Final Project Report

Berkeley Willis

Database Project: Final Report, Professor: Dr. Neel
COMP 3115 Section 002, Date: 4/18/2016

Abstract. The fortune function on Unix devices have several thousand fortunes, but it is difficult to sift through all of the files to find any particular fortune. The Fortune's Database takes all of these fortunes files and puts them into one accessible database, allowing users to find any fortune they are looking for.

1 Introduction

The Fortunes database takes many files from the “fortune” program and simplifies searching for fortunes that pertain to a certain subject, a quote from a specific person or possibly to find a humorous one. To create this database the fortunes are parsed and inserted into the created database, and the tables are created along with the relations among them. Additionally, a website with an application would make the database more secure as well as displaying the information that the user is looking for in a much more readable format. Therefore, this project creates a simple user experience when looking through fortunes from the famous program.

2 Related work

This project was completed on my personal machine running on Ubuntu 14.04.1 LTS. The MySQL database version being used 5.5.47 from Oracle. The parsing and inserting scripts were written in Python 2.7.6. A virtual environment was created to run Python for the web-framework called by the virtualenv commands uses version 1.11.4, and is set to run a version of Python 3. The web-framework running in this virtual environment is Flask which is a module written in Python 3.4.3, and the version of Flask is 0.10.1. The remainder of the scripts that were written for the web-application and all the plug ins were all written in Python 3.4.3. The website was written in a combination of HTML 5, Bootstrap CSS, and some in-line CSS.

3 Design

The design of the parsing script takes fortune files that were found and generates three text files, one for each base table of the MySQL design. The insertion script reads these three text files and inserts into the MySQL database. Instead of having a single Python script to read all of the files parse them and insert them all at once, they have been split into two different scripts. This has been done because there is absence of uniform formatting between all the fortunes, having fortunes “types” mixed together in each file, and the need to verifying of correct parsing through the text files. The design of the MySQL logical schema is based off of three base tables where the bulk of information, multiple mapping tables containing the IDs or more specific entries, and views that do a join between a base table and a mapping table showing only the information that is meaningful to the query. The web-framework was a combination of Python modules using scripts to access the Administrator account of MySQL on my machine, handle queries, and sanitation of user input in a separate module. The SQL script written to create the key mapping tables were all written by going into the database with only the base tables and finding some repeated subject matter. Each mapping table that is added to more than once also has queries that check to make sure that there are no duplicate entries in the tables.

3.1 Design of the Parsing Python Script

This script creates the resulting three different files, searches for the fortune files using exact paths on my computer, and then reads the file line by line. While reading the fortune files line by line, it uses specific regular expressions to determine what “type” of fortune it is, reads all the lines in the fortune that are left before the start of the next fortune, and then writes it in another format that will be uniform throughout that specific file. The output files that this script creates were then found to have some errors which is why another version for the parser was created.

Parser script version 2. The first parser script worked for exactly what it was supposed to do, but unfortunately even with a regular expressions that identifies numerous formats of quotes, and questions and answers it could not catch them all. For example the only way really to identify a quote is by space followed by two – symbols, but there were several that only had no spaces before the two – symbols. Even some with only a single – symbol, so the first version didn't pick these up and they ended up in the Fortunes table. To fix this it was necessary to take the Fortunes.txt, receiving file from the first parser, and re-developed slightly more accurate regular expressions when it came to the number of spaces, but then hand change many of the Quotes that did not picked up which had

extra symbols instead of just letters. The reason why a script to automate this process couldn't be developed was because of the nature of many of the other fortunes. Many contain --, or -, with no spaces at the beginning of a line, so it first had to be identified by a person to be a quote, and then give it the necessary space before the – symbols so it would be caught in the pattern matching.

3.2 Design of the MySQL Fortunes Database

The MySQL database was determined after direction was given by Dr. Neel on a possible structure. There are three tables that are the three different types of fortunes that all fortunes can be separated into. The Questions_And_Answers table holds all of the fortunes that are in a question answer format, and have five different attributes. These attributes are a QA_ID for identification of each unique fortune, Questions containing all of the questions that appear, Answers containing all of the answer to the questions, Question_And_Answer which contains the entire fortune, and subject which one may or may not have but can be used for identification of many of a certain type such as “Fortune's Real-Life Courtroom Quote”. There is also a Quotes table containing all quotes that have three attributes. Quotes have a Q_ID for identification of unique quotes, the Quotee for showing where the quote came from, and Quote containing the actual quote. There are several key-mapping tables, their names being derived from their subject and the base table that it it came from is attached as a postfix. Each one of these key tables has a single entry which is the ID number from each base table. A view is then created as a result from a query from these base tables and the key mapping tables.

Example of Fortunes Key-Mapping Table. The MarkTwain_Q table is a table of the Q_IDs from Quotes that are made by Mark Twain. It has a single attribute called Q_ID, which is a Foreign Key with a reference to Q_ID in the Quotes table. This way I can even make a view to output all Mark Twain Quotes quicker by doing an inner join on the two tables, and it will be quicker than executing a query to find all Quotes with the Quotee “like” Mark Twain.

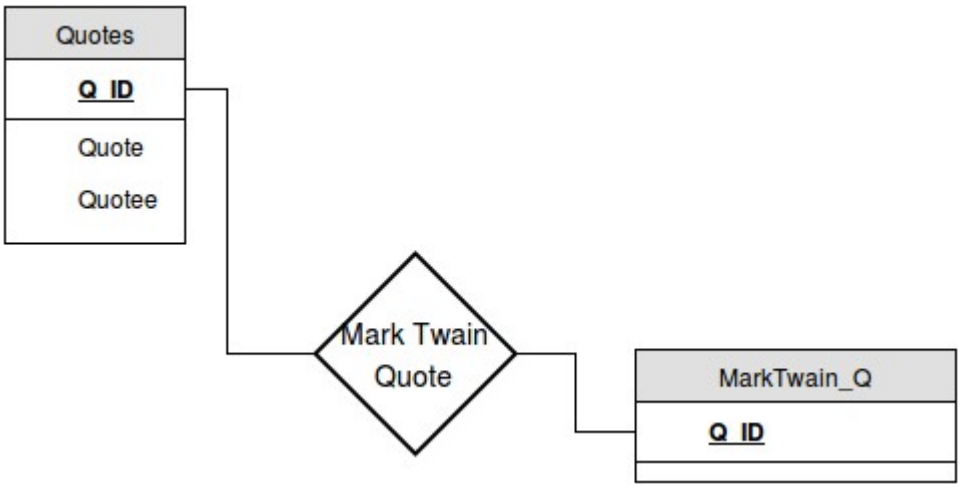

Fig 1. If a Quotee from the Quotes table has been found to be Mark Twain, then it is added to MarkTwain_Q. Not shown here but after this process of searching Quotes for Quotees with Mark Twain and adding their Q_ID to MarkTwain_Q a view is made from a query on the joining of these two tables.

3.3 Design of the Insertion Python Script.

This insertion Python script took all the previously parsed files and insert them into the database. To accomplish a Python module called “MySQLdb” that gave the ability to create a connection to the database, execute MySQL commands, and commit all work to the database or rollback if needed. The query that this script uses not only attempts to insert into the database but also checks to make sure that it is not inserting duplicates. When inserting all of the information truncation errors did pop up for several quotes, but there was no visible errors or problems when checking in the database. There was also a very particular fortune that was not able to insert, but there was no reason that was seen why it wouldn't work, again no harm to the rest of the entries in the database.

3.4 Flask (Web-Module, and Application to Run Module)

First a virtual environment was created that would use Python3, because it was seen how MySQLdb would work on the the Python2 version of Flask.

```
virtualenv –python=/usr/bin/python3 {nameOfEnvironment}
```

After successfully setting up the virtual environment, was the creation of the Python

application that would run the server, which imports Flask. This first application directs the users to all of the web-pages, along with importing and running other applications designed to interact with the database. The HTML and CSS files needed to be stored in certain directory away from the “index.html” file, because how the Flask module searches for them. When starting the main application, it sets up all of its URL paths that a user may be in, connects to the database so it can get a list of currently registered users, creates the list of BOFH excuses that can be displayed, and then hosts either on-line or the local machine depending what arguments were given with its run command. After hosting begins, it merely creates sessions when it needs to, and communicates with other modules to the database for user selections.

3.5 Web-Pages

The web-Pages that were designed for this were very simple overall using simple HTML, CSS, and Bootstrap CSS to create their visuals. There is a home page that a user is first greeted with, and there is a sign up and sign in connections for a user to use. After the user signs up or signs in then they have a query page, and they then have the option to view their profile, or log out. There is a simple text area that a user can put in a query. After submitting the query they have the same page displayed but with a table with a limited number of the query results in a table for them to view. From this page they can enter another query and get new results if they want, or go to the next page to view more results from the same query.

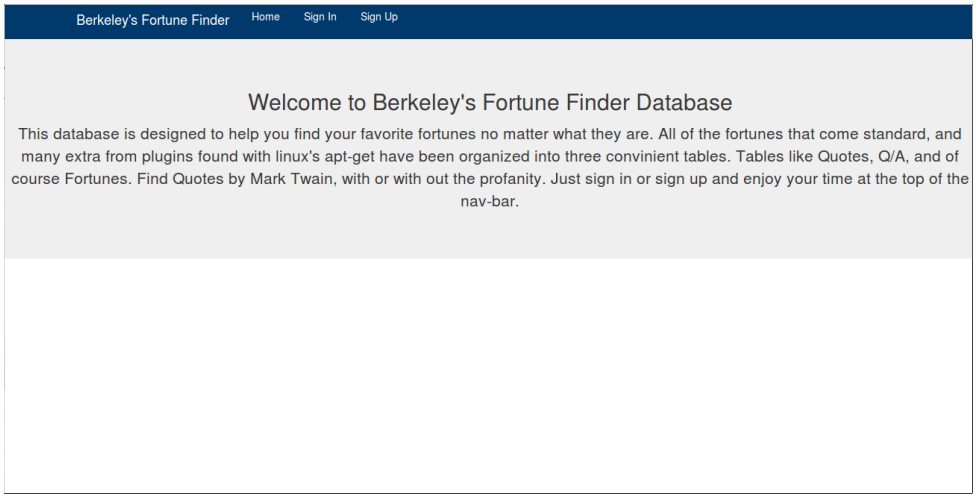

Fig 2. Home page to greet users, and direct them to either sign in, or register as a new user.

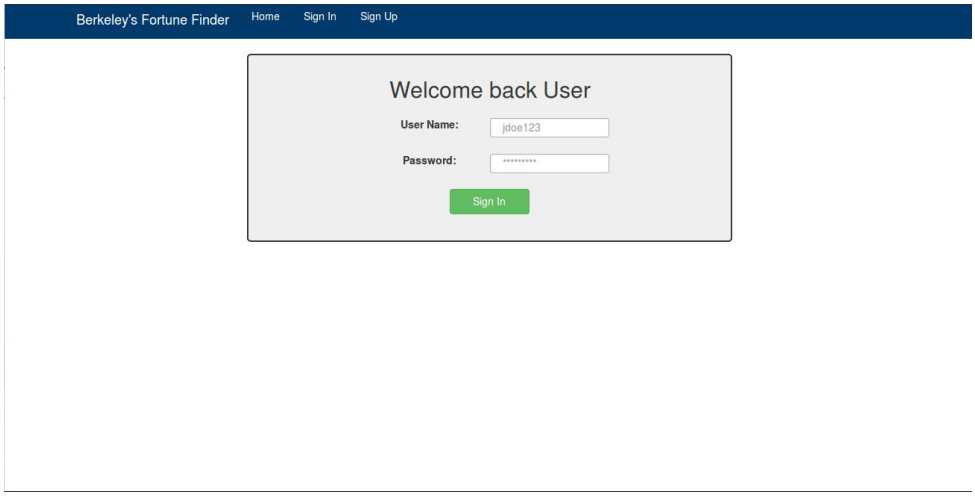

Fig 3. The user sign in page for previous users to log back in.

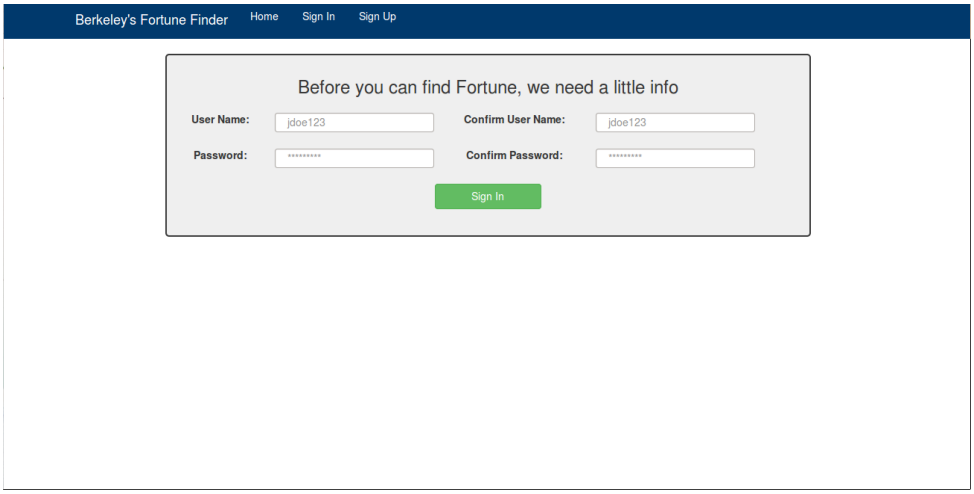

Fig 3. Sign up page for new users to register to be granted access to the database.

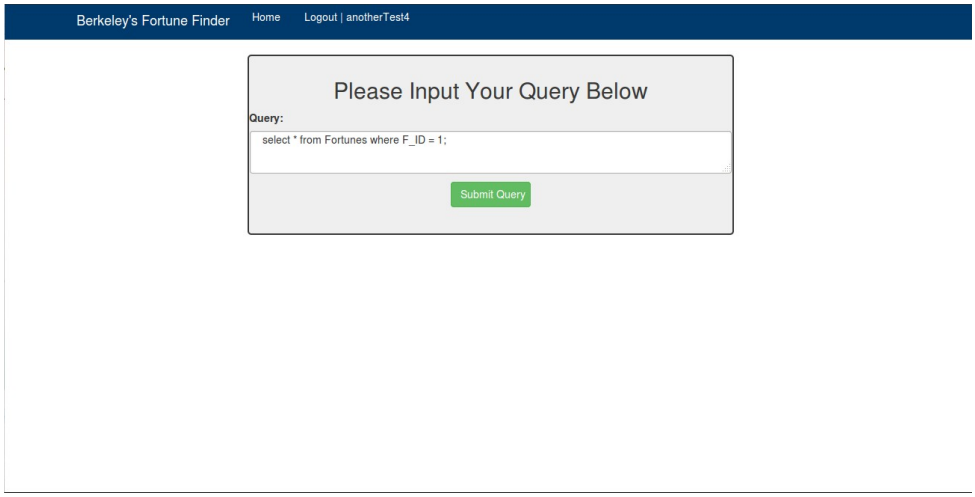

Fig 4. The query page before a valid query is submitted.

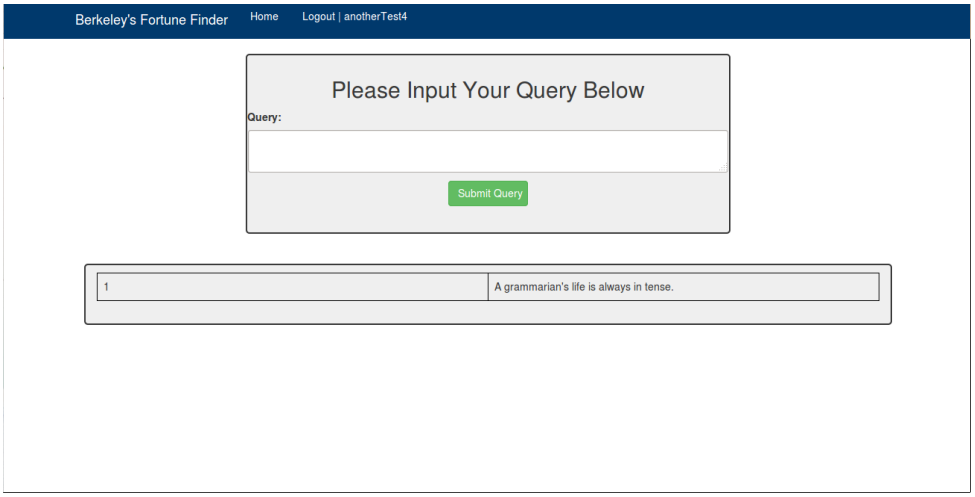

Fig 5. The query page after a valid query is submitted, all the results of the query are placed in a table for human readable formatting.

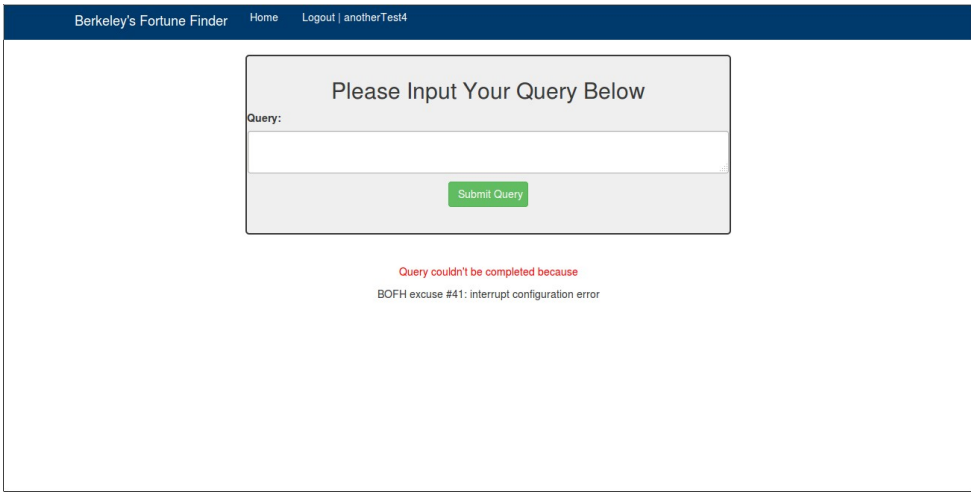

Fig 6. The query page if an invalid query was submitted, either if it were an injection that was detected, an attempt at access of a table or privilege that was not granted to the user. Instead of query results, a BoFH excuse is displayed on the screen along with a message saying that the query could not be completed.

3.6 Extra Python Modules

Additional Python modules were all designed for various uses that would have cluttered the main web hosting application if all present. The sanitizing module takes in user-names, passwords, and queries from users and looks for any obvious code injections. If there is anything suspicious found, then it doesn't allow it to do anything and just alerts the user that the query or logging in or registration couldn't be completed. Another module is used to create a current list of users allowed on the database, adding users to the database, creating a list of the BoFH excuses, and returning a random BoFH excuse to the user if one is needed. The workhorse of all the modules is one that creates a dictionary used to keep up with user's and their valid connections, and getting the user's query results to the main application.

4 Analysis

Keeping the parsing and insertion scripts separate seems to have paid off, that way it was easily visible in the text files that there was some type of logical error in the first parser. This also gave the opportunity to directly have access to certain types that were not parsed

correctly into one convenient file, allowing direct testing of the regular expressions with the values that did not match but should have. It also allowed to group them all back into a new text file and parse them correctly the last time. A better designed state-machine could have solved this problem though, one that would take current lines and the previous line to better track if it were a quote or enough lines to see if the format was similar to a question and answer format. The key-mapping tables were one of the few normalizations that was available for this project, though they do allow for views to easily be built for a user to view certain queries that were created previously for them. The web-site would have two options for querying if there is time to have it made, but time grows quite short as the project is coming to a close.

5 Conclusion

Fortune files and their fortunes are more traversable with just the database being created. With the web-site and a hosting application for users to sign up or sign in and find the fortunes they are seeking in a simpler more human readable format. . The goals of the entire project have have been achieved.

6 Future works

If there was to be anymore progress on this project is constantly adding more tables from the database along with the view that would be create from queries from joining them. Another aspect that would be developed were the profiles for a user, a better looking web-site on the front end, and a list of previous queries or most popular queries for each user to look back through. One thing to be added would be a simple query experience by having drop down menus of tables and other ways to search for non-MySQL literate users. Possibly add more modules would be added the hosting application to give the web-site better abilities or features on the front end of the web-site.

7 Appendix

This last section contains more specific and in-depth information about the entire schema of the database, certain Python scripts, and any other information seeing fit.

7.1 The Database Schema

The visual breakdown below, is the entire database bud does not include virtual tables. The three base tables have a relationship to a mapping table, if said table's entry is relates to the subject matter that composes the name of the key mapping table. A view is created for each relationship to have a simple way for a user to see this relationship. These tables are split into the types as well because to do that I would make the primary key a tuple between the entry that would be displayed to and the distinct ID. Taking this structure for each of these tables, which could be more numerous, would cause redundancy of information. The redundancy of these large entries would be wasteful in respect to hard disk space.

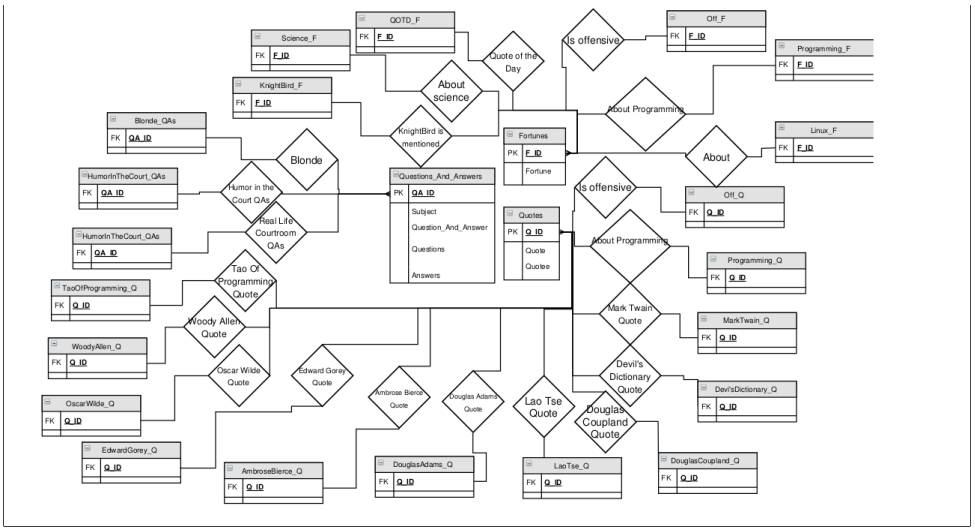

Fig 7. The database schema with the three base table in the center, having a many to one relationship to each outer table using the membership to qualify if the ID is inserted into those mapping tables.

7.2 Python parser.py and parser2.py

Both of these scripts were designed to be a state-machine to determine whether or not an entire entry was a Quote, QA, or a Fortune. First a file from the set of copies made from the program's storage is chosen. Then the state-machine starts with a loop for each

line in the file, giving a particular value to a set variable to identify if it is a Quote, QA, or just a plain Fortune. To identify the type it looks through each line to see if there is a single match to Quote, one match for a question and another to see if it has an answer to determine if its a QA, and if the value is still zero by the time it sees that it is on a new file entry it is seen as a Fortune.

parser.py Regex. The following regular expressions that get "compiled" by Python were to identify if line matched, and if it did fully it was considered to be a quote, question and answer, or some in between of question and answer. The question expression counts exactly two space characters and then two – symbols, a single whitespace character, and then any alphabet entry. The Question and Answer expressions needed to be separated but still had to work together, both had to be tripped as for it to be considered a Question and Answer. They simply search for a Q/A, a single : symbol or . symbol, and exactly one whitespace character.

```
quoteReg = re.compile("^(\s\s-- [a-zA-Z]+)")

qReg = re.compile("^(Q: )|^(Q. )")

aReg = re.compile("^(A: )|^(A. )")
```

parser2.py Regex. These finalizing regular expressions were created to catch more quote and QA types. This time the quote expression doesn't required but at least on space before – symbols and then even took multiple symbols after exactly one single whitespace. The question and answer expressions remained mostly the same functionally.

```
quoteReg = re.compile("^((\s+)-- ((\"?)|([0-9]?)|(\'?)|
(/?)|[a-zA-Z]+))")

qReg = re.compile("(^(Q:\s+)|^(Q\.\s+))")

aReg = re.compile("(^(A:\s+)|^(A\.\s+))")
```

7.3 Python Web-Framework

These modules handle the work that the Flask module distributes to them.

msqlHelper.py. This module in particular creates a User object that contains a dictionary of the cursor objects from MySQLdb of each user, and the user-name. It uses the user's name for the key, identifying their database privileges from cursor to cursor with

functions to get these entries or delete them when a user logs out. The module also executes user's queries, and returns their results.

getReady.py. This module is the one that is dangerous to have active all the time, because it contains the information for an administration level account that creates users, generates users lists, creates the BoFH list, and returns BoFH excuses when needed.

Acknowledgments

Dr. Neel for help in giving the idea of the basic schema for the Fortunes Database.
Robert Edstrom for information on how key-mapping tables function, and why they function faster than a normal query.
http://www.draw.io for creating a visual break down of the schema