Name: Brian Jon Stout
Date: 05JUNE2017
Current Module: SQL Development
Project Name: safetydome

Project Goals

The goal of this project is to create a website that can access data inside of a database and display that information using python and CGI scripts.

Considerations

- Avoid SQL injection attacks
- Pages should be dynamically created because data in the database is not guaranteed to stay the same

Initial Design

There exists a safetydome.sql file. This file is used to create and populate the database used for the server. The three main folders are cgi-bin, containing all the cgi-scripts, misc_img, which contains images that are not related to profile pictures, and profile_pictures which contains images for the fighter's profile. Server.py is an executable program that handles the server responsibilities.

The website's home page is by default index.html. In index.html are three links which all go to python scripts in the cgi-bin folder. The three main scripts in the folder are battle,py, combatant.py, and ranking.py

battle.py generates a list of all the battles which have occurred in the safety dome. In addition to the data generate, there are links for the combatant's names which link to the combatant's detailed page, as well as links that go to a more detailed version of each individual battle.

The python script of the detailed battle is called battledetails.py. Battle.py's link to battledetails.py provides URL variables that are used to generate the web page. Not much dynamic coding is done on this page at all to avoid query calls for a mostly superfluous webpage.

Combatant.py generates a bunch of links to every combatant's detailed page. The link passes a single variable containing the combatant's id. The cgi script is called combatantid.py.

Combatantid.py uses the combatant's id to query all the relevant information for that fighter from the combatant, species, and fight tables.

Ranking.py just generates a table of every combatant sorted by their arena wins. Like the previous pages you can select the names to go to the detail page.

Additional modules include queries.py and extra_output.py. Queries.py contains python queries that are not specific to a single website (aren't specifically tailored for that page's purpose.) Inside are two functions called get_record() and name_lookup(). Get record returns a tuple of the total battles won and total battles fought, and name_lookup() does a query based off a passed Id and return the name of that Id.

Extra_output.py contains two functions called generate_img_tag. Generate_img_tag when given the name of a fighter, checks to see if that fighter has a profile_picture. If it does it generates an img tag based on that file, and if it doesn't it generates an img tag based off a default picture (a question mark in this case.)

The instructed provided mysql_utils.py is also in the cgi-bin and is called from other python programs to establish a connection to the mysql server.

Data Flow

The website starts at index.html. There are three main links on the home page for combatants, battles, and rankings. Going to combatants loads up a cgi-script which contacts the SQL server and generates a list of hyperlinks to the same cgi-script, the only difference is the passed value for cid (combatant id) which is pulled by combatantid.py so it knows which records it needs to pull from the database.

When you click on the link it takes you to combatantid.py with the id of the name you clicked. Combatantid.py runs a query which pulls up all the information about that particular combatant it. It starts with pulling the id stored in "cid" from the url. It then passes the id to the select_combatant function. The select_combatant function's purpose is to return all the rows used by python to fill up the table. It's one of the few versions of similar functions that return data instead of just printing it out, for the specific reason that it only returns one record. The function also does a few quick checks on combatantId to make sure it's actually a digit, and above zero. If the function fails it returns a none type.

The next function run is from the queries module. It's used to grab the battle record (wins and total battles) for the combatant. This function returns none by default if it can't find record, but in hindsight it should have also contained id checks to make sure valid input was being given, or at least checked to see if select_combatant returned a none_type already to avoid doing work twice.

Combatantid.py then checks to see if either of the functions returned None. If it has, then it prints out html code that displays an error message. If it hasn't, then it runs the generate_img_tag from the extra_output module. Generate_img_tag generates a path based on a combatant name, and checks the path to see if the combatant has a profile picture. If it does it generates a img tag based on that picture, and if it doesn't it displays a question mark.

Then it runs the print_combatant function. Since select_combatant doesn't do it's own printing, the printing is handled by the print_combatant function. Unfortunately the detailed table for the combatant's profile has a lot of print lines. There are several rows and two columns specifying each specific data point being displayed. At the end of the table there's links provided that go back to the home page, rankings page, battle's page, and combatant's page.

When you click the link to go to the battles page, it loads up battle.py. The script does basic CGI html setup and then calls the print_battle function. The print_battle function grabs all the information from the fight table with a few modifications. Instead of grabbing the finish datetime stamp from the last column, it uses the TIMESTAMPDIFF SQL function to calculate how many seconds the battle lasted.

From here the print_battles function sets up a table with the headers Match Link, First Combatant, Second Combatant, Victor, Start Time, and Match Length. Format strings for string.format() are made for both generating a link to the combatant's profile page, a link to the more detailed battle page. The function then steps into a for loop that enumerates each row from the query. The row data is stored in the row variable, and the iteration number is stored in the index number. Index is used to number the list as well as server as the link to the main battle page. Based off the combatant_1 and combatant_2 id fields, the name_lookup function is called and the returned strings are stored in appropriately named variables. After this there is a quick check that compares the ids for combatant one and combatant two to the id in the victor column. Depending on which one are equivalent the victor variable gets stored that fighter's name to avoid another SQL lookup. The function the prints out each row based off the format strings. At the end there's links provided to navigate the page.

When you clink a link to the main battle the website gets changed to battledetails.py and all the fields for that record are stored as values in the URL to avoid making another lookup for the same exact information. The website formats the data in a way that dramatizes the single battle since there's no actual in depth information that is new. While it would easy for the user to change the website to display whatever they want, it's mostly pointless since there isn't any extra SQL look up there's no risk of exploitation.

Clicking on the rankings link sends you to ranking.py script. Ranking.py is a very simple script that generates a table of a fighter, the number of wins he's made, and sorts it so the most wins show on top. When the script first starts it does basic HTML setup then jumps into the print_ranks function. The print_ranks function does an SQL query based on winner_id, the count of winner_id and then orders it by winner_id which presents a bug. The order by should be the count of winner_id. Whoops. The function then goes through each record and generates a table based off the information.

Communication Protocol

HTML web servers

Potential Pitfalls

- SQL Injection vulnerability from string appends.
- Not verifying database integrity

Test Plan

User Test:

1. Click on all the links to make sure the web pages work as expected.
2. Modify url variables to test for SQL injection

Test Cases:

1. Websites work as expected
2. There is little to no risk of SQL injection

Conclusion

The website works as expected though probably not as well as it could. To avoid spending too much time perfecting each little element (you could spend hours tweaking one thing on a website.) I decided not to focus too much on the visual aspects of the program. Even with that, web design work is extremely tedious and HTML is ugly by itself, let alone stuck in print statements.

Trying to get the SQL injections to work was a good exercise though, even if most of the API tools we were using to generate the website with python prevent all the very common injections.

One of the things that I forgot to implement in the website was checking for database integrity, like there being no values to look up. In real life it'd be important since a web designer might not be the database administration and things could easily get messed. In hindsight at the bare minimum I should have checked for no records being returned at all, and in some scripts I don't where others I didn't because I made the assumption that the database WOULD have information in it, which isn't up to me to debate if I should try to solve a test case that makes it impossible for the user to use the website when it exists anyway.