Getting Started

This assignment comes with some distribution code that you'll need to extract before starting. Attached is zip file called *cse_project.zip*.

1. Navigate to your web servers web root directory (this is usually the www or htdocs folder depending on your webserver choice).

2. Once inside extract the contents of *cse_project.zip*

you should see that `pset7` contains three subdirectories: `includes`, `public`, and `templates`. But more on those soon.

3. Next, ensure a few directories are world-executable by executing

```
chmod a+x pset7

chmod a+x pset7/public
```

so that the web server (and you, from a browser) will be able to access your work.

4. Then, navigate your way to `/pset7/public`
   you should see that `public` contains four subdirectories and three files. Ensure that the former are word-executable by executing the below.

   ```
   chmod a+x css fonts img js
   ```

5. Finally, ensure that the files within those directories are world-readable by executing the below.

   ```
   chmod a+r css/* fonts/* img/* js/*
   ```

For security's sake, don't make `pset7/includes` or `pset7/templates` world-executable (or their contents world-readable), as they shouldn't be accessible to the whole world (only to your PHP code, as you'll soon see).

Alright, time for a test! Open up Chrome and visit http://localhost/pset7

Okay, time for a heads-up. Anytime you create a new file or directory in `pset7` or some subdirectory therein for this assignment, you'll want to set its permissions with `chmod`. Thus far, we've relied on `a+r` and `a+x`, but let's empower you with more precise control over permissions.

Henceforth, for any PHP file, file, that you create, execute

```
chmod 600 file
```

so that it’s accessible only by you (and the webserver). After all, we don’t want visitors to see the contents of PHP files; rather, we want them to see the output of PHP files once executed (or, rather, interpreted) by the web server.

For any non-PHP file, file, that you create (or upload), execute

```
chmod 644 file
```

so that it’s accessible via a browser (if that’s indeed your intention).

And for any directory, directory, that you create, execute

```
chmod 711 directory
```

so that its contents are accessible via a browser (if that’s indeed your intention).

What’s with all these numbers we’re having you type? Well, `600` happens to mean `rw-------`, and so all PHP files are made readable and writable only by you; `644` happens to mean `rw-r--r--`, and so all non-PHP files are to be readable and writable by you and just readable by everyone else; and `711` happens to mean `rwx--x--x`, and so all directories are to be readable, writable, and executable by you and just executable by everyone else. Wait a minute, don’t we want everyone to be able to read (i.e., interpret) your PHP files? Nope! For security reasons, PHP-based web pages are interpreted "as you" (i.e., under you pc’s username).

Okay, still, what’s with all those numbers? Well, think of `rw-r--r--` as representing three triples of bits, the first triple of which, to be clear, is `rw-`. Imagine that `-` represents `0`, whereas `r`, `w`, and `x` represent `1`. And, so, this same triple (`rw-`) is just `110` in binary, or `6` in decimal! The other two triples, `r--` and `r--`, then, are just `100` and `100` in binary, or `4` and `4` in decimal! How, then, to express a pattern like `rw-r--r--` with numbers? Why, with `644`.

Actually, this is a bit of a white lie. Because you can represent only eight possible values with three bits, these numbers (`6`, `4`, and `4`) are not actually decimal digits but "octal." So you can now tell your friends that you speak not only binary, decimal, and hexadecimal, but octal as well.

Yahoo!

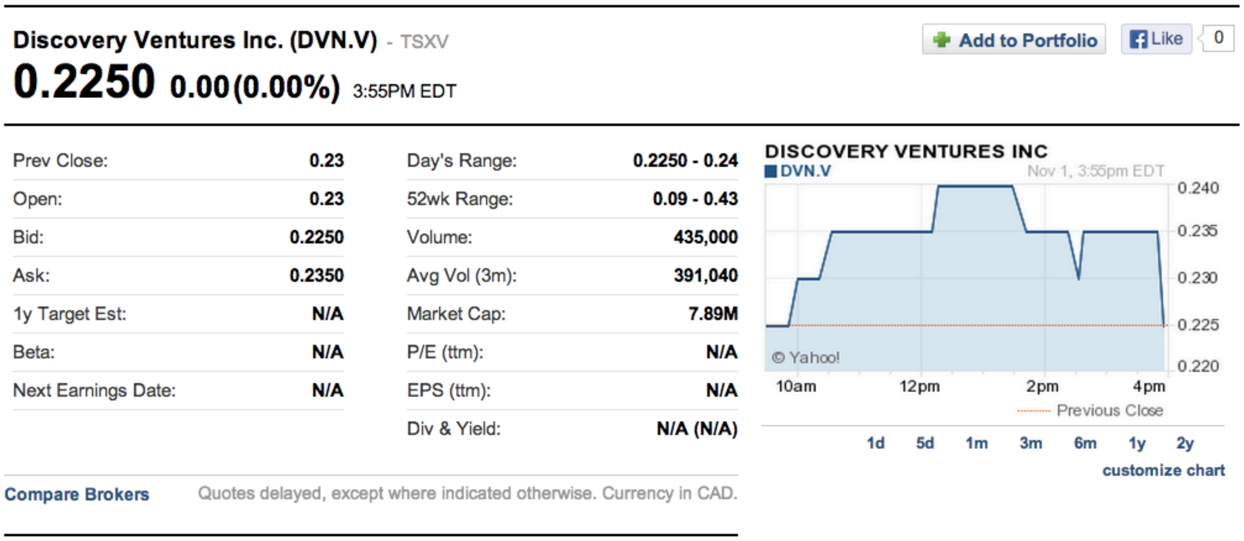
- If you’re not quite sure what it means to buy and sell stocks (i.e., shares of a company), surf on over to http://www.investopedia.com/university/stocks/ for a tutorial.

  You’re about to implement C$50 Finance, a Web-based tool with which you can manage portfolios of stocks. Not only will this tool allow you to check real stocks' actual prices and portfolios' values, it will also let you buy (okay, "buy") and sell (fine, "sell") stocks! (Per Yahoo’s fine print, "Quotes delayed [by a few minutes], except where indicated otherwise.")

- Just the other day, I received the stock tip below in my inbox!

  "Discovery Ventures Signs Letter Of Intent To Acquire The Willa Gold Deposit"

  Let’s get in on this opportunity now. Head on over to Yahoo! Finance at http://finance.yahoo.com/. Type the symbol for Discovery Ventures, **DVN.V**, into the text field in that page’s top-left corner and click **Get Quotes**. Odds are you’ll see a table like the below, which no one has apparently yet Liked!



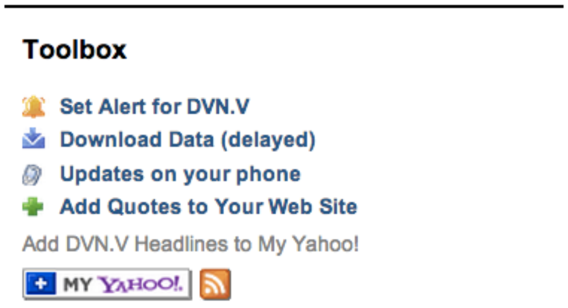
Discovery Ventures Inc. (DVN.V) - TSXV

0.2250 0.00(0.00%) 3:55PM EDT

Prev Close:	0.23	Day's Range:	0.2250 - 0.24
Open:	0.23	52wk Range:	0.09 - 0.43
Bid:	0.2250	Volume:	435,000
Ask:	0.2350	Avg Vol (3m):	391,040
1y Target Est:	N/A	Market Cap:	7.89M
Beta:	N/A	P/E (ttm):	N/A
Next Earnings Date:	N/A	EPS (ttm):	N/A
		Div & Yield:	N/A (N/A)

Quotes delayed, except where indicated otherwise. Currency in CAD.

  Wow, only 22.5 cents per share! That must be a good thing. Anyhow, notice how Yahoo reports a stock’s most recent (i.e., "Last Trade") price ($0.27) and more. Moreover, scroll down to the page’s bottom, and you should see a toolbox like the below.



- Looks like Yahoo lets you download all that data. Go ahead and click **Download Data** to download a file in CSV format (i.e., as comma-separated values). Open the file in Excel or any text editor (e.g., `atom`), and you should see a "row" of values, all excerpted from that table. It turns out that the link you just clicked led to the URL below.

  http://download.finance.yahoo.com/d/quotes.csv?s=DVN.V&f=sl1d1t1c1ohgv&e=.csv

  Notice how Discovery Ventures' symbol is embedded in this URL (as the value of the HTTP parameter called `s`); that's how Yahoo knows whose data to return. Notice also the value of the HTTP parameter called `f`; it's a bit cryptic (and officially undocumented), but the value of that parameter tells Yahoo which fields of data to return to you. If curious as to what they mean, head to http://www.gummy-stuff.org/Yahoo-data.htm.

  It's worth noting that a lot of websites that integrate data from other websites do so via "screen scraping," a process that requires writing programs that parse (or, really, search) HTML for data of interest (e.g., air fares, stock prices, etc.). Writing a screen scraper for a site tends to be a nightmare, though, because a site's markup is often a mess, and if the site changes the format of its pages overnight, you need to re-write your scraper.

  Thankfully, because Yahoo provides data in CSV, C$50 Finance will avoid screen scraping altogether by downloading (effectively pretending to be a browser) and parsing CSV files instead. Even more thankfully, we've written that code for you!

  In fact, let's turn our attention to the code you've been given.

- Navigate your way to `pset7/public` and open up `index.php` with `atom`. (Remember how?) Know that `index.php` is the file that's loaded by default when you visit a URL like http://localhost/pset7/. Well, it turns out there's not much PHP code in this file. And there isn't any HTML at all. Rather, `index.php` "requires" `config.php` (which is in a directory called `includes` in `index.php`'s parent directory). And `index.php` then calls `render` (a function implemented in a file called `functions.php` that can also be found inside of `includes`) in order to render (i.e., output) a template called `portfolio.php` (which is in a directory called `templates` in `index.php`'s parent directory). Phew, that was a mouthful.

  It turns out that `index.php` is considered a "controller," whereby its purpose in life is to control the behavior of your website when a user visits http://localhost/pset7/ (or, equivalently, http://localhost/pset7/index.php). Eventually, you'll need to add some more PHP code to this file in order to pass more than just title to render. But for now, let's take a look at `portfolio.php`, the template that this controller ultimately renders.

  Navigate your way to `pset7/templates` and open up `portfolio.php` with `atom`. Ah, there's some HTML. Of course, it's not very interesting HTML, but it does explain why your website is "under construction," thanks to the GIF referenced therein.

  Now navigate your way to `pset7/includes` and open up `config.php` with `atom`. Recall that `config.php` was required by `index.php`. Notice how `config.php` first enables display of all errors (and warnings and notices, which are less severe errors) so that you're aware of any syntactical mistakes (and more) in your code. Notice, too, that `config.php` itself requires two other files: `constants.php` and `functions.php`. Next, `config.php` calls `session_start` in order to enable `$_SESSION`, a "superglobal" variable via which we'll remember that a user is logged in. (Even though HTTP is a "stateless" protocol, whereby browsers are supposed to disconnect from servers as soon as they're done downloading pages, "cookies" allow browsers to remind servers who they or, really, you are on subsequent requests for content. PHP uses "session cookies" to provide you with `$_SESSION`, an associative array in which you can store any data to which you'd like to have access for the duration of some user's visit. The moment a user ends his or her "session" (i.e., visit) by quitting his or her browser, the contents of `$_SESSION` are lost for that user specifically because the next time that user visits, he or she will be assigned a new cookie!) Meanwhile, `config.php` uses a "regular expression" (via a call to `preg_match`) to redirect the users to `login.php` anytime they visit some page other than `login.php`, `logout.php`, and `register.php`, assuming `$_SESSION["id"]` isn't yet set. In other words, that block of code requires users to log in if they aren't logged in already (and if they aren't already at one of those three pages).

Okay, now open up `functions.php` with `atom`. Interesting, it looks like `functions.php` requires `constants.php`. More on that file, though, in a moment. It looks like `functions.php` also defines a bunch of functions, the first of which is `apologize`, which you can call anytime you need to apologize to the user (because they made some mistake). Defined next is `dump`, which you're welcome to call anytime you want to see the contents (perhaps recursively) of some variable while developing your site. That function is only for diagnostic purposes, though. Be sure to remove all calls thereto before submitting your work. Next in the file is `logout`, a function that logs users out by destroying their sessions. Thereafter is `lookup`, a function that queries Yahoo Finance for stocks' prices and more. More on that, though, in a bit. Up next is `query`, a function that executes a SQL query and then returns the result set's rows, if any. Below it is `redirect`, a function that allows you to redirect users from one URL to another. Last in the file is `render`, the function that `index.php` called in order to render `portfolio.php`. The function then "extracts" those values into the local scope (whereby a key of `"foo"` with a value of `"bar"` in `$values` becomes a local variable called `$foo` with a value of `"bar"`). And it then requires `header.php` followed by `$template` followed by `footer.php`, effectively outputting all three.

In fact, navigate your way back to `pset7/templates` and open up `header.php` and `footer.php` in `atom`. Ah, even more HTML! Thanks to render, those files' contents will be included at the top and bottom, respectively, of each of your pages. As a result, each of your pages will have access to Twitter's Bootstrap library, per the link and script tags therein. And each page will have at least four `div` elements, three of which have unique IDs (`top`, `middle`, and `bottom`), if only to make styling them with CSS easier.

Even more interestingly, though, notice how `header.php` conditionally outputs `$title`, if it is set. Remember how `index.php` contained the below line of code?

```
render("portfolio.php", ["title" => "Portfolio"]);
```

Well, because `render` calls `extract` on that second argument, an array, before requiring `header.php`, `header.php` ends up having access to a variable called `$title`. Neat, eh? You can pass even more values into a template simply by separating such key/value pairs with a comma, as in the below.

```
render("portfolio.php", ["cash" => 10000.00, "title" =>
"Portfolio"]);
```

Okay, now open up `constants.php` in `pset7/includes` (which, recall, `config.php` required). Suffice it to say, this file defines a bunch of constants, but you shouldn't need to change any of them.

Navigate your way back to `pset7/public` and open up `login.php`, another controller, with `atom`. This controller's a bit more involved than `index.php` as it handles the authentication of users. Read through its lines carefully, taking note of how it how it queries the web server's MySQL database using that `query` function from `functions.php`. That function (which we wrote) essentially simplifies use of PDO (PHP Data Objects), a library with which you can query MySQL (and other) databases. Per its definition in `functions.php`, the function accepts one or more arguments: a string of SQL followed by a comma-separated list of zero or more parameters that can be plugged into that string, not unlike `printf`. Whereas `printf` uses `%d`, `%s`, and the like for placeholders, though, `query` simply relies on question marks, no matter the type of value. And so the effect of

```
query("SELECT * FROM users WHERE username = ?", $_POST["username"]);
```

in `login.php` is to replace `?` with whatever username has been submitted (via POST) via an HTML form. (The function also ensures that any such placeholders' values are properly escaped so that your code is not vulnerable to "SQL injection attacks.") For instance, suppose that President Skroob tries to log into C$50 Finance by inputting his username and password. That line of code will ultimately execute the SQL statement below.

```
SELECT * FROM users WHERE username='skroob'
```

Beware, though. PHP is weakly (i.e., loosely) typed, and so functions like query can actually return different types. Indeed, even though query usually returns an array of rows (thanks to its invocation of PDO's `fetchAll`), it can also return `false` in case of errors. But, unlike `SELECT`s, some SQL queries (e.g., `DELETE`s, `UPDATE`s, and `INSERT`s) don’t actually return rows, and so the array that `query` returns might sometimes be empty. When checking the return value of `query` for `false`, then, take care not to use `==`, because it turns out than an empty array is `==` to `false` because of implicit casting. But an empty array does not necessarily signify an error, only `false` does! Use, then, PHP’s `===` (or `!==`) operator when checking return values for `false`, which compares its operands' values and types (not just their values), as in the below.

```
$result = query("INSERT INTO users (username, hash, cash) VALUES(?, ?, 10000.00)", $_POST["username"], crypt($_POST["password"]));
if ($result === false)
{
    // the INSERT failed, presumably because username already existed
}
```

See http://php.net/manual/en/language.operators.comparison.php for more details.

Anyhow, notice too that `login.php` "remembers" that a user is logged in by storing his or her unique ID inside of `$_SESSION`. As before, this controller does not contain any HTML. Rather, it calls `apologize` or renders `login_form.php` as needed. In fact, open up `login_form.php` in `pset7/templates` with `atom`. Most of that file is HTML that’s stylized via some of Bootstrap’s CSS classes, but notice how the HTML form therein POSTs to `login.php`. Just for good measure, take a peek at `apology.php` while you’re in that directory as well. And also take a peek at `logout.php` back in `pset7/public` to see how it logs out a user.

Alright, now navigate your way to `public/css` and open up `styles.css` with `atom`. Notice how this file already has a few "selectors" so that you don’t have to include style attributes the elements matched by those selectors. No need to master CSS for this assignment, but do know that you should not have more than one `div` element per page whose `id` attribute has a value of `top`, more than one `div` element per page whose `id` attribute has a value of `middle`, or more than one `div` element per page whose `id` attribute has a value of `bottom`; an `id` must be unique. In any case, you are welcome to modify `styles.css` as you see fit.

You’re also welcome to poke around `pset7/public/js`, which contains some JavaScript files. But no need to use or write any JavaScript for this assignment. Those files are just there in case you’d like to experiment.

Phew, that was a lot. Help yourself to a snack.

Task #1

Alright, let's talk about that database we keep mentioning. So that you have someplace to store users' portfolios, the comes with a MySQL file (called `pset7.sql`). Open up `pset7.sql`, which you downloaded. You should see a whole bunch of SQL statements. Highlight them all, select **Edit > Copy** (or hit control-c), then return to phpMyAdmin. Click phpMyAdmin's **SQL** tab, and paste everything you copied into that page's big text box (which is below **Run SQL query/queries on server "127.0.0.1"**). Skim what you just pasted to get a sense of the commands you're about to execute, then click **Go**. You should then see a greenish banner indicating success (i.e., **1 row affected**). In phpMyAdmin's top-left corner, you should now see link to a database called **pset7**, beneath which is a link to a table called **users**. (If you don't, try reloading the page.)

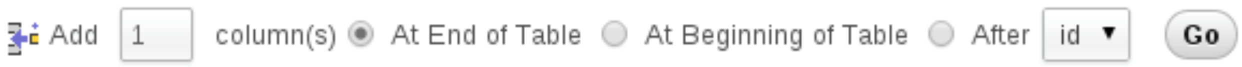
Return to `http://localhost/pset7` and reload that page. Then try to log in again with a username of **skroob** and a password of **12345**. 0:-)

- Head back to http://localhost/phpMyAdmin/ using Chrome to access phpMyAdmin. Log in and you should then find yourself at phpMyAdmin's main page. in the top-left corner of which is that table called **users**. Click the name of that table to see its contents. Ah, some familiar folks. In fact, there's President Skroob's username and a hash of his password. Now click the tab labeled **Structure**. Ah, some familiar fields. Recall that `login.php` generates queries like the below.

```
SELECT id FROM users WHERE username='skroob'
```

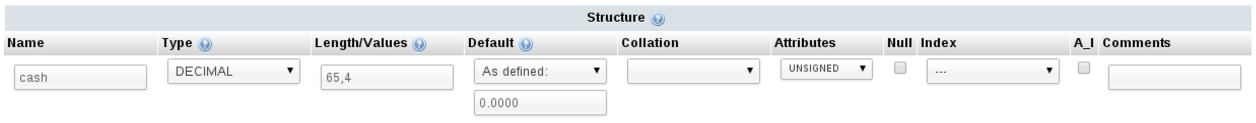
As phpMyAdmin makes clear, this table called users contains three fields: `id` (the type of which is an `INT` that's `UNSIGNED`) along with `username` and `hash` (each of whose types is `VARCHAR`). It appears that none of these fields is allowed to be `NULL`, and the maximum length for each of each of `username` and `hash` is `255`. A neat feature of `id`, meanwhile, is that it will `AUTO_INCREMENT`: when inserting a new user into the table, you needn't specify a value for `id`; the user will be assigned the next available `INT`. Finally, if you click **Indexes** (above **Information**), you'll see that this table's `PRIMARY` key is `id`, the implication of which is that (as expected) no two users can share the same user ID. Recall that a primary key is a field with no duplicates (i.e., that is guaranteed to identify rows uniquely). Of course, `username` should also be unique across users, and so we have also defined it to be so (per the additional **Yes** under **Unique**). To be sure, we could have defined username as this table's primary key. But, for efficiency's sake, the more conventional approach is to use an `INT` like `id`. Incidentally, these fields are called "indexes" because, for primary keys and otherwise unique fields, databases tend to build "indexes," data structures that enable them to find rows quickly by way of those fields.

Make sense?

- Okay, let’s give each of your users some cash. Assuming you’re still on phpMyAdmin’s **Structure** tab, you should see a form with which you can add new columns. Click the radio button immediately to the left of **After**, select **hash** from the drop-down menu, as in the below, then click **Go**.
- 

Add 1 column(s) [✓ At End of Table] At Beginning of Table After id ▾ Go

Via the form that appears, define a field called cash of type `DECIMAL` with a length of `65,4`, with a default value of `0.0000`, and with an attribute of `UNSIGNED`, as in the below, then click **Save**.



Structure									
Name	Type	Length/Values	Default	Collation	Attributes	Null	Index	A_I	Comments
cash	DECIMAL	65,4	As defined: 0.0000		UNSIGNED	☐	---	☐	

If you pull up the documentation for MySQL at http://dev.mysql.com/doc/refman/5.6/en/numeric-types.html, you’ll see that the `DECIMAL` data type is used to "store exact numeric data values." A length of `65,4` for a `DECIMAL` means that values for `cash` can have no more than 65 digits in total, 4 of which can be to the right of the decimal point. (Ooo, fractions of pennies. Sounds like **Office Space**.)

Okay, return to the tab labeled **Browse** and give everyone $10,000.00 manually. (In theory, we could have defined `cash` as having a default value of `10000.000`, but, in general, best to put such settings in code, not your database, so that they’re easier to change.) The easiest way is to click **Check All**, then click **Change** to the right of the pencil icon. On the page that appears, change `0.0000`to `10000.0000` for each of your users, then click **Go**. Won’t they be happy!

- It’s now time to code! Let’s empower new users to register.

  Return to a terminal window, navigate your way to `pset7/templates` and execute the below. (You are welcome, particularly if among those more comfortable, to stray from these filename conventions and structure your site as you see fit, so long as your implementation adheres to all other requirements.)

```
cp login_form.php register_form.php
```

  Then open up `register_form.php` with `atom` and change the value of form’s `action` attribute from `login.php` to `register.php`. Next add an additional field of type `password` to the HTML form called `confirmation` so that users are prompted to input their choice of passwords twice (to discourage mistakes). Finally, change the button’s text from `Log In` to `Register` and change

```
or <a href="register.php">register</a> for an account
```

to

```
or <a href="login.php">log in</a>
```

so that users can navigate away from this page if they already have accounts.

Then, using `atom`, create a new file called `register.php` with the contents below, taking care to save it in `pset7/public`.

```
<?php

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

    // if form was submitted
    if ($_SERVER["REQUEST_METHOD"] == "POST")
    {
        // TODO
    }
    else
    {
        // else render form
        render("register_form.php", ["title" => "Register"]);
    }

?>
```

Alright, let’s take a look at your work! Bring up http://localhost/pset7/login.php in Chrome and click that page’s link to `register.php`. You should then find yourself at http://localhost/pset7/register.php. If anything appears awry, feel free to make tweaks to `register_form.php` or `register.php`. Just be sure to save your changes and then reload the page in the browser.

Of course, `register.php` doesn't actually register users yet, so it's time to tackle that `TODO`! Allow us to offer some hints.

- If `$_POST["username"]` or `$_POST["password"]` is empty or if `$_POST["password"]` does not equal `$_POST["confirmation"]`, you'll want to inform registrants of their error.
- To insert a new user into your database, you might want to call

  ```
  query("INSERT INTO users (username, hash, cash) VALUES(?, ?,
  10000.00)", $_POST["username"], crypt($_POST["password"]));
  ```

  though we leave it to you to decide how much cash your code should give to new users.
- Know that `query` will return `false` if your `INSERT` fails (as can happen if, say, `username` already exists). Be sure to check for false with `===` and not `==`.
- If, though, your `INSERT` succeeds, know that you can find out which `id` was assigned to that user with code like the below.

```
o $rows = query("SELECT LAST_INSERT_ID() AS id");
```

```
$id = $rows[0]["id"];
```

- If registration succeeds, you might as well log the new user in (as by "remembering" that `id` in `$_SESSION`), thereafter redirecting to `index.php`.

• All done with `register.php`? Ready to test? Head back to http://localhost/pset7/register.php using Chrome and try to register a new username. If you reach `index.php`, odds are you done good! Confirm as much by returning to phpMyAdmin, clicking once more that tab labeled **Browse** for the table called `users`. May that you see your new user. If not, it's time to debug!

Be sure, incidentally, that any HTML generated by `register.php` is valid, as by ctrl- or right-clicking on the page in Chrome, selecting **View Page Source**, highlighting and copying the source code, and then pasting it into the W3C's validator at http://validator.w3.org/#validate_by_input and then clicking **Check**. Ultimately, the **Result** of checking your page for validity via the W3C's validator should be **Passed** or **Tentatively passed**, in which case you should see a friendly green banner. Warnings are okay. Errors (and big red banners) are not. Note that you won't be able to "validate by URI" at http://validator.w3.org/#validate_by_uri, since your appliance isn't accessible on the public Internet!

- Do bear in mind as you proceed further that you are welcome to play with and learn from the live implementation of C$50 Finance at http://finance.cs50.net/
- In particular, you are welcome to register with as many (fake) usernames as you would like in order to play. And you are welcome to view the pages' HTML and CSS (by viewing our source using your browser) so that you might learn from or improve upon our own design. If you wish, feel free to adopt our HTML and CSS as your own.

  But do not feel that you need copy the design. In fact, for this assignment, you may modify every one of the files we have given you to suit your own tastes as well as incorporate your own images and more. In fact, may that your version of C$50 Finance be nicer than this!

Task #2

- Okay, now it's time to empower users to look up quotes for individual stocks. Odds are you'll want to create a new controller called, say, `quote.php` plus two new templates, the first of which displays an HTML form via which a user can submit a stock's symbol, the second of which displays, minimally, a stock's latest price (if passed, via render, an appropriate value).

  How to look up a stock's latest price? Well, recall that function called `lookup` in `functions.php`. Odds are you'll want to call it with code like the below.

  ```
  $stock = lookup($_POST["symbol"]);
  ```

  Assuming the value of `$_POST["symbol"]` is a valid symbol for an actual stock, lookup will return an associative array with three keys for that stock, namely its `symbol`, its `name`, and its `price`. Know that you can use PHP's `number_format` function (somehow!) to format price to at least two decimal places but no more than four decimal places. See http://php.net/manual/en/function.number-format.php for details.

  Of course, if the user submits an invalid symbol (for which lookup returns false), be sure to inform the user somehow. Be sure, too, that any HTML generated by your templates is valid, per the W3C's validator.

Task #3

- Now it's time to do a bit of design. At present, your database has no way of keeping track of users' portfolios, only users themselves. (By "portfolio," we mean a collection of stocks (i.e., shares of companies) that some user owns.) It doesn't really make sense to add additional fields to users itself in order to keep track of the stocks owned by users (using, say, one field per company owned). After all, how many different stocks might a user own? Better to maintain that data in a new table altogether so that we do not impose limits on users' portfolios or waste space with potentially unused fields.

  Exactly what sort of information need we keep in this new table in order to "remember" users' portfolios? Well, we probably want a field for users' IDs (`id`) so that we can cross-reference holdings with entries in `users`. We probably want to keep track of stocks owned by way of their symbols since those symbols are likely shorter (and thus more efficiently stored) than stocks' actual names. Of course, you could also assign unique numeric IDs to stocks and remember those instead of their symbols. But then you'd have to maintain your own database of companies, built up over time based on data from, say, Yahoo. It's probably better (and it's certainly simpler), then, to keep track of stocks simply by way of their symbols. And we probably want to keep track of how many shares a user owns of a particular stock. In other words, a table with three fields (`id`, `symbol`, and `shares`) sounds pretty good, but you're welcome to proceed with a design of your own. Whatever your decision, head back to phpMyAdmin and create this new table, naming it however you see fit. To create a new table, click **pset7** in phpMyAdmin's top-left corner, and on the screen that appears, input a name for your table and some number of columns below **Create table**, then click **Go**. On the screen that appears next, define (in any order) each of your fields.

  If you decide to go with three fields (namely `id`, `symbol`, and `shares`), realize that `id` should not be defined as a primary key in this table, else each user could own no more than one company's stock (since his or her `id` could not appear in more than one row). Realize, too, that you shouldn't let some `id` and some `symbol` to appear together in more than one row. Better to consolidate users' holdings by updating shares whenever some user sells or buys more shares of some stock he or she already owns. A neat way to impose this restriction while creating your table is to define a "joint primary key" by selecting an **Index** of `PRIMARY` for both `id` and `symbol`. That way, `INSERT` will fail if you try to insert more than one row for some pair of id and symbol. We leave it to you, though, to decide your fields' types. (If you include `id` in this table, know that its type should match that in `users`. But don't specify `AUTO_INCREMENT` for that field in this new table, as you only want auto-incrementation when user IDs are created for new users. And don't call your table `tbl`.) When done defining your table, click **Save**!

- Before we let users buy and sell stocks themselves, let's give some shares to President Skroob and friends at no charge. Click, in phpMyAdmin's left-hand frame, the link to `users` and remind yourself of your current users' IDs. Then click, in phpMyAdmin's left-hand frame, the link to your new table (for users' portfolios), followed by the tab labeled **Insert**. Via this interface, go ahead and "buy" some shares of some stocks on behalf of your users by manually inserting rows into this table. (You may want to return to Yahoo! Finance to look up some actual symbols.) No need to debit their `cash` in `users`; consider these shares freebies.

Once you've bought your users some shares, let's see what you did. Click the tab labeled **SQL** and run a query like the below, where `tbl` represents your new table's name.

```
SELECT * FROM tbl WHERE id = 7
```

Assuming `7` is President Skroob's user ID, that query should return all rows from `tbl` that represent the president's holdings. If the only fields in table are, say, `id`, `symbol`, and `shares`, then know that the above is actually equivalent to the below.

```
SELECT id, symbol, shares FROM tbl WHERE id = 7
```

If, meanwhile, you'd like to retrieve only President Skroob's shares of Discovery Ventures, you might like to try a query like the below.

```
SELECT shares FROM tbl WHERE id = 7 AND symbol = 'DVN.V'
```

If you happened to buy President Skroob some shares of that company, the above should return one row with one column, the number of shares. If you did not get buy any such shares, the above will return an empty result set.

Incidentally, via this **SQL** tab, you could have inserted those "purchases" with `INSERT` statements. But phpMyAdmin's GUI saved you the trouble.

Alright, let's put this knowledge to use. It's time to let users peruse their portfolios! Overhaul `index.php` (a controller) and `portfolio.php` (a template) in such a way that they report each of the stocks in a user's portfolio, including number of shares and current price thereof, along with a user's current cash balance. Needless to say, `index.php` will need to invoke `lookup` much like `quote.php` did, though perhaps multiple times. And know that a PHP script can certainly invoke `query` multiple times, even though, thus far, we've seen it used in a file no more than once. And you can certainly iterate over the array it returns in a template (assuming you pass it in via `render`). For instance, if your goal is simply to display, say, President Skroob's holdings, one per row in some HTML table, you can generate rows with code like the below, where `$positions` is an array of associative arrays, each of which represents a position (i.e., a stock owned).

```
<table>
    <?php
```

```
        foreach ($positions as $position)
        {
            print("<tr>");
            print("<td>" . $position["symbol"] . "</td>");
            print("<td>" . $position["shares"] . "</td>");
            print("<td>" . $position["price"] . "</td>");
            print("</tr>");
        }

    ?>
</table>
```

Alternatively, you can avoid using the concatenation operator (`.`) via syntax like the below:

```
<table>
    <?php

        foreach ($positions as $position)
        {
            print("<tr>");
            print("<td>{$position["symbol"]}</td>");
            print("<td>{$position["shares"]}</td>");
            print("<td>{$position["price"]}</td>");
            print("</tr>");
        }

    ?>
</table>
```

Note that, in the above version, we've surrounded the lines of HTML with double quotes instead of single quotes so that the variables within (`$position["symbol"]`, `$position["shares"]`, and `$position["price"]`) are interpolated (i.e., substituted with their values) by PHP's interpreter; variables between single quotes are not interpolated. And we've also surrounded those same variables with curly braces so that PHP realizes they're variables; variables with simpler syntax (e.g., `$foo`) do not require the curly braces for interpolation. (It's fine to use double quotes inside those curly braces, even though we've also used double quotes to surround the entire argument to `print`.) Anyhow, though commonly done, generating HTML via calls to `print` isn't terribly elegant. An alternative approach, though still a bit inelegant, is code more like the below.

```
<?php foreach ($positions as $position): ?>


    <tr>
        <td><?= $position["symbol"] ?></td>
        <td><?= $position["shares"] ?></td>
        <td><?= $position["price"] ?></td>
    </tr>


<?php endforeach ?>
```

Of course, before you can even pass `$positions` to `portfolio.php`, you'll need to define it in `index.php`. Allow us to suggest code like the below, which combines names and prices from `lookup` with shares and symbols, as might be returned as `$rows` from `query`.

```
$positions = [];
foreach ($rows as $row)
{
    $stock = lookup($row["symbol"]);
    if ($stock !== false)
    {
        $positions[] = [
            "name" => $stock["name"],
            "price" => $stock["price"],
            "shares" => $row["shares"],
            "symbol" => $row["symbol"]
        ];
    }
}
```

Note that, with this code, we're deliberately create a new array of associative arrays (`$positions`) rather than add names and prices to an existing array of associative arrays (`$rows`). In the interests of good design, it's generally best not to alter functions' return values (like `$rows` from `query`).

Now, much like you can pass a page's title to render, so can you pass these positions, as with the below.

```
render("portfolio.php", ["positions" => $positions, "title" =>
"Portfolio"]);
```

Of course, you'll also need to pass a user's current cash balance from `index.php` to `portfolio.php` via `render` as well, but we leave it to you to figure out how.

To be clear, in the spirit of MVC, though, do take care not to call `lookup` inside of that (or any other) template; you should only call `lookup` in controllers. Even though templates (aka views) can contain PHP code, that code should only be used to print and/or iterate over data that's been passed in (as via render) from a controller.

As for what HTML to generate, look, as before, to https://www.cs50.net/finance/ for inspiration or hints. But do not feel obliged to mimic our design. Make this website your own! Although any HTML and PHP code that you yourself write should be pretty-printed (i.e., nicely indented), it's okay if lines exceed 80 characters in length. HTML that you generate dynamically (as via calls to `print`), though, does not need to be pretty-printed.

As before, be sure to display stocks' prices and users' cash balances to at least two decimal places but no more than four.

Incidentally, though we keep using President Skroob in examples, your code should work for whichever user is logged in.

As always, be sure that the HTML generated by `index.php` is valid.

Task #4

And now it is time to implement the ability to sell with a controller called, say, `sell.php` and some number of templates. We leave the design of this feature to you. But know that you can delete rows from your table (on behalf of, say, President Skroob) with SQL like the below.

```
DELETE FROM tbl WHERE id = 7 AND symbol = 'DVN.V'
```

We leave it to you to infer exactly what that statement should do. Of course, you could try the above out via phpMyAdmin's **SQL**tab. Now what about the user's cash balance? Odds are, your user is going to want the proceeds of all sales. So selling a stock involves updating not only your table for users' portfolios but `users` as well. We leave it to you to determine how to compute how much cash a user is owed upon sale of some stock. But once you know that amount (say, $500), SQL like the below should take care of the deposit (for, say, President Skroob).

```
UPDATE users SET cash = cash + 500 WHERE id = 7
```

Of course, if the database or web server happens to die between this `DELETE` and `UPDATE`, President Skroob might lose out on all of that cash. You need not worry about such cases! It's also possible, because of multithreading and, thus, race conditions, that a clever president could trick your site into paying out more than once. You need not worry about such cases either! Though, if you're so very inclined, you can employ SQL transactions (with InnoDB tables). See http://dev.mysql.com/doc/refman/5.6/en/sql-syntax-transactions.html for reference.

It's fine, for simplicity, to require that users sell all shares of some stock or none, rather than only a few. Needless to say, try out your code by logging in as some user and selling some stuff. You can always "buy" it back manually with phpMyAdmin.

As always, be sure that your HTML is valid!

Task #5

Now it's time to support actual buys. Implement the ability to buy, with a controller called, say, `buy.php` and some number of templates. (As before, you need not worry about interruptions of service or race conditions.) The interface with which you provide a user is entirely up to you, though, as before, feel free to look to https://www.cs50.net/finance for inspiration or hints. Of course, you'll need to ensure that a user cannot spend more cash than he or she has on hand. And you'll want to make sure that users can only buy whole shares of stocks, not fractions thereof. For this latter requirement, know that a call like

```
preg_match("/^\d+$/", $_POST["shares"])
```

will return `true` if and only if `$_POST["shares"]` contains a non-negative integer, thanks to its use of a regular expression. See http://www.php.net/preg_match for details. Take care to apologize to the user if you must reject their input for any reason. In other words, be sure to perform rigorous error-checking. (We leave to you to determine what needs to be checked!)

When it comes time to store stocks' symbols in your database table, take care to store them in uppercase (as is convention), no matter how they were inputted by users, so that you don't accidentally treat, say, `dvn.v` and `DVN.V` as different stocks. Don't force users, though, to input symbols in uppercase.

Incidentally, if you implemented your table for users' portfolios as we did ours (with that joint primary key), know that SQL like the below (which, unfortunately, wraps onto two lines) will insert a new row into table unless the specified pair of `id` and `symbol` already exists in some row, in which case that row's number of shares will simply be increased (say, by `10`).

```
INSERT INTO table (id, symbol, shares) VALUES(7, 'DVN.V', 10) ON
DUPLICATE KEY UPDATE shares = shares + VALUES(shares)
```

As always, be sure to bang on your code. And be sure that your HTML is valid!

Task #6

Alright, so your users can now buy and sell stocks and even check their portfolio's value. But they have no way of viewing their history of transactions.

Enhance your implementations for buying and selling in such a way that you start logging transactions, recording for each:

- Whether a stock was bought or sold.
- The symbol bought or sold.
- The number of shares bought or sold.
- The price of a share at the time of transaction.
- The date and time of the transaction.

Then, by way of a controller called, say, `history.php` and some number of templates, enable users to peruse their own history of transactions, formatted as you see fit. Be sure that your HTML is valid!

Task #7

- Phew. Glance back at `index.php` now and, if not there already, make that it somehow links to, at least, `buy.php`, `history.php`, `logout.php`, `quote.php`, and `sell.php` (or their equivalents) so that each is only one click away from a user's portfolio!
- And now the icing on the cake. Only one feature to go, but you get to choose. Implement at least one (1) of the features below. You may interpret each of the below as you see fit; we leave all design decisions to you. Be sure that your HTML is valid.
  - Empower users (who're already logged in) to change their passwords.
  - Empower users who've forgotten their password to reset it (as by having them register with an email address so that you can email them a link via which to do so).
  - Email users "receipts" anytime they buy or sell stocks.
  - Empower users to deposit additional funds.

Sanity Checks

Before you consider this assignment done, best to ask yourself these questions and then go back and improve your code as needed! Do not consider the below an exhaustive list of expectations, though, just some helpful reminders. The checkboxes that have come before these represent the exhaustive list! To be clear, consider the questions below rhetorical. No need to answer them in writing for us, since all of your answers should be "yes!"

- Is the HTML generated by all of your PHP files valid according to http://validator.w3.org/?
- Do your pages detect and handle invalid inputs properly?
- Are you recording users' histories of transactions properly?
- Did you add one (1) additional feature of your own?
- Did you choose appropriate data types for your database tables' fields?
- Are you displaying any dollar amounts to at least two decimal places but no more than four?
- Are you storing stocks' symbols in your table(s) in uppercase?