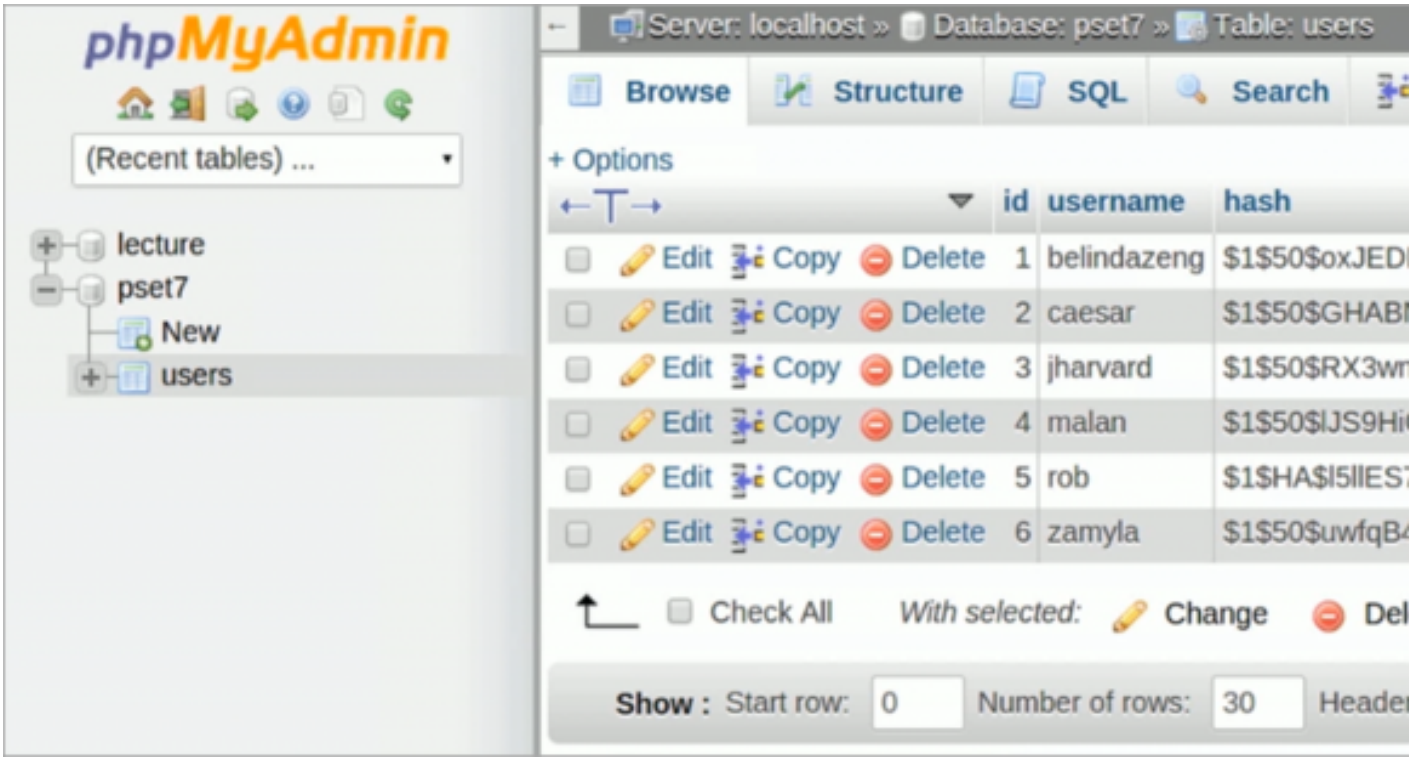
And your first challenge will be to implement the backend database for the users' names, passwords, and eventually the stocks they own, among many other things.

- Let's look more closely at databases. In the appliance, we can go to `localhost/phpmyadmin/`, and click the **+** next to `pset7`:

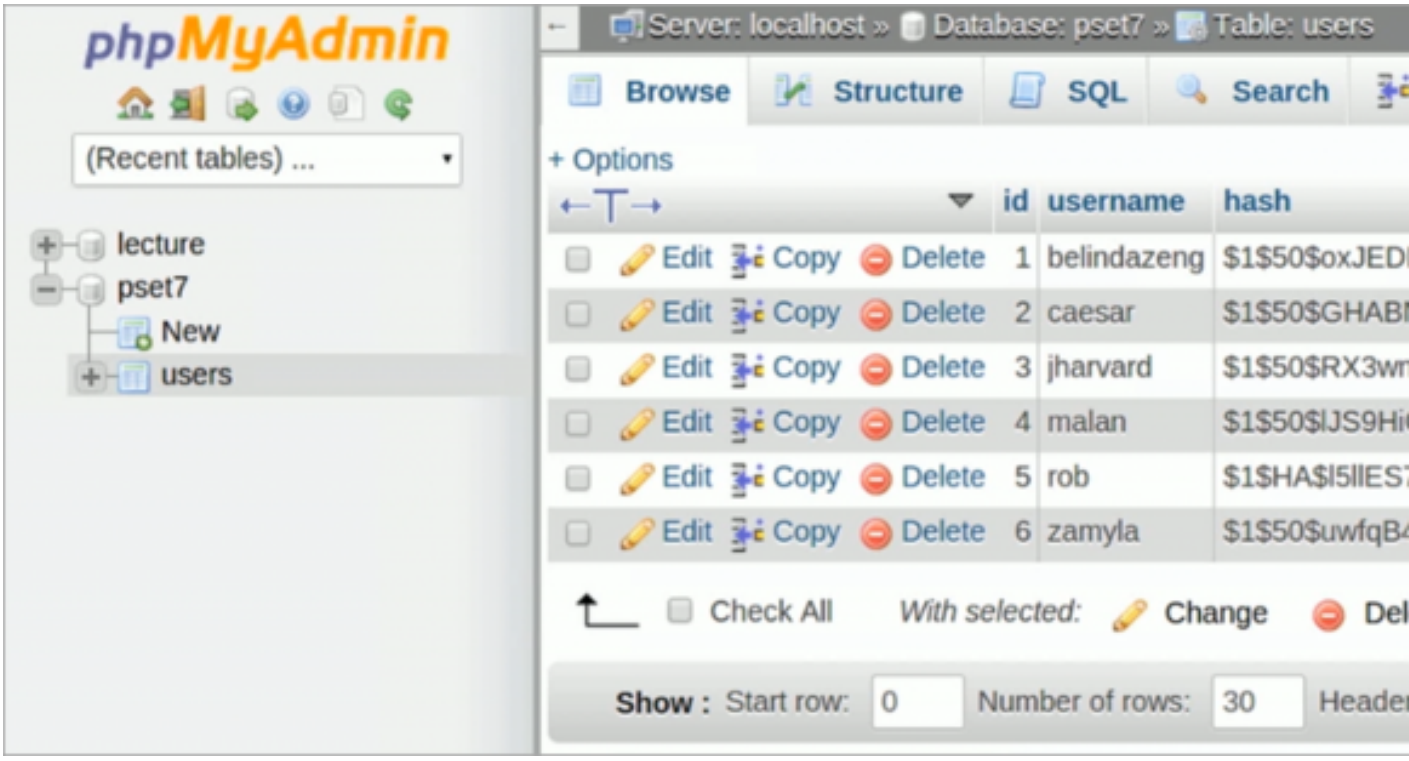


We see that the `pset7` database has a link to `New`, for creating a new table, but also `users`, a table we've already created.

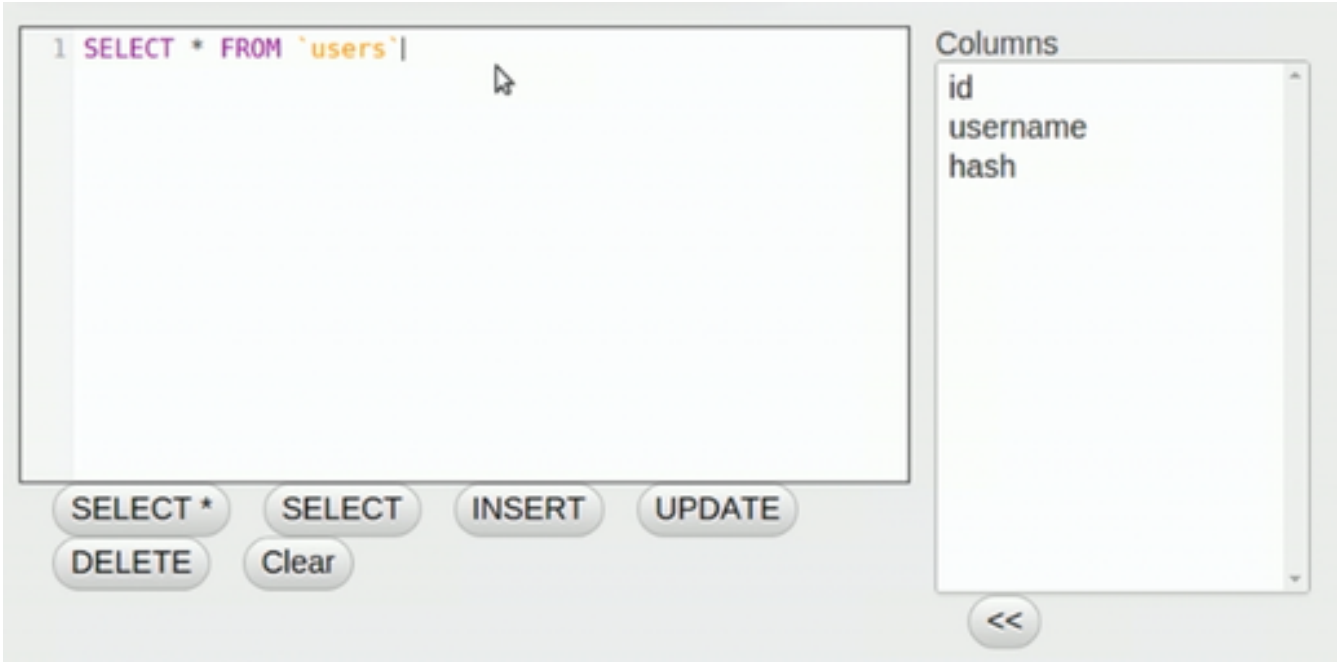
- A **table** is like an individual spreadsheet with its own rows and columns, represented by the tabs along the bottom of the screen in Excel or Numbers files. A database is just made up of one or more of these tables.
- If we click on `users` and scroll down a bit, this is what that table looks like:



- We see that there are columns `id`, `username`, and `hash`, and in Problem Set 7 we give you this information to start you off. Notice that there are 6 usernames, each with unique IDs from 1 to 6. To the right are hashes, or basically encrypted passwords. (The hashes and names are from Problem Set 2's Hacker Edition.) And to the left are links to GUI operations, like editing, copying, or deleting rows.
- We can now click on the **SQL** tab and try out queries. (Remember that phpmyadmin is just a tool for us to poke around, but we'll really program all of these queries into our `.php` files when we want to use them.) Let's take a look at that tab:



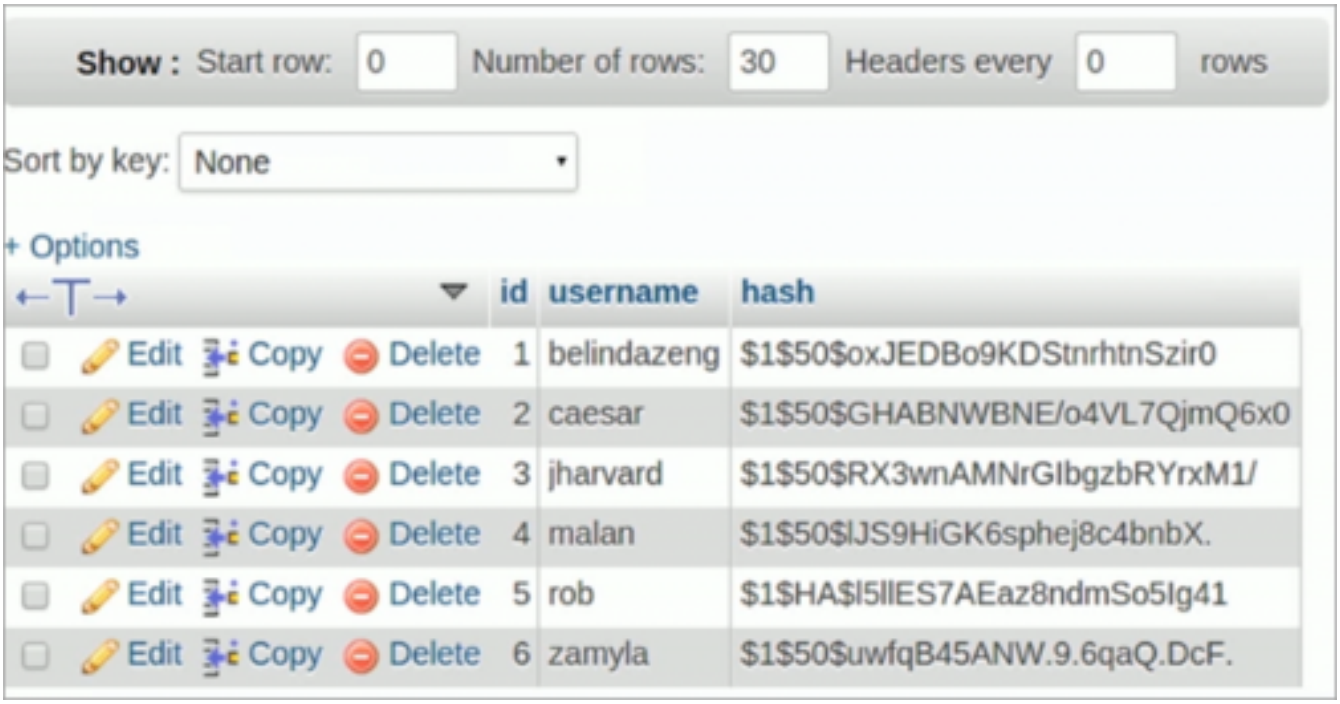
- The big text box is where we can input a query, and let's try to run this:



`SELECT` is a keyword that means we're getting something from the table, and that something is `*`, which means all the columns, `FROM` the table called `` `users` ``.

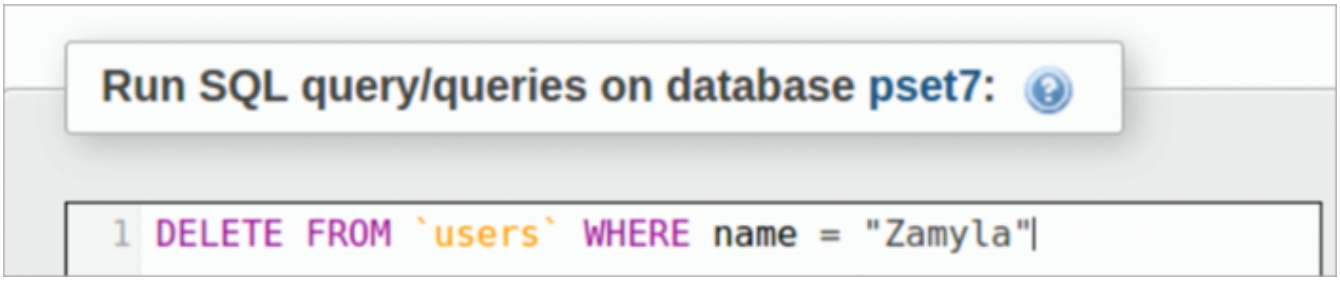
(The backtick, `` ` ``, character is used to surround the name of a table, not single or double quotes, and it's probably on the top left of your keyboard to the left of the `1` key.)

- We can scroll down and click **GO**, and we'll see this:



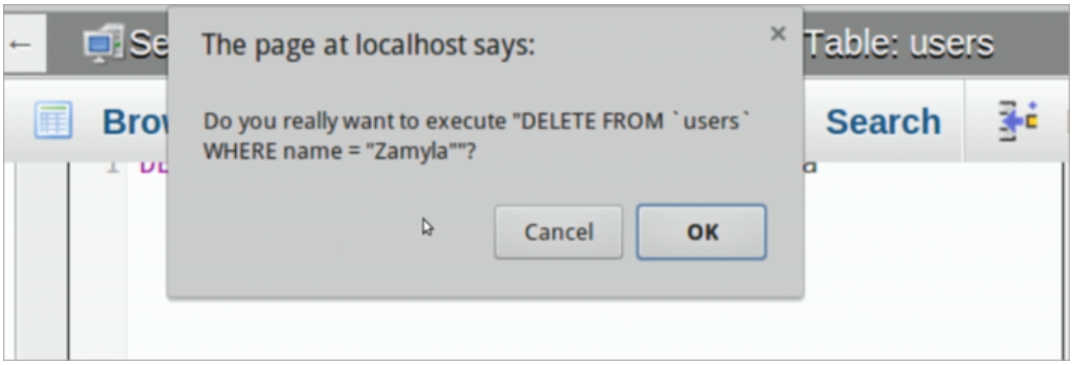
This is the same thing as before, since our query was to get everything.

- Now let's try to run this:

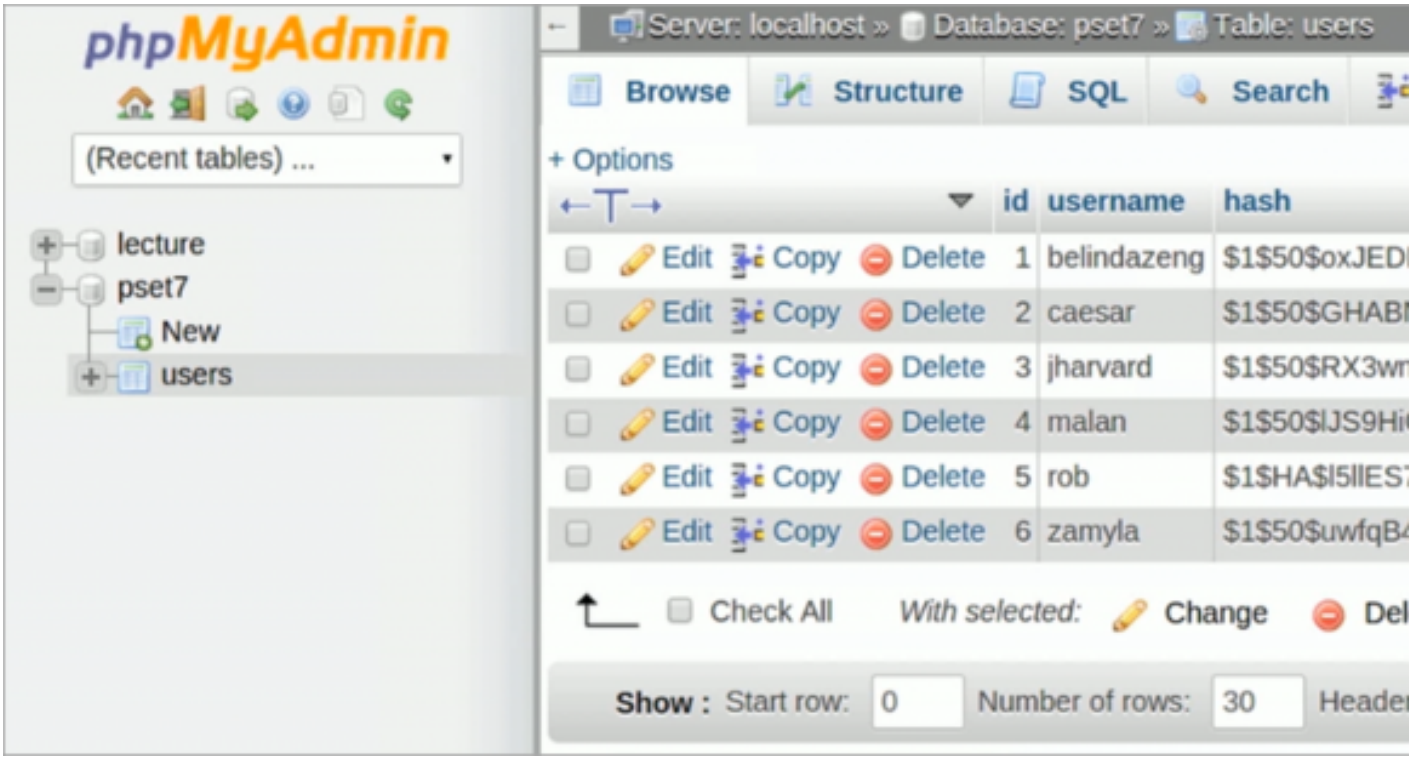


Notice that we use double quotes around strings as usual.

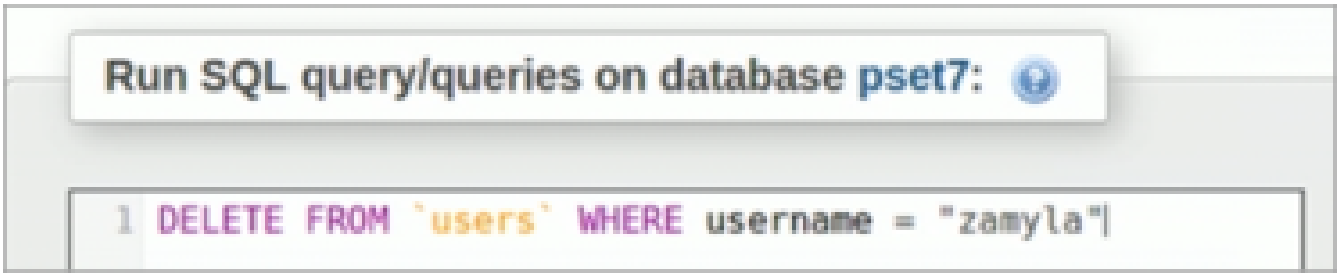
- And when we run it we get a warning, to be sure we really wanted to `DELETE` something:



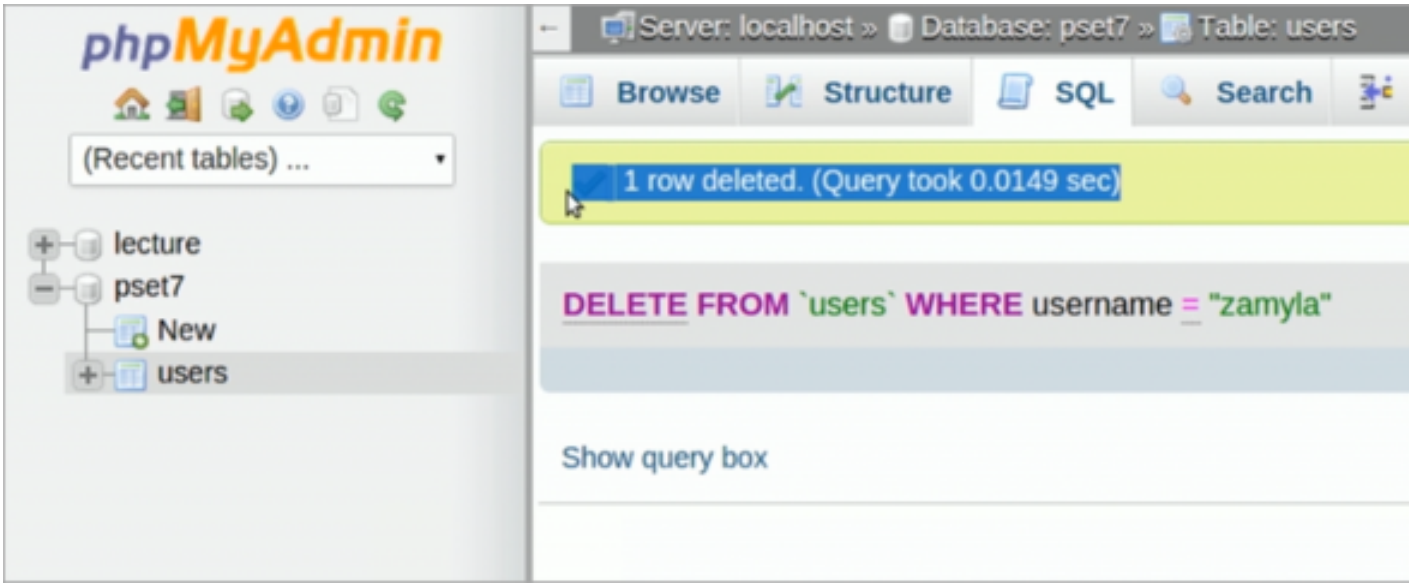
- We click OK, but when we go back to the `users` table, it looks exactly the same:



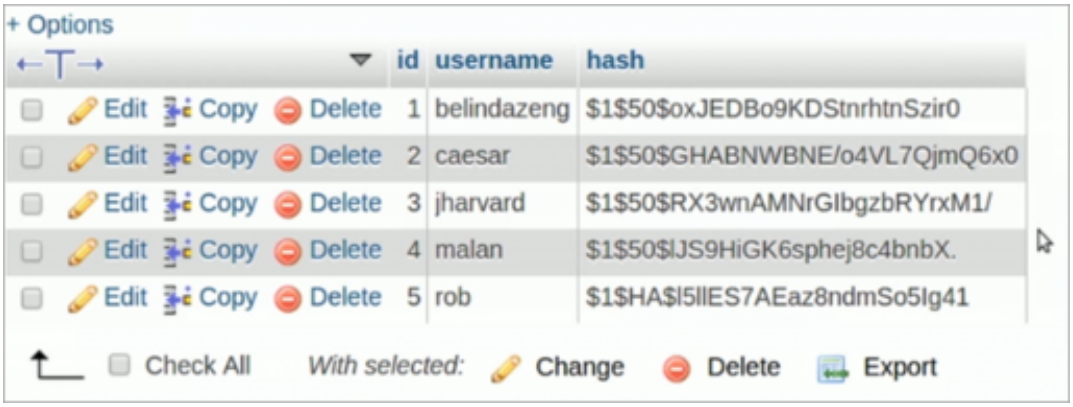
- Turns out we made a couple of mistakes. The column is named `username`, not `name`, and her username is stored as `zamyla` with a lowercase, not uppercase, "z."
- So let's try this again:



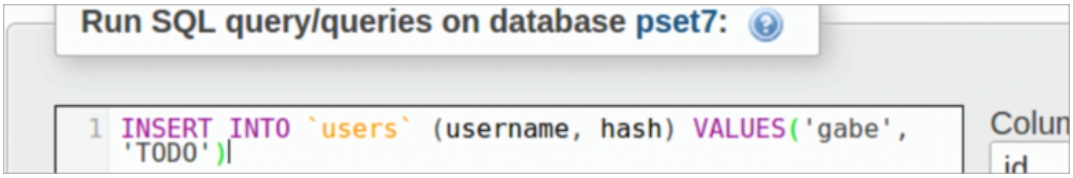
- And now we actually see a confirmation page that tells us we completed this query successfully:



- To be sure, we can go back to our table and see all the rows:

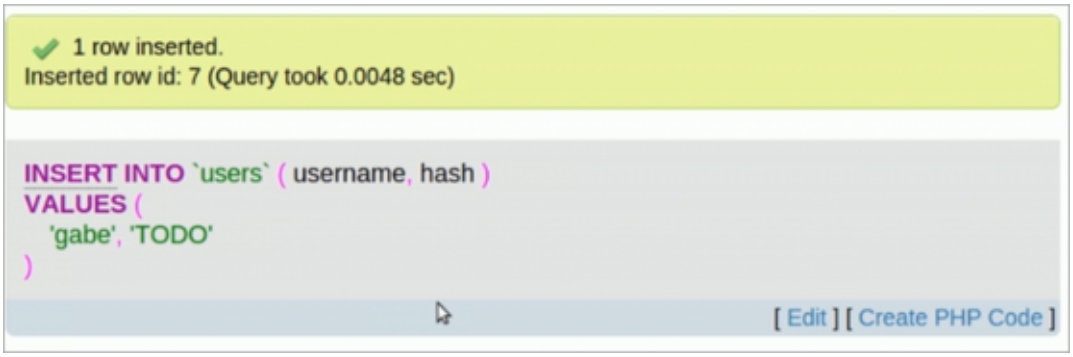


- We can also register Gabe. We can do something like this:



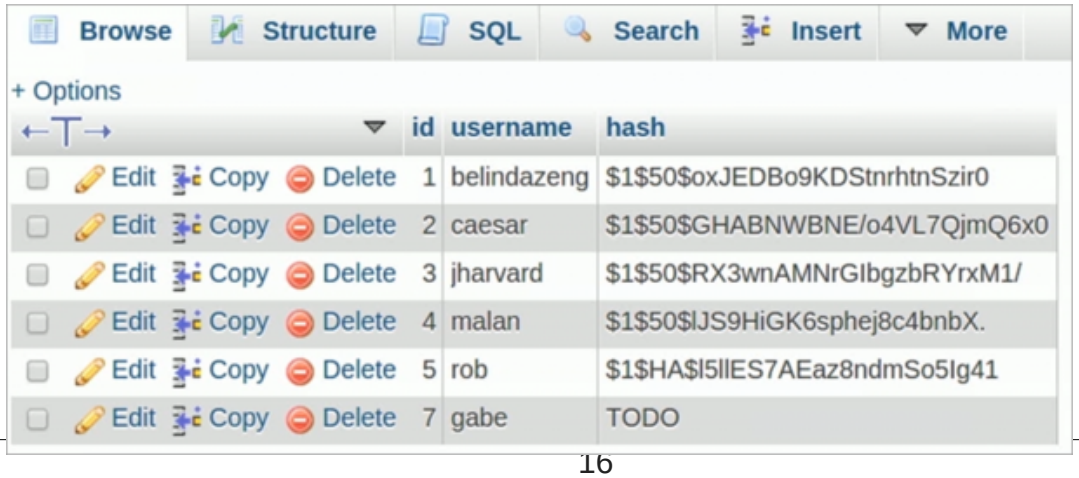
The syntax is bit more cryptic now, but we are telling the database that we want to `INSERT INTO` the table called `` `users` `` a row with the columns `(username, hash)` specified with the `VALUES('gabe', 'TODO')`, with the values we want in the same order of the columns we listed. (Even though David used single quotes, which do work, probably best to use double quotes, as in C.) We don't really care what the `id` column is, and we don't know what the `hash` is going to be, so we'll leave it as `TODO` for now.

- We click **GO**, and we get:



It looks like we successfully inserted Gabe into the table.

- And when we go back to the table, we see that his `id` is actually:

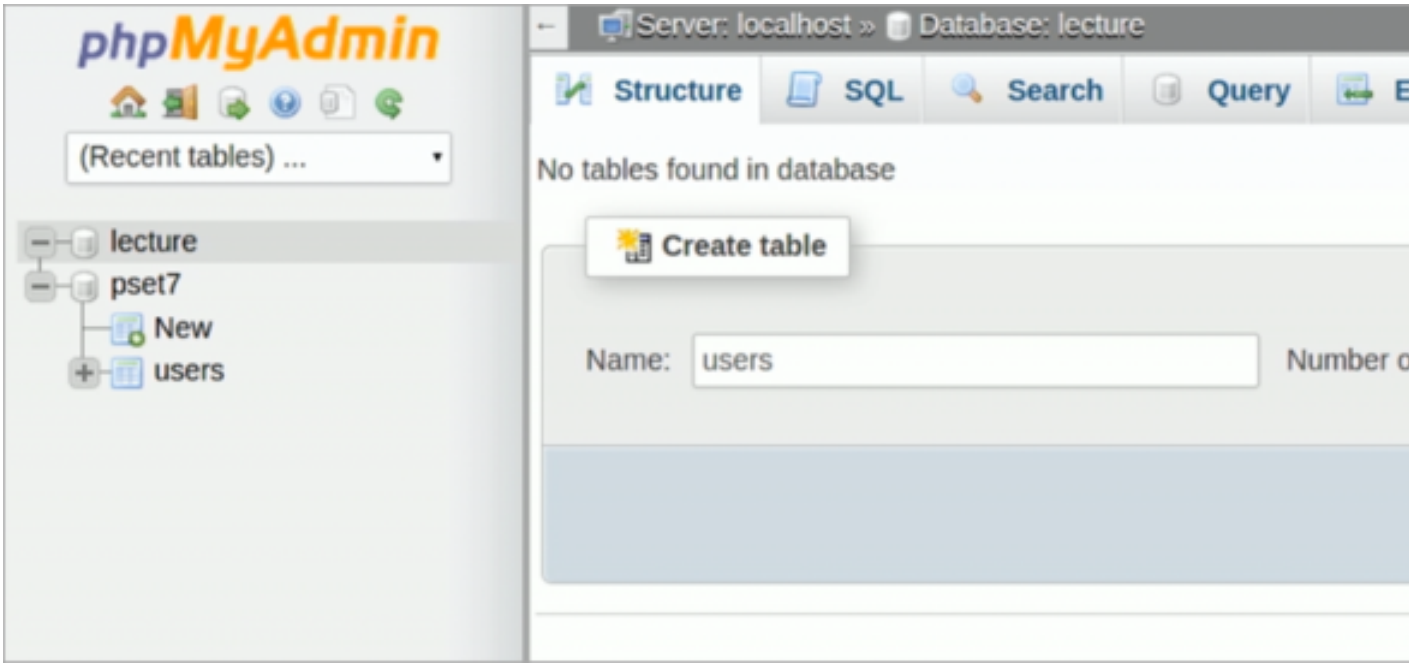


    - This is a built-in feature of the database, where `id` can automatically be incremented and assigned, even though we didn't specify it. (When the table was created, we specified that `id` would be a special field that does this.)

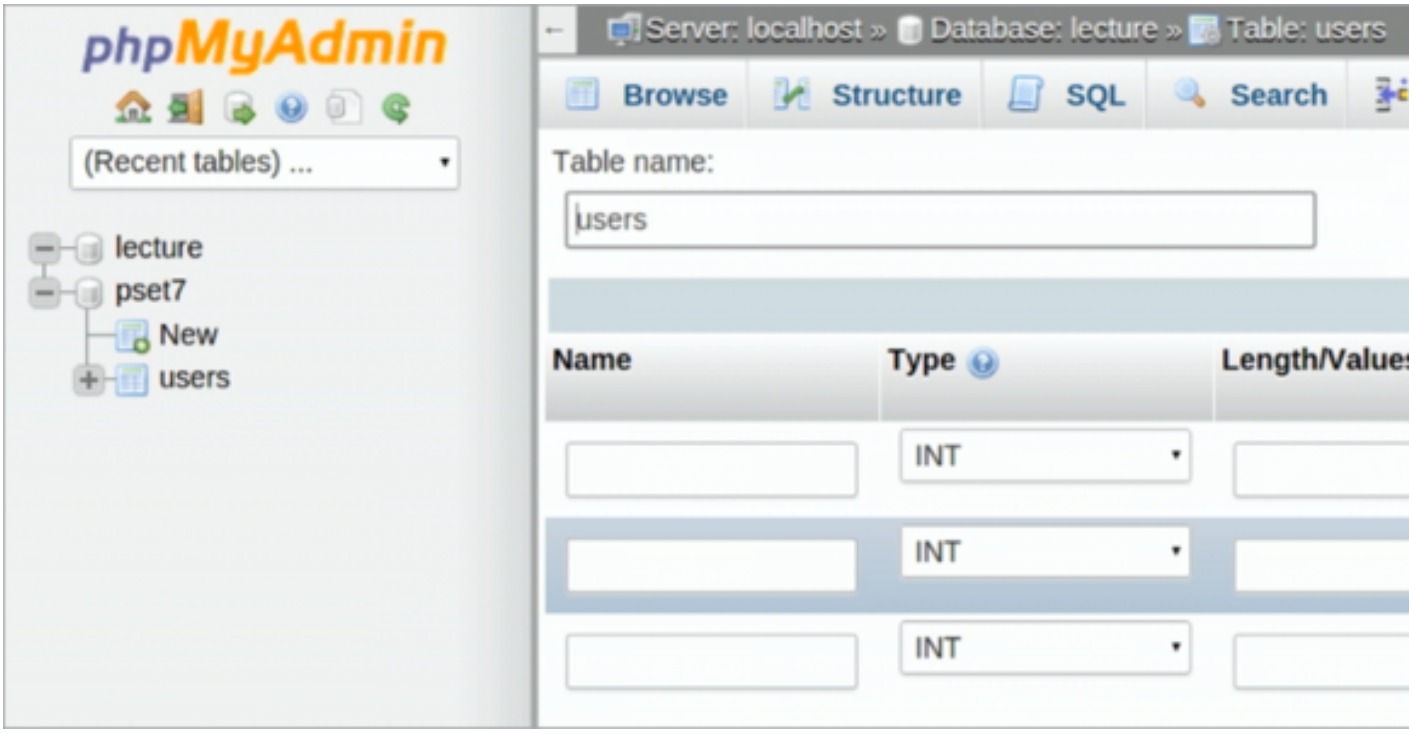
- As an aside, Facebook has an API where you can get all sorts of data about users. Mark Zuckerberg's account had an ID of 3, with users 1 and 2 being test accounts. And all of us have numbers much larger. (In fact, they've moved from using an `int` to the equivalent of a `long long` so they can eventually accomodate more users.)
- So that's just an introduction to the syntax of SQL, with which we can do many more powerful things.

Creating a Table

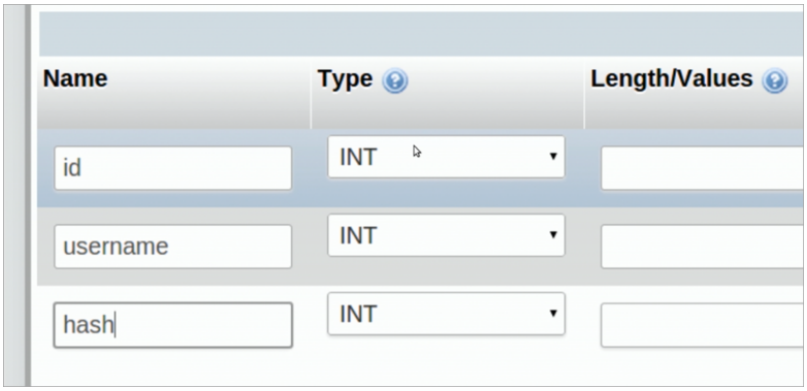
- In Problem Set 7 we'll allow you to make a number of design decisions, one of which is the types of data to use.
- Just like in C, there are datatypes in SQL including:
    - `CHAR`
    - `VARCHAR`
    - `INT`
    - `BIGINT`
    - `DECIMAL`
    - `DATETIME`
    - …
- Let's go to the `lectures` database that David has, and create a new table called `users` with `3` columns:



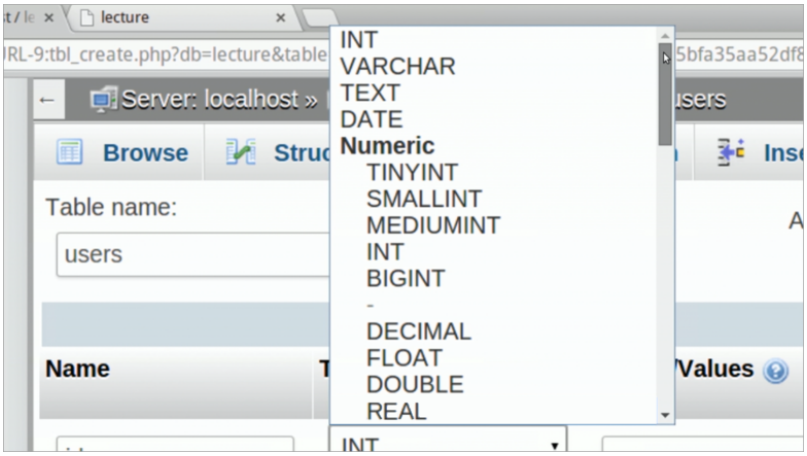
- Once we click **GO**, we get to a screen that looks like:



- Here we can name each column:

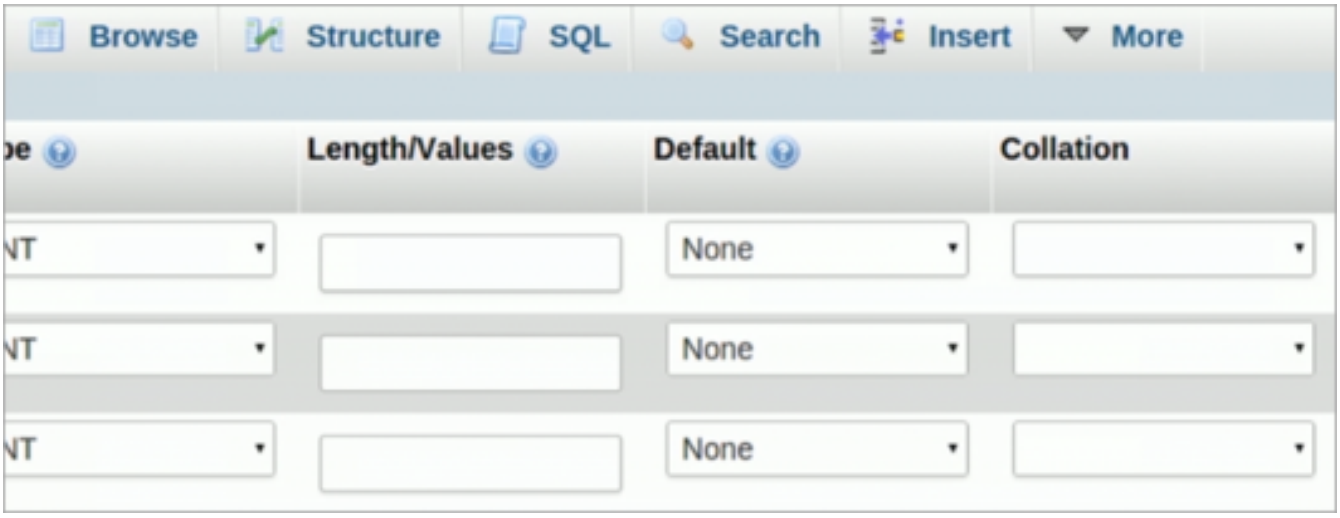


- And when we click on **Type**, we get a menu of the whole list of types possible:

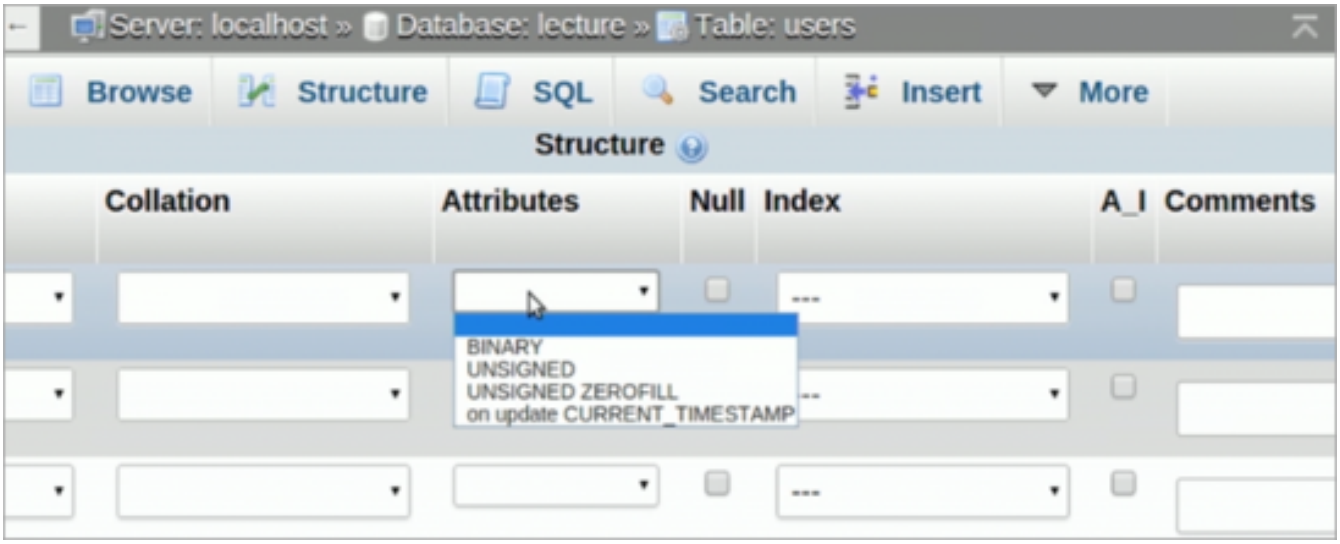


But we'll stick to `INT` for the `id` column.

- We can scroll right and see more fields, which we'll leave blank since none of them are applicable:

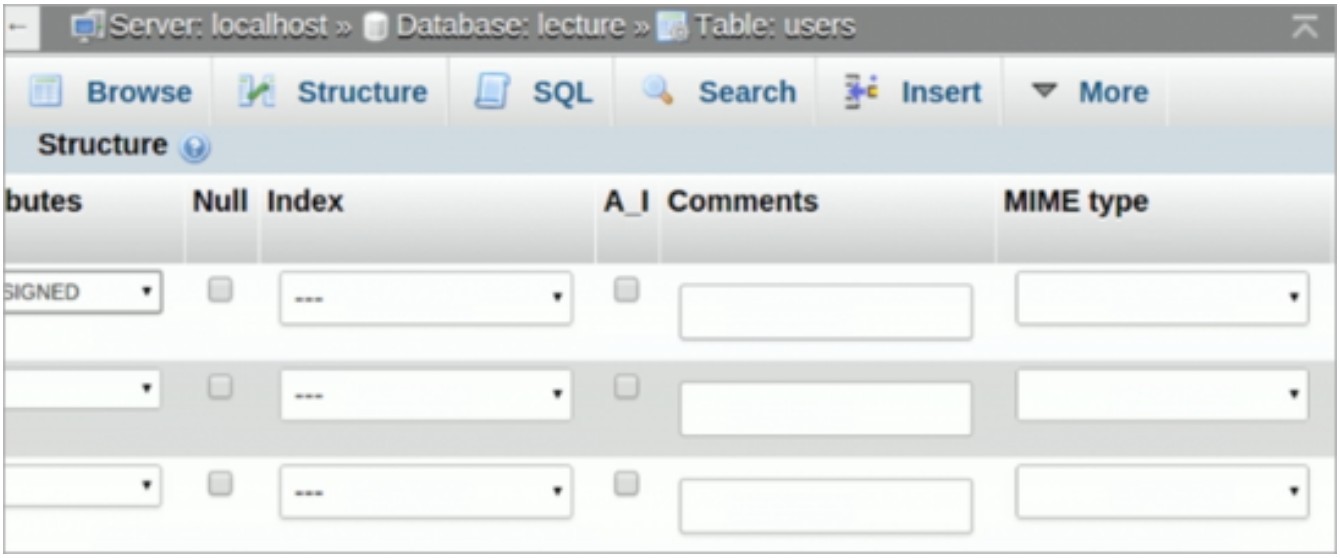


- When we get to **Attributes**, we have a design decision to make, where we can choose `UNSIGNED` for the `id` column:



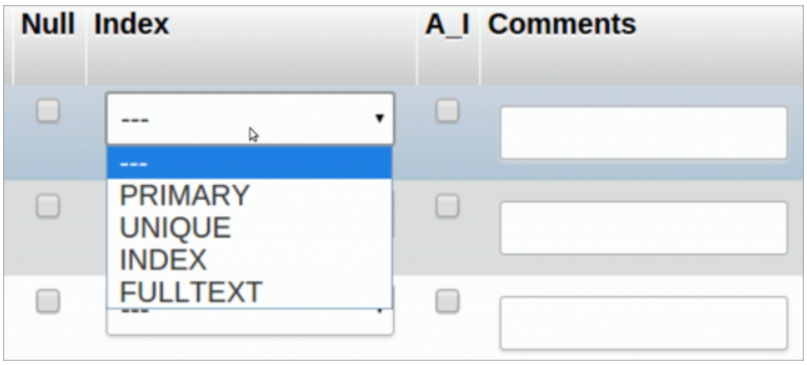

```
# This specifies that the int has to be non-negative, 0 or higher.
```

- If we keep scrolling right, we see a few more columns:



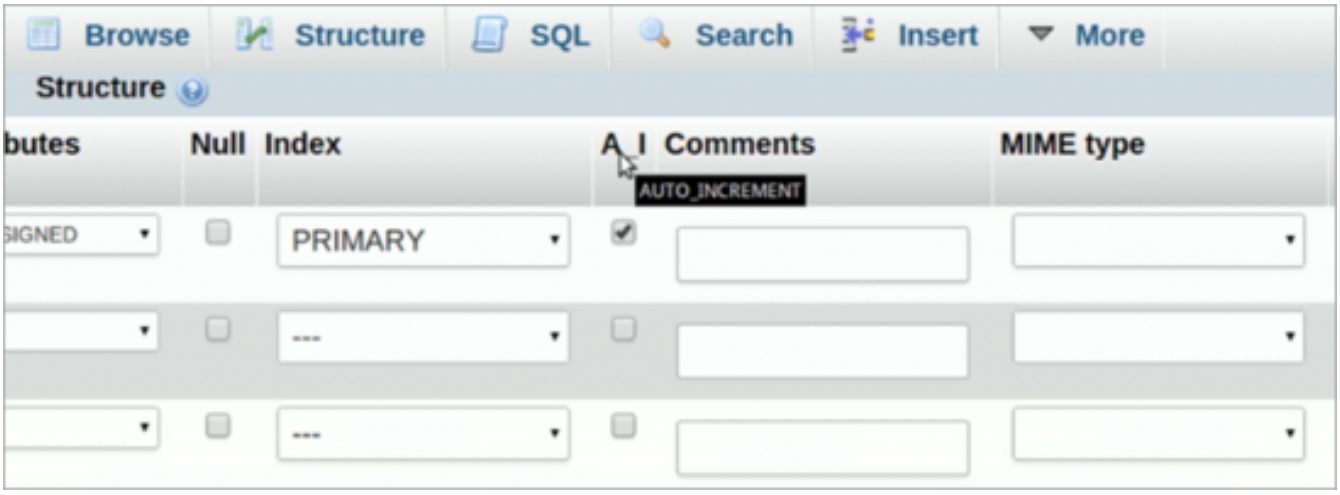
We don't check **Null** because we want every user to have an `id`.

- If we click on **Index**, we see these options:

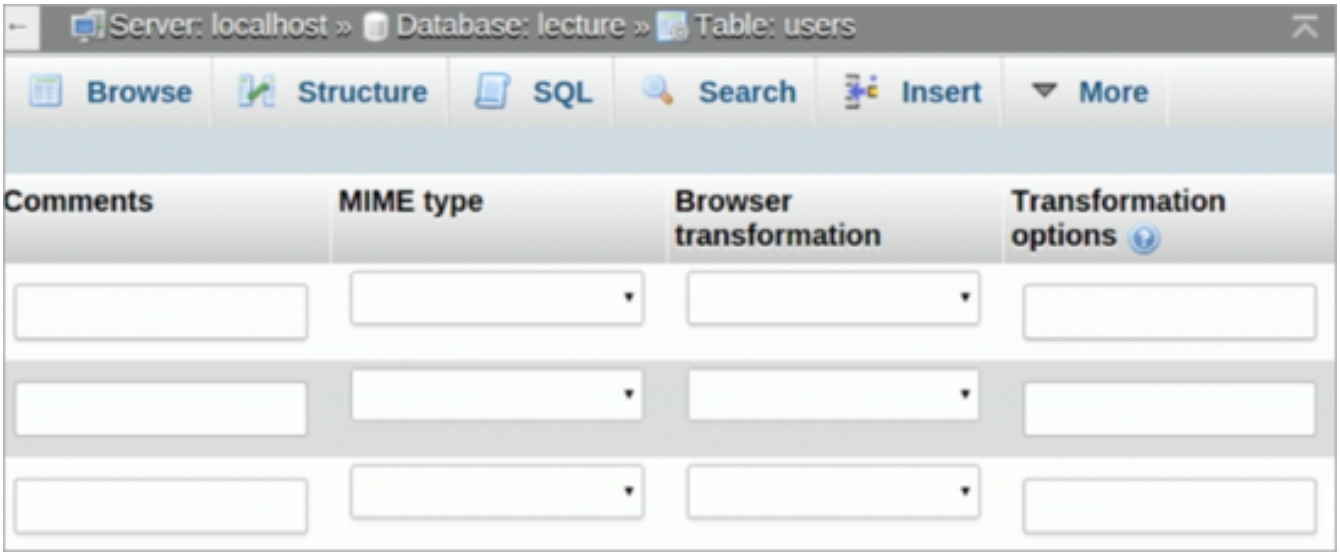


Another feature of a database server is that it can optimize the way it structures the database, so operations are faster. In Problem Set 5 you had to make sure your hash table or trie was fast, but here we can use what other people have built. And here, by selecting `PRIMARY` in the **Index** column for `id`, we are telling the database server that `id` will be the primary way of identifying users in the database. We could use their `username`, but that's a string, and less efficient to sort and access than an `int`.

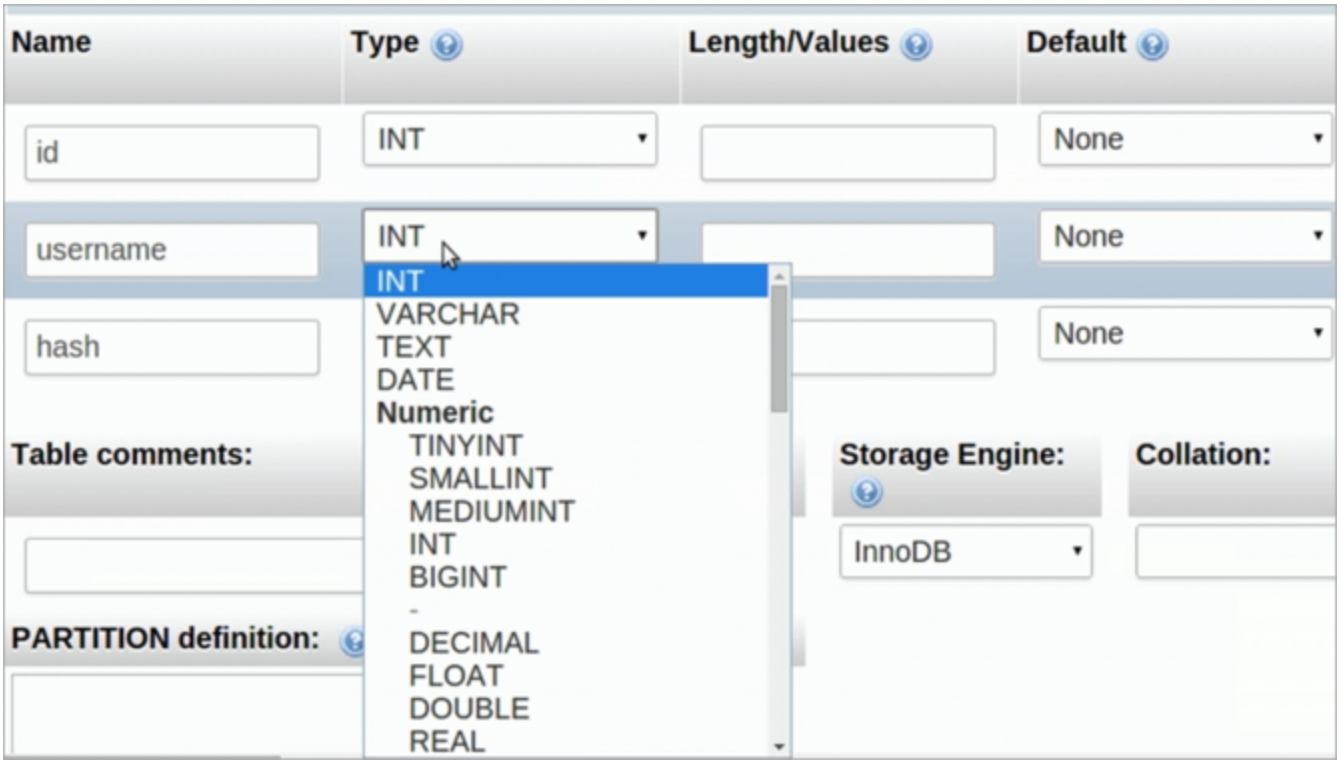
- The next field, **A_I**, just means AUTO_INCREMENT, which we want to check so the `id` column is automatically updated:



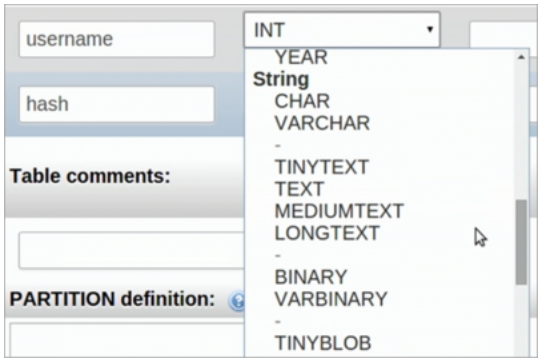
- And if we go further to the right to the last few columns, not many interesting things are there:



- So we can go back to the beginning, and take a look at the types for `username` and `hash`:



We probably don't want `username` to be an `INT`, so we should choose a `STRING` … but there are many choices if we scroll a bit down:

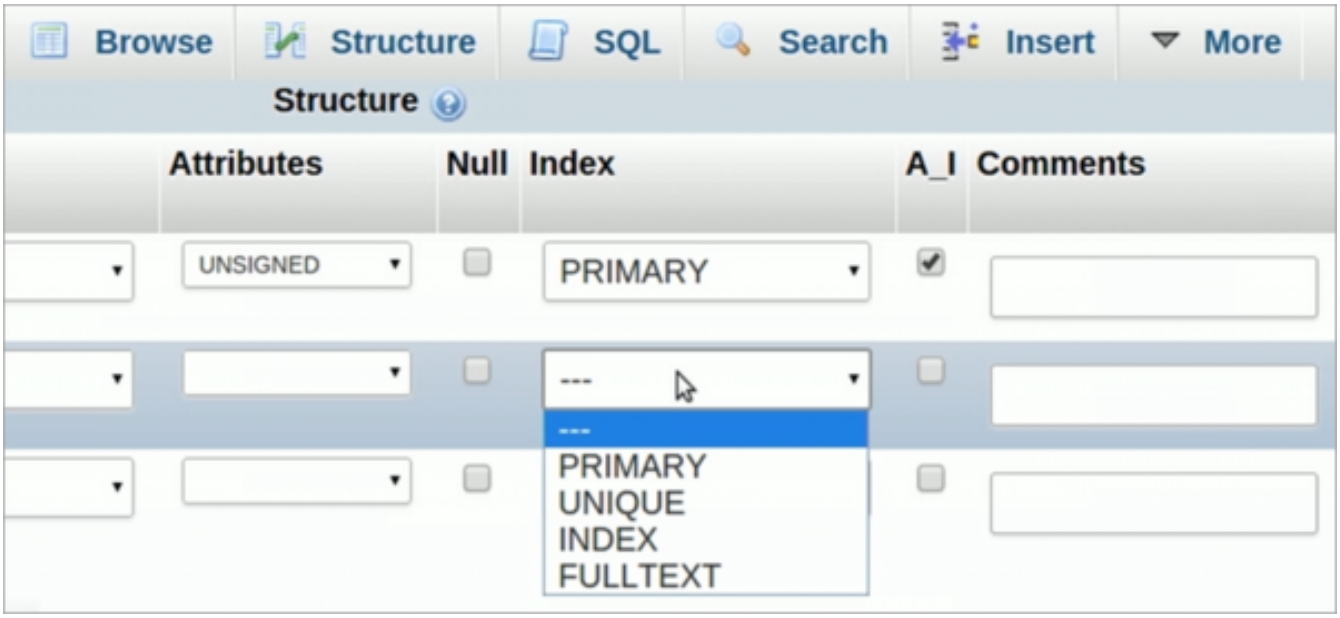


`CHAR` is not a single character, but a particular number of characters that we'd have to specify in the next column, **Length/Values**. We could decide to make all usernames have a length of 8, for example. But some people might want a longer or shorter one, so we can choose the `VARCHAR` type, which allows us to have a variable number of characters. If we select that, then the value in the **Length/Values** column will be the maximum number that field can be. `TEXT` would also

work, but it's meant for strings with a maximum length of 65,535 characters, and probably too much for a simple username.

    # We'll set `hash` to also `VARCHAR`, but not focus on its length for now.

- We'll scroll a bit to the right, and make a stop at the **Index** column:



    # We can't make two indexes both `PRIMARY`, but we should choose to make `username` `UNIQUE`, since we don't want multiple people singing up with the same username. When that's selected, the database will help us enforce this by not letting us insert rows with the same username as another row.
    # `INDEX` tells the database to create an index of this field, to increase the speed of our searches on this field.
    # `FULLTEXT` tells the database we might run searches on the full text of this field, which might be paragraphs or longer pieces of text.

- Another design decision you might have is the storage engine you use, but we'll come back to that someday.

MVC with Texting

- Let's open this week's `source code`[4] in our Terminal:

[4] http://cdn.cs50.net/2014/fall/lectures/8/w/src8w/

```
jharvard@appliance (~/vhosts/localhost): ls
includes  public  templates
```

Notice that we have three directories, where we left off on Monday. `public` will contain all the files that we want users to actually visit. `templates` will have reusable components, like the header and footer of a webpage. This is the equivalent of the "View" part of MVC, where we put a lot of the aesthetics. `includes` has the following:

```
jharvard@appliance (~/vhosts/localhost/includes): ls
config.php  constants.php  functions.php
```

- Let's take a quick look at `config.php`[5]:

```
<?php

    // display errors, warnings, and notices
    ini_set("display_errors", true);
    error_reporting(E_ALL);

    // requirements
    require("constants.php");
    require("functions.php");

    // enable sessions
    session_start();

?>
```

Like the `#include` statement in C, the `require` statements in PHP allow us to essentially copy and paste code from those files. And `session_start()` means that we'll be keeping a session on this website, with cookies being sent back and forth.

- We can also open `constants.php`[6]:

[5] http://cdn.cs50.net/2014/fall/lectures/8/w/src8w/includes/config.php
[6] http://cdn.cs50.net/2014/fall/lectures/8/w/src8w/includes/constants.php

```
<?php

    // your database's name
    define("DATABASE", "lecture");

    // your database's password
    define("PASSWORD", "crimson");

    // your database's server
    define("SERVER", "localhost");

    // your database's username
    define("USERNAME", "jharvard");

?>
```

Again like C, PHP supports constants, even though the syntax is a bit different.

- And in the file `functions.php`[7], there are lots of functions, but let's look at `function query`:

```
    /**
     * Executes SQL statement, possibly with parameters, returning
     * an array of all rows in result set or false on (non-fatal) error.
     */
    function query(/* $sql [, ... ] */)
    {
        // SQL statement
        $sql = func_get_arg(0);

        // parameters, if any
        $parameters = array_slice(func_get_args(), 1);

        ...
```

Earlier we used phpMyAdmin to set up and experiment with our database, but to actually use it in code we'll start calling this `query` function.

[7] http://cdn.cs50.net/2014/fall/lectures/8/w/src8w/includes/functions.php

- The file also contains `function redirect` that we can call to send the user to another URL:

```
/**
 * Redirects user to destination, which can be
 * a URL or a relative path on the local host.
 *
 * Because this function outputs an HTTP header, it
 * must be called before caller outputs any HTML.
 */
function redirect($destination)
{
...
```

- `render` renders a template, but more about these functions in Problem Set 7's walkthroughs.

Registering

- Let's look now at `index.php`[8] in the `public` directory:

```
<?php

    // configuration
    require("../includes/config.php");

    // render portfolio
    render("form.php");

?>
```

    # Here we `require` `config.php` which is in the `includes` directory that is in our parent directory, `..`.

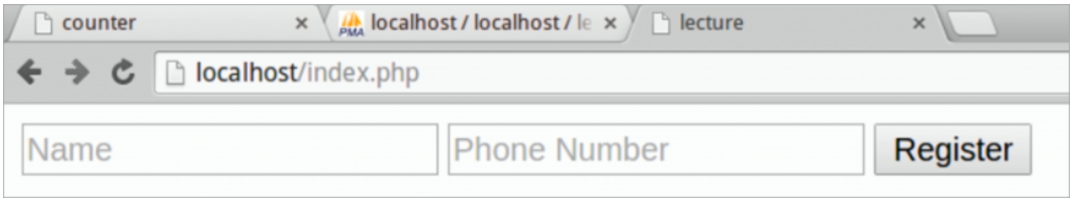
- Then we just `render` `form.php`[9], which is in our `templates` directory:

[8] http://cdn.cs50.net/2014/fall/lectures/8/w/src8w/public/index.php
[9] http://cdn.cs50.net/2014/fall/lectures/8/w/src8w/templates/form.php

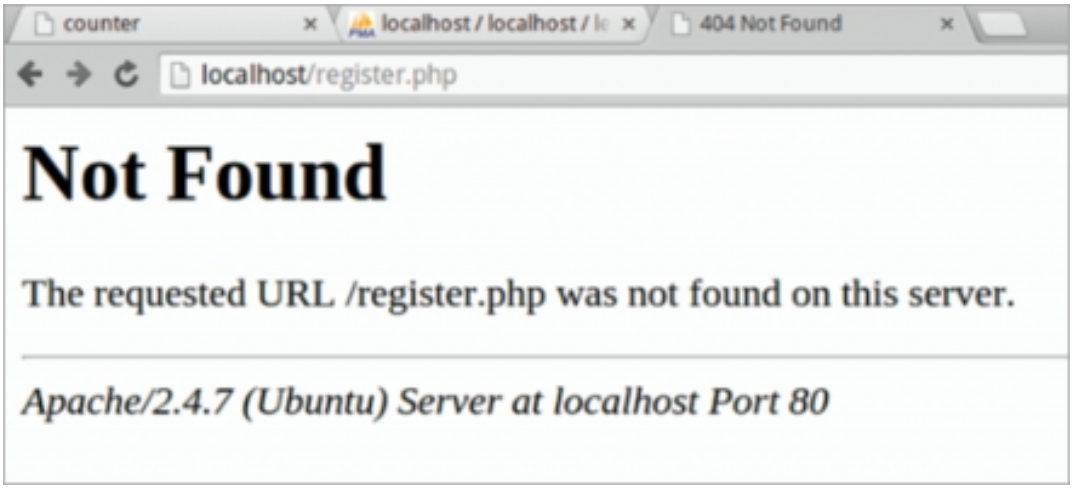
```
<form action="register.php" method="post">
    <input name="name" placeholder="Name" type="text"/>
    <input name="number" placeholder="Phone Number" type="text"/>
    <input type="submit" value="Register"/>
</form>
```

So this form is using the `post` method to hide the information from the URL, and submit it to a file called `register.php`.

- Putting it all together, `http://localhost/index.php` will end up looking like this:



- If we fill out the form and click **Register**, we'll get this because `register.php` is not yet implemented:

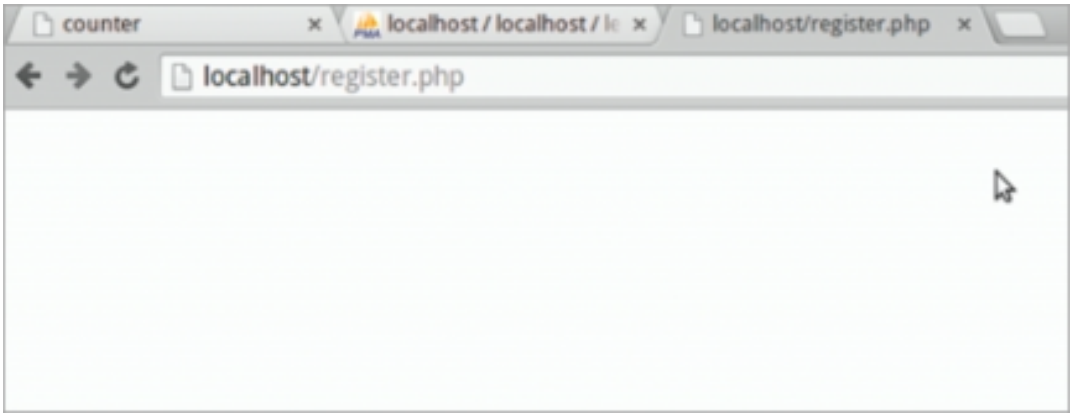


- So let's create `register.php` in our `public` directory with `gedit`, and start with this:

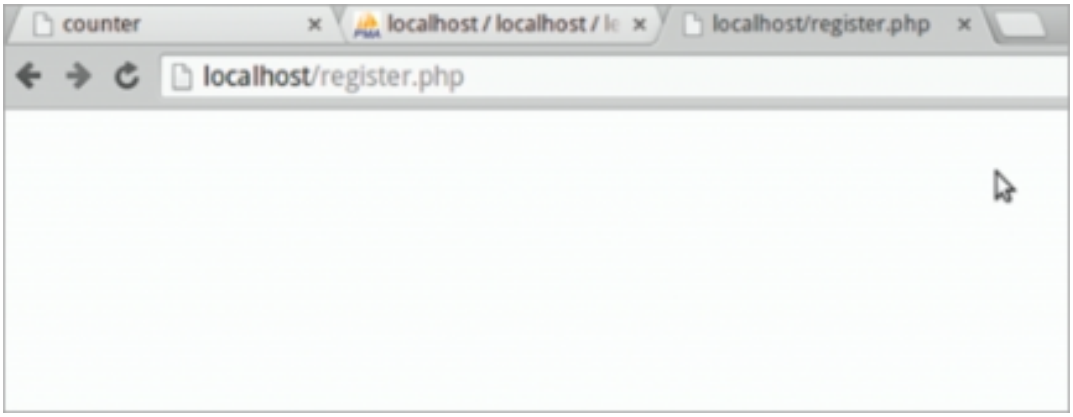
```
<?php

    if (empty($_POST["name"]))
    {
        apologize("Missing name");
    }
    else if (empty($_POST["number"]))
    {
        apologize("Missing number");
    }

?>
```

If either of the fields are `empty`, we call the `apologize` function (that we wrote in `functions.php`, not a built-in PHP function) to tell the user which one is missing.

- Now if we register, we get a blank screen, because the form was complete and there were no errors:



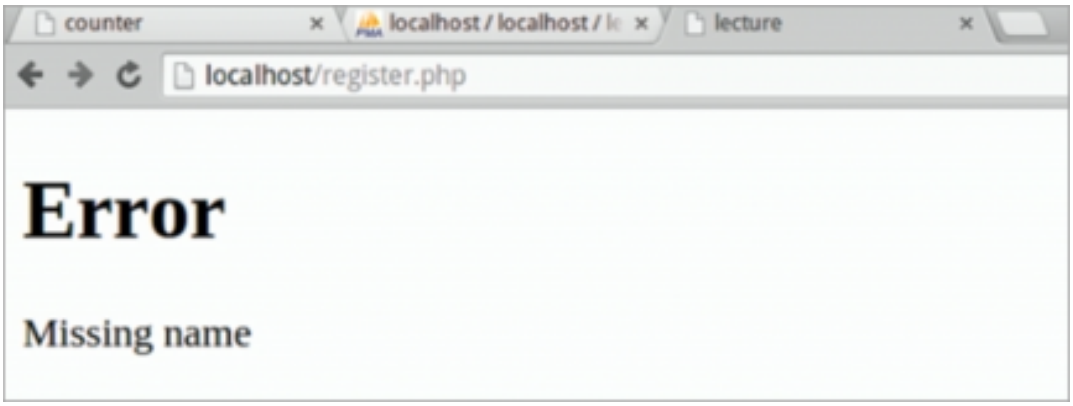
- Let’s go back and purposefully leave both fields blank. Hmm, the same thing:



- Turns out we forgot the most important part, requiring our `config.php` that sets up our constants and functions. So let's add that:

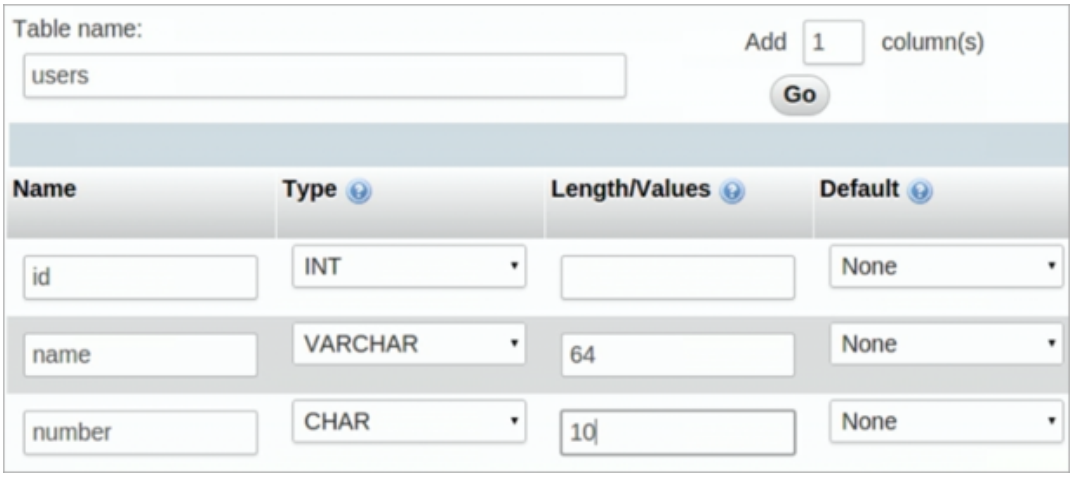
```
<?php

    require("../includes/config.php");

    if (empty($_POST["name"]))
    {
        apologize("Missing name");
    }
    else if (empty($_POST["number"]))
    {
        apologize("Missing number");
    }

?>
```

- Now if we reload the page, we get an error as we expected:



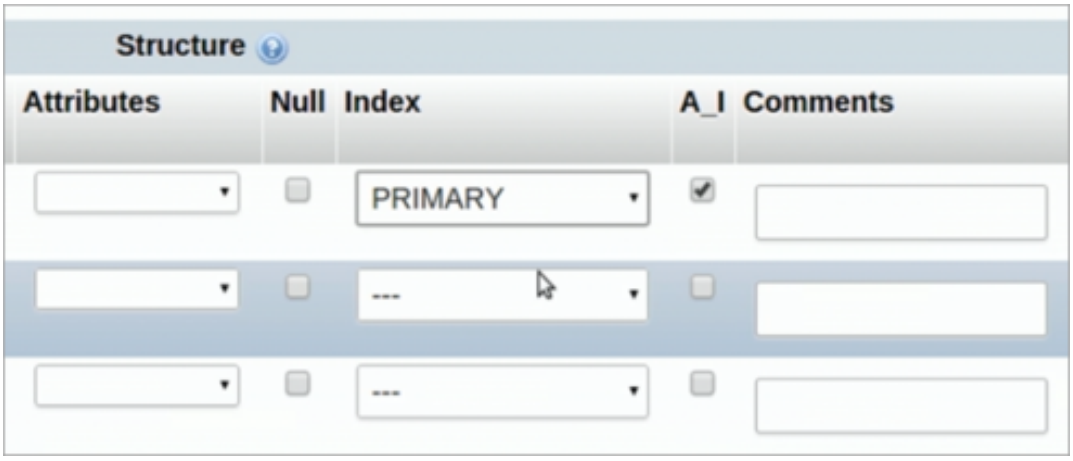
    # What we see is the `apologize` function printing out whatever we gave it as an argument.

- Now we should do something when the user provides us the information correctly. Let's go back to phpMyAdmin and quickly create a `users` table:

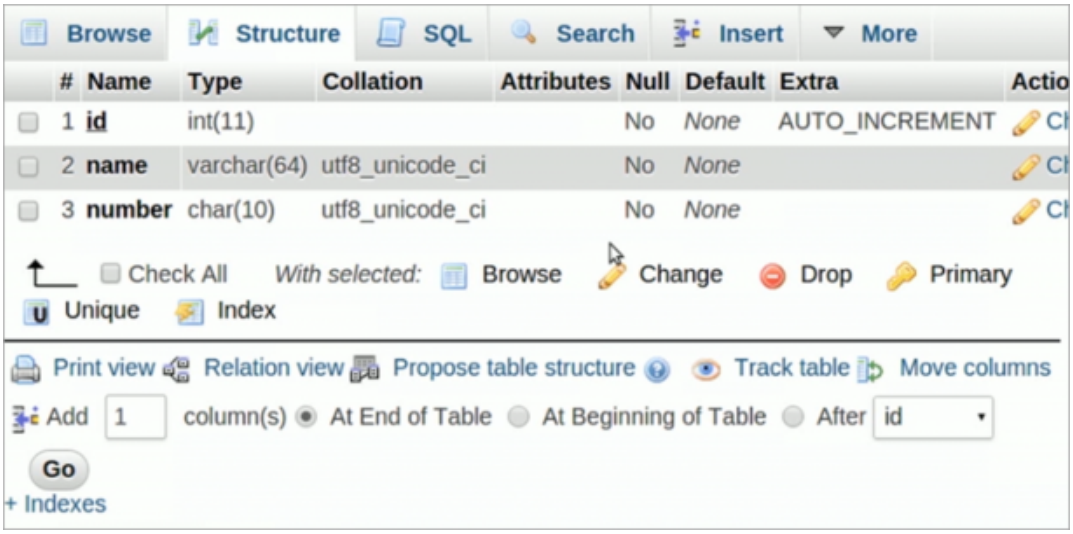


    # We'll make `name` at most `64` characters, and since we'll support US numbers for now, we'll make that fixed at `10` characters long.

- Then we'll make the `id` field `PRIMARY` and automatically increment it, but leave the others blank for now:



- After we click **Save** towards the bottom of that page, we can go back to our `users` table and click the **Structure** tab:



So we see the types of each field, along with other information.

- Now we can try to run a query on that table in our `register.php` source code:

```
<?php

    require("../includes/config.php");

    if (empty($_POST["name"]))
    {
        apologize("Missing name");
    }
    else if (empty($_POST["number"]))
    {
        apologize("Missing number");
    }

    query("INSERT INTO users (name, number) (?, ?)", $_POST["name"],
$_POST["number"]);

    render("success.php");

?>
```

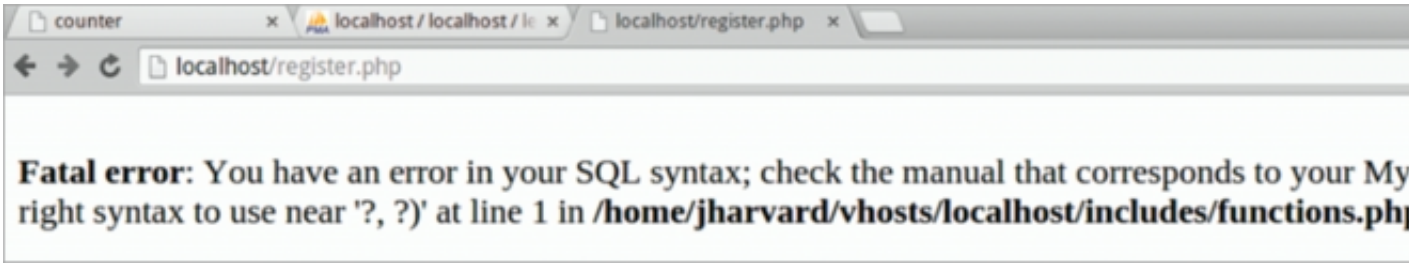
In the query we don't have to have backticks around `users` since it's relatively safe.

In the values for `name` and `number` we use `?`s, question marks, as placeholders, and add the variables afterwards.

And if all goes well, we'll just `render` a template called `success.php` which will just be:

```
<h1>Success!!!!</h1>
```

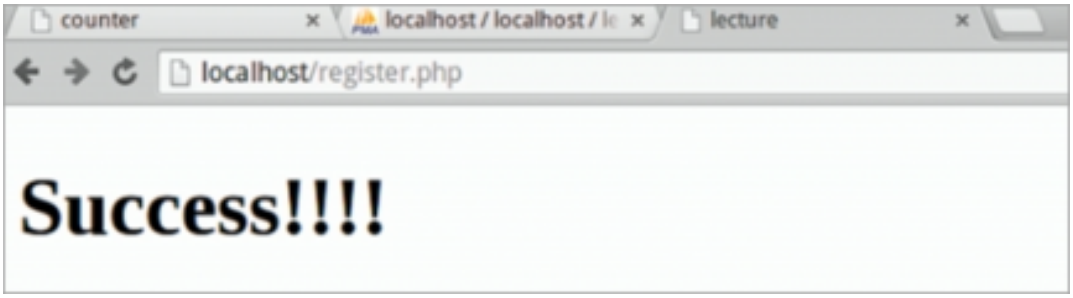
- So let's go to `register.php` in our browser, and when we click **Register**:



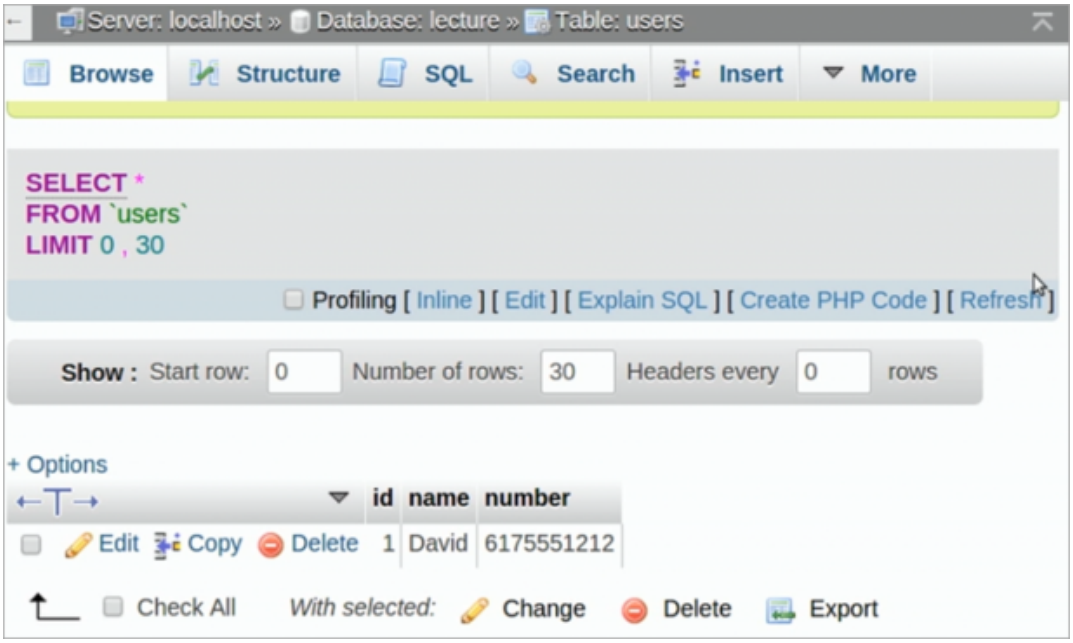
Dammit, we have an error in our SQL syntax. Right, we needed `VALUES` in our query:

```
query("INSERT INTO users (name, number) VALUES(?, ?)", $_POST["name"],
 $_POST["number"]);
```

- Now we can reload, and see "Success!!!!":



- And if we go back to phpMyAdmin and look at our table, we indeed see our information saved:



Texting

- We'll try to text David now, programmatically.
- Let's write a quick program called `text`[10]:

```
#!/usr/bin/env php
<?php

    require("includes/config.php");

    $rows = query("SELECT * FROM users");

    foreach ($rows as $row)
    {
        printf("Name is %s, and number is %s\n", $row["name"],
 $row["number"]);
    }

?>
```

[10] http://cdn.cs50.net/2014/fall/lectures/8/w/src8w/bin/text

We start by including our `config.php` file. Then we select all the rows in the `users` table, saving that in a variable called `$rows`, and for each of them, we print the `name` and `number` stored in the `$row`.

- Then we should `chmod a+x text` in our Terminal so we can execute it, and now it looks like it's working:

```
jharvard@appliance (~/vhosts/localhost): chmod a+x text
jharvard@appliance (~/vhosts/localhost): ./text
Name is David, and number is 6175551212
```

- Notice that we've written a script in PHP, that we can run in our command line, that has access to our entire database because of `config.php`. We can quickly use `register.php` to add Rob, and see that we can get both numbers back:

```
jharvard@appliance (~/vhosts/localhost): ./text
Name is David, and number is 6175551212
Name is Rob, and number is 6175551212
```

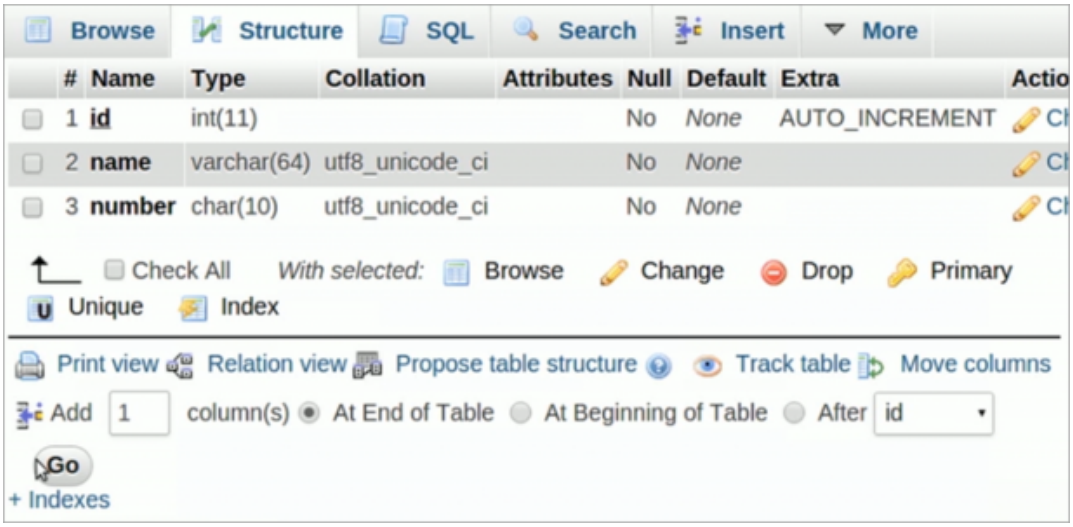
- As an aside, the last time David tried to give this demo two years ago, he used the entire CS50 database, but a bug in his loop caused him to send one email the first iteration, two emails the next, and so on. Anyways these emails were actually texts that read "Why aren't you in class?" (#davidhumor), and David got back lots of apologetic emails about how they were sorry for missing lecture "just this once" … anyways, this year we'll try this with just David's phone.
- In `functions.php`[11] we have the following function that we've written in advance:

[11] http://cdn.cs50.net/2014/fall/lectures/8/w/src8w/includes/functions.php

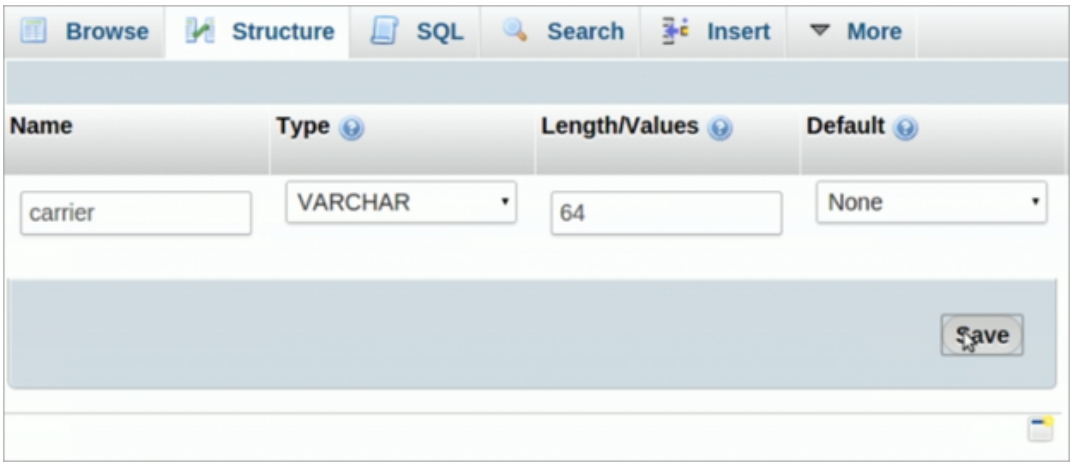
```
/**
 * Sends a text. Returns true on success, else false.
 */
function text($number, $carrier, $message)
{
    // determine address
    switch ($carrier)
    {
        case "AT&T":
            $address = "{$number}@txt.att.net";
            break;

        case "Verizon":
            $address = "{$number}@vtext.com";
            break;
    }
    if (!isset($address))
    {
        return false;
    }

    // instantiate mailer
    $mail = new PHPMailer();

    // use SMTP
    $mail->IsSMTP();
    $mail->Host = "smtp.fas.harvard.edu";
    $mail->Port = 587;
    $mail->SMTPSecure = "tls";

    // set From:
    $mail->SetFrom("jharvard@cs50.harvard.edu");

    // set To:
    $mail->AddAddress($address);

    // set body
    $mail->Body = $message;

    // send text
    if ($mail->Send())
    {
        return true;
    }
    else
    {
        return false;
```

So this function takes three arguments, and we see that a `switch` statement in PHP can take strings. It turns out, that with AT&T and Verizon, you can send an email to addresses like `{$number}@vtext.com`, and it will go to that number as a text message.

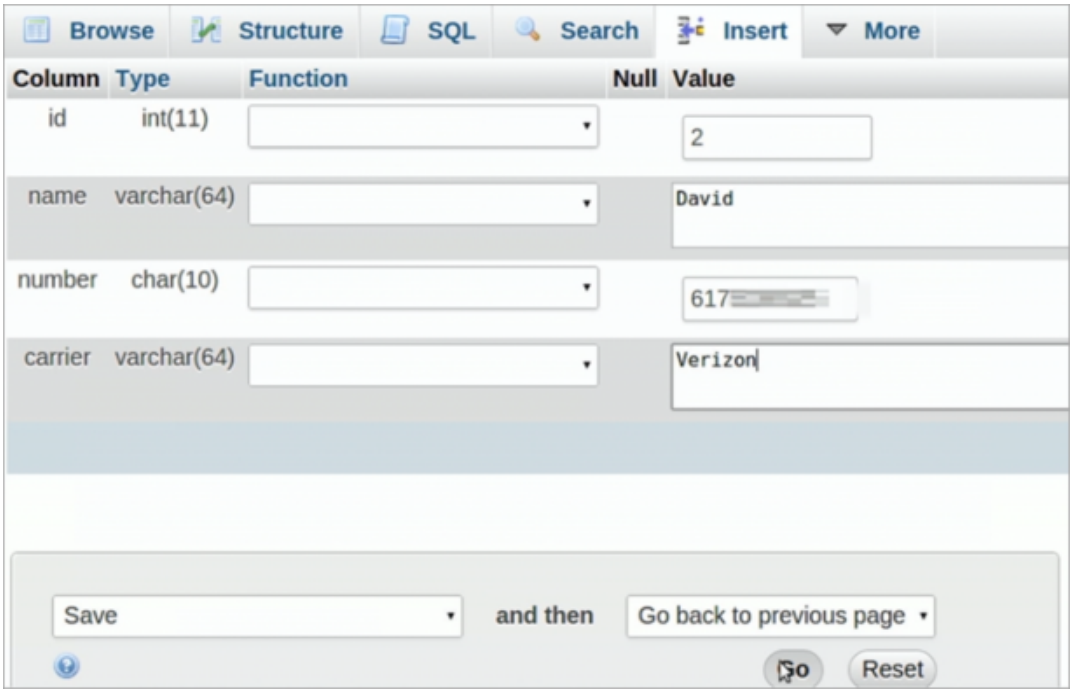
- We'll need to quickly add a field:



- We'll call this `carrier` as a `VARCHAR` and save it:



- Then we can click **Edit** for David and manually change his `carrier` to `Verizon`:



- And now we can change our `text` script to print out the `carrier`:

```
#!/usr/bin/env php
<?php

    require("includes/config.php");

    $rows = query("SELECT * FROM users");

    foreach ($rows as $row)
    {
        printf("Name is %s, and number is %s, carrier is %s\n",
 $row["name"], $row["number"], $row["carrier"]);
    }

?>
```

- And that works fine:

```
jharvard@appliance (~/vhosts/localhost): ./text
Name is David, and number is 6175551212, carrier is Verizon
```

- Now inside the `foreach` loop we'll not only `printf` the information, but send an actual message to the number:

```
#!/usr/bin/env php
<?php

    require("includes/config.php");

    $rows = query("SELECT * FROM users");

    foreach ($rows as $row)
    {
        printf("Name is %s, and number is %s, carrier is %s\n",
 $row["name"], $row["number"], $row["carrier"]);
        text($row["number"], $row["carrier"], "Don't screw up this year");
    }

?>
```

We'll call the `text` function, and pass in the three arguments, `$number`, `$carrier`, and `$message`.

- So let's run it:

```
jharvard@appliance (~/vhosts/localhost): ./text
Name is David, and number is 6175551212, carrier is Verizon
PHP Notice: Undefined variable: mail in /home/jharvard/vhosts/localhost/
includes/functions.php on line 166

Notice: Undefined variable: mail in /home/jharvard/vhosts/localhost/
includes/functions.php on line 166
PHP Fatal error: Call to a member function IsSMTP() on a non-object in /
home/jharvard/vhosts/localhost/includes/functions.php on line 166

Fatal error: Call to a member function IsSMTP() on a non-object in /home/
jharvard/vhosts/localhost/includes/functions.php on line 166
```

Turns out we were missing a line that includes the PHPMailer library.

- We make a few more fixes to various files, but then this error appeared:

```
jharvard@appliance (~/vhosts/localhost): ./text
Name is David, and number is 6175551212, carrier is Verizon
SMTP Error: The following recipients failed: 6175551212@vtext.com
```

- After a bit more debugging, we give up on this example, to be fixed Monday. :(