Hello class!

I hope your week is going well. Like last time, I will go over the homework and then do a recap of all the concepts we went through in class

Homework
To get the homework, go to www.github.com/izzyjohnston/gdi-php-mvc. In the middle, left hand side of the page click on the branches button. Look for the "class2" branch and download the zip file (or check out the branch if you want to use git).

These files are a bit different from the class 1 files because I have actually removed code in the applications/models folder so you can get practice building models. The code that you worked on for the views will still be there, but there are extra comments in the views to let you know where the forms will go.

This week we learned about models, or the data structures used to interact with the data in our database. Your homework will be to create models for posts and bloggers, found in application/models and to create html forms to submit data in the views worked on in class1 (application/views/bloggers/list.php and application/views/posts/list.php).

Models
Each model (post.php and blogger.php) will have 5 small functions.
You can see the examples for the posts model on slides 10-13.

1. getAll() will do a SELECT query to get all the posts or bloggers from the database (depending on which model you are working on).
2. getOne($id) will do a SELECT query to get one post or blogger based on an id
3. create($fields) will do an INSERT query to create a new post or blogger
4. update($fields, $id) will do an UPDATE query to update a post or blogger based on an id
5. destroy($id) will do a DELETE query to delete a post or blogger based on an id

Views
You will add three forms to each of the list.php views that you worked on last week.
You can see the examples for the posts form on slides 14-17.

1. A create new form. It will have input fields for each field in the database EXCEPT date_created and id. Place this above the foreach loop that prints out the list of posts or bloggers, respectively.
2. An edit form. This will have all the fields of the create new form. But for each input field you will add a "value" that is the value of the current post. It will also have a "hidden" input field that contains the value of the id of the post or blog the form belongs to and a "hidden" input field that has the value of the user_id associated with the blogger or post. Place this inside the foreach loop right after the code for showing the data about the blogger or the post.

3. A delete from. This will have only a confirmation button and "hidden" fields of the id of the post or blogger, and the user_id associated with the post or blogger. Place this form inside the foreach loop after the edit form.
4. Buttons. In my completed example, these forms are displayed and hidden by clicking a button. If you want that functionality, add a create new button above the create new form, and an edit and a delete button below those forms inside the foreach loop.

Models 101

In MVC architecture, Models are the data structure that define how we can interact with the data in our database. All models have basic CRUD (Create, Read, Update, Delete) functionality.

Robust models can also add restrictions on different fields. For example, if you know a field in our database is color, and you only want to ever see the colors red, green, and blue, your model can put restrictions on the what data will be sent to the database. The user will get an error if they put a color not approved.

Models always initially reflect the structure of the database. For every table in your database, you will have a model file. This seems like a lot of work, but setting up the models once will save you so much time in the future.

Relational Databases
Our sample application is using a MySQL database. MySQL is the most commonly used relational database. In general, there are two kinds of databases, relational and document. Relational databases include MySQL, PosgreSQL, and even Microsoft Access. Document databases include MongoDB and CouchDB. We are only going to learn about relational databases in this class.

A relational database splits all the data in your application up into tables. Each table generally represents a noun. When dividing up your data into tables, think to yourself "is this data a separate noun"? For example, a library database would have a books table (which would have title, author, ISBN, date, subject, etc.). Now authors are separate nouns from their books, particularly because ONE author can write MANY books. So, we would make a separate table for author. The same thing would happen for subject.

In order to explain how one table is connected to another table, we use "foreign keys". In our library database above, our books table would now have title, author_id, ISBN, date and subject_id. The fields ending in _id will have a number in them. That number relates to the author's id number or the subject's id number.

A little inhumanly, no matter how much personality you have, in databases you will always be reduced down to a number! But this is a good thing. Because if an author changes their last name, or we learn that they were born in a different year than

previously thought, we can change the data in the author table, but our books table will be unaffected, because the author of the book is identified by their id and not data that could be changed around.

Connecting to a database with PHP
Whenever you set up a new website with PHP (or any language), you will have to establish a connection to the database and write code that allows us to talk to it.

Database Constants
In PHP, constants are like variables in that they hold information. They are different than variables because once they hold data, that data won't change. So, instead of creating a variable with a $, we use define.

```
define('DATABASE_NAME', 'gdi_php_mvc');
define('DATABASE_USER', 'root');
define('DATABASE_PASSWORD', 'root'); // '' for PCs
define('DATABASE_HOST', 'localhost');
```

DATABASE_NAME will have the value of whatever you named your database
DATABASE_USER will have the value of the user of your database
DATABASE_PASSWORD will have the value of the password to your database
DATABASE_HOST will have the value of the host (or url).
You will always have to set these up and the values will change if you put your website on the web and buy a domain name. However, your hosting service (Dreamhost, Bluehost, etc.) will tell you what each of these are.

DB Class
In the config/db.php file there is a class DB.

Classes in PHP are just ways of putting similar functions in the same file, so you know where to look for them. For example, our "convertFeetToInches($feet)" function, might go in a Conversions class that has a bunch of different conversion formula functions.

In the DB class, there is one function, called connect().
It first tries to connect to the database with a php function mysql_connect() which needs to know the DATABASE_HOST, DATABASE_USER, and DATABASE password.

$connection_to_database=mysql_connect(DATABASE_HOST, DATABASE_USER, DATABASE_PASSWORD);

If that goes well (it was able to connect), then it tries to find the database through mysql_select_db.

mysql_select_db(DATABASE_NAME, $connection_to_database)

If that goes well, it returns the $connection_to_database. This connection is like a tunnel to the database. Whenever you interact with the database, you first have to check if that tunnel exists and if it does, you can continue on with your interaction. With no connection, it would pointless to try to create, read, update or delete.

Database responses
When interacting with the database, there are two cases.

1. You are trying to read from the database and expect data back (e.g. all the posts in the database)
2. You are creating, updating or deleting something in the database and all you expect back is a thumbs up (everything is ok and your request was completed) or a thumbs down (something, for whatever reason, went wrong).

In config/model.php, you will see examples of both.

1. select($sql)
   This is the read function. When calling this function you expect either nothing back (e.g. there is no blog post with the id 407). Or data about the appropriate rows in your table back. Those rows are then converted into arrays of data that PHP can use.

   The arrays have two levels.
   The first level is an array of each set of data. If you are trying to get all the posts in your database, your first level will be an array of each post.
   $posts[0], $posts[1], $posts[2].

   But posts themselves have data, so each value of the top array is actually an array too! So, if we wanted to know the title and text of the first post in our database, we would look for $posts[0]['title'], $posts[0]['text']. You will always know what the word (title, text, etc.) will be because it will be exactly what you have called the corresponding field in your database.

   Another example… if you want the username of the 4th blogger in your database? $blog[3]['username']
   What about…if you want the date_created of the 5th post?
   $post[4]['date_created']

   Well, this is all great, but you just learned about arrays a week ago. Would I really make you format these complicated arrays from a database? What kind of cruel teacher do you think I am?

   I'm not. PHP has functions that do all the work for you. They aren't pretty, but they do their job.
   First we check if our "database tunnel" is available
   if(DB::connect())

Then, we create an empty array for our results to go to
$rows= array();

Then, we send our SQL statement through the tunnel to the database.
$results_from_database = mysql_query($sql, DB::connect());

If all goes well, and no errors came back from the database, we loop through each of the rows of results and use PHP to turn those into those structured arrays described above.

```
while ( $row_from_database = mysql_fetch_assoc($results_from_database) ) {
        $rows[] = $row_from_database;
      }
   }
```

It's a scary looking while loop, but mysql_fetch_assoc is the function that turns results from databases into usable data. Then we put each piece of data into a usable array.

If there were results, we return our structured array.
If not we return false.

2. insert($sql), update($sql), delete($sql)
   As mentioned above, all other interactions with the database will either return true (everything completed properly) or false (there was an error). So the insert, update, and delete functions in the Model class all look very similar.

   First we check if our "database tunnel" is available
   if(DB::connect())

   Then, we send our SQL statement through the tunnel to the database.
   $results_from_database = mysql_query($sql, DB::connect());

   If there is an error, tell the user what it is:

```
if (!$results_from_database){
     return 'Error adding submitted data: ' . mysql_error();
   }
```

   mysql_error() is a php function that prints out why something went wrong when trying to interact with a database

Cleaning Data

SQL queries often look like very Yoda-like English. This means that it's possible for someone to try to put in data into a database that could take over the statement and do damage to your database.
Fortunately, there is a way of preventing that "SQL injection".
The last function in the Model class is cleanData($fields).
The $fields are fields of data from the user, these would be made when a user is creating a new blog post. The fields would be 'title' and 'text'.

We loop through each $field and clean up the data through a PHP function.
$field = mysql_real_escape_string($field, DB::connect());
This cleans up the field so any data sent to the database won't be interpreted as a command by the database.

Defining a model
As mentioned in the homework section above, most models will be very robust to allow a user to interact with their database in very complex ways, such as finding all the blog posts written be a certain user on a given week. For our model, we will have the 5 functions described above.
The SQL query is basic English, everything in ALL CAPS is a SQL word. The lower case words are specific to the table you are looking for.

getAll()
This is a READ call.
You build a query
$sql = 'SELECT * FROM table_name_here ORDER BY field_name_here DESC';
You send it through the Model class to the database
$results = Model::select($sql);
And return your results.

getAll($id)
This is a READ call, where you only get one result from the database, the post or blog with the appropriate id.
You build a query
$sql = 'SELECT * FROM table_name_here WHERE id = '. $id . ' LIMIT 1';
You send it through the Model class to the database
$results = Model::select($sql);
And return your results.

create($fields)
This is a CREATE call, where you create a new post or blog from $fields sent by the user.
First, create a "timestamp" of what date and time it is right now.
$date = date ("Y-m-d H:i:s");
Then you clean the data to prevent bad things from happening to the database.
$fields = Model::cleanData($fields);
Then you create an UPDATE query for all the field names (except id, because that will be automatically created by the database).
'INSERT INTO table_name_here (field1, field2, field3)
VALUES ("' . $fields['field1'] . '", "' . $fields['field2'] . '", ' . $fields['field3']. '")';
You send it through the Model class to the database
$results = Model::insert($sql);
And return your results.

edit($fields, $id)

This is an UPDATE call, where you update a blogger or post from $fields sent by the user by an $id.
You clean the data to prevent bad things from happening to the database.
$fields = Model::cleanData($fields);
Then you create an UPDATE query for all the field names.
$sql = 'UPDATE table_name_here SET field1 = "' . $fields['field1'] . '", field2 = "' . $fields['field2'] . '" WHERE id = ' . $id;
You send it through the Model class to the database
$results = Model::update($sql);
And return your results.

destroy($id)
This is an DELETE call, where you delete a blogger or post by an $id.
You create an DELETE query.
$sql = 'DELETE FROM table_name_here WHERE id = ' . $id;
You send it through the Model class to the database
$results = Model::delete($sql);
And return your results.

Forms
Although we will go into forms in more depth in class3, some basics are:
In HTML a form begins with <form> and ends with </form>

All forms have an action (where you send the data) and a method (get or post).
<form action="some url" method="post"></form>

The name in elements is necessary so the controller knows which data sent belongs to which field. Value is optional depending on whether you want data pre-filled in

Some form elements:
Small text box.
<input type="text" placeholder="Enter text here" name ="field1" value=""/>
Big text box
<textarea placeholder="Enter text" name = "field2">Value</textarea>
Hidden field
<input type = "hidden" value ="true" name="name"/>
Button
<button type = "submit" value="Submit">Submit</button>