Classroom Connect

Test Plan Documentation

Kyle Wright, Anthony Chan, Jacob Wyner, Joseph Cao

1. Introduction

Classroom Connect is a product designed to help facilitate communication between students and professors in a lecture setting. Often times, many students are confused by the same topics during lecture. Classroom Connect software aims to help students communicate their confusion to the professor anonymously. Classroom Connect software also provides a platform for a professor to poll students about specific questions and receive results in real time, assisting both parties in the overall goal of creating a productive learning environment.

2. Quality Control

2.1 Test Plan Quality

In order to test our backend connections and insertions, deletions, and reading of the database, we used the PHPUnit testing framework along with the DBUnit extension. We ran these unit tests locally before each commit, however, Travis CI automation was not fully implemented on our main Github repository until late into our development process. This is because using Travis required many inconvenient changes to the flow of how we synced our deployment server with Github commits. A major reason for this is that early on in the development process we neglected to use namespaces sufficiently throughout our code and there were very many instances of specific hard coded file paths that would be incorrectly referenced on Travis. Another problem we ran into with Travis is because of how we utilized the DBUnit extension. We neglected to take into account different environments and we created a test database that did not exactly reflect the eventual deployment database structure (in addition another problem is that the deployment DB was initially hosted separate from the main server on a Google SQL Instance) in order for our tests to reference the correct DB structure we found it easiest to test on the localhost with with a locally created DB with the same tables and general structure as our deployment DB. Our early test scripts to check insertions into the database essentially wrote in rows into our actual database and then removed them after it was done checking assertions. This was troublesome to recreate on Travis because our code had hardcoded references and

authentication to a remote SQL server that the DB was hosted on (in order to access the DB, the external IP would have to be added every time which is impossible with Travis). We eventually read through Travis's DB testing documentation and changed some of our code to be more general in its reference to the database. This allowed us to run the scripts after adding into the .travis.yml configuration file to recreate our an unpopulated version of our DB on the test VM instance.

2.2 Adequacy Criterion

As a result of the simple nature of our class functions, a test suite consisting of insertion, deletion, and reading of the database rows and columns and then testing the consistency of the database structure after these operations is adequate coverage of all functionality of our program if every test in the suite passes.

3. Test Strategy

All of our test scripts are unit tests that isolate some functionality of how the software should function and check its reliability separately. Our approach for integration testing was to manually test the combined functionality of the separate components by using the software and noting any unexpected behavior. We also tried to analyze the logical structure of our code and ensured that the structure and organization was reasonable. If a problem still arose we would create new tests that check operational artifacts at certain expected points. For example, if the poll function was not incrementing correctly, we would first check the value at the point of the AJAX request by using the browser, then we would check the value on the server side using a debugger, and then finally the query the database to ensure that it isn't being incorrectly inserted. Our main methodology for integration was implementation while integrating at the same time. We also utilized some non-execution based testing such as discussing code changes and pull requests before commits.

4. Test Cases

Note that since our remote database connection needs authentication, there will be segments of code that fail because of authentication requirements.

Testcase	Steps	Expected Result	Actual Result	Additional Notes
SQL Get	1. Query Database at specific table 2. Compare Query to expected	We created a "flat xml" dump of a correct	The actual result is obtained by running the code, and then querying	This testing is done in the PHPUnit framework with the DBUnit extension.

	table	database structure and expected values after a certain query. This is the “default.xml” file.	the live database and dumping the results of this query into a “seed.xml”	
SQL Table State	1. Insert a value into database 2. Check that only the correct table is updated and all other tables are unaffected	After the insertion code is run, a dump of the database should only show that the affected table is updated and no other tables are changed	The actual result is obtained by running the PHP code and then dumping the database into an xml file, this is then compared against the expected file.	Testing is done on an empty database with the same schema as the deployment database. Code is run to populate this empty database and allows us to create an expected version of the database dump.
SQL multiple users with same email	1. Insert two new users with the same email address 2. Check if the USER table in the CLASSROOM CONNECT database only shows the latest entry with the same email.	A query to the database for the test user’s email should return a table with one row containing the latest information for that user.	The actual result is stored in an xml dump named “seed.xml” after our PHP code is run. The dump is compared against the expected table, given in xml format.	