Camelot Final Summary

A summary of the Camelot server. Created and maintained by Ian Howell, Hunter Mathews, Illya Starikov, William Thurman, Zachary Wileman. *Server Team #1*

Last Revised: *May 5, 2017*
Illya Starikov

Michael Gosnell
Immediate Supervisor
mrghx4@mst.edu

Status: **Approved**
Version: **1.0**

For our project, we decided to implement an asynchronous server written in Python3 and uses a PostgreSQL database to store user and channel information. The Python module `pytest` was also used for writing unit tests so that we may try to have have full code coverage on the project and "bring to light" any bugs that may exist in our project in terms of the Server and Database class files.

Some of the main accomplishments that were made on this project include: designing a database which was used for storing the user and channel information, creating a class file for both the server and database where the server file was dedicated to server functions (most functions mainly used by clients) and the database file was dedicated to interacting with the PostgreSQL database, writing unit tests to help better test our code so we could find any bugs/problems that may have existed, and created a multi-threaded asynchronous server for use with clients along with a client file used for testing purposes.

When designing the architecture, we decided to follow a Model View Controller paradigm. The controller was the server, the model was the database, and the view was the respective Client teams graphical user interface. In true MVC fashion,

- the server acts as the bridge between the client and the database
- the database stores all the data
- the client provides a view for the data

Our server runs on an asynchronous model. This means that we are able to service multiple clients independently of each other. This is accomplished by making use of Python3's threading library. Each client is thrown onto its own thread, and each thread is thrown into a list of threads. When a client sends a message, we loop over each client (within the respective channel), sending the message to each. This is extended to all other actions performed by a client, such as logging in and deleting channels. When a client closes the connection to the server, it is removed from the client list.

The database we designed uses three tables, one for the user, one for the channel, and one for the user in channel relationship. The `USER` table stores the username and password of the user. The `CHANNEL` table stores the channel name and the admin of the channel. The `CHANNELS_JOINED` table stores which users are in which channels. Not only does the database allows us to query which users should receive a message, but it also allows us to perform other more complex operations such as notifying all users when a channel is deleted. An additional PostgreSQL file was also included that adds some initial channels to the database that all users are, by default, added to when they create their account.

While we waited for the client teams to finish their clients and begin testing the connection to our server, we created a handful of unit tests that would allow us to test certain functions that we would provide to the client teams to use. These functions included creating accounts, logging in, creating channels, deleting channels, deleting accounts, sending messages, along with many other things. For each different test, we covered all of our bases, both fail and success. Some examples include:

- If the user was not logged in, then what would the server tell the client
- If there was an invalid JSON file, then what would the server tell the client
- If the function was to fail, make sure the server handled the exception properly

As far as any deviations from the requirements document, none seemed to occur for our team, but one of our client teams did experience a change to their requirements document that affected us. Specifically, one of the client teams was originally wanting us to implement private messaging as a feature but the team decided to abandon that idea after they found out they werent going to be able to implement this feature. So, due to that decision, the team decided to no longer pursue the implementation of the private messaging feature.