Phase One

I first created the actual database in MySQL, and using SQL to build tables and fields. These included tables for question, answer, and competition submission, and include a join table named question-answer that linked potential answers to questions and tracked whether a particular answer was the correct answer to the related question.

My first plan of attack was to download a random question with its related three answers from the database, because this process was able to follow the MVC architecture by directly interacting between the view and the model without any need for the go-between of a new controller. It would provide confirmation that the database connection with the model was working correctly, and doing this would consequentially also provide all of the data needed for the submission process that would be needed later.

I created a CompModel class and created an object for this at the top of the NavController. I created a new CompDB class to represent the database connection where the SQL would all exist. I copied the DB file that Todd had used in an example he had provided in class that directly interacts with the database through such processes as querying, fetching prepared rows from a query, and freeing up the query last made. The first steps used in confirming these worked was temporarily trying to create objects beneath the code for the competition model class and use them with echo statements.

Phase Two

Once the connection was working I inserted three questions into the question table, a number of answers into the answer table, and in the question\_answer table I inserted records to marry up the appropriate questions with the required answers and tagging each as correct or incorrect.

Having created a competition view that used a form and appropriate input types to obtain an entry details to the competition during class earlier that week, the file was essentially ready to have a question and its related answers inserted into it beside the radio buttons. With a CompModel object included in the navigation controller using “require\_once” to include that file, what was needed now was a function to select a random question of the three available and the three related answers. Inside the CompModel I created a function that did that by calling another function inside the CompDB where the SQL is expected to be kept. Inside that function I used the random number generator mt\_rand() (which I found at W3 Schools) to assign to a variable used in the SQL code to identify the id number of the random question desired and its three answers. After running the query, the CompDB function would then use the DB getNext function to retrieve the single row of required data to the CompModel.

Inside the CompModel class I chose to include a list of variables that would keep track of the question and answers being asked, and inside the function that requested the question and answer details these variables would be filled with the appropriate contents after a successful array of strings was returned from the CompDB. I included another function inside the CompModel that used a parameter to identify information needed (question text, answer1 text, etc) to return that variable contents. That function was then called within the CompView to retrieve the text needed while displaying the question and three answers.

Later in class while helping me, Todd suggested to use inheritance to include the DB class into the CompDB class code rather than having to use an extra object and functions between them, since the CompDB basically is a database connection anyway. I also used W3 Schools to discover “expands” is the code word used for class inheritance in place of the colon used in the .Net situations.

Phase Three

After getting the data displaying on CompView I went into the stage of trying to get the entered data submitted into the database, which would take more interaction with a controller now being needed to link between the CompView and the CompModel. I created a CompController file to provide a class and two functions to do this: the object constructor, and a submit function that was called upon a isset() test initiated and waiting for the click of a submit button on the CompView.

At some stage while putting the submit function together to suck up the input fields needed from the view I encountered problems getting a $\_REQUEST to perform the “sucking” of the data, so I used $\_POST for each variable needed instead. I first managed to get the controller to pass the variables on to the CompModel, where I stored each in a local variable under the impression that I could later use these to return the competition data on to the follow-up thanking view. The model then called a function of the same name in the CompDB class, acting in a way that made it seem like the controller, model, and database were passing a rugby ball. The CompDB then called the inherited query function to store the details to the database.

Upon getting this to work with legitimate shaped variable contents, I looked into creating the follow up CompThankView and getting it to access the variables that I believed were waiting for it in the CompModel object. Here rose a hiccup to my plan, and this wouldn’t work with errors claiming that the CompThankView knew nothing of the CompModel or any of its previously stored contents.

Phase Four

After talking to Todd about this issue, and he explained that it was due to the fact that the CompModel object which stored the variables of the submitted competition entry was created on the CompView form. Therefore, when that page was closed by requesting CompThankView then the CompModel object was out of context so consequentially forgotten. He suggested that the solution to this problem was to create and use a super global variable to track the ID of the record submitted to the database when it was inserted. I did this inside the CompDB using $\_SESSION[‘lastCompID’]. Therefore, I could call another function from the CompThankView that would use this super global in a basic SQL search for the information needed to display on the thanking page. This process was successful, but my use of “require\_once” of the thanking view instead of accessing it in the standard way through the navigation controller led to errors trying to access the login model – similar to the message I was receiving with accessing my CompModel object. I was unable to work it out at the time, and logins are a critical part of this at this stage, so I simply avoided the navigation and login model usage and displayed the information on a page without those parts and including a link at the bottom of the thanking text. This would initiate a call to the navigation controller and set things back to normal, albeit in a less than uniform fashion.

Phase Five

The next thing I looked into was trying to take care of the SQL escaping. I put together a function called fixString in the CompModel that could be called to tend to variables that the entrant had entered. It included some code that I found (again through W3 Schools) that would change any change any undesirable characters (htmlspecialchars), trim off any spaces (trim), and remove any HTML tags included (strip\_tags). I made the mistake of not throwing many undesirable input at this simply assuming that it would work. A couple of days before submission I included a <div>, and the outcome was errors when trying to display the altered submitted code. I therefore decided to alternatively check any altered strings against what they looked like before “fixing” them, and to request the user to try again if any undesirable content was found in their entered text. I tried to reload the CompView page instead of submitting to the database, but this seemingly gave my website a heart attack in that I couldn’t get it to even reload index.php normally again. I spent a few hours on this, with even Stewart unclear about what was going wrong when I asked him for any ideas. I finally worked it out a couple of hours later and discovered that it was because the super global that I had used to track the current view in action was hanging around in memory long after the website was closed, after Xampp was disconnected, and after the laptop was turned off. I therefore included an isset test for the [‘view’] super global in the first part of the index.php and if it was set then it would become unset and ready for standard use.

The last problem I encountered after this was the reloading of the CompView page when illegitimate text content was found. Similar to the problem encountered with the LoginModel object when loading the CompThankView, errors would appear claiming the object didn’t exist, but the alternative approach used then was not feasible at this point. My investigation discovered that this was due to the LoginModel object being created by the NavController, and the only place that this was called from was way back in the index.php. Again based on the understanding that the login is not yet required for this milestone, I put a quick-fix in by adding an isset($LoginModel) to the if test inside the LoginView where the error was occurring. This diverts the issue for now, but will need to be addressed when users and logging in becomes a key part of the process in milestone three.

This semi-solution was also able to be used to make the competition thanking screen have the navigation bar and login parts fit in without errors, and didn’t need a link to activate the navigation