Inebriation Proclamation User-Interface and Server Documentation

Tyler Fox, Nathan Lapinski, Andrew Miller, Varun Varada

CSCI 3308 — Fall 2012

Contents

UI Design and Motivations

The *Inebriation Proclamation* website is a tool for users to research, classify, catalogue, and rate various cocktails and beverages. As such a website will necessarily realize an abundant knowledge base of user submitted data, it was decided early in the development stage that we would be implementing a database driven website. Some options we considered were:

- Django Framework
- Rails Framework
- PHP/MySQL

We wound up implementing the latter. In part because a group member had existing familiarity with the PHP/MySQL, and because every member possessed a working grasp of the rudiments of database-driven content using PHP/MySQL after the in-class labs (Week 7 and Week 8). This was viewed as being beneficial in that it greatly enhanced the amount of testing we could do, since every group member had some knowledge of the technology and was therefore able to practice regression testing in a more in-depth and efficient manner.

The front-end UI is written primarily in HTML5, CSS3, and JavaScript. When necessary, have used delegation to third party APIs and frameworks, such a jQuery and Google Maps, in order to provide functionality as well as code reuse.

The motivations for the overall visual layout of the page are as shocking as they are educating. The color scheme was originally a soothing shade of red vino, in order to instill a theme of placid inebriation. This was later replaced with a more modern image of a beverage. The main page layout is replicated on every page. This decision was made in order to provide the user with a consistent browsing experience from page to page, and it in some ways echoes the tenants of the Master-Detail UI design pattern.

Site Navigation and Structure

The website consists of 6 HTML pages, each with its own auxiliary CSS and JavaScript files. Briefly, the pages are:

- **Login.html:** This is simply the landing page of the website. From here, a user is greeted with the standard menu that appears on the left hand side of all pages, as well as with a search bar that they can use to search for arbitrary keyword that occurs with a drink recipe. The system will then return all such entries containing that keyword.
- **SearchResults.html:** This page is returned when a user executes a search query on the Login.html page. It will simply display the results of the keyword search. When a user clicks on one of these results, they will be taken to the page corresponding to that drink.
- **DrinkPage.html:** This page is dynamically populated by the database depending on what drink the user has selected. It will provide information on the drink recipe, such as which ingredients go into the drink, how the drink is to be prepared, etc. There is also a system for the user to rate the drink on a scale of 1–5. Lastly, this page also offers

the user the ability to associate a particular establishment with a drink, or to see which establishments other users have linked this drink to. This feature, along with the rating system, allows a user to get the best drink at the best location. Currently, locations are all located in Boulder, but the technology (Google Maps API) supports tagging locations anywhere in the world.

- **MapPage.html:** This is simply an implementation of the Google Maps API running through a jQuery widget. It shows users a list of bars in Boulder that have been tagged to the currently selected drink. Note that this map is fully functional, so that a user may enter street view, search for other locations, etc.
- **MapPageTag.html:** Similarly, this page allows a user to tag a bar to a particular drink. This is accomplished by having the user type the name of the bar in a search box. A JQuery autocomplete widget is activated to aid the user in locating a specific establishment.
- **AddDrinkPage.html:** This page allows the user to insert a new drink recipe into the system.

Site Implementation and Technologies

As previously noted, the site is implemented primarily in two sections. The database backend, which is driven by PHP and MySQL, and the front-end UI, which is implemented in HTML, CSS, and JavaScript. In addition to these, we have also used a variety of additional technologies and tools on the UI front end to enhance the user experience. The jQuery library is used on a variety of pages to provide certain functionality, such as an autocomplete feature on the Map Tag page. jQuery is also used to run the Google Maps API.

Testing and Tools

In keeping within the spirit of this course, a huge part of our project was learning how to use new tools, and performing rigorous unit testing of the system. From a UI standpoint, a variety of new tools that we were mostly unfamiliar with were used to aid in development and testing, such as:

- **jQuery:** Used to provide slick UI functionality. Allowing features like text box auto completion.
- **Google Maps API:** This is provided by Google, and is used via a jQuery plugin
- **Firebug:** Firebug is an add-on for the Mozilla Firefox web browser. It proved invaluable in debugging and unit testing the HTML, CSS, and JavaScript. In addition to allowing one to examine arbitrary elements of an HTML page, as well as fully investigating the DOM, Firebug contains a JavaScript debugger, which enabled us to find and debug JavaScript errors in a more efficient fashion than would have been possible without the aid of this excellent tool.
- **NoScript:** This add-on also proved quite useful in debugging HTML pages containing JavaScript. Specifically, we wanted to isolate and test the use case where a user has JavaScript turned off in their browser. Since our pages use JavaScript rather

extensively, we felt it was necessary to make sure that rudimentary fallback functionality was in place in the event that JavaScript was not executable in the users browser.

- **W3C HTML and CSS Validator:** These tools, found on the W3C website, were used to assert that all of our HTML and CSS was standards compliant.
- **Browsershots.org:** This website greatly increased the clip at which we were able to test for cross browser compatibility of our HTML pages. By simply uploading an HTML file, this utility will emulate it in a variety of browsers and provide screenshots of what the page will look like on those browsers. This enabled us to catch and fix and cross-browser compatibility issues.
- **GitHub:** No software project would ever function properly without adequate use of a version control system. While we initially considered the idea of using a subversion repository, we decided that GitHub would be a better version control option for our project.

For unit testing, we initially did a fair amount of searching for a unit testing systems like JUnit that could automate HTML and JavaScript tests. As we were unable to find one that adequately suited our purposes, many of the UI tests are simply tests we ran by executing use cases and observing the resulting system behavior. The following unit tests were derived from the requirements presented in our iteration proposal.

Iteration 1

"*Run through use cases regarding the UI*": For this requirement, we mostly discussed what we wanted the overall layout of the pages and system to look like, and conjectured upon ways in which users would navigate the system. This enabled us to create a set of very basic UI mockups that eventually became the templates for the actual HTML. While it was mostly "theoretical regression testing" at this point, this particular phase of the design was of utmost importance in achieving consistency in the final site design.

Iteration 2

"*Test integration of backend system with UI*": This had a variety of sub tests that needed to be performed, as is outlined below:

"*Test User Search for Drink keyword*": The test here involved typing a keyword ingredient into the search box on the login.html page and checking the resulting behavior. When a keyword that is found in the database was typed in, it produced a list of drinks containing that ingredient. From there, clicking on one of the drinks returned that drinks page. This was the desired behavior. In order to more thoroughly test, we also tried the scenario where a user types an ingredient that does not exist in the database. In this case, the behavior that occurred was that the user was given an error message explaining that no drinks in the database use that ingredient. This was the expected behavior for this use case as well.

Iteration 3

"*User attempts to access a drink result from the drink page*": In this case, a user will have just queried the database for a specific keyword. When they click on one of the returned results, they should be taken to the page displaying that drink. While this worked in all tests, we managed to catch a bug in this phase of unit testing. It turned out that the HTML div element that held the list of ingredients was too small, so that in certain scenarios (we tested for several drinks), the results would actually overflow the div. After catching this error during testing, we made the necessary adjustments to the CSS, and fixed the problem. All other unit/regression testing of this part of the system passed.

"*User attempts to tag a bar with a specific drink*": This scenario occurs on the drink page when a user clicks on the "add location" button. This will take them to a page containing an embedded instance of the Google Maps API (MapTag.html) . From there, they may begin entering the name of a bar into a text box, which has a jQuery autocomplete feature. For testing, we simply entered the name of a bar that existed in the system, and then hit tag. The functionality we observed after running several tests was that the embedded map would be updated correctly with a new map marker on the location of the bar. In addition, the database necessarily tracked this update, so on subsequent visits to the same drink page, the bar would remain tagged with that particular drink. This was the desired behavior, so we counted this as passing the unit test.

"*User attempts to view tagged locations for a specific drink"*: In this case, the user wants to view which locations other users have tagged for a specific drink. To test this, we simply made sure the Google Map was reading its location markers from the database correctly. After tagging several locations, we then reloaded the system and checked to see if the map had updated correctly. Since it had, we counted this unit test as being passed.

"*User attempts to rate a drink"*: This scenario also occurs in the drinkpage.html. In this case, a user has a small image of 5 stars, from which they can rate the drink. To test this, we tried rating the drink as the user would several times, and then checked back to assert that the database was storing these results properly. We also had to check to make sure that the ratings were being averaged properly. That is, if a drink received a rating of 5 stars and then one of 3 stars, the page should display a rating of 4 stars for that particular drink. After several unit tests, the system exhibited this behavior, so we counted this unit test suite as being passed.

"*Implement an intelligent search heuristic*"*:*** For this requirement, we used a jQuery widget that enables intelligent auto completion when a user is searching for a bar to tag. This feature will begin populating the text field with suggestions of locations as soon as the user begins typing, based on what inputs they have provided so far. Unit testing of this feature consisted of providing inputs on the frontend, and asserting that the database was populating the autocomplete correctly (e.g., only bars that have some substring match with what the user has typed so far). All of these tests passed, so we counted this unit test as a success.

This concludes the majority of the unit/regression tests we executed on the frontend of the system. Through these we were able to catch many bugs and usability issues, and were able to ensure that the UI was interfacing with the server correctly. The unit tests above were described at a slightly higher level, as we had no real way to automate them. However, the testing was

quite rigorous, and each test was performed several times in make sure that it really was passing consistently.

Server Documentation

Field Research

In order to acquire the data we used to populate the database, field testing was required. Only a handful of the bars we used in the initial database posted their cocktail menus online. Therefore, we needed to visit the locations in person, inquiring about the various drinks offered at each and choosing the best. It should be noted that only group members of the legal drinking age partook in this research.

PHP Unit Testing

Unit testing for the backend code aspect of the project was done manually in PHP. The main aspects of the code that needed to be tested were adding drinks, searching for drinks, rating drinks, finding drinks at bars, and tagging bars to drinks.

Adding Drinks

This part of the code was tested by first implementing checks to see if all the required information fields were filled in. Then, each of those fields were sanitization checked to see if they contain the correct type of input (e.g., the quantity of an ingredient should be a number). This sanitized information was then inserted into the database into the appropriate table. Then, the database was checked (using phpMyAdmin) to see if the new drink had indeed been added to the database without any problems.

This process was repeated several times with different types of input to ensure the code's robustness. One of those times, the code was not adding the drinks to the database because it was not allowing the ampersand (&) as part of the name of a drink. As such, the drink name sanitization check had to be relaxed to allow for any type of characters (we figured that a lot of drinks could have special characters, accented characters, or even numbers) so that we could make our application as flexible as possible in terms of names. We fixed further issues with the process in a similar fashion.

Searching for Drinks

As with any code that interacts with a database, the code had to sanitize input as part of the first step. Then, database was queried based on the sample keywords we input to see if it would return the correct result set for the query that we executed.

During this process, we encountered a few roadblocks in trying to get query working. One of them was trying to get the query to scour through not only recipe names, but ingredient names and instructions for recipes as well. This is when we had to do extensive research into SQL to

see how to execute table joins in order to take advantage of the relational database capabilities of MySQL. Another challenge was to address the problem of multiple keywords being inputted. This was where we had to devise a way to allow for “OR logic” between the keywords such that entries matching any one of the keywords would be included in the result set. Such issues, and some other minor ones, were the ones that we were able to catch and account for through unit testing.

Rating Drinks

This part of the unit testing was relatively straightforward, as the input was a rating (an integer between 1 and 5). We basically had to implement some boundary checks to make sure that ratings were not below 1 and above 5. Then, we just had to commit the results to the drink entry in the database.

This process went rather smoothly, as there were not as many moving parts to get this code to work.

Finding Drinks at Bars

The unit testing for this part was very similar to that of searching drinks, except on a smaller scale. Seeing as the code between searching for drinks and finding drinks at bars was very similar (it just involved database querying), we reused the code from that function for the purposes of this part. The only major testing was making sure that the code from the other function was fully adapted for this purpose.

This process went relatively smoothly as well, apart from the few instances where we had typos in code, or had missed changing one of the variables from the search code.

Tagging Bars to Drinks

This process involved a multiple steps. First, we had to make sure that the specified drink existed in the database in the first place. Next, we had to make sure that the specified bar to tag existed in the database as well. Then, we had to make sure that the to-be-tagged bar has not already been tagged to the drink in question. Finally, the bar could be tagged to the specified drink.

The third step in this process was where we had some hiccups, as we were trying to figure out how to search for already tagged bars seeing as the IDs of tagged bars are stored in CSV format in one field of each entry in the database table. When we originally tried this step, it worked initially but then mysteriously stopped returning the correct results. We found that we did not account for the fact that the IDs of tagged bars could exceed one digit (as the database was populated with more and more bars). As such, we had to change our query algorithm to account for this fact, and the system was returning the correct result sets thereafter.

Database Unit Testing

The unit testing we performed on the database was less code based than most conventional testing. We had five main pillars to test in the user-accessible portion of the database: adding drinks, searching for drinks, searching for drinks offered at bars, tagging a drink at a bar, and preventing SQL injections by sanitizing user input via stored procedures. To test the addition of a drink to the database, we simply added cocktails with various levels of completeness for the different fields (zero, one, or more directions entered, no name entered, various amounts of instructions, etc). Searching the database was similarly tested by entering multiple types of keywords and ensuring the correct results were returned even when the keywords consisted of ingredients, instructions, or recipe names. Tagging was also tested by performing the action as many ways as possible. Edge cases were included in the tests, such as when no input is given, or when a non-existent bar or drink was specified. Proper sanitation methods were asserted by entering problem characters and strings and noting the results. Other database-related testing was covered as well, such as removing cocktails, editing fields, and adding bars, though these actions are only available to the database administrator.

Database Tools

Most of our team had minimal experience in databases coming into this project. Therefore, we had to put quite a bit of research into which tools would best meet our needs. Using this research combined with the experience of our database-savvy member, we settled upon a LAMP stack similar to that used in the class lab. To add a level of abstraction, we used the phpMyAdmin tool to interact with our database, which had the added bonus of making the testing of our database's basic functionality much simpler and much more efficient.

Agile

For this assignment, we decided to pursue the agile software development framework known as Scrum. Andrew was assigned the title of Scrum Master. He determined that we should hold scrum meeting every other day to discuss the progress of the current sprint, since meeting every day wasn't realistic due to schedule conflicts. We had a total of three sprints, as defined by the project requirements. After completing an iteration, we met and reviewed the last sprint, as well as planned the next one and discussed to make it better than the last. Any unfinished components were rolled into the following sprint, though there weren't many of those. During all sprints, we used paired programming to improve our coding efficiency, as well as the quality of code, as is demanded by Scrum.