Camelot
System Requirements

All relevant documentation for system requirements.
Created and maintained by Ian Howell, Hunter Mathews, Illya Starikov, William Thurman, Zachary Wileman. *Server Team #1*

Last Revised: *March 12, 2017*
Illya Starikov

Michael Gosnell
Immediate Supervisor
mrghx4@mst.edu

Status: **Proposed**
Version: **1.1**

Contents

1 Introduction

Camelot is JSON-based server written exclusively in Python 3. The purpose of Camelot is to provide an IRC-esque chatroom on the server, leaving implementation details for the client up open to interpretation.

1.1 Purpose

This document is intended to guide development of Camelot. Not only will it provide the requirements (i.e. system requirements, user requirements, and interface requirements), but it will provide a decent outline for what the server entails (i.e. functions, constraints, dependencies, etc.). Understanding the system requirements will give a more clear and concise understanding of the Application Programming Interfaces (APIs) and their purpose.

1.2 How to Use This Document

Because this document will be reviewed by various different skill sets, this section will break down which parts should be reviewed by which type of reader.

1.2.1 Types of Reader

This document is going to be aimed at two different type of people: developers and end-users. The first set, client/server side developers, will be the Python3 Programmers, server-side programmers, Information Technology (IT) personal, and others that might have communication needs. These people are going to be the ones who wish to build upon or plan to use the code commercially. The other set will be the end-users. These people will be running the server in order to hold a chat session(s).

1.2.2 Technical Background Required

Programming competency *is imperative* to understanding the documentation. Programming methodology, jargon, and general concepts will be assumed in subsequent sections. Server side programming is recommended, but not required.

The document will stay to a high level server design. For the purpose of this document, source code will not appear. Specific tools/programming languages capability is not required.

1.2.3 Overview Sections

To get a a general understanding of the document, this following sections and subsections should be (chronologically) read:

1. User Characteristics (Section 2.3)
2. Assumptions and Dependencies (Section 2.5)
3. Specific Requirements (Section 3)

1.2.4 Reader-Specific Sections

Types of readers are described as follows.

- Server-side Developers: This would entail anyone interested in working on this particularly instance of the server. All of this document should be read.
- Client-side Developers: This would entail all who would want to get involved in creating a client. The following sections should be read:
  1. Description (Section 2)
  2. Specific Requirements (Section 3)
- End Users: This would entail anyone that is generally interested, but has no intentions of truly using it standalone.
  1. Product Perspective (Section 2.1)
  2. Product Functions (Section 2.2)
  3. User Characteristics (Section 2.3)

1.3 Scope of the Product

Truthfully, the intention of this project is to get an A in software engineerings. But I actually have to put something here, so.

There is an expectation at the end of the development lifecycle to open source this server. After open sourcing, the server team hopes that Camelot will serve as a model for server-side programming. Because the JSON-based framework would be familiar to people, the team hopes for a some market adoption. Overall, this would be a good model for how a simple server would be maintained.

1.4 Business Case for the Product

There is a real pandemic: there are not enough chat applications. Sure, there's Messenger, Slack, Skype, Viber, WeChat, WhatsApp, Line, GroupMe, Snapchat, Voxer, but they are garbage. The market needs a great server to tackle on these giants, and Camelot will do that.

1.5 Overview of the Requirements Document

1. Raspberry Pi 3 needed to host the server
2. Knowledge of Python to program the server
3. JSON based framework

2 Description

This section will give the reader an overview of the project, including why it was conceived, what it will do when complete, and the types of people we expect will use it. We also list constraints that were faced during development and assumptions we made about how we would proceed.

The Camelot Server project will create a means of communication between chat clients, in order to allow end users to communicate with each other.

2.1 Product Perspective

We have chosen to develop this project in order to create a standard form of communication between the chat clients that are being developed to use the server. The developers will use this in order to create an interface to house the information being transmitted.

2.2 Product Functions

The server will have the following capabilities.

2.2.1 User Authentication and Login

Users Will have the option to either create an account or to log in with an existing account.

Users will be able to create accounts by submitting a username that hasn't been taken. The client will send a JSON encoded file that looks something like:

```
{
    "tags": {
        "create_account": True
    },
        "create_account": {
        "username": "some username",
            "password": "some password",
        "server_password": "some server password"
    }
}
```

If the username has already been taken, then the client will receive a JSON encoded file that looks something like:

```
{
    "tags": {
        "account_creation_success": False
    },
        "message": "Some string"
}
```

When the client sends a request to login, it should send a JSON encoded package to the server. It should look something like this:

```
{
    "tags": {
        "login": True
    },
    "login": {
        "username": "some username",
        "password": "some password",
        "server_password": "some server password"
    }
}
```

If the username doesn't exist or if the password the user enters doesn't match the password associated with the user, then the client will receive a JSON encoded file that looks something like:

```
{
    "tags": {
        "login_success": False
    },
    "message": "Some string"
}
```

There will also be a server password given to each client that the user doesn't have to enter but the client will have to pass along with the user login so that the client can be authenticated.

2.2.2 Channels

Channel Creation

As of right now, there will be a set number of channels created by the server that the clients will have the option of joining. Later on in development, we may add the option for users to create channels. Upon creation, the creator will become the admin of that channel. Each user will have a limit on the number of channels they may create.

Channel Deletion

As of this writing, only the server team will have the option of deleting specific channels. When users gain the ability to create channels, they will also gain the ability to delete channels that they have created[1].

[1]The initial channels will be owned by the server team.

Initial Joining of Channels

After a user has logged in, the server will send the client a list of default channels that the user has the option of joining[2]. It will look something like this:

```
{
    "tags": {
        "login-success": True
    },
     "channels": [
        "channel 1",
        "channel 2",
        "channel 3"
  ]
}
```

The client will then need to send back a JSON encoded file to the server describing what channels the user would like to join. The JSON file should look something like this:

```
{
    "tags": {
        "joinChannel": True
    },
    "join-channel": [
        "channel 1",
        "other channels they may want to join"
    ]
}
```

After the user successfully joins the channels, the specified user will have access to the channels that they decided to join.

Joining Channels After Selecting Initial Channels to Join

Users will have the option to list the channels when logged in. They will be able to join channels in the same fashion as they did initially[3].

```
{
    "tags": {
        "newMessage": True
    },
    "channel_receiving_message": "some channel name",
    "message": {
        "user": "the username",
        "timestamp": "the time at which the message was received",
        "message": "the actual message that the user posted"
    }
}
```

[2]Later on, the user will have the option to search for channels based on a keyword.
[3]Later on, the user will have the option to leave channels.

When a user (client) wishes to send a message to a certain channel, the JSON object should look something like this:

```
{
    "tags": {
        "newMessage": True
    },
    "channel_receiving_message": "some channel name",
    "message": {
        "user": "the username",
        "timestamp": "the time at which the message was received",
        "message": "the actual message that the user posted"
    }
}
```

Notice that both the JSON file being received by the client and the JSON being sent out by the server look exactly the same. This is done so that the server can simply broadcast messages to all users without having to decode a JSON file server side. This will give the least amount of delay in messages being sent out to each user. Also, the user that sends the message out will also receive the message. It's up to the client in how they want to approach this situation.

2.2.3 Send/Receive Messages

Whenever the server receives notice that a new message has been posted to a given channel, the server will send out a message to each user who is connected to that given channel. The message will be a JSON file that looks something like this:

2.3 User Characteristics

Most users will be developers who will make clients to interface with the server. They will be mostly of a more technical background, with education background involving computer programming. They will be interfacing with the server in order to create a client for end users to communicate with each other. They may encounter obstacles with reading messages if they have no experience with parsing JSON.

2.4 General Constraints

For the constraints of our server, we didn't have any specific constraints as to what Integrated Development Environment (IDE) each person used or the platform that they developed on. The only specific constraints that we have so far is that the server is going to be developed with Python3 using JSON for data transfer and also that each person use Git for collaborating on the development of the server. We have not run into any issues with making our server compatible with other software as of yet.

2.5 Assumptions and Dependencies

The assumptions made with the development of this project is that each person that will be working on the server has a working knowledge of some programming language (most likely C++) and is willing to learn python. Its also assumed that at least some people working on the server have some background knowledge as to how to program in python as to help others within the server group who aren't as familiar with Python.

3 Specific Requirements

The follow sections will go in-depth about the specific requirements of the Camelot system.

3.1 User Requirements

Our user base will be split across two use cases:

1. The end users, who will be referred to as the user, and
2. The client teams, who will be referred as the developers

3.1.1 End User Requirements

The user will need to have a client that can connect with the Camelot server. They will need to be able to communicate with other users across channels. They should be able to send and receive messages. Messages should be sent and received in a logical order.

3.1.2 Developer Requirements

The developers will require the ability to request a list of channels. The list will contain information regarding the channel names and the users currently in each channel. With a successful login, they should be able to create to create a unique user with a specific identifier. Users should be deleted upon signing out or error. The developers should be able to request message information, such as message text, senders, requested channel, and timestamps.

3.2 System Requirements

The Camelot server will be run on a Raspberry Pi 3 Model B. It will need need to have internet access. The hardware will need to have Python3 installed as well as PostgreSQL for database management.

3.3 Interface Requirements

The Camelot will need to interface with a client using JSON formatting as a data transfer protocol.

4 References

Below is a list of all relevant documentation/references for this document.

C++ http://www.cplusplus.com

Git https://git-scm.com

JSON http://www.json.org

PostgreSQL https://www.postgresql.org

Python3 https://www.python.org/download/releases/3.0/

5 Version History

Below are major releases and their respective changes.

- **1.0** Initial release
- **1.1** Bug Fixes
  - Fixed typos, mistakes, inproper wording, etc
  - Fixed Pandoc spacing issue where a `\n` would appear after 80 characters
  - Further explained functionality

6 Glossary

C++ A general-purpose, object oriented programming language. 10, 12

Git A free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. 9, 12

IDE Integrated Development Environment. 9

JSON JSON is a syntax for storing and exchanging data, written with JavaScript object notation. 9, 11, 12

PostgreSQL Sophisticated open-source Object-Relational DBMS supporting almost all SQL constructs. 11, 12

Python An interepreted programming language popular for server side programming. 9, 11, 12