SENG 299 Project Milestone 4

The Implementation of Hissper

http://hissper.herokuapp.com

Prepared for Caleb Shortt

SENG 299

August 4, 2017

Created by:
Aman Bhayani
Jordan Vlieg
Andrew Li
Seth Tolkamp

Table of Contents

Purpose

Purpose of this report is to give a comprehensive overview of the implementation and design of the web based chat app Hissper. This is the third report detailing Hissper and it goes in depth into its actual construction and creation and the changes that were made to the design proposed in Milestones 1 and 2. This report introduces what Hissper is, the objectives of Hissper and the project as a whole are expected to meet, the implementation and design, and finally problems encountered and changes that were made throughout the design process.

Introduction

The Hissper Chat system is software that provides real time communication between two or more parties with the use of network sockets. Throughout the development process a few changes regarding this model has changed, but the essence of this process for the most part has remained the same. The techniques used to drive the development of this product has until this point been unclear. That is, it was not fully explained what *form* this software will take (i.e. a web app, a command line interface…, etc). In this document, information pertaining to these decisions, changes, and design techniques will be discussed, argued and justified. In addition, modifications to functionality and the addition/removal of features will also be explained. The Hissper Chat system has become an evolving project that has changed throughout the phases of its Software Development Life Cycle (SDLC). However, Hissper provides significant use, meets outlined goals, and incorporates user experience into its development far beyond originally described.

The content in the below pages explain in detail the implementation process and the management decisions made for software construction. It is expected that the reader has familiarized themselves with previous documents regarding this system, and has a grasp of technical concepts, vocabulary, and programming prior to reading the contents of this document.

Software Implementation

Before referring to modifications, changes, and reference to original specifications of the Hissper Software System, it is important to clearly describe what the software has become. This helps solidify what the software is, and what it is not, and acts as a guide when comparing differences to the original software plan. In this section of the document the mechanics of the software are discussed, referring to *how* the software works, and the architecture of the system components at this development stage. This document specifies the design patterns and techniques used, explains implementation through the use of code snippets/visuals, and helps define the system as a whole to provide breadth for other sections of this document. It is important to note that this section of the document does not refer to, compare, or discuss any design changes made to the product from previous phases, and simply explains the Hissper Chat System at its current stage. These issues are covered later in the document.

Note: Hissper was not reviewed in Milestone 3 by an outside source and as such there were no changes made based on another party's recommendations.

Overview

The Hissper Chat system is a web application implemented using a Model-View-Control (MVC) approach. It comprises of a webstack that contains back-end and front-end development. It utilizes a variety of technologies that can be thought of as layers to the software as a whole. The back-end houses a Python server, while the front-end takes advantage of Javascript, and markup. Below are the details regarding this stack and the mechanics of how each component works together.

Back-end

The Hissper Chat System makes use of Python code, which is the backbone of the software. Without this component the application would fail, since the rendering of front-end markup is equally depended on the server-side of this application. The use of four main frameworks are used in this back-end component (see figure 1):

1. Flask
2. Flask-SocketIO
3. Gunicorn
4. Eventlet

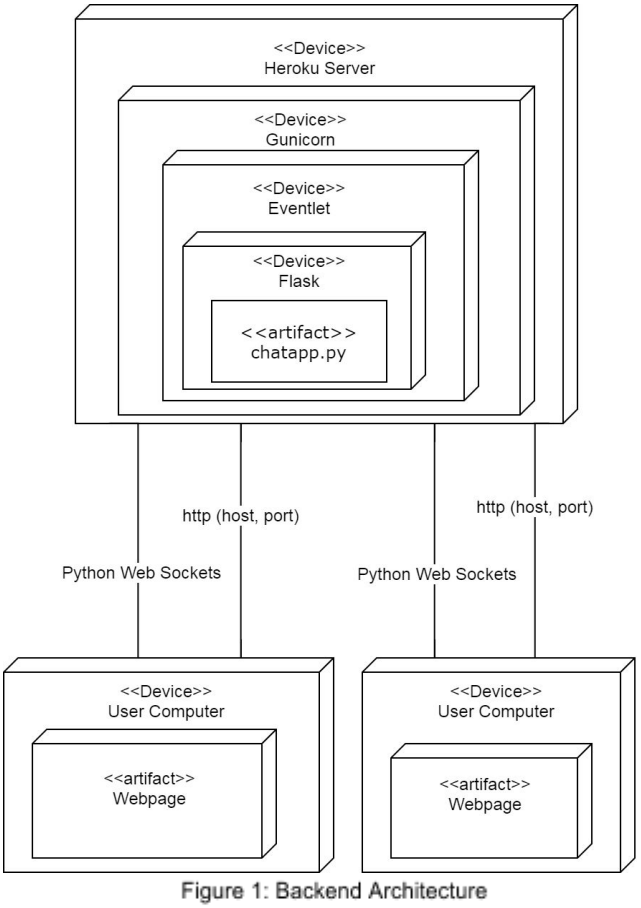


Figure 1: Backend Architecture

Below are simple explanations of how what these frameworks do and how they work for the Hissper System.

1. Flask

Flask is a web application framework written in Python that allows developers to avoid the need to write low-level code. It allows developers to write a server in a much simpler way. When writing a server the need for I/O, threads, and logic is usually required to handle requests received from client-side. However, Flask skips this process and interprets all requests received from the client. It utilizes 3 additional Python frameworks that help with this process:

- ➔ WSGI
- ➔ Werkeug and
- ➔ jinja2.

WSGI
Web Server Gateway Interface — WSGI is an adopted standard for Python web application development, and acts as the interface between the web server and the web application.

Werkeug
Werkeug is a toolkit. It allows for requests, and any communication to Flask to be properly interpreted. It is the base used by Fask to allow for any web framework to be built on top of it.

Jija2
This is a Flask extension for template rendering. It allows for dynamic web pages on the front-end. Any “events” that occur on the web application by a user's actions are interpreted by Flask and render changes in the GUI of the application.

2. Flask-SocketIO

Flask Socket IO are websockets meant to go hand-in-hand with Flask. Since Flask helps supply the server, Flask-SocketIO allows for bidirectional communication between the client side and the server. This module also handles a lot of the heavy lifting and management of users and chat rooms. The emit function is used to broadcast information to any users in a chat room, while the join_room, leave_room, and close_room handle chat room management within the backend.

3. Gunicorn

The Gunicorn "Green Unicorn" is a Python Web Server Gateway Interface (WSGI) HTTP server and is commonly used to run Python web applications. It is an implementation of the WSGI servers concept and syncs well with Flask. The built in flask server is not intended for deployment and intended for testing only, Gunicorn is the recommended replacement as it syncs well with both Flask, and eventlet.

3. Eventlet

Eventlet is a concurrent python networking library that allows for the use of both websockets and polling in a flask application. Integrating eventlet into our project was a matter of importing it, and launching our application using it through the procfile.

Backend Mechanics

The above mentioned stack is simply the skeleton of how the software will be contained. In a sense, the Back-end description above can be thought of as the development environment for implementation. Nonetheless, code must be written (in a particular manner) to make use of this stack. Python is the choice of language for this task, and therefore below is an overview of how the mechanisms of the back-end is written.

API

Functionally Flask works to create all of the server's RESTful API endpoints and listeners which can be reached by the various clients. There are 9 server endpoints and socket listeners in total and they are as follows.

'/'

The endpoint for the main landing page. The sole purpose of this endpoint is to return the splash page that users see when navigating to the web page (see figure 2)

```
@myapp.route('/')
def index():
    return render_template('splash.html')
```

Figure 2

'/chat/<chat_room>'

The endpoint that returns the chatting page, this endpoint allows users to retrieve the HTML, CSS, and JavaScript required to be a part of a chat room, send, and receive messages. (see Figure 3)

```
@myapp.route('/chat/<chat_room>')
def chat_room(chat_room):
    return render_template('ChatAppPage.html')
```

Figure 3

'/dashboard'

This returns the page that allows users to join, create, and delete chat rooms. It also sends back information regarding current status such as the names of all existing rooms, and how many users are part of them. (See figure 4)

```python
@myapp.route('/dashboard')
def dashboard():
    numUsers = []
    sortedRooms = sorted(list(room_history.keys()))

    for item in sortedRooms:
        numUsers.append(len(room_users[item]))

    return render_template('dashboard.html', rooms=sortedRooms, users=numUsers)
```

Figure 4

‘/getFileName’

This endpoint returns the audio file required to play the sound when a new user joins the chat room. (See Figure 5)

```python
@myapp.route('/getFileName')
def get_file_name():
    return 'static/content/connected.mp3'
```

Figure 5

‘message’

This allows users to send and receive messages when already a part of a chat room. When this endpoint is hit, it adds the incoming message to the history cache of the room, then broadcasts the message to every users that is currently a member of the room.
(See Figure 6)

```python
@socketio.on('message')
def on_message(json):
    print('received something chat:' + str(json))
    room = json['room']
    room_history[str(room)].append(json)
    emit('message response', json, room=room)
    if len(room_history[str(room)]) > 25:
        room_history[str(room)].pop(0)
```

Figure 6

‘createroom’

This is used by the dashboard to allow for the creation of new chat rooms, it creates a history and users list in the backend to store the 25 most recent messages in the room, and the members of the room. (See Figure 7).

```python
@socketio.on('createroom')
def on_create_room(json):
    room_name = str(json['room'])
    room_history[room_name] = []
    room_users[room_name] = []
    if room_name not in room_history:
        room_history[room_name] = []
```

Figure 7.

‘leaveroom’

This deletes a user from the list of users that are a part of the room, as well as closes any connections they have with the server. See figure 8

```python
@socketio.on('leaveroom')
def on_leave_room(json):
    print("LEAVING ROOM")
    alias = str(json['alias'])
    room = json['room']
    leave_room(room)
    if alias in room_users[str(room)]:
        room_users[str(room)].remove(alias)
    emit(alias + ' has left the room', json, room=room)
```

Figure 8

‘deleteroom’

This deletes an entire room, the history associated with it, and removes all the users from the room.(See Figure 9).

```python
@socketio.on('deleteroom')
def on_close_room(json):
    room = json['room']
    del room_history[str(room)]
    del room_users[str(room)]
    emit("Room is being closed, please leave", json, room=room)
    close_room(room)
```

Figure 9

‘join’

This checks first that the room is not already full, if not, it proceeds to add the user to the room and sends them the appropriate join message, be it the 25 most recent messages, an empty room message, or a freshly created room message. If the room is full, it sends the user a message informing them of such. (See Figure 10)

```
@socketio.on('join')
def on_join(json):
    alias = json['alias']
    room = json['room']
    room_name = str(room)
    if len(room_users[room_name]) < MAX_ROOM_SIZE:
        join_room(room)
        room_users[room_name].append(alias)
        emit('join response', json, room=room)
        if len(room_history[room_name]) == 0:
            if room_name == 'general':
                emit('general room msg', json)
            else:
                emit('new room msg', json)
        else:
            emit('history req', json)
            for item in room_history[room_name]:
                emit('message response', item)
    else:
        print("ROOM HAS: " + str(len(room_users[room_name])))
        emit('full response', json)
```

Figure 10

Front-end

Moving from the Back-end to the Front-end, the Hissper Chat System also requires code to be pleasantly display an output to the user. The design of the Chat system utilized another traditional front ‘stack’ that is used to depict both design and functionality when the user interacts with the browser.

The front-end of the Hissper Chat System is written in HTML, CSS and JavaScript. This makes up the front-end ‘stack’ for user experience. HTML and CSS is the markup and the styling of the web application and is what the user sees when they interact with the application. UI/UX design was discussed as a group and ensured that the color and elements of the page are in optimal positions. (See Figure 11)

```
<title>Hissper Chat Client</title>

<!-- Bootstrap -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="
anonymous">
<!-- Custom CSS-->
<link rel="stylesheet" href="../static/css/splash.css">

<!-- Animate CSS -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/animate.css/3.5.2/
animate.min.css" />

<body>
<div class="container">
<div class="ng-scope" id="splash-page">
    <div class="fade-element-in ng-scope">
        <div class="grid-block header center">
            <div class="grid-container">
              <img id="logo" src="../static/content/logo.png"></div>
            </div>
        </div>
        <div class="grid-block container v-align">
            <div class="login-buttons grid-container">
                <span class="slogan">Python Powered.</span>
                <div class="login-button">
                  <button id="mylogin" type="button" class="btn btn-primary btn-lg">Start</button>
                </div>
```

Figure 11 — Example of HTML Splash Page

The front-end communicates to the back-end by socket programming in the JavaScript. Each Python end point (described in the Back-end section above) is called by using the “emit” function in JavaScript. Emit takes in two arguments, the first argument is the string that matches the Python endpoint and the second argument is an object of variables to pass to the back-end. This allows the user to interact with the back-end to perform additional actions. (See Figure 12)

```
var socket = io.connect();
var alias = sessionStorage.getItem("alias");
socket.on('connect', function () {
    var path = window.location.pathname.split('/');
    room = path[2];
    socket.emit('join', {
        alias: alias,
        room: room
    });
    document.getElementById("chat_room_name").innerHTML = room.replace(/%20/g,'');

    document.getElementById("alias-banner").innerHTML = "Alias: " + alias;
});

//Capture users joining
socket.on('join response', function (msg) {
    if (typeof alertify !== 'undefined') {
        alertify.set('notifier', 'position', 'top-right');
        alertify.success(String(msg.alias) + " has joined the room");
    }

    var audio = document.getElementById('sound');
    if (audio) {
        audio.src = '../static/content/connected.mp3';
        audio.load();
        audio.oncanplaythrough = function () {
            this.play();
        }
    }
});
```

Figure 12 — Example of communication from Javascript (Front-end) to Python Flask Sockets

Design Pattern

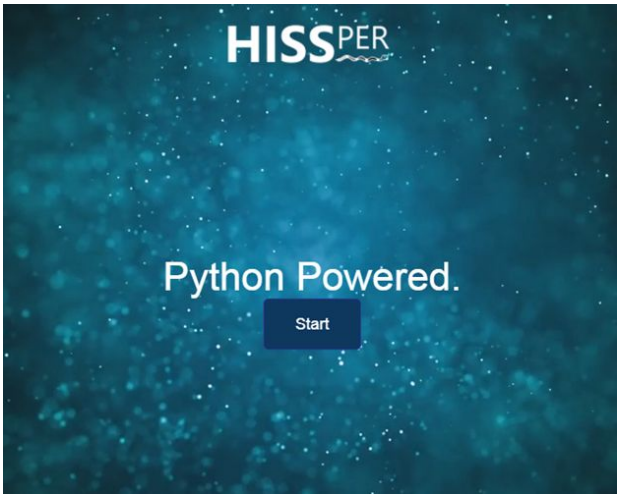
The design pattern utilized by the Hissper Chat System comprised of **Information Expert**. That is, the software was written by separation of duties. This means that each module was given a task to complete, and can easily be thought of in terms of template pages. For example: the splash page determines the information of the Alias, which is then used throughout the system. Therefore all code pertaining to that particular page is the information determined to fulfill. Since this software project was relatively small in size, no classes were utilizes, nonetheless, down the line Hissper may include other design patterns as the software scales and evolves.

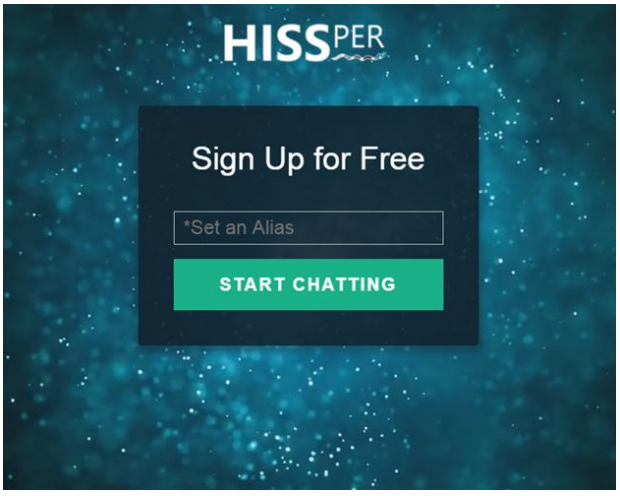
User Experience & User Interface Gallery

In this section of the document, an overview of what a user sees, does, and can do when using Hissper. Visuals that capture the user interface and user experience help guide this section. It is recommended to review this section to understand the software's entire functionality. The web application is created with the user in mind. As such a clean and simple design was implemented for users to be able to navigate the software without any instructions. Nonetheless, documentation pertaining to all functionality and desired outputs are mentioned here. It is expected that the software will output the correct results, but since software maintenance is a constant necessity, this section serves as a description for output comparison and evaluation.

Start

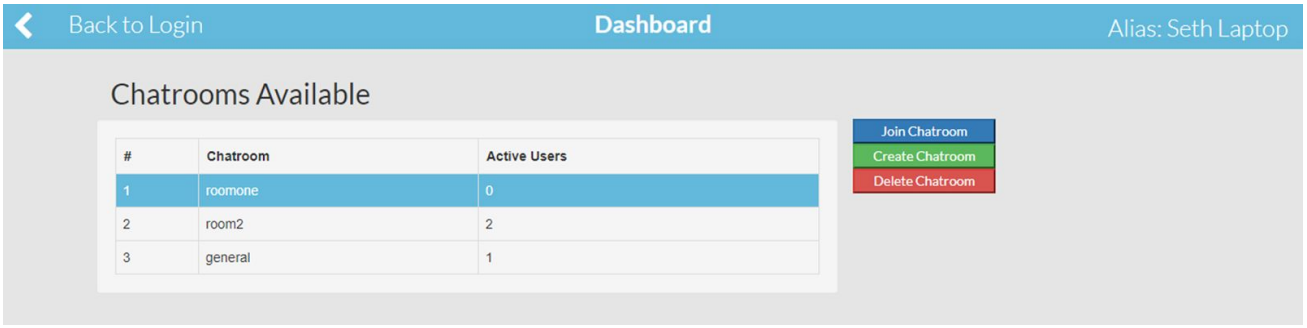
When users travel to hissper.herokuapp.com, they are presented with the following start screen. Here they click ‘Start’ and are prompted to set their alias. After setting an alias, the user clicks ‘Start Chatting’ and is brought to the dashboard.





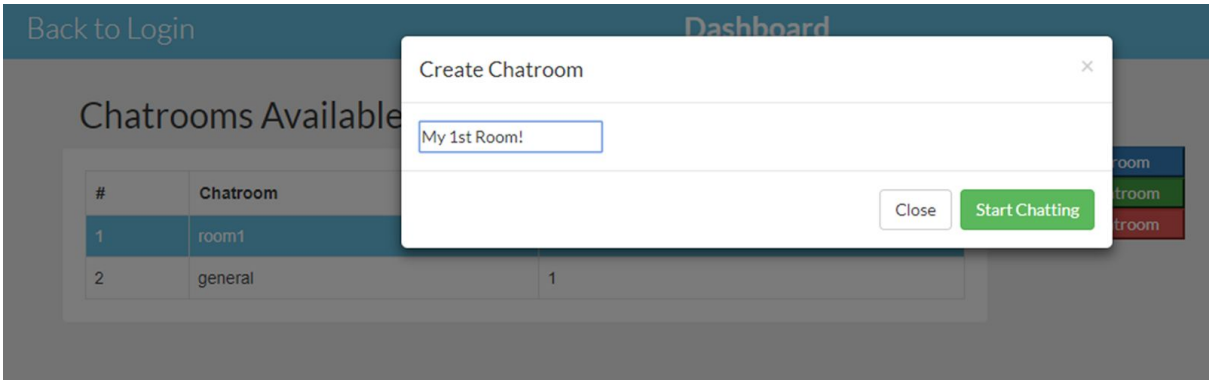
Dashboard

The dashboard is the central navigation page for users. Here, users can view all the chat rooms, see the number of users in the chat rooms, join, create, and delete chat rooms. In this figure ‘roomone’ is selected and is the room the user would join if they clicked ‘Join Chatroom’ and delete if they clicked ‘Delete Chatroom’.



Make a Room

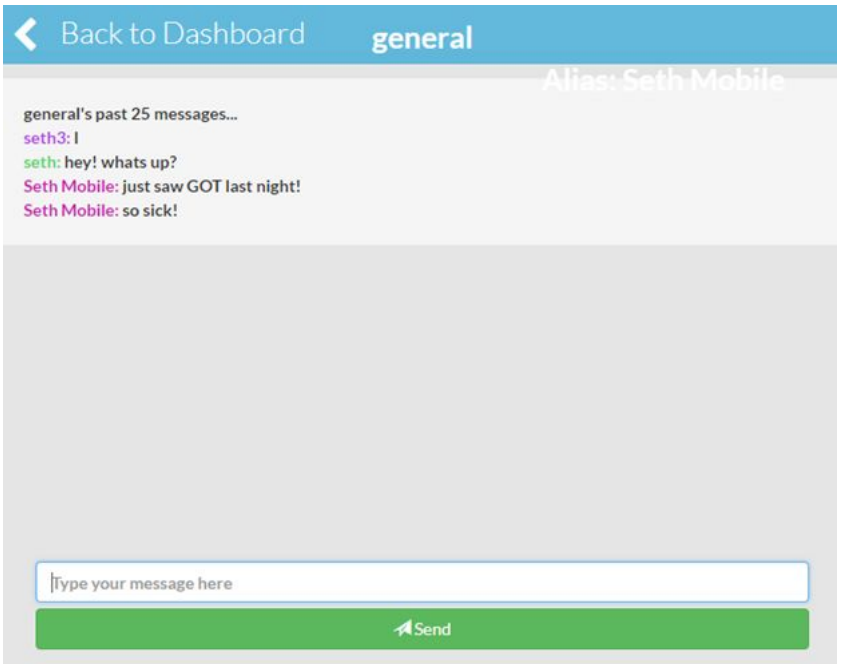
When making a room the users can enter any name they want, spaces are removed and capitals changed to lowercase. The new room is added to the dashboard Chatroom table and the the creator automatically joins it.



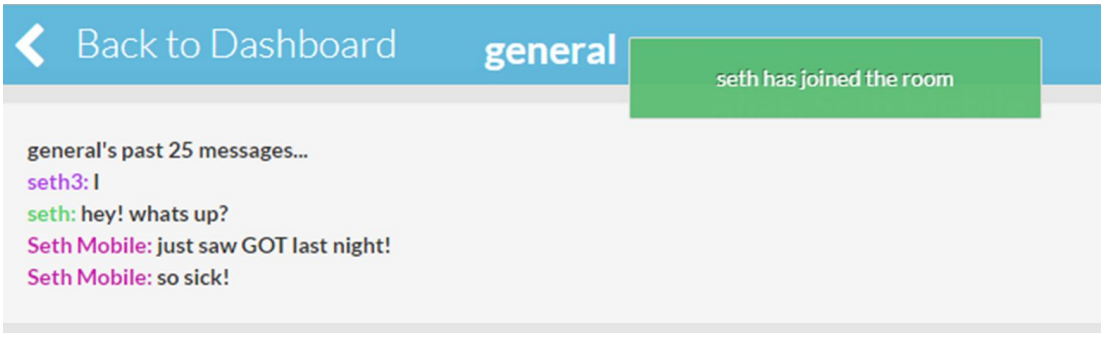
Chatting

When a user joins a new room without history they are alerted it is new and that there is no history yet. When the chat room already has some messages the past 25 are printed out on the new user's display (but not on the other users’). There is no need for a ‘display history’

command because all the messages are continuously stored in the scrollable webpage display.



When sending messages each user's alias is displayed and assigned a random color to help keep the chat organized and quickly discern between different users.
Additionally, when a user joins a room a pop up notification appears and makes a pleasant sound to alert the current users to the new members arrival.



Interaction Diagram

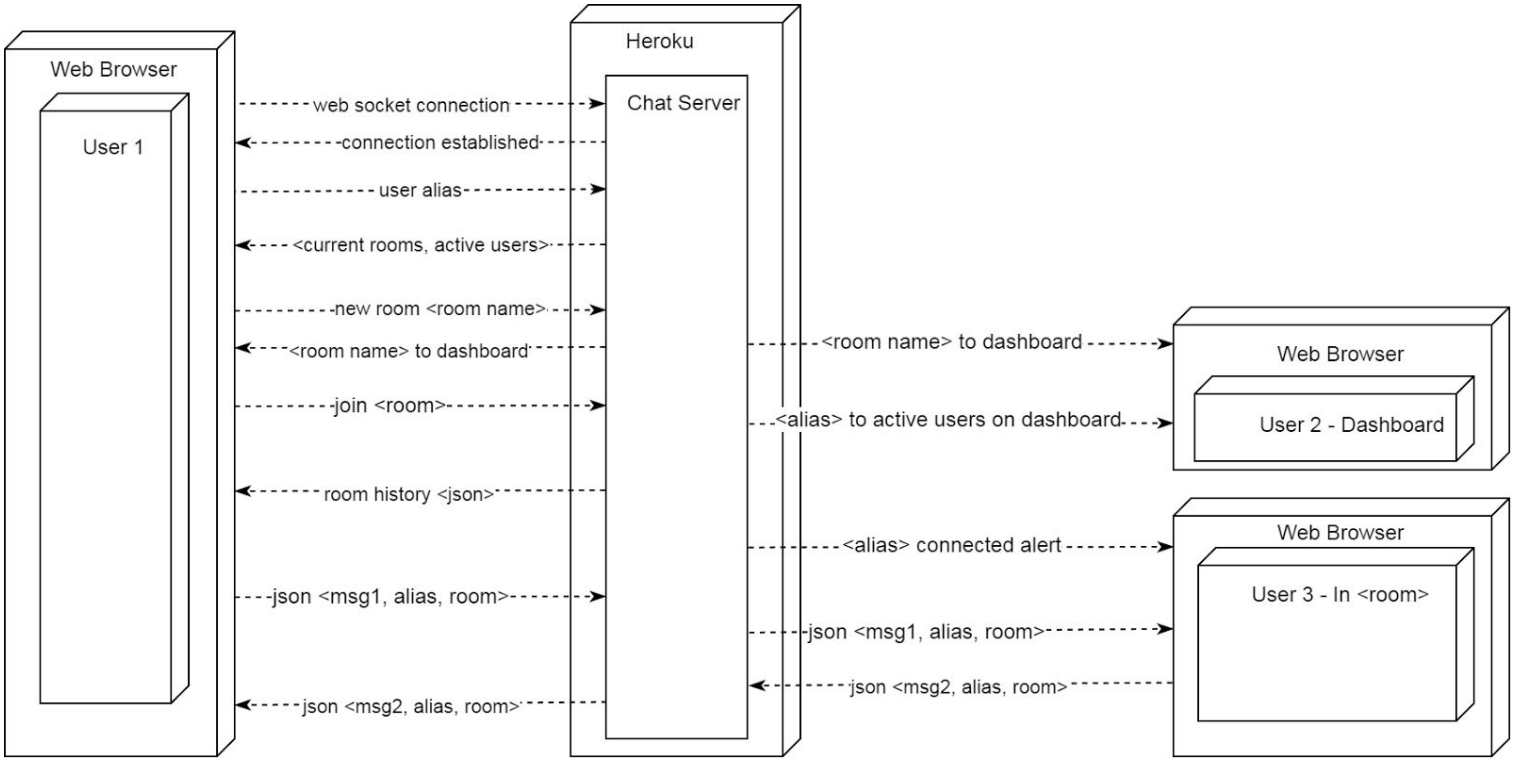


Figure 13: The server side of Hissper is hosted on Heroku. Chatapp.py has many dependencies. It sits on and Flask and Eventlet and the Heroku server interfaces with them using the Python server Gunicorn. To use Hissper, users navigate to hissper.herokuapp.com. The server communicates with the user's' browser using http and Python websockets.

High Level Changes and Problems

During the designing phase at the beginning of the development of the web application, our group as a whole let feature creep hit hard. We considered most of all the requirements and went beyond; which led to many in many features being included in the initial design documents not being included in the final product; such as a fully persistent database, google authentication, and more. A full extensive list of the requirements are listed in the section below.

Getting Heroku deployed and working with our stack and code was one of the problems the group faced. Specifically, the web application does not function 100% using HTTPS because of the mixed content of using web sockets. Another issue we faced during the development phase was the clearing of the client cache to remove old code. Since the client browser caches the HTML, CSS and JavaScript, often it was believed the new code was being used, but in fact legacy code was still on the client. This caused much confusion during

debugging. Once the issue was discovered, the solution was a simple matter of clearing the cache after a fresh deploy.

In addition to the development phase, scheduling meetings became harder closer to the end of the school term due to the teams' examinations and final projects being due in a similar time frame. As a solution, the team collaborated online or had partial group meetings and filled in the member who couldn't make it at a later time.

Requirement Satisfaction

This section details the completion of each requirement. Requirements are organized by number and title. This section should be read with section ‘3.0 Requirements’ of Milestone 1 handy to allow cross referencing and provide full descriptions of each requirement.

Requirement	**Completed**	**Justification if No**
3.1 External Interfaces	No	A database was not used as there was no use for one after it was realized all the info that needed to be stored could easily be held on the Python server. Hissper uses Heroku as its main external service now.
3.2 Functional Requirements		
3.2.1 Registration	No	There was not time or the need to implement a full registration and authentication for users. This was realized during Milestone 2 and further rescoped in the week of July 24th while implementing the front end landing pages.
3.2.2 Login	Partially	While there is no authentication, you do “log in” with an alias.
3.2.2.4 Valid Characters	Yes	
3.2.3 Instant Messaging	Partially	
3.2.3.1 Instant Messaging Platform	Yes	
3.2.3.2 Real Time Communication	Yes	
3.2.3.3 Error State on Failed Delivery	No	During the architecture design phase of Milestone 2 this feature was removed because it is not necessary for the system.
3.2.3.4 Valid Characters	Yes	
3.2.4 Persistent Messaging	Partially	During the architecture

		design phase of Milestone 2 this feature was removed as it would require the use of an external database to permanently store all messages. Instead, the 25 most recent messages of each active room are stored.
3.2.5 Chatrooms	Partially	
3.2.5.1 Chatroom Functionality	Yes	
3.2.5.2 Chatroom Ownership	No	Any user can delete a chat room at this point in the systems development. This issue was discovered in the week of July 31st while testing and making fixes. There was not time to fix this issue but it was noted as one of the areas that need work the most.
3.2.5.3 Error Behaviour Definition	Yes	
3.2.6 Aliases	Partially	
3.2.6.1 Setting Unique Alias	Partially	Users can set their alias and their alias is how they are identified by other users. However, during the week of July 17th while implementing chat rooms and the front end unique aliases were removed because there was not enough time and they were not strictly required
3.2.6.2 Valid Characters	Yes	
3.2.6.3 Modification of Alias	Yes	
3.2.6.4 Error Behaviour	No	See 3.2.6.1
3.2.7 Blocking and Unblocking	No	All the blocking and unblocking functionality was removed in the week of July 10th while implementing a basic chat system. It was decided that this would be

		too much work and was not worth spending time on.
3.2.8 Logging Out	No	The team decided to remove this functionality while designing the architecture of the system in Milestone 2.
3.3 Non Functional Requirements		
3.3.1 Desktop Based	Exceeded	While deploying the basic chat system to Heroku the team realized that Hissper would be functional for mobile phones as well as desktop computers.
3.3.2 Available	Yes	Currently the system is available whenever Heroku is available. Only time will tell if there are catastrophic bugs or deployment issues that will affect availability. Heroku has a track record of excellent availability.
3.3.3 Security	No	In the week of July 10th while deploying to Heroku and creating the base app the team decided not to implement security features. The only personal information that is stored is the 25 most recent messages in a room but there are no security measures in place to protect this information because any user can enter any room.
3.3.4 Users	Exceeded	A full stress test has not been done but Hissper can easily support many more than 20 users.
4.0 Stretch Goals		
4.2 1000 Concurrent Users	Partially	See 3.3.4
4.5 GUI	Yes	

Recommendations / Known Bugs

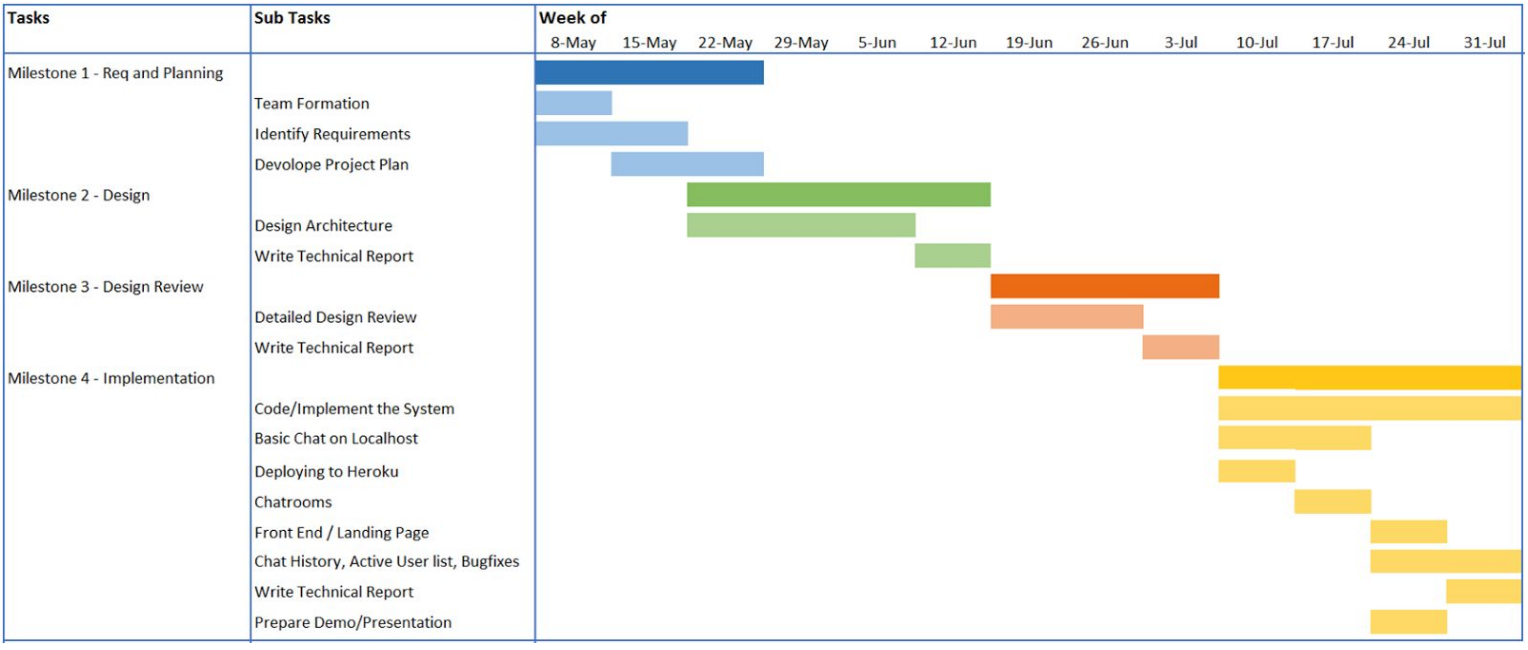
A number of known bugs have been raised and is documented below, they are listed in order of priority from most severe to least severe. It is recommended that if this project is to be pursued in the future, these bugs should be removed before release.

1. The app is not accessible through HTTPS, only HTTP is supported. HTTPS causes multiple failures due to mixed content.
2. The dashboard and chat rooms are accessible through direct use of the URL, which will lead to unexpected behaviour due to the alias not being set.
3. Any user can delete a chat room, only the creator should be able to delete a chat room.
4. Leave notifications are not working for when a user leaves a room.
5. Close notifications are not working for when the room is closed.
6. Implement password protected rooms.

Design Process Used

This project was completed based on a Waterfall model because that is how the course milestones structured it. Despite this, the actual building of the system followed a more agile approach with multiple short sprints to add different features. Development consisted of several revisions and additions to Hissper. The Gantt chart below shows the workflow of the entire project and breaks down the work done for Milestone 4 - Implementation
(See Figure 14).

Figure 14: Gantt Chart describing waterfall model.



Initially, a basic chat system with one room was implemented by Aman. This phase involved bringing together Flask, Javascript, HTML, and Python to create a chat application that runs on the users localhost. This was the most frustrating and difficult phase.

Next, Jordan deployed the basic app on Heroku so it was live and could be accessed by anyone. This required Gunicorn and interfacing Hissper as it was with the requirements of Heroku for Heroku to host the app.

The next phase was implementing chat rooms which Andrew developed and the Front End / Landing page that Aman created. These sections together provided most of the functionality of Hissper and what was left was implementing many other small features, fixing bugs, and satisfying outstanding requirements. Chat history was added by Seth, active users and ability to delete rooms by Andrew, and multiple bugs fixed and found by Jordan. This phase was worked on by all four members.

The demo occurred after deploying to Heroku but before Chatrooms had been implemented. All members participated in the demo and Seth made the PowerPoint presentation.

The report was written by all members. Seth wrote the Purpose, User Experience Gallery, Deployment Diagram, Design Process Used, and Requirements sections. Aman wrote the Introduction, Overview, Architecture. Andrew wrote the Front-end Architecture and High-Level Changes and Problems sections. Jordan worked wrote about the backend and improved grammar and structure.

The actual implementation and writing of code did not happen on schedule with the original plan. The team originally wanted to begin development in the week of June 19th but they procrastinated for three weeks before beginning to work on it and a week after that before meeting.

Conclusion

In total, Hissper exceeded the minimum requirements outlined by the course instructor, but fell short of all of the goals laid out by the team in the initial design document. The team feels that the product produced is very solid, and forms an excellent prototype which could be built upon in the future.

References

http://flask.pocoo.org/docs/0.12/
https://flask-socketio.readthedocs.io/en/latest/
http://pythonhow.com/building-a-website-with-python-flask/
https://www.tutorialspoint.com/flask/flask_overview.htm