**Team TBA**

**Dr. Deb McAvoy**

**Russ College Chatbot**

**Software Design Document**

Name (s):

Jon Feige

Joshua Wright

Michael Dinger

Logan Gordon

Date: (4/13/2020)

**TABLE OF CONTENTS**

[1. INTRODUCTION 2](#_Toc4987)

[1.1 Purpose 2](#_Toc4988)

[1.2 Scope 2](#_Toc4989)

[1.3 Overview 2](#_Toc4990)

[1.4 Reference Material 2](#_Toc4991)

[2. SYSTEM OVERVIEW 2](#_Toc4993)

[2.1 Teachnical Stack 2](#_Toc4992)

[3. SYSTEM ARCHITECTURE 3](#_Toc4994)

[3.1 Architectural Design 3](#_Toc4995)

[3.2 Decomposition Description 4](#_Toc4996)

[3.3 Design Rationale 5](#_Toc4997)

[4. DATA DESIGN 6](#_Toc4998)

[4.1 Data Description 6](#_Toc4999)

[4.2 Data Dictionary 6](#_Toc5000)

[5. COMPONENT DESIGN 7](#_Toc5001)

[6. HUMAN INTERFACE DESIGN 7](#_Toc5002)

[6.1 Overview of User Interface 7](#_Toc5003)

[6.2 Screen Images 8](#_Toc5004)

[6.3 Screen Objects and Actions 8](#_Toc5005)

[7. REQUIREMENTS MATRIX 9](#_Toc5006)

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Date** | **Reason For change** | **Version** |
| Original | 10/09/19 | Original Copy | 1.0 |
| Reformatted Original | 10/21/19 | Reformatted to properly align with specifications | 1.1 |
| Semester 2 Updates | 1/17/20 | Updates for changers over the course of last semester | 2.0 |
| Spring Final Update | 4/13/20 | Updates for spring final | 3.0 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# INTRODUCTION

## Purpose

This software design document describes how to implement a chatbot that can answer questions about the Russ College on the Ohio University Website for incoming students.

## Scope

The goal of this project is to create a chatbot using Python, JavaScript, CSS, and HTML that can accurately respond to a user’s questions based on a built-in database of common questions and answers and set up this package on a website where it can easily be accessed.

## Overview

This document will go over the specifics of our design in implementing the specified functionality. The project was requested by Ohio University for the benefit of incoming students.

## Reference Material

Chatterbot libraries were used. Its documentation can be found here: https://chatterbot.readthedocs.io/en/stable/index.html

# SYSTEM OVERVIEW

A button can be found in the lower right corner on an Ohio University webpage that, when clicked, will expand into a texting window where the user can ask questions and receive answers from the chatbot. Any questions the chatbot fails to answer will prompt the user to contact a human who can better answer that question.

## Technical Stack

* Python
* JavaScript
* NLTK
* HTML5/CSS Client dependent option
* AWS Cloud9
* Ohio University Website

Python is chosen due to the access it gives to the NLTK (natural language tool-kit) and Chatterbot libraries. These form the base of the chatbot’s language comprehension and confidence mapping. JavaScript will be used to create scripts for the dynamic portion of the website interface. The website will likely use a mixture of HTML and CSS, but that portion of the project is dependent on the client’s current website which we will be given further clarification on at a later date. Finally, the AWS Cloud9 software suite will be used for testing and creation of test cases.

# SYSTEM ARCHITECTURE

## Architectural Design

The Russ Rufus Chatbot will be rooted in a machine-learning based Python library called Chatterbot. This library will handle all natural-language processing to train the chatbot on our training data. In response the bot will learn which phrases to return to the user while returning a confidence level between 0 and 1. This library has the following dependencies:

* NLTK
* Chatterbot-corpus
* SQL-derivative
* Pandas
* Tk (Tkinter)
* Eel
* Requests
* Bs4
* Selenium

This Python program will serve as the input handler to the chatbot object or component on the Russ College recruitment website. Text will be received as input to a messaging window on the webpage and this text will be sent to the server hosting the chatbot. By calling the chatbot script with HTML/CSS/JS, the chatbot will be triggered to return a learned response with at least 80% confidence. Training data of inputs and outputs for the chatbot will be stored in .yml files. On the occasion that our Python script fails to return a response with the specified confidence an email-initializing script will run. This script serves as a secondary method of communicating with the user. If the user has a specific question or would like to carry-on a conversation with another human, the user will be prompted to input an email for use by our client.

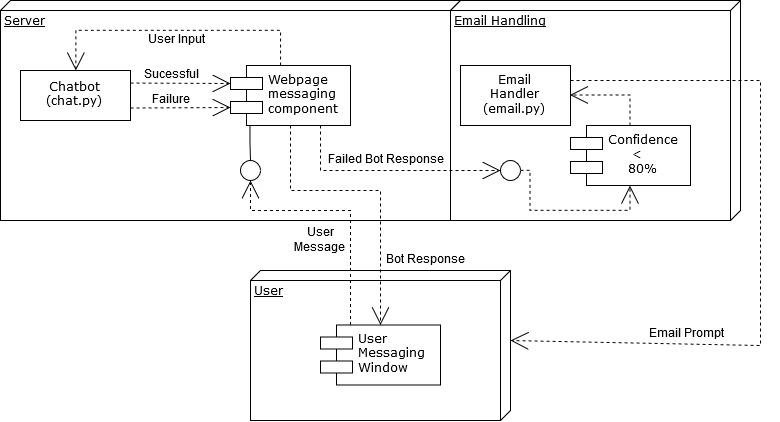


Figure 3.: Chatbot system architecture mapped from client to server and response paths back to the user

It starts at the Webpage messaging window. The user enters a message and it gets sent to the webpage messaging component where the input is displayed on-screen as it also gets sent to chat.py for processing. It results in either a success of failure. A success gives a response to the webpage messaging component to be displayed on-screen and awaits further response from the user to cycle again. If it resulted in failure, the confidence level was less than 80% and the email handler prompts the user to enter their information to get further help.

## Decomposition Description

Decomposition of the web component: Starting with the user, their input is sent to the web server through a JavaScript messaging component which display the conversation between user and chatbot. A loop between the user, the web component handling text input and output, and the chatbot are the endpoints for all data in this architecture. The web component drives conversation by waiting for user input and delivering it the chatbot and vice-versa. The component will be implemented in the web page to run on the user’s machine. It also serves as the go-between for the back end and front-end. This system mimics the model-view-controller design with the model being the container for the user input and bot responses, a JavaScript view component to display the conversation to the user, and the controller being the JavaScript calling the chatbot the formatted input.

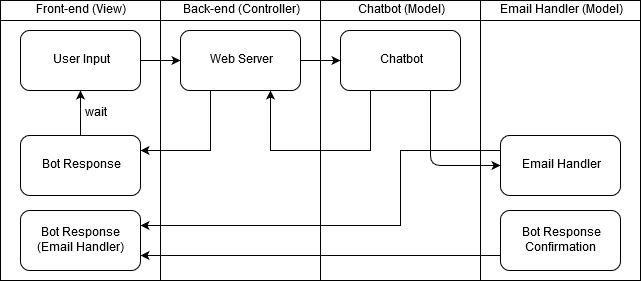


Figure 3.2: Decomposed view of the Russ Rufus Chatbot Architecture

The user input is sent to the webserver where it is sent to the chatbot. If the chatbot fails to pick a response it goes to the email handler and the bot responds to get the user more help. If the chatbot succeeds it gives the response to the webserver which the bot response will display, waiting for the next user input.

Decomposition of Chatbot:

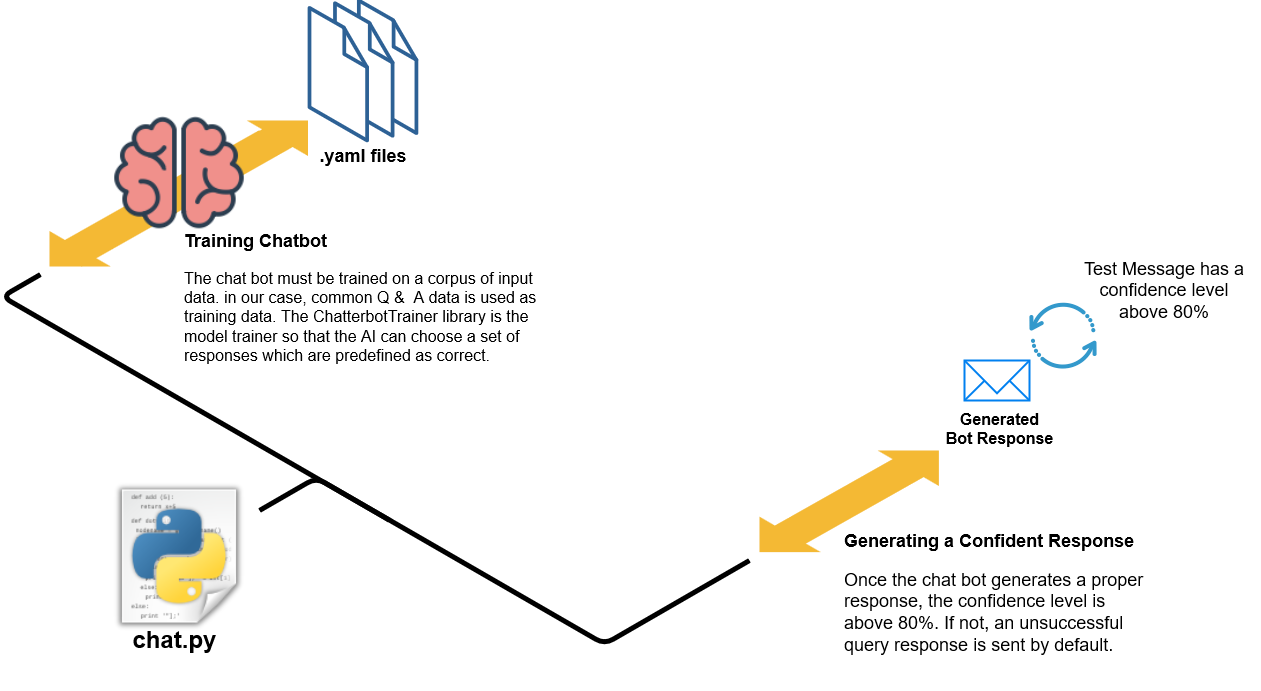


Figure 3.3: Decomposed view of the chat.py functionality

The chatbot is trained on the provided .yaml files. This training will allow it to generate a confident response from the input.

The chatterbot driven program, chat.py, is the response generator. The bot runs in the background waiting for input from the server. Upon running the program, a model for deriving a response is trained based on the input (.yml files) that we specify. After this initial training, the bot can be run indefinitely waiting for input and delivering response via the JavaScript component in real-time. Since it will sit with the website there is little latency in response time.

Decomposition of Messaging Component:

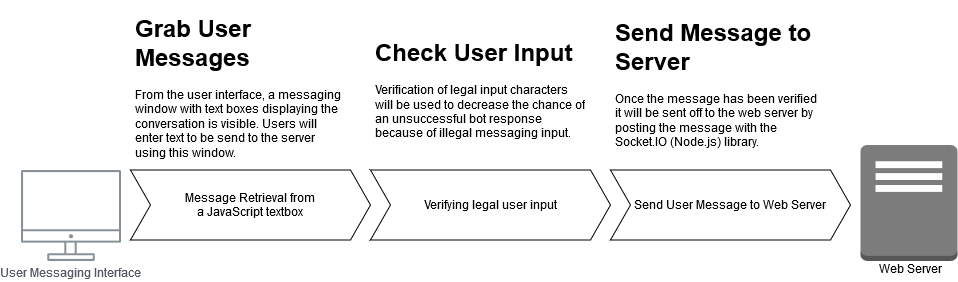


Figure 3.4: Decomposed view of the JavaScript messaging component

The message is retrieved from the user messaging interface and checked to see it’s valid. If so it is sent off to the web server using the Socket.IO library.

The messaging component retrieves and sends the user input from the webpage to the server. After verifying a valid message, JavaScript code will send the message using the Socket.IO library. After input is sent, the JavaScript messaging component will wait until a response is given by the chat.py file sitting on the web server. This component utilizes JavaScript, Socket.IO, and HTML/CSS to handle the processing of front-end user interaction.

Decomposition of Email Handler:

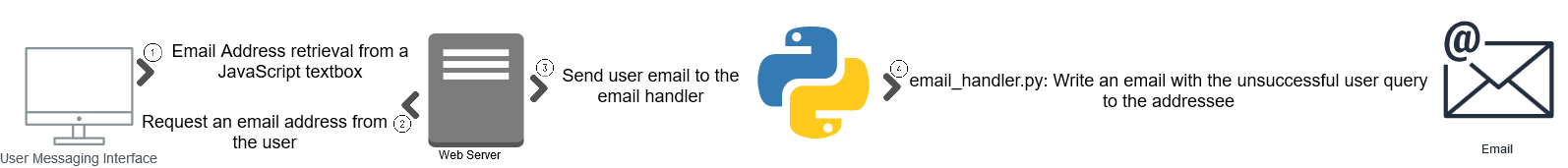


Figure 3.5: Decomposed view of the email handler

The email is retrieved using JavaScript and sent to the webserver. This input is handed off to the email handler and sets the user up to send an email to the appropriate recipient that can help them.

The functionality of the email handling program needs more requirements from the client before a design can be drawn out. The main functions it will serve will be the following:

* Conversational acquisition of email or email registration
* Initialization of the user’s default email program
* Conversationally confirm the email has been sent to our client

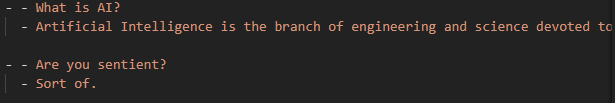
## Design Rationale

The chatterbot library allows us to house our chatbot on the server rather than in cloud storage as many technologies now require. The calls then are now local to the server and have the effect of reducing latency, shortening the bot re-training and re-deployment time, as well as saving money from the cost of cloud-storage. Integration with the webpage can be more flexible as well since the technology is not hidden from developers as other chatbot technology alternatives are such as AWS Lex, Microsoft Azure Bot, or Google’s Dialogflow. These technologies remove the machine-learning technicalities from the developers view and only require configuring the bot and serving it training data the same as chatterbot. The difference is price and storage. They are all cloud-based technologies which have a payment model based on use. The more use our chatterbot-based solution receives will not increase the cost of the chatbot.

# DATA DESIGN

## Data Description

The chatbot takes .yml documents formatted in such a way that each call and response is separated by either “- -“ or “-“ This is shown in the figure below.



As you can see a user call is denoted with “- -“ and a bot response is denoted with “-“. Once loaded into the bot using the built-in call “trainer.train(‘File’)”, the bot will have all data loaded and ready to respond based on a given user call. When all files are loaded the, the chatbot builds its own database denoted “db.splite3”.

## Data Dictionary

Ohio University Q/A Catalog

English\_Corpus.yml

Ai.yml

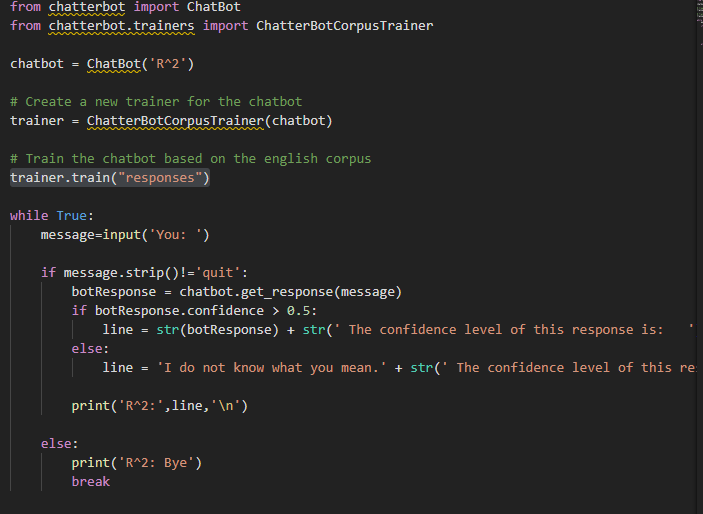
Jokes.yml

Trvial.yml

Db.sqlite3 -Chatbot created database file

# COMPONENT DESIGN

The way this bot will function is that you immediately call for the trainer, this trainer will give the bot all call responses. Then in the while loop, while the user does not request to quit it will constantly take user input. For the chatbot to respond you run it through the pre-built function chatbot.get\_response. This function takes the user input and attempts to pattern match the existing database. This returns a class with a bot-response and a bot confidence. We use the confidence to better tune the chatbot. This functions as a correctness filter. If it finds a close response, it will respond, and if not, it will respond with an “I don’t know” statement.



# HUMAN INTERFACE DESIGN

## Overview of User Interface

Opening the chatbot prompts for their full name and email to log into the chatbot initially. From here the user inputs their questions and the chatbot attempts to match the question closely to one in its database then output the corresponding answer from the database. The chatbot is set to send an automated email to a staff member if a corresponding question cannot be found so that all questions can be answered as quickly as possible.

## 

## Screen Images

## Figure 1 Figure 2 Figure 3

## A close up of a logo Description automatically generatedA screenshot of a cell phone Description automatically generated

## Screen Objects and Actions

A circle with the label R2 seen in Figure 1 will overlay the website in the bottom right corner. This circle may have a small pop-up attached to it and is able to be clicked. Selecting this circle opens the text box seen in Figure 2. Logging in with name and email as prompted will change the layout of the bot to that seen in Figure 3. The user will be prompted and can input questions in the bottom text field. Answers and previous questions will be displayed in a fashion similar to text messages on a cellular phone.

# REQUIREMENTS MATRIX

|  |  |
| --- | --- |
| Component | Functional Requirements Satisfied |
| Database Component | 4.1.3 REQ-1  4.1.3 REQ-2  4.1.3 REQ-3 |
| U.I Component | 4.2.3 REQ-1  4.2.3 REQ-2 |
| Email Component | 4.3.3 REQ-1  4.3.3 REQ-2 |
| Website Integration Component | 4.2.3 REQ-3  4.2.3 REQ-4 |
| Filter Component | 4.3.3 REQ-3  4.3.3 REQ-4 |