Alex Lin, Helen Wu, Joy Jin
12/6/15

Design for Budgitt

We designed our project by using bits and pieces of HTML, CSS, PHP, and mySQL code—all in tandem with one another. We followed the MVC (Model/View/Controller) structure. All of our data for the users was stored in a mySQL database, which could be manipulated by PHP code that we wrote on the server. The PHP acted as a transferring agent between the user and the data; it extracted the information in the databases and displayed it on the website through the rendering of HTML pages. Similarly, we also used PHP to take user input from the individual pages and update the mySQL tables.

Each page that can be accessed via the header of our website is rendered using a PHP controller. This allows our pages to be versatile and respond differently based on whether the user requests a "GET" method or a "POST" method. For instance, on our home page, index.php, the "GET" method serves two functions: 1) to display the user's portfolio (i.e. relevant pieces of information from the mySQL table) and 2) to send them an email alert if they are overspending. However, if the user decides to submit a small form at the top of index.php that allows them to start a new spending period, then our program executes completely different code. Instead, we update our mySQL tables based on the user's input and then redirect them back to index.php via the "GET" method so that they can view their changes. Thus, this design decision to render all of our pages through PHP allows us to easily respond with appropriate responses to user requests.

The pages "Add Item", "Purchases", and "History" all operate via the "GET" method only. For each of these, we have a PHP file to take information from mySQL and render another PHP file (called "form") that will create the actual HTML that is to be displayed. "Settings" operates like the home page, as it has separate implementation for "GET" methods and "POST" methods.

In fact, one of the major design decisions concerned the settings page. We had many options we wanted to give users of our website, and we knew that it would probably be best to group these options in one location where everything can be easily accessible for their convenience. However, each of these options was effectively a "mini-form" that needed to be submitted independent of the other mini-forms. For instance, when a user wants to "add a new category", we needed to make sure that our program completely ignores fields relating to "edit category ideal spending" and "change alert options". The issue was that we could not submit each mini-form individually; the entire page was effectively submitted via a POST request once any "submit" button is pressed. To solve this problem, we added a dropdown menu that first prompted them for what action they wished to complete on the settings page. Then, based on the user's selection for this drop down menu, we used switch statements in the PHP code to respond to the appropriate request, thereby ignoring the other ones.

Another design decision had to do with how we wanted to record spending history. We thought that it would be too cumbersome and possibly confusing (as well as infeasible) for users to view every single item they've ever purchased. However, we did acknowledge that it would be nice for users to be able to view at least some of the items they bought, especially the more recent ones that are probably more relevant. Thus, we decided to split "spending history" into two pages—one that would allow users to view all the items they purchased in the *current* spending period, and a second that would display the data of the previous spending periods through pie charts. In this way, users could still view their long spending histories in a condensed, more aesthetically pleasing fashion. They would also not have to forgo any significant specificity, because they could still view individual item information for their current spending period.

The pie charts themselves are coded in PHP and powered by a library called Libchart. The implementation structure is imported from online, but we completely generated the code ourselves. We also had the option of creating pie charts by using HTML in tandem with PHP *and* Javascript, but we realized that the code would have become unnecessarily complex, involving the transferring of variables from PHP to Javascript and vice versa. We find the current design of Libchart to be much cleaner in comparison. It is for this same reason that we chose to use PHP Mailer, another free library from the Internet, to run our email alert service. The design of these internet libraries make them easy to integrate into the specific MVC model that we are employing—a model that is already dominated by PHP code.