# Technical Specification

## Public Transport Chatbot

## 3rd Year Project

## Michael O'Hara 16414554

## Matthew Nolan 16425716

## Supervisor: Prof. Gareth Jones

# Table of Contents

## 1.Introduction

1.1 Overview

1.2 Glossary

## 2. System Architecture

## 3. High Level Design

3.1 Old System Diagram

## 4. Problems and Resolution

4.1 Bus API

4.2 Hosting the chatbot

4.3 Responding to the user

## 5. Installation Guide

5.1 Adding the contact

5.2 Joining the sandbox

# 1. Introduction

## 1.1 Overview

Our project is a chatbot service that is based on the popular social platform WhatsApp. As a majority of people already use WhatsApp, we felt it was a good idea to integrate it with WhatsApp rather than develop our own app. We concluded that if it’s built into an app that people already use, they would be more open to using it. It is designed to minimise the stress around getting info on public transport services, the service allows users to find when the next train is from whatever station that the user is at. This eliminates the need to go to the required public transport website and rolls all the services into one platform.

We used a service known as Dialogflow to handle the content of the responses to small talk interactions and other interactions such as greetings. This means that we could handle a large number of interactions without having to write an unnecessary amount of if/else statements and functions and making the code very cluttered. Dialogflow also handles entities which will be passed as parameters to our backend to gather the needed information.

We interact with multiple services and API in order to gather the required information and relay it back to the user. We do this in such a way that it is easy for the user to do and would not add any unnecessary steps to the user’s routine. Our backend interacts with the real time API provided by either Dublin Bus and Irish Rail and returns requested info to the user in the form of a text on WhatsApp through a service known as Twilio.

We also implemented a Google Search feature meaning that if the chatbot doesn’t know what the user means it would offer search google for it and return a number of links to articles that relate to the input given by the user. With the way the chatbot is set up to use machine learning meaning if more users search for the same thing the bot will be able to return the results faster.

## 1.2 Glossary

API is an Application Programming Interface a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.

Twilio

Dialogflow is the chatbot service we use to handle the response that will be sent to the user

Flask

Json

XML

# 2. System Architecture

**A close up of a device

Description automatically generated**

**API’s**

The APIs are used in order to gather the data about public transport services, they take in the station and can return all the train, in XML format, that are going to serve that station in the next 90 minutes. The XML returned is comprised of twenty-one pieces of information for each train. This contains information relating such as train destination, origin, Estimated time arrivals and departure.

**Flask**

Flask is a python library for web development. We use it to handle the json requests to and from Dialogflow.

**PythonAnywhere**

PythonAnywhere is hosting service tailored towards web development in python. Which is free to use and has its own virtual environment allowing for the installation of modules required for the web app.

**Dialogflow**

Dialogflow is used to handle the content of replies to the user. It allowed us to eliminate the need to clutter the code with inefficient if and else statements to handle interactions. This allowed us to be able to spend more time focusing on getting the information from the APIs back to the user in an efficient manner. Dialogflow can also handle entities which can be defined by us i.e. the stations and then pass those identities as parameters to our backend in order to use them when calling on the API.

**Twilio**

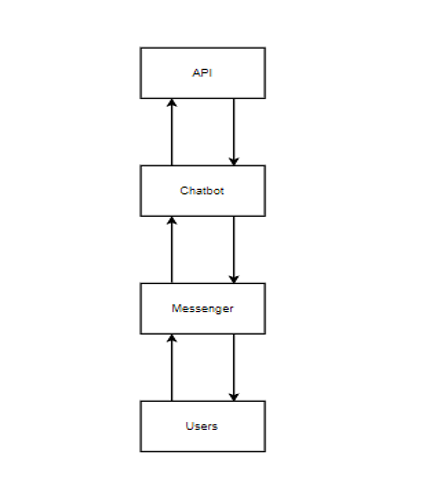
Twilio is a service which allows for the sending and receiving messages to and from the user. This allows the user to send the messages to Twilio which then forwards it to Dialogflow. Dialogflow will then populate a response and send it back to Twilio and back to the user. It also allows for multiple users to connect to the same sandbox without clashing. Twilio also allows our project members to see logs of each message its status to see if any error occur and why they occurred.

**WhatsApp**

WhatsApp is the social messaging platform which the user sends the messages on and receives the replies from our bot on. The messages that are sent get sent to Twilio and the responses are sent back by Twilio to the user.

# 3. High-Level Design

Figure 3.1



As you can see from figure 3.1, we thought that the system would be fairly simply and straight forward but as we soon found out it was going to be more complex. The first big change that was made was the move to WhatsApp instead of Facebook Messenger. This was due to the larger audience and the fact that most people already have WhatsApp installed on their smartphone. For reference, according to an article on businessofapps.com 1.5 billion users in 180 countries makes WhatsApp the most-popular messaging app in the world with 0.2 billion more than stablemate Facebook Messenger. From this statement alone it was a clear choice to use WhatsApp.

You will also see that we assumed the chatbot would be self contained where as early tests of the bot showed that we needed somewhere to host the bot in order for it to be accessible 24/7.This meant changing our system to integrate with a hosting service, the one of choice being PythonAnywhere. We migrated our flask web app, that we were able get running successfully on the local host to our PythonAnywhere environment. PythonAnywhere was perfect for this due to its ability to “pip install” modules such as flask for the web app and requests for the HTTP requests. This allowed us to have the chatbot active and meant we could make changes and push them to the PythonAnywhere and deploy them seamlessly. This seamlessness allowed for more fluid user testing as we could update without having to restart the bot or take the bot offline.

By the end of the development cycle we had a few major changes to the initial idea. But through trial and error and discussion with our supervisor for guidance we quickly locked in an idea we were happy with. The final idea also gave us a good learning opportunity in relation to topics such as hosting web apps and integrating web hooks into that web app in order to link it to another service. We also got good experience with making a chatbot and the work that goes into to making sure that the bot is intelligent enough to reply to certain queries. We also got good experience using APIs and http requests in order to get information we need, then we learned how to process that info and relay it back in an effective manner.

As we will discuss later on in the problems and resolution section, we also learned the hard way how external factors can have a drastic effect on a project. With the rolling back of the Dublin Bus API it meant we had to shift the focus of the project. Due to the nature of the roll back we couldn’t know if the API would come back in the duration of the project. So, we had to shift focus and not sit around hoping for it to come back. This was a difference from the original idea as we had looked into the bus and it seemed like a very viable option of an API to use.

# 4. Problems and Resolution

### 4.1 Bus API

From early on in the project we began to run into problems. We started off development by trying to get information from the Dublin Bus real time API, which at first was going well but then was halted when the API was taken down. We contacted the support team of the Dublin Bus API wondering if we could get some info as to why the API was turned off. We received word back that the servers were getting too many requests per hour to be able handle meaning they had to restrict access. This restriction means that access to the API is limited to official Dublin Bus channels such as the Dublin Bus app or website. This was a major setback as it ruled out the bus indefinitely as we had no idea when the Bus API would be made available to the public if at all. Since we required the API and it is controlled by external factors, we had no way to rectify this problem meaning it was not implemented. However, if in the future access to the API were to be made public again the code could be implemented to the chatbot.

### 4.2 Hosting the chatbot

Another problem we ran into early on was the fact that we needed to host the flask app somewhere in order to have the chatbot running at all times. At the beginning we were hosting it on localhost and using ngrok to allow the bot to tunnel in. Very quickly this method would prove to be inefficient. As without the bot being hosted somewhere, we would need to start it locally on one of our machines whenever a user wants to use it in order for the bot to be able to respond.

We looked at a number of places to host the service with the main two services we looked at being Heroku.com and pythonanywhere.com. Initially we looked at Heroku and found that it wouldn’t be able to cater for our needs as it wasn’t geared towards web dev in python.

We then found pyhtonanywhere.com and this was a more viable option as it was based around web development in python. Hosting on PythonAnywhere meant creating a virtual environment and install modules need for our app to run such as flask and requests (to handle json) and google search. Another upside was that the site was free to use as long as you log into your account to once every 3 months, this means that we could keep the chatbot running indefinitely without having to have it running on our machines. This was one of the major problems resolved and also allowed us to have users test it and let us know if it was working.

### 4.3 Responding to the user

Another slight problem we had was that because Twilio integration with WhatsApp is still in beta it means that we needed a service to allow us to reply to the user. We used Dialogflow in order to handle the content that is in the replies. This is done by web hooking Dialogflow to Twilio. This allows Dialogflow to create the response and Twilio to relay the response to the user. It will also allow us also to have Twilio send the response from the user to Dialogflow to be processed and if needed t call the back end to get info requested from the API.

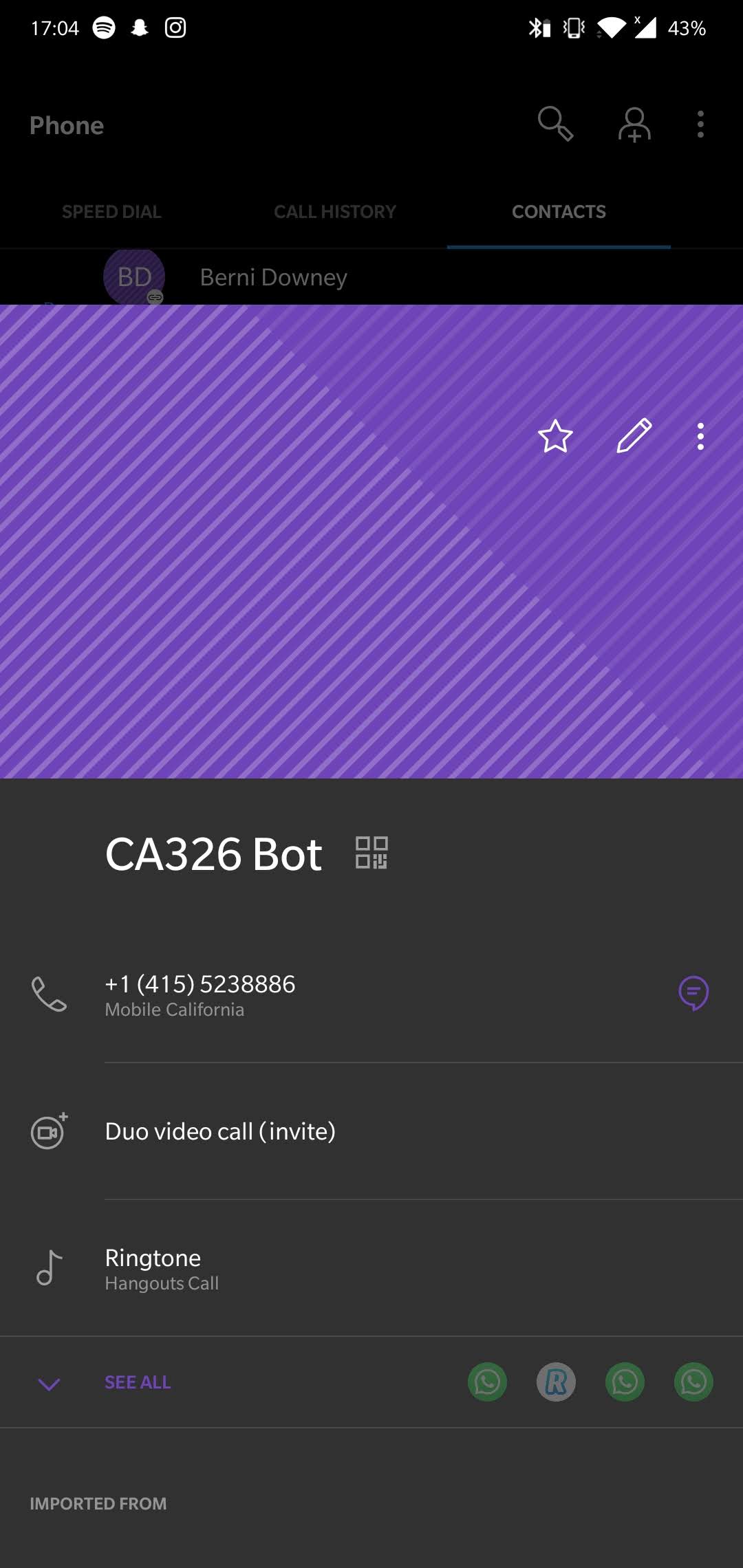
# 5. Installation Guide

The installation process is very straightforward as our chatbot service does not require the installation of any other services by the user as the chat bot is self-contained hosted on PythonAnywhere and is all online.

The user must have WhatsApp installed on their phone and set up with their phone number. This should take no more than 5 minutes.

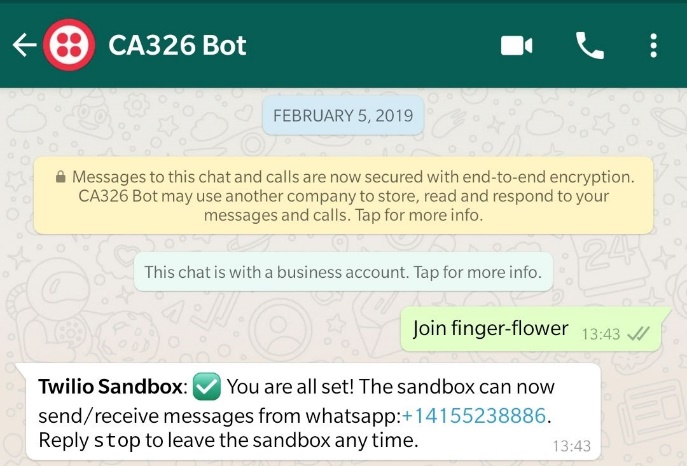
### 5.1 Adding the contact

Once that is done the only steps required by the user is to add the bots contact details to their phonebook. The name of the bot can be put in as anything (CA326 Bot for example as shown below) but the number must be the one shown below ( +1 (415) 523 8886).



### 5.2 Joining the sandbox

Once the user has WhatsApp set and the contact added to their phone then the user has to simply open a chat with the contact on WhatsApp and type in the join phrase in order to join the WhatsApp sandbox that the chat bot operates out of. The sandbox phrase we are using is “join finger-flower”



This allows our sandbox to remember your number and thus be able to send/receive messages to and from the sandbox. And that’s all the steps the user needs to take in order to get access to the chatbot service. The bot will only reply when prompted first by the user meaning there is no need to worry about spam messages coming from it.