

Technical Guide

Performing Ecological Momentary Assessments (EMA) via
Picroft or Mobile App

CA400

Santiago de Arribas de Renedo • 16449272

Pedro Marques Canes • 16770821

Supervisor: Dr. Tomas Ward

Date Complete: 16/05/2020

Table of Contents

Table of Contents	2
Abstract	4
Overview	4
Glossary	5
Motivation	5
Research	6
Database	6
RESTful API	7
Mycroft	7
Android	8
Design	8
System Architecture	8
High Level Design	9
API and Database	11
Designing Mycroft Skills	12
Designing the Android App	13
Implementation	14
Building the Database	14
Deploying the API	15
Building Mycroft Skills	15
Building the Android App	16
Sample Code	17
API and Database	17
Mycroft	22
Android	26
Problems Solved	29
Initially Using Flask	29
Switching to Amazon Web Services (AWS)	29
Heroku	30
Pairing System	30
Developing Skills	30
Automated Question Check	30
GPS Tracking	31
Asking Questions	31
Android Background Operation Issues	31

Results	32
Successful Integration	32
Future Work	37
Machine Learning	38

Abstract

Ecological Momentary Assessments (EMA) are surveys conducted in real time to when they are most valuable. This can range from consumer testing to behaviour analysis. Electronic Patient Reported Outcomes (ePRO) is a method of data retrieval used in clinical trials to record patient information. Typically this is done after the clinical trial by using a series of questions set by researchers. However, this usually leads to recall bias where patients have trouble remembering accurate information to respond to the questions with. A system that could perform EMA or ePRO methods in real time would be useful in gathering much more accurate data. This is what our system (Daisy) does.

Overview

The Daisy system consists of a mobile app and a home assistant, in this instance - a picroft. There are three types of users that can use the system:

- **General User:** Asked general survey questions.
- **Clinical trial User:** Asked clinical trial related questions.
- **Researcher:** Sends questions to either the general or clinical trial user.

For a general or clinical trial user to use the system they must install the daisy mobile app on an android device, create an account, install the daisy skill and pair with the home assistant via a pair code issued by the app, and they must await questions to be asked on either device at any time deemed fit by a researcher.

The researcher also creates an account on the app but does not pair with any home assistant. They select users to ask questions to and send them questions with possible responses at any time they wish. The system uses *context detection* to decide which device to send the questions to. There are three different scenarios in which users can be asked questions:

Scenario	Device used:
User is near the home assistant and is alone.	Home Assistant
User is near the home assistant and has company.	Phone
User is away from the home assistant.	Phone

Glossary

<i>mycroft</i>	An open source home assistant.
<i>daisy</i>	The system that handles asking the questions and storing the answer via the mobile app and mycroft home assistant.
<i>mycroft skill</i>	A program that runs on the mycroft to perform tasks for the user e.g. Checking the weather.
<i>utterance</i>	Anything spoken by the user to Mycroft.
<i>dialog</i>	Anything returned to the user by Mycroft.
<i>android</i>	The platform on which the mobile application runs.
<i>marketplace</i>	A website where mycroft skills are shared.
<i>github</i>	A website where code is stored.
<i>raspberry pi</i>	A small single board computer.
<i>user</i>	A person that receives questions.
<i>researcher</i>	A person that sends questions and receives answers.

Motivation

One of the main things that drew us to this project was the challenge of trying to build a system that hadn't existed before. It would require learning some new concepts such as deploying and building API's, learning how to build a database and learning how to build a complete android app. It would require learning about a novel device - the mycroft - how to build skills for this home assistant and trying to understand how context detection could work. Most importantly, it would require trying to figure out what would be needed in the first place to create such a system and how to integrate all these different components together.

The challenges this project presented us with as well as the potential to build something that had useful real world applications was very inviting. We decided this was the type of project we would like to work on and hoped that over the course of trying to build this system we would learn a lot more than we had known previously.

Research

Understanding how to build a backend for this system proved to be one of the most challenging aspects of this project. We initially believed that the system architecture would be client-server and tried to deploy our own server using Flask. We knew that a RESTful Application Programming Interface (API) would be needed to communicate with the server but it became apparent that deploying Flask locally would not be useful for our purposes. We then looked into using Amazon Web Services (AWS) to provide this functionality for our server.

AWS provides various microservices such as database hosting and API Gateway which is used to manage API calls. Although we tried to build our API this way it became problematic to use and we switched back to using Flask. Progressing through these problems we learned about databases, APIs and function as a service infrastructure.

We had to look into various home assistant options and settled on using Mycroft. We looked into how to develop skills for this novel platform which were built in the Python programming language. Another major component to our system is our android application and developing this was something we had not done before so needed to look into it.

Database

We had learned about using databases in the course but had little experience with developing our own. There were issues with trying to understand where to host the database or what system to use. First we had to decide between a NoSQL database like Firebase or a relational database like MySQL, Postgres or Microsoft SQL. We looked at the reasons we needed a database and in the end chose to use Firebase for the android app and Postgres for the overall system.

The reason for using Firebase with the app is that it is difficult to communicate straight to an app from an external server. Thus Firebase is the middle-man between our API and our app. However we mentioned that Firebase is a NoSQL database and as such is non relational meaning it's difficult to connect entities to each other. Looking at the relational database options we chose Postgres because it's free and scales very well.

As we looked into using a relational database we looked at how to host it and the easiest option was to store it locally. However, although this may be useful for testing purposes it would not be ideal for our system. Thus, we looked into hosting options and AWS Relational Database System (RDS) proved to be a very feasible choice. This was due to its *free tier* option which allowed us to use a certain amount of space monthly for free. We could host a postgres database that would scale up accordingly with the amount of users that could potentially use the system.

RESTful API

We had no experience with designing APIs and this was something we had to look into developing. The first stage was understanding the different API types: REST and Soap. REST is used for sending data via HTTP protocols while Soap is only used for XML data. As such we knew we needed to use a RESTful API. We then needed to understand the different HTTP requests - GET, POST, PUT, and DELETE.

- **GET:** Requests data from a resource.
- **POST:** Send data to create a resource.
- **PUT:** Update a resource.
- **DELETE:** Delete a resource.

We had to figure out how to use these different requests to interact with our database and perform our different SQL queries - SELECT, INSERT, UPDATE and DELETE.

Our initial idea was to use a local Flask application as our server and this would act as our basic RESTful API. We were able to get it to deal with HTTP requests sent from another device on the same network but had trouble with communicating to this API outside of the local network. We then decided to look into other options and AWS became a strong contender for building our API. We could use AWS Lambda to perform our HTTP methods and AWS API Gateway as our management system for receiving our API calls. However, we had trouble connecting this to our postgres database despite it being hosted by AWS RDS. We then turned to a different option while keeping our database hosted on AWS.

We decided to try to deploy our Flask server again but from our AWS research we had discovered Heroku, another cloud platform. Heroku would enable us to deploy our Flask server and use it as an API. We could push the Flask files to Heroku using Git. Additional files such as a **Procfile** and a requirements file were needed to make sure Heroku understood how to run the Flask server. Once the API was deployed on Heroku we could communicate to our database from our two clients, Mycroft and the android app.

Mycroft

We understood that a major component in our system would be the use of a home assistant. However, we had no previous experience with using or developing programs for any home assistant devices. There are many different brands available on the market from the Amazon Echo to Google Home. There are also home assistant kits that enable you to build your own home assistant and use open source software. One of these open source home assistants was Mycroft and this was what we decided to use as our home assistant.

The main reason we chose Mycroft as our home assistant is because the components are available for cheap, we don't need custom parts only those that already exist such as a raspberry pi, a playstation eye and a pair of logitech speakers.

The software is also freely available and completely open source with a large community that could help us develop our skill. The Mycroft system is built using Python as are the skills and these are deployed on a linux OS.

Android

We believed that, in addition to the home-assistant, our system needed another efficient way to reach the users. We, therefore, decided upon using a mobile application to act as the middle ground interface of the system. After much deliberation, we chose to build an Android mobile application. This was due to the fact that both members of the team had no prior experience in mobile development, and that the android environment is very popular due to its vast range of documentation and resources online, specifically Android Studio. The app was developed mostly with Java, but also incorporated various other formats, such as XML.

Android offered us the possibility of using Firebase, specifically Firebase Firestore and Firebase Authentication, in order to build a swift and reliable users system. This way we could easily implement a registration system that properly split regular users and researchers.

We believed using an Android App as the main interface of the system would allow us to build a strong question answering system by acting as the link between users and researchers. In addition, we believed using the device's gps location and the microphone input would help enhance our context detection system which was the main core of the project.

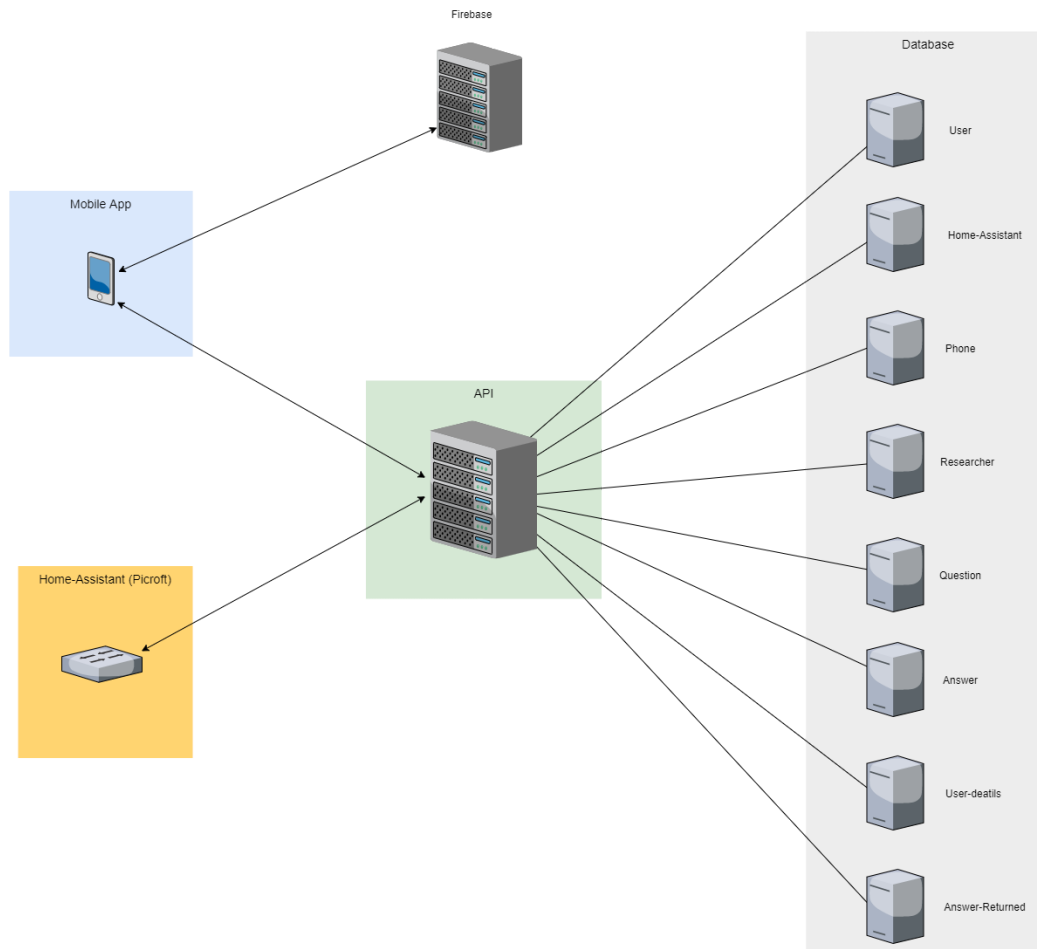
Android studio was the software used to develop most of the android application. This software was freely available and open source with an extremely large community, documentation and resources that helped us create this application.

Design

The design of this system contains many moving parts. The primary backend of the system is the API through which the Mycroft and app communicate to the database. When designing the system we wanted a server to perform our context detection. However, as we were designing the other components we realised we would not have time to implement that correctly and decided to use the home assistant to perform context detection. Thus, the user's home assistant performs the context detection to check what device to use and updates the database accordingly. Depending on which scenario the user is in, a different device is used.

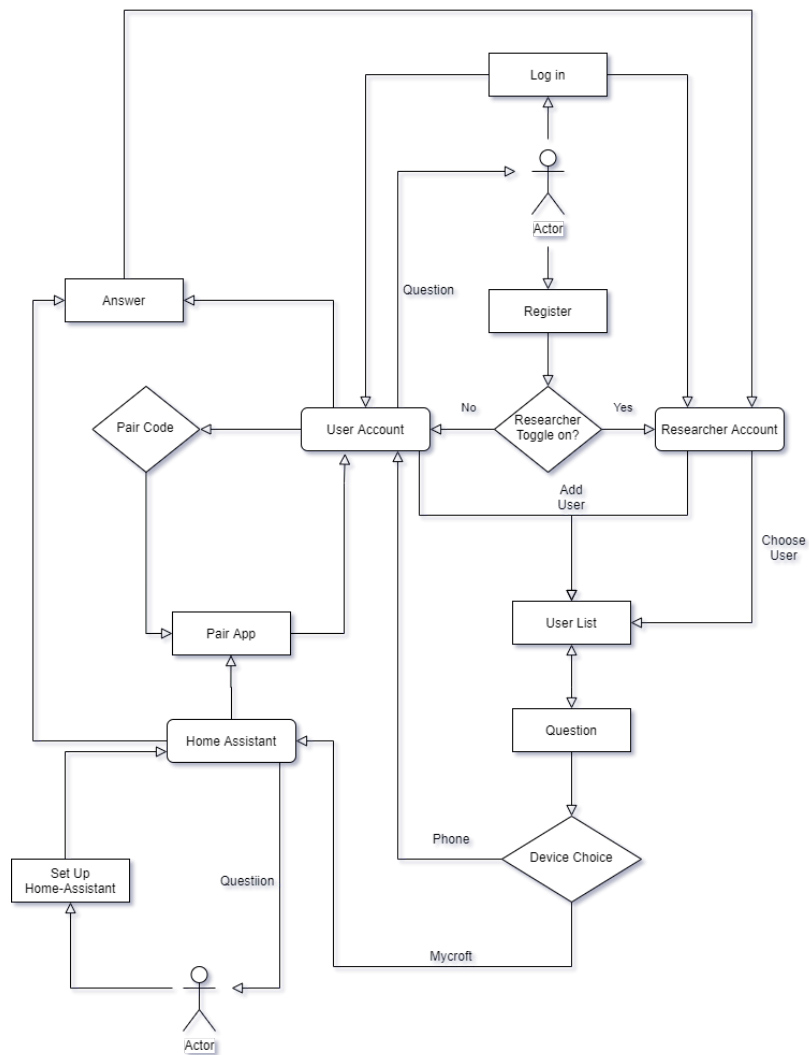
System Architecture

The system uses 4 major components and implements a client server architecture. The app and Mycroft communicate to the API which in turn sends information to the database. The following is a system architecture diagram:



High Level Design

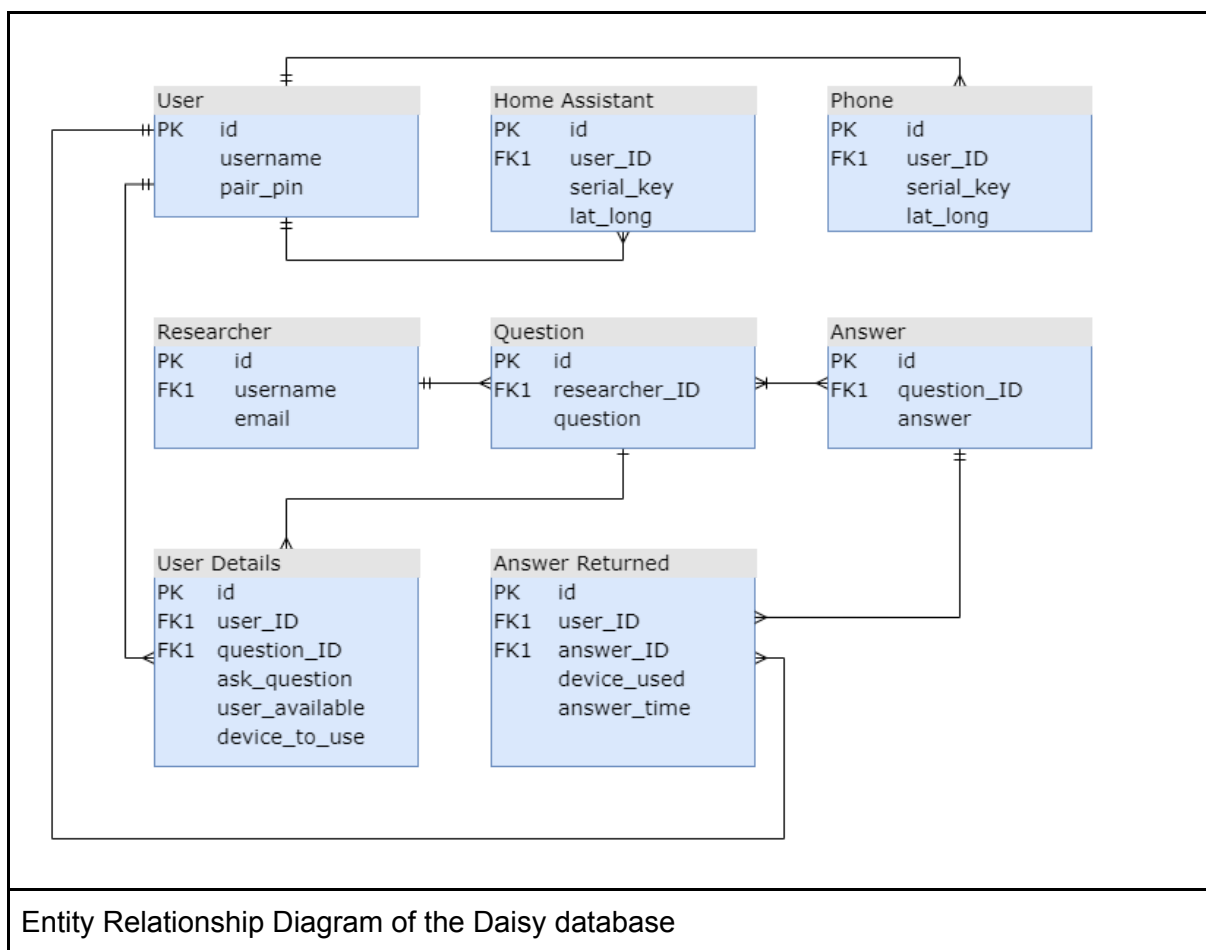
The following is a High Level Design of how the overall system functions:



API and Database

The database consists of 8 tables that can be communicated to via the use of our API. There are 8 resources, each for a table in the database. There's a GET, POST, PUT and DELETE method for every resource. However certain resources have additional GET methods to perform more complex SELECT queries where it makes more sense for example selecting a Home Assistant entity by user ID instead of by the home assistant ID. This is the same for the Phone entity as it makes it easier to get information from a particular user's devices for example GPS to be used for context detection.

The following is an Entity Relationship Diagram (EDR) which covers the relationship between tables in the database.



We knew designing the database would initially involve 6 main tables: **user**, **researcher**, **home assistant**, **phone**, **question** and **answer**. We needed to think about the relationship between these different entities and how to eliminate redundancy as much as possible. We were careful with selecting what foreign keys to use for each table since an addition of foreign keys would slow down the performance of database queries.

As we progressed through building the database we realised we needed an addition of more tables. In particular we needed a table to store the state of the user when they had been set questions to be asked. We also needed a table to record the answers as pointing to the **answer** table was not ideal since this table was only to store answer entities not the actual answer returned by a user. Thus we developed a **user details** and **answer returned** table.

The **user detail** table stores the values needed when the system needs to perform the context detection. The *user_ID* field allows us to check what users have questions that need to be asked. This is performed with a GET request to a /user/user-details/<user_ID> resource. What's returned is a list of rows as items where each item is a dictionary. We can check the *ask_question* field to see if any questions need to be asked. This is updated with a PUT request when the researcher sets a user's question to be asked. Specifically, the PUT request updates the *ask_question*, *device_to_use* and *user_available*. The *user_available* field relates to the microphone service performed by the app and whether the user is available for Mycroft to ask questions. For example if the microphone picks up that there are other people with the user, *user_available* is set to **False** and Mycroft is not used.

Once context detection has been performed by checking the users home assistant and phone GPS information as well as the *user_available* field, the *device_to_use* field is set to whichever device to use: **phone** or **home-assistant**. By default, when the PUT request initially sets *ask_question* to **True** and updates the *user_available* field accordingly, the *device_to_use* field is set to **null**. Once registered, the user's home assistant checks the **user details** table every minute to see if any questions need to be asked, performs context detection and updates *device_to_use* accordingly with *user_available* and *ask_question* set to **False**. The phone checks this table every 15 minutes and checks if *device_to_use* is set to **Phone**. If this is the case, the phone then asks the question to the user.

Once a question has been asked the **answer returned** table can be set with the user that returned a response, their response, the device used and time answered. Once the researcher has asked a question to a user the app checks this table every 15 minutes and if it sees that the user has answered it pulls their answer along with additional information to display to the researcher.

Designing Mycroft Skills

The Mycroft uses skills which perform various tasks for the user. These can range from checking the weather to allowing the user to play voice commanded games. Skills are available from the Mycroft marketplace and are built by an open source community which means anyone can build a skill. Building a skill was not something we'd had previous experience with so there was a learning curve with understanding the Mycroft environment. We also needed our skill to do things which would be outside the scope of the usual components it uses - the microphone and speakers, specifically, the GPS location tracking.

A skill is primarily built using one Python `__init__.py` file to identify the skill as a package as well as a few additional files to identify certain user utterances and use Mycroft dialogue.

An *utterance* is whatever the user says to Mycroft and *dialog* is whatever Mycroft says to the user. While the bulk of the users interaction with Mycroft would be contained in the `__init__.py` file we needed additional programs to check the GPS data from the raspberry pi and to start automatically asking questions and send those questions to Mycroft. These programs need to be stored in the skills package.

The GPS tracking uses a program, *gps.py*, from github by user **Lauszus** to retrieve GPS data from the raspberry pi GPS module. This module returns data in National Marine Electronics Association (NMEA) format which is difficult to process so the program formats the data to be more human readable. We designed an additional program, *get_gps.py*, to use *gps.py* and retrieve the latitude and longitude only. This data is then sent to the database to update the **home assistants** *lat_long* field to be used in the context detection. The home assistant *lat_long* GPS info is only updated once, when the user registers since it is necessary to constantly update the home assistants location as it is practically stationary once installed.

In order to perform context detection and ask questions we designed a *get_questions.py* program to check the status of the user's question, pull any details necessary to perform context detection. If the *device_to_use* was to be set to **phone**, this was done and no further action necessary. However, if the *device_to_use* is to be set to **home-assistant** then this is done and the question and answers to that question are pulled from the database to be formatted in JSON and stored in a *questions* file. The Mycroft Daisy skill is then notified that there is a new question and it checks the contents of this file for the data needed to ask the question and return the correct information to the database.

Designing the Android App

An android application is made primarily of different activities. An activity represents a single screen with a user interface, for example a window or frame of Java. All features in the app correspond to an activity in the code. For instance, the register activity will be responsible for creating a user. This activity will link to a specific register activity XML file which will be responsible for displaying the register screen and will be composed of all the different elements in the screen, such as the username and email fields and any buttons on the screen. The activity file will find these elements in the XML file and respond to any actions performed by the user, such as clicking a button or typing in a username. For this app, all the screens had a respective activity.

In addition to activities, a big part of the app was the use of services. Services behave similarly to activities, however they do not link to an XML file. A Service is an application component that can perform long-running operations in the background of the app. It doesn't provide a user interface and once started, it might continue running for some time, even after the user switches to another application. We believed using services for the GPS tracking and the Microphone input could be beneficial.

Another feature we wanted to implement in the app was a fully functional register/login system. Due to the lack of experience in android development, we opted to use Firebase.

Firebase is a mobile platform that is made up of complementary features that can be mix-and-matched to fit various needs, with Google Analytics for Firebase at the core. Firebase has large amounts of documentation which would be extremely helpful, especially since this was a learning experience. As we got more accustomed to Firebase, we started getting more creative and adventurous with it. In the end, most of the app's working flow incorporated Firebase in some way. The two Firebase features we used for this app were the Firebase Authentication and the Firebase Firestore. Firebase Authentication was used to create accounts and to make sure no two users shared the same email address. Firebase Firestore is a real time flexible NoSql database system. Being updated in real time allowed us to explore the location updates for the user in real time.

Another important feature we wanted to prioritize was the communication between the app and the home-assistant. This was done mostly through api calls to the database. Android does not originally support api communications, hence we had to use Retrofit 2, a type-safe HTTP client for Android and Java that turns a HTTP API into a Java interface. This allowed us to create an interface in Java that allows us to directly communicate with the API, and hence with the home-assistant.

Implementation

We knew we needed our backend to allow us to store user information. However, it was not necessary to have this completed before we could work on anything else so as the backend was being developed, the Daisy mobile app was also under development, primarily the user interface and connected to a Firebase NoSQL database. This type of database was fine for the app as it could be used to store user information but would not be useful in the long run as we needed a relational database to connect multiple entities for example users to their home assistant and phones. Thus was used a postgres database hosted on AWS.

Since the app and mycroft needed to be implemented separately and needed to function independently of each other the API and database had to be built first. Thus we needed to make sure we understood the control flow of the system so that we knew what information we needed to store to be used at the right time.

For example, we knew the microphone check for the context detection was going to take time to implement but we knew we needed it. Therefore, we had the *user_available* field already implemented in the database and used dummy data for testing before the actual functionality was implemented.

Building the Database

Implementing the database was a matter of using SQLAlchemy to build models for each of the tables and using Alembic (a SQLAlchemy migration tool) to update the postgres database whenever there was an edit to the models. We initially implemented each of the database models into one *create_database.py* file which, when run, would create the database that we wanted. We could use this as our initial database to test our API on.

If there were any issues with key constraints or errors from the database, running this program would let us know. This was done on a local postgres database.

We then needed to connect postgres to AWS and to allow external networks to access the database. This is because AWS services can work within their own environment on a virtual network but in this case we needed our API to be hosted elsewhere and to access our database on AWS. Once our database instance was set up on AWS and we allowed external connections, we created a new server on our postgres pgAdmin dashboard - this is the user interface for postgres - and added our Amazon instance endpoint. Our database was now ready to be used by our API.

Deploying the API

We had used one file for creating our database but for maintaining our different tables and using multiple resources to access each table we needed to decouple everything. This would make it much easier to debug specific table/ resource issues and make the code much cleaner. Thus we built a Flask package for our API. This package contained multiple other packages specifically: a *models* package for all our database models, a *controller* package for handling our HTTP requests, a *services* package for performing the database operations determined by the method invoked from whichever request was used for example running a SELECT query for a GET method invoked in the controller, and a *testing* package for performing unit testing.

Each of the packages above contained files for each of the models in the database. For example, a *user* model had one *user_services.py* file in the *services* package and one *user_controller.py* file in the controller package. It also had one *user_tests.py* file in the testing package. This would make it very easy to write methods specifically for any user queries or to test the user resource separate to the other resources.

A *manage.py* file was used to handle running the overall Flask application and this is what would be executed at runtime by Heroku. The *config.py* contains 3 different methods that run the API differently depending what environment it's in. For instance, if set to **dev**, the API is run in a development environment with the Flask debugging feature set to True.

The local postgres database is used. In **prod**, the production environment, this feature is set to False and the AWS hosted database is used. In **test**, for a testing environment, unit tests are run to check the methods for each resource.

Building Mycroft Skills

The first step in implementing the Mycroft skill was to create a basic *__init__.py* file that would perform the core functionality of the skill. This is the program that is used in allowing the user to interact with the Daisy skill. A basic skill allowed us to understand the Mycroft environment better and to find out how to respond to users' utterances to try create a conversation between the user and Mycroft. While writing this skill, we realised we needed additional files to store information locally on the raspberry pi. A *cred* file was created to store the users credentials to be used for retrieving questions and registering correctly.

Retrieving the Mycroft home assistants location to update the **home assistant** table was done using two programs. Since the GPS information returned by the GPS module is in NMEA format, this had to be changed to something more human readable and was done using the *gps.py* file from user **Lauszus** on github. The *update_gps.py* file then updated the home assistant table with the relevant latitude and longitude information.

In order to check what questions needed to be asked to the user, a *cron* job was written once the user registered with the Daisy skill to start retrieving questions every minute. This was done using the *get_questions.py* file. This program performs context detection and updates which device to use accordingly. If the home assistant is to be used, a *questions* file is updated with JSON data of what questions and answers to use and a prompt is sent to Mycroft to start the Daisy skill and check this file.

Building the Android App

When building the App, we wanted to make sure it performed a few major features: A User logging system, constant location tracking system, communication with API and microphone input tracking system.

As previously mentioned we used Firebase for the user logging system. Firebase Authentication was used to store user's emails and (encrypted) passwords and Firebase Firestore was used to store user's info, such as the user's details, questions and answers, locations, researcher's user lists, etc. Using Firebase would allow the app to safely store info remotely. Its fast speed also meant the app could very quickly fetch any info it needed. Firestore is a NoSql database system that works in collections and documents rather than tables. We had 3 major collections: Users, Usernames and PairCodes. Users essentially stored user details. Usernames stored all usernames used so that the app could check if a username was already being used. PairCodes worked similarly but to make sure no two pair codes were identical.

In order to have a fully functional location tracking, a service was used. This service would be started whenever the user registered or logged in and would only be closed if the user logged out. By starting this service, the app will create a persistent notification notifying the user that the app is tracking the gps coordinates. This will allow the location to be tracked even when the app is closed. This service will retrieve the device's GPS coordinates every few seconds (can easily be changed) and make an API call to the phone table in the database. This way the database gets updated every few seconds.

The microphone input of the device will be used for context detection. The way we implemented this feature was by creating a Job Scheduler instance. This job scheduler will, just like for the location tracking, be started whenever the user registers or logs in and will only be stopped when the user logs out. The job will be scheduled to perform an action every 15 minutes, even if the app is closed. This action will be to start a microphone service that once started will create a persistent notification. This service will use Speech Recognition to listen to the microphone for 20 seconds.

If it recognises any words, it will decide the user is busy. If no words have been recognised, then the app will decide the user is not busy and can be sent a question through the home-assistant. It will then update the database through an API call with the user availability. Once this happens, the app will then stop the microphone service and remove the notification. Essentially, the app will listen to 20 seconds of audio every 15 minutes to decide whether the user is available and hence send it to the database.

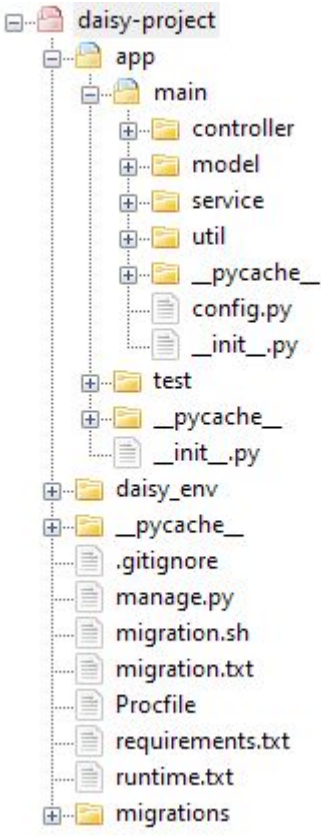
Communicating with the API was a challenge. Android does not originally support API calls. A way around it is to use third party systems. One of these third party systems is Retrofit 2. Retrofit 2 allowed us to create an interface that makes API calls through different methods. This way we were able to successfully update the database. The API is called various times throughout the flow of the application. We used API calls to store user and researcher details, phone details, questions and answers.

Sample Code

There two main programming languages used while building this system: Python for the API and Mycroft skill and Java for the android mobile app. The API was complex to design as we had 8 different tables in the database that needed 8 different resources to be used. The android app needed to communicate with firebase as well as our api and it needed to respond to user input.

API and Database

We used SQLAlchemy with Flask to allow us to use Python to communicate to our database. We had 8 different models, one for each table in the database and we used Alembic to perform migrations if there were any changes made. Each model had a controller, a service and a test class. This made debugging and testing parts of the code much easier as well as keeping the code clean.

 <pre> daisy-project ├── app │ ├── main │ │ ├── controller │ │ ├── model │ │ ├── service │ │ ├── util │ │ ├── __pycache__ │ │ ├── config.py │ │ └── __init__.py │ ├── test │ │ ├── __pycache__ │ │ └── __init__.py │ └── daisy_env │ ├── __pycache__ │ ├── .gitignore │ ├── manage.py │ ├── migration.sh │ ├── migration.txt │ ├── Procfile │ ├── requirements.txt │ ├── runtime.txt │ └── migrations </pre>	<p>The Flask API contains:</p> <ul style="list-style-type: none"> • A <i>controller</i> package for handling the HTTP requests. • A <i>model</i> package for all the models in the database. • A <i>service</i> package to perform the database queries. • A <i>test</i> package for testing each model. • A <i>util</i> package containing the namespaces for all the resource endpoints. • A <i>config</i> file with different environments to set while running the application. • A <i>manage.py</i> file which is run to start the Flask app. • A <i>Procfile</i> with information on how to run the app in Heroku. • A <i>requirements.txt</i> file with all the requirements needed to run the app. • A <i>migration.sh</i> script which runs the migration commands needed to perform database migrations and update the database if there are any changes to the models.
--	---

If we take a look at the model *user* we can see one of its *controller* and *service* classes:

<pre> from .. import db class User(db.Model): tablename = "user" id = db.Column(db.String(28), primary_key=True) username = db.Column(db.String(20), unique=True) pair_pin = db.Column(db.String(9), unique=False) home_assist_ID = db.relationship("HomeAssist", backref="user") phone_ID = db.relationship("Phone", backref="user") answer_returned_ID = db.relationship("AnswerReturned", backref="user") user_details_ID = db.relationship("UserDetails", backref="user") def __init__(self, id, username, pair_pin): self.id = id self.username = username self.pair_pin = pair_pin def json(self): return {'ID': self.id, 'Username': self.username, 'Pair Pin': self.pair_pin} </pre>	<p>The <i>user</i> model contains the model for how the user table is built.</p>
---	---

```

'''
The user controller class handles all the incoming HTTP
requests relating to the user
'''

from flask import request, jsonify
from flask_restplus import Resource

from ..util.dto import UserDto
from ..service.user_service import save_new_user, get_all_users, get_a_user, delete_user, update_user

api = UserDto.api
_user = UserDto.user

parser = api.parser()
parser.add_argument('pair_pin', type=str, default='pair_pin', required=True)

@api.route('/')
class UserList(Resource):
    @api.doc('list_of_registered_users')
    @api.marshal_list_with(_user, envelope='data')
    def get(self):
        """List all registered users"""
        return get_all_users()

    @api.response(201, 'User successfully created.')
    @api.doc('create a new user')
    @api.expect(_user, validate=True)
    def post(self):
        """Creates a new User """
        data = request.json
        response_object = {
            'status': 'success',
            'message': 'Successfully registered.',
            'user': data
        }
        save_new_user(data=data)
        return response_object

```

This sample of code from the *user_controller* shows what happens if a *post* method is invoked. The *save_new_user* method from the *user_service* class is used.

```

import datetime

from app.main import db
from app.main.model.user import User
from flask import jsonify
import json

'''
Creates a new user by first checking if the user already exists;
it returns a success response_object if the user doesn't exist
else it returns an error code 409 and a failure response_object
'''

def save_new_user(data):
    user = User.query.filter_by(username=data['username']).first()
    if not user:
        new_user = User(
            id=data['id'],
            username=data['username'],
            pair_pin=data['pair_pin']
        )
        save_changes(new_user)
        response_object = {
            'status': 'success',
            'message': 'Successfully registered.'
        }
        return response_object, 201
    else:
        response_object = {
            'status': 'fail',
            'message': 'User already exists. Please Log in.'
        }
        return response_object, 409

```

Here we can see the `save_new_user` method which is invoked when a POST request is sent to the API.

We need ways to save different namespaces associated with each resource. That is, the URL endpoint for each model we want to interact with.

```

api = Api(blueprint,
    title='Daisy API',
    version='1.0',
    description='The Daisy API is used to interact with the daisy-db for use within the Daisy system.'
)

api.add_namespace(user_ns, path='/user')
api.add_namespace(researcher_ns, path='/researcher')
api.add_namespace(home_assistant_ns, path='/home-assistant')
api.add_namespace(phone_ns, path='/phone')
api.add_namespace(question_ns, path='/question')
api.add_namespace(answer_ns, path='/answer')
api.add_namespace(answer_returned_ns, path='/answer-returned')
api.add_namespace(user_details_ns, path='/user-details')

```

These namespaces are stored in the `__init__.py` file of the main app.

The `dto.py` file allows us to save descriptions for each resource as well as their fields. It tells the API what values to accept.

```

from flask_restplus import Namespace, fields

class UserDto:
    api = Namespace('User', description='User related operations')
    user = api.model('User', {
        'id': fields.String(required=True, description='User Identifier'),
        'username': fields.String(required=True, description='User Username'),
        'pair_pin': fields.String(required=True, description='User Pair Pin')
    })

```

The *UserDto* class is related to the *user* model.

Additionally, we have a *config.py* file which allows us to change the environment as appropriate and the *manage.py* file which runs the app.

```

class DevelopmentConfig(Config):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = postgres_prod_base
    SQLALCHEMY_TRACK_MODIFICATIONS = False

class TestingConfig(Config):
    DEBUG = True
    TESTING = True
    SQLALCHEMY_DATABASE_URI = postgres_test_base
    PRESERVE_CONTEXT_ON_EXCEPTION = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False

class ProductionConfig(Config):
    DEBUG = True
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SQLALCHEMY_DATABASE_URI = postgres_prod_base

config_by_name = dict(
    dev=DevelopmentConfig,
    test=TestingConfig,
    prod=ProductionConfig
)

```

The *config.py* file allows us to run the app with different environment variables set depending on the environment we're in.

```

import os
import unittest
from flask_migrate import Migrate, MigrateCommand
from flask_script import Manager
from app.main import create_app, db
from app import blueprint
from app.main.model import user, researcher, question, answer, answer_returned, home_assistant, phone, user_details

app = create_app('prod')
app.app_context().push()
manager = Manager(app)
migrate = Migrate(app, db)
manager.add_command('db', MigrateCommand)
app = create_app('prod')
app.register_blueprint(blueprint)
app.app_context().push()

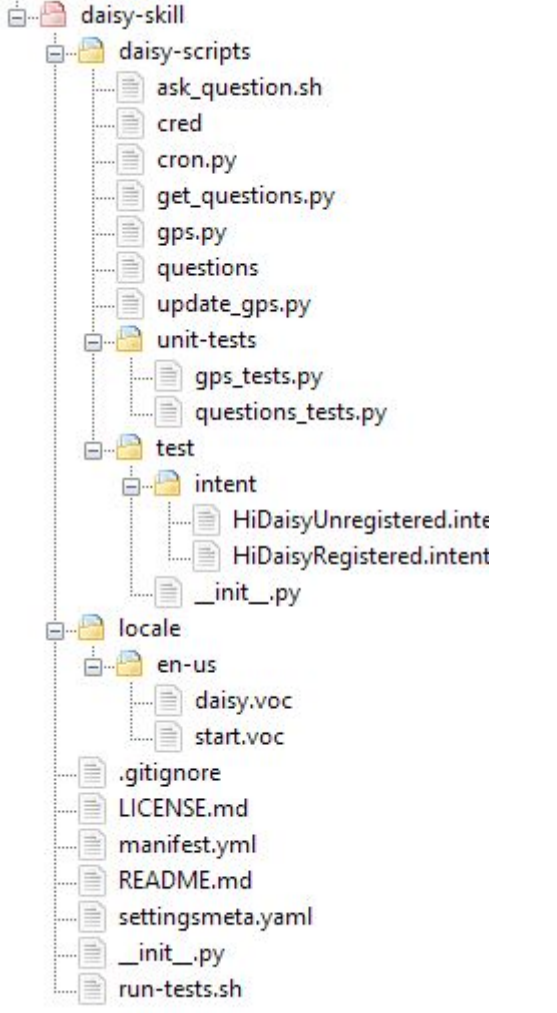
@manager.command
def run():
    app.run()

```


The *manage.py* file is our entrypoint in running the Flask API.

Mycroft

As mentioned before, Mycroft uses an *__init__.py* file for the main user interaction. The *gps.py* and *update_gps.py* files are used to update the home assistants GPS information. The *cron.py* and *get_questions.py* check every minute if the user has any questions and performs context detection to set devices appropriately.

 <pre>graph TD daisy-skill --> daisy-scripts daisy-skill --> unit-tests daisy-skill --> test daisy-skill --> locale daisy-skill --> .gitignore daisy-skill --> LICENSE.md daisy-skill --> manifest.yml daisy-skill --> README.md daisy-skill --> settingsmeta.yml daisy-skill --> __init__.py daisy-skill --> run-tests.sh daisy-scripts --> ask_question.sh daisy-scripts --> cred daisy-scripts --> cron.py daisy-scripts --> get_questions.py daisy-scripts --> gps.py daisy-scripts --> questions daisy-scripts --> update_gps.py unit-tests --> gps_tests.py unit-tests --> questions_tests.py test --> intent intent --> HiDaisyUnregistered.inte intent --> HiDaisyRegistered.intent intent --> __init__.py locale --> en-us en-us --> daisy.voc en-us --> start.voc</pre>	<p>We can see the files stored in the Daisy skill package. The files that perform in the background are stored in the <i>daisy-scripts</i> directory.</p>
--	---

Working through the user interaction, the main program that is executed is the *__init__.py* file:

```
class Daisy(MycroftSkill):
    def __init__(self):
        MycroftSkill.__init__(self)
        self.serial_key = getserial()
        self.home_assistant_id = str(uuid.uuid4())[0:28]
        self.answers_returned_id = str(uuid.uuid4())[0:28]
        self.user_id = None
        self.username = None
        self.registered = False
        self.questions_answers = {}
        self.answers = {}

        self.start_cron = join(self.root_dir, 'daisy-scripts/cron.py')
        self.ask_questions_bool = False
        self.update_gps = join(self.root_dir, 'daisy-scripts/update_gps.py')
        self.cred_file = join(self.root_dir, 'daisy-scripts/cred')
        self.questions_file = join(self.root_dir, 'daisy-scripts/questions')
```

The *init* method sets a number of variables to be used for the skill. The file paths to the extra programs are defined here.

```
@intent_handler(IntentBuilder('StartDaisy').require('Start').require('Daisy'))
def handle_start_daisy(self, Message):
    self.check_cred()
    if self.registered == False:
        response = self.get_response("Hi, have you registered on the daisy app")
        if response == "yes":
            code = self.get_response("Whats your pair code")
            self.check_user(code)
            if self.registered == False:
                self.speak("User does not exist. Please register on the daisy app and try pairing again with hi daisy")
            elif self.registered == True:
                if self.register_home_assist() is "SUCCESS":
                    self.save_cred()
                    self.update_location()
                    self.speak("Welcome {}. You have been registered".format(self.username))
                    self.start_cron_job()
                else:
                    self.speak("There has been an error. Please wait and try pairing again with hi daisy later")
            else:
                self.speak("There has been an error. Please wait and try pairing again with hi daisy later")
        elif response == "no":
            self.speak("Please register on the daisy app and try pairing again with hi daisy")
        else:
            self.speak("Invalid response use yes or no. try pairing again with hi daisy")
    else:
        self.speak("Welcome {}".format(self.username))
        self.get_questions()
        if self.ask_questions_bool == False:
            self.speak("You have no questions.")
        else:
            self.ask_questions()
```

This is the main *intent_handler* method. An intent handler handles an intent uttered by the user. An intent is a specific phrase said by the user, in this case, a combination of a start word *Hi*, *Start*, *Hello* stored in the *Hi.voc* file and *Daisy* stored in the *Daisy.voc* file.

Once the users pin has been successfully identified, the location is sent from the raspberry pi:

```

#!/home/pi/mycroft-core/.venv/bin/python
# Original Code: https://gist.github.com/Lauszus/5785023#file-gps-py
# Created by: Kristian Sloth Lauszus

import time
import os
import sys
from serial import Serial

def readString():
    ser = Serial('/dev/ttyACM0', 9600, timeout=1) # Open Serial port
    while 1:
        while ser.read().decode("utf-8") != '$': # Wait for the begging of the string
            pass # Do nothing
        line = ser.readline().decode("utf-8") # Read the entire string
        return line

def getTime(string, format, returnFormat):
    return time.strftime(returnFormat,
                        time.strptime(string, format)) # Convert date and time to a nice printable format

def getLatLng(latString, lngString):
    negate_lat = 1
    negate_lng = 1
    if "S" in latString:
        negate_lat *= -1
    if "W" in lngString:
        negate_lng *= -1
    lat_lst = str(float(latString[:-1]) / 100).split(".")
    lng_lst = str(float(lngString[:-1]) / 100).split(".")
    deg_lat = float(lat_lst[0])
    deg_lng = float(lng_lst[0])
    min_lat = lat_lst[1].lstrip("0")
    min_lng = lng_lst[1].lstrip("0")
    lat = (deg_lat + float(min_lat[0] + "." + min_lat[1:]) / 60) * negate_lat
    lng = (deg_lng + float(min_lng[0] + "." + min_lng[1:]) / 60) * negate_lng
    return lat, lng

```

This *gps.py* file from user **Lauszus** on github. The *getLatLng* method is what we can use to retrieve the home assistants latitude and longitude.

```

def main():
    print("Running...")
    root_dir = "/opt/mycroft/skills/daisy-skill/daisy-scripts"
    cred_file = join(root_dir, "cred")
    home_assistant_user_url = "https://daisy-project.herokuapp.com/home-assistant/user/"
    home_assistant_base_url = "https://daisy-project.herokuapp.com/home-assistant/"
    user_id = return_user_id(cred_file)
    if user_id is not None:
        home_assistant_id = get_home_assistant_id(home_assistant_user_url, user_id)
        put_gps(home_assistant_base_url, home_assistant_id)
        logs = root_dir + "/access_logs.txt"
        myFile = open(logs, "a")
        myFile.write("\nUPDATE_GPS.PY: Accessed on " + str(datetime.now()))
    else:
        pass

if __name__ == "__main__":
    main()

```

A sample from our *update_gps.py* file which has the URL endpoints needed to save the users credentials and update the home assistants GPS information using *gps.py*.

Once the location has been updated, the cron job is then started to begin checking for questions using the *get_questions.py* file:


```

from crontab import CronTab
import os

cron = CronTab(user='pi')
root_dir = "/opt/mycroft/skills/daisy-skill/daisy-scripts"
script_path = root_dir + "/get_questions.py"
job = cron.new(command='python3 ' + script_path)
job.minute.every(1)

cron.write()

```

The *cron* program will set the *get_questions.py* file to run every minute, checking for questions and performing context detection.

```

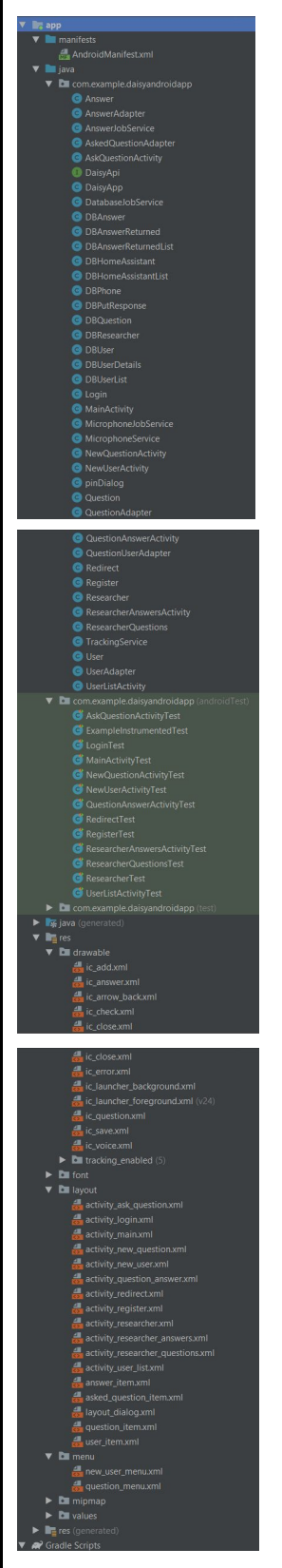
def main():
    print("Checking for questions...")
    root_dir = "/opt/mycroft/skills/daisy-skill/daisy-scripts"
    logs = root_dir + "/access_logs.txt"
    myFile = open(logs, "a")
    myFile.write("\nGET QUESTIONS.PY: Accessed on " + str(datetime.now()))
    user_details_user_url = "https://daisy-project.herokuapp.com/user-details/user/"
    questions_url = "https://daisy-project.herokuapp.com/question/"
    answers_url = "https://daisy-project.herokuapp.com/answer/"
    phone_url = "https://daisy-project.herokuapp.com/phone/"
    home_assistant_url = "https://daisy-project.herokuapp.com/home-assistant/"
    user_id = get_user_id()
    phone_gps = get_gps(phone_url, user_id)
    home_assistant_gps = get_gps(home_assistant_url, user_id)
    questions_id_list = check_true_questions(user_id)
    user_availability = check_user(user_id)
    if user_id is not None and questions_id_list is not None and user_availability is not None: #check if there are any questions
        print("Found questions to ask!")
        #test_dist = 9.9 #to use for testing purposes and checking if the system picks the right device
        dist = find_dist(phone_gps, home_assistant_gps)
        if choose_device(dist, user_availability) == "home-assistant":
            print("Sending to home-assistant...")
            questions_dict = get_questions(questions_id_list)
            question_answers_dict = questions_answers(questions_dict)
            print("Saving questions...")
            print(question_answers_dict)
            save_details(question_answers_dict)
            set_device("home-assistant", user_id)
            prompt_mycroft()
        else:
            print("Sending to phone...")
            set_device("phone", user_id)
    else:
        print("Found no questions.")
        pass

if __name__ == "__main__":
    main()

```

The *main* method in the *get_questions.py* file. This is where most of the control flow for how the program runs is written. The program uses methods above to check GPS coordinates and user availability to perform context detection. If the Daisy skill is to be used to ask questions to the user a *questions* file is updated with the question and answers formatted in JSON. A prompt is sent to the Daisy skill notifying it of new questions and it checks this file.

Android

	<p>We can see the files stored in the app package.</p> <ul style="list-style-type: none">• The manifest file describes essential information about the app to the Android build tools, the Android operating system and Google Play. It defines all the activities and services as well as all the permissions required.• The java directory will have all the classes used as well as the testing files.• The res folder will have all the design elements of the app. This includes any images or icons used in the app, all the screen XML files and any menus used in the app.
--	--

```

public interface DaisyApi {

    @GET("user")
    Call<DBUserList> getUsers();

    @POST("user/")
    Call<DBUser> createUser(@Body DBUser dbUser);

    @PUT("user/{id}")
    Call<DBPutResponse> updatePairPin(@Path("id") String id, @Query("pair_pin") String pairPin);

    @POST("phone/")
    Call<DBPhone> createPhone(@Body DBPhone dbPhone);

    @PUT("phone/{id}")
    Call<DBPutResponse> updatePhoneCoords(@Path("id") String id, @Query("lat_long") String gpsCoords);

    @POST("researcher/")
    Call<DBResearcher> createResearcher(@Body DBResearcher dbResearcher);

    @GET("home-assistant")
    Call<DBHomeAssistantList> getHomeAssistants();

    @POST("question/")
    Call<DBPutResponse> createQuestion(@Body DBQuestion question);

    @GET("question/{id}")
    Call<DBQuestion> getQuestionWithId(@Path("id") String id);

    @POST("answer/")
    Call<DBPutResponse> createAnswer(@Body DBAnswer answer);

    @GET("answer/question/{questionId}")
    Call<List<DBAnswer>> getAnswersWithQuestionId(@Path("questionId") String questionId);

    @POST("user-details/")
    Call<DBPutResponse> createUserDetails(@Body DBUserDetails dbUserDetails);

    @GET("user-details/user/{id}")
    Call<List<DBUserDetails>> getUserDetails(@Path("id") String id);

    @PUT("user-details/user/{user_id}")
    Call<DBPutResponse> updateUserDetails(@Path("user_id") String userId, @Query("ask_question") Boolean askQuestion, @Query("device_to_use") String deviceToUse, @Query("
    @DELETE("user-details/{id}")
    Call<DBPutResponse> deleteUserDetails(@Path("id") String id);

    @POST("answer-returned/")
    Call<DBAnswerReturned> createAnswerReturned(@Body DBAnswerReturned dbAnswerReturned);

    @GET("answer-returned")
    Call<DBAnswerReturnedList> getAnswerReturned();
}

```

This is the API interface file. Each different call used will have its own method.

```

private void requestLocationUpdates() {
    LocationRequest request = new LocationRequest();

    //Specify how often your app should request the device's location//
    request.setInterval(30000);

    //Get the most accurate location data available//
    request.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
    final String path = "location";
    int permission = ContextCompat.checkSelfPermission(context, this,
        Manifest.permission.ACCESS_FINE_LOCATION);

    //If the app currently has access to the location permission...//
    if (permission == PackageManager.PERMISSION_GRANTED) {
        //...then request location updates//
        client.requestLocationUpdates(request, locationCallback = (LocationCallback) onLocationResult(locationResult) -> {
            Location location = locationResult.getLastLocation();
            if (location != null) {
                //Save the location data to the database//
                Log.w(tag, "TrackService", msg: "Service is trying to send location");
                documentReference.collection(path).document(path).set(location)
                    .addOnSuccessListener(new OnSuccessListener<Void>() {
                        @Override
                        public void onSuccess(Void aVoid) {
                            Log.w(tag, "TrackService", msg: "Service has updated location");
                        }
                    }).addOnFailureListener(new OnFailureListener() {
                        @Override
                        public void onFailure(@NonNull Exception e) {
                            Log.w(tag, "TrackService", msg: "Service has failed to update location: " + e.toString());
                        }
                    });
                Log.d(TAG, location.toString());

                String latLong = location.getLatitude() + "," + location.getLongitude();
                Log.d(TAG, msg: "LOCATION: " + latLong);
            }
        });
    }
}

```

This is a portion of the TrackService.java file. This portion shows the interval in which the location gets updated as well as firestore being updated.

```

private void listen() {
    SpeechRecognitionListener h = new SpeechRecognitionListener();
    mSpeechRecognizer = SpeechRecognizer.createSpeechRecognizer(this);
    mSpeechRecognizer.setRecognitionListener(h);
    mSpeechRecognizerIntent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
    mSpeechRecognizerIntent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
        RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
    mSpeechRecognizerIntent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, Locale.getDefault());
    Log.d(TAG, msg: " " + mSpeechRecognizer.isRecognitionAvailable( context: this));
    if (mSpeechRecognizer.isRecognitionAvailable( context: this))
        Log.d(TAG, msg: "onBeginningOfSpeech");
    mSpeechRecognizerIntent.putExtra(RecognizerIntent.EXTRA_CALLING_PACKAGE,
        this.getPackageName());
    Log.d(TAG, msg: "Listening now");
    listening = true;
    mSpeechRecognizer.startListening(mSpeechRecognizerIntent);
}

```

```

@Override
public void onError(int i) {
    Log.d(TAG, msg: "ERROR: " + i);
    if ((i == SpeechRecognizer.ERROR_NO_MATCH)
        || (i == SpeechRecognizer.ERROR_SPEECH_TIMEOUT))
    {
        Log.d(TAG, msg: "didn't recognize anything");
        userResults += "";
        // keep going
        if (listening) {
            Log.d(TAG, msg: "Listening again");
            mSpeechRecognizer.destroy();
            listen();
        }
    }
}

@Override
public void onResults(Bundle resultsBundle) {
    Log.d(TAG, msg: "onResults: " + resultsBundle.toString());
    ArrayList<String> matches = resultsBundle.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION);
    if (matches.get(0) != null)
        userResults += " " + matches.get(0);
    else
        userResults += "";
    if (matches != null)
        Log.d(TAG, msg: "matches: " + matches.get(0));
    Log.d(TAG, msg: "I have listen to: " + userResults);
    if (listening) {
        Log.d(TAG, msg: "Listening again");
        mSpeechRecognizer.startListening(mSpeechRecognizerIntent);
    }
}

```

This image shows the microphone service. This specific sample shows the app start listening and the results being analysed once listening has completed or an error has occurred. Speech recognizer only listens to a sentence at the time, so the app has to keep calling listen until stopListening has been called.

```

public class Question {
    private String title;
    private String id;
    private String researcherId;
    private String userId;
    private ArrayList<String> answers;

    //empty constructor needed
    public Question() {}

    public Question(String title, String id, String researcherId, String userId, ArrayList<String> answers) {
        this.title = title;
        this.id = id;
        this.researcherId = researcherId;
        this.userId = userId;
        this.answers = answers;
    }

    public String getTitle() { return title; }

    public String getId() { return id; }

    public String getResearcherId() { return researcherId; }

    public String getUserId() { return userId; }

    public ArrayList<String> getAnswers() { return answers; }
}

```

This sample shows the Question class. This class gets used anytime a question is either being created, asked or answered. Each question will have an id, a title, a researcher id, a user id and an array with the possible answers.

Problems Solved

As previously mentioned, we had issues with understanding how to build the backend of our system. We knew we needed a server to perform our context detection but were unsure of how to develop this or how to deploy it. We also needed a way for our Mycroft and app to be paired in some way to be certain we could link a user to their account and home assistant.

One of our biggest issues was understanding the control flow in which the system would work. For example, when sending a question to the user, we need to keep in mind that first *ask_question* in **user details** is set to **True**. Then, the users app needs to perform a mic check and update *user_available* to either **True** or **False** appropriately. Once that's all done, the system can perform context detection and update the *device_to_use* appropriately.

By fleshing out how the system would function step by step, we were able to break down the order of operations and make sure the system would do what we wanted it to. Otherwise, there could be problems with what device the system would send questions to or other issues.

Initially Using Flask

We mentioned initially using Flask to use as our server but we were unfamiliar with developing a server or understanding where to deploy it. At first, a simple flask server was deployed locally with a connection to a local postgres database. However, there were problems with using Flask locally as we couldn't make any calls to the API outside of the local network. Thus we needed a solution.

Switching to Amazon Web Services (AWS)

Once we realised deploying a Flask server was going to be more difficult than we thought we looked at other options to build our backend. One of these options was AWS. We chose this platform in particular because of the availability of a *free tier* option. AWS provided us with a *cloud* platform with serverless hosting meaning we didn't need a dedicated server to host our API and database.

Although we were able to migrate the database fairly easily to AWS, we had issues with developing an API. Understanding the AWS environment became very difficult. AWS requires its services to run within its own network for security reasons and a developer needs to create this environment and grant permissions across services. When developing functions with AWS Lambda and trying to connect them to resources in API Gateway became too problematic we decided to switch to a different solution.

Heroku

We had looked into using Heroku before, early in development when we were trying to deploy our Flask server but at the time it seemed quite daunting. Once we had more familiarity with using APIs from AWS, we started looking into trying to build a Flask API that could be deployed on Heroku again. We rebuilt the API and were able to deploy it on Heroku.

Pairing System

Early in development we quickly realised that we needed a way to connect users to their app and home assistant. We needed a way for users to *sign in* to the home assistant but this was difficult considering they can only interact with voice commands. Thus we came up with the solution of having a pin being used to identify users to their account.

Originally we thought of having users have their pin issued by the home assistant as this would mean they could create an account on the app and thus eliminating the need for speaking to Mycroft to sign in. However, we found it would be problematic to create completely unique codes and multiple users might be issued the same code so when checked from the app, their account wouldn't be linked correctly. We switched issuing the code to the app. Firebase made it much easier to create unique pins and we could guarantee that no two users would share the same pin. Then it became a matter of saying the pin to Mycroft and that would pair the user.

Developing Skills

While we were developing the Daisy skill for Mycroft there were a few issues we encountered. These were problems that were to do with unconventional tasks we needed to perform with Mycroft. Two such tasks we needed to have were: to set an automated way to check the database every minute once the user had registered and to start GPS information tracking and update the database once the user had registered.

Since a skill is only active when a user is interacting with it, we also needed a way to start the skill and issue a question to the user once the home assistant had decided it was time to do so. The Mycroft community was a wonderful resource in allowing us to troubleshoot any skill issues we came across and we were able to solve a few problems with the aid of the Mycroft forum.

Automated Question Check

One of the problems we had was with automatically checking for questions. We needed this to be done in the background of the raspberry pi as it could not be performed while a skill was active. This was because the skill could not stay active indefinitely and is only available when a user is interacting with it.

A solution to this was to issue a cron job to start once the user had paired with the home assistant. A cron job can be set to run any programs at a specific time. Once the user had registered, a program was executed to write the cron job starting the question check every minute.

GPS Tracking

GPS information is one aspect of the context detection that we needed to identify which device to send questions to. It was difficult to read the NMEA data sent straight from the GPS module but from looking online the *gps.py* file we found helped us convert this. However, although this data was now more readable, we discovered that the GPS latitude and longitude was giving us incorrect information. It also didn't match the format as that from the phone. This wouldn't help us in calculating the distance between the two devices. The *getLatLng* method in the *gps.py* file had to be edited to fit the standard we needed.

Asking Questions

A problem that the Mycroft community helped us with was in sending questions from the *get_questions.py* file. We needed to prompt the skill without it requiring a user utterance first. We found out that the Mycroft *messagebus* could be used to prompt the skill. A *question_handler* method could be used to handle this prompt and announce that new questions were ready to be asked. Since it was difficult to send all JSON data directly through this prompt we stored the data in a *questions* file. The format of this data was:

```
{question id: [question, {answer id: answer}]}
```

The question id enables us to identify which question is being asked. We then have the actual question along with potential answers. Once the user selects an answer, its answer id is sent to the **answers returned** table.

Android Background Operation Issues

When designing the application, we wanted to build an app that listens to audio and tracks location in the background. This seemed feasible, however, since Android Oreo (8.0), Google has imposed a range of limitations with background running operations due to battery and cpu. Essentially, an app running cannot keep a service running in the background anymore. This was an issue because the app would incorporate various background tasks such as location tracking, microphone input, question and answer updating.

The way around this was to instead of create background services, make use of the Android Job Scheduler and Foreground services. Job Scheduler allows us to schedule an action to be performed under certain circumstances. One of these circumstances is setting the job periodically. This was used for implementing the microphone feature.

Another way to run operations in the background is to move a service to the foreground. This works similarly to the background service, however, the operations are performed in the main UI thread.

In addition, they require a persistent notification to be active, hence why the use of notifications for the microphone service and the tracking service to allow the user to know the app is running in the background.

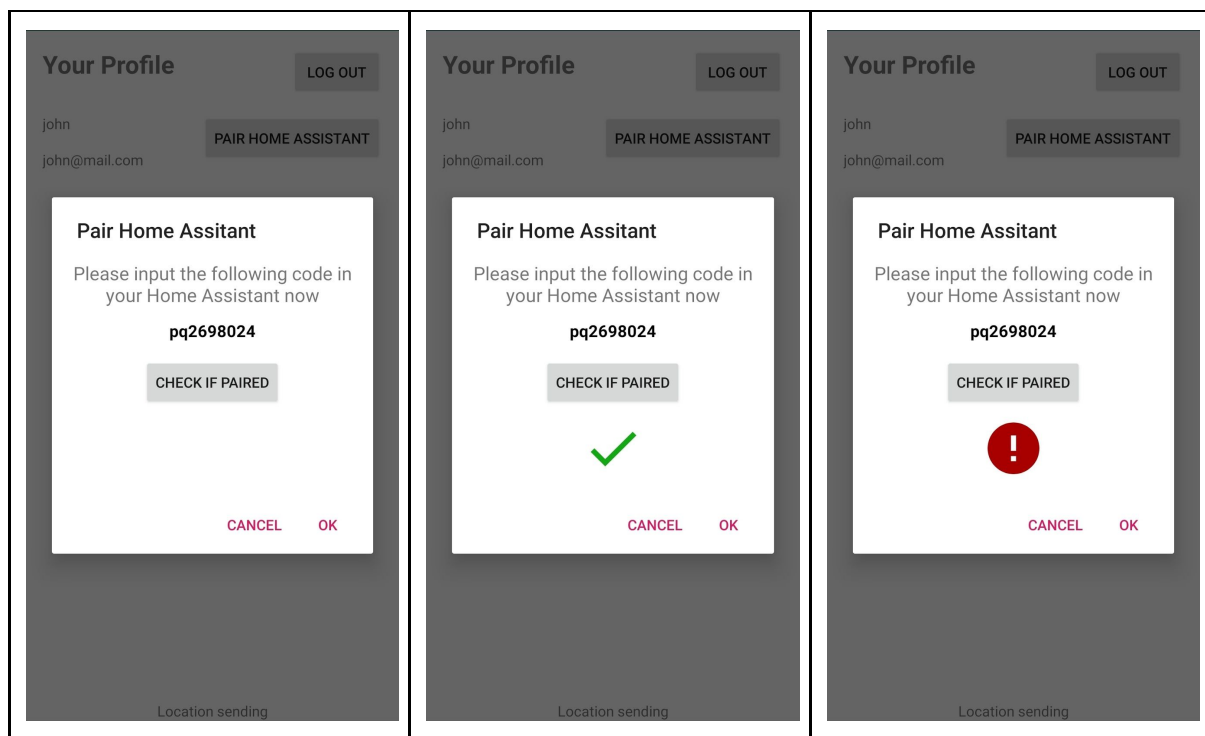
Job Scheduler was used to create a foreground service that runs the microphone and then stops itself. Tracking the location was done through a foreground service active throughout the whole lifecycle of the application. Job Scheduler was also used to update and notify the user and researcher that a new question has arrived or an answer has arrived respectively.

Results

Having successfully developed the API, the database, the mobile app and the Mycroft skill, it was a matter of integrating all these components together. We first started by developing the database, then the API. While this was being developed we were developing the daisy app user interface and storing information with Firebase. Once the API was developed, we could integrate the app and develop the Mycroft skill properly to communicate with the API. We needed the skill to have access to information sent from the app in order to register the user successfully and to send them questions. Once this was done, the answers could be sent back to the researcher.

Successful Integration

Once the user created an account on the app, their information is stored in the database. When they created a pair pin from the app, the *pair_pin* field in the database is updated and the home assistant will look for this when the user utters their pin to the Daisy skill.



Pairing code issued.	Clicking <i>Check If Paired</i> will show a tick symbol if you have successfully paired with Mycroft.	Clicking <i>Check If Paired</i> will show a warning symbol if you have unsuccessfully paired with Mycroft.
----------------------	---	--

We can see this user saved in the **user** table in the database:

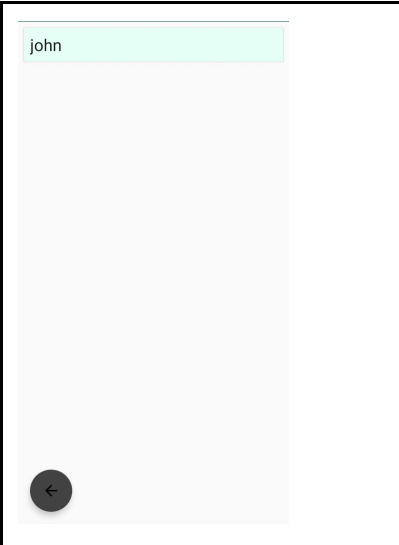
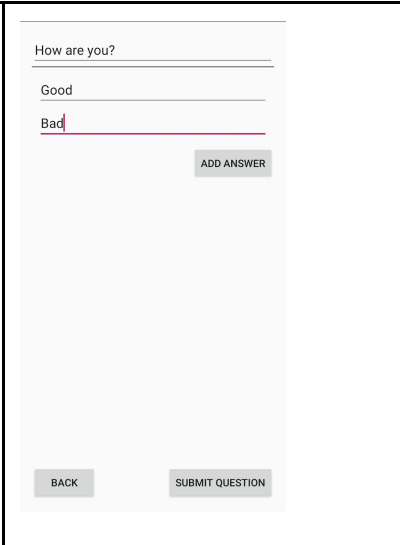
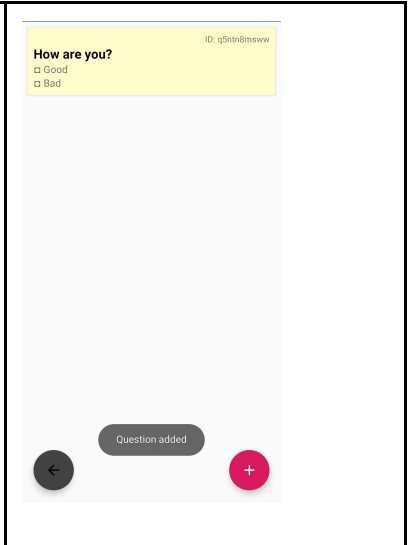
Data Output	Explain	Messages	Notifications
	id [PK] character varying (28)	username character varying (20)	pair_pin character varying (9)
1	9YJSdnL0wUUwqV1ijGuDql...	john	pq2698024

User **john** created a *pair pin* **pq2698024**. The Daisy skill checks for users with this pair pin when uttered to register the Mycroft as their home assistant.

Once a user creates a pin using the app they must say this pin to Mycroft and they will be successfully registered and ready to receive questions through either device.

Initial interaction with the Daisy skill.	<pre> History ===== hi daisy >> Hi, have you registered on the daisy app Log Output Legend ===== Mic Level === DEBUG output skills.log, other voice.log --- 7.86 Input (':' for command, Ctrl+C to quit) ===== > </pre>
Daisy asks for the Pairing Code.	<pre> History ===== hi daisy >> Hi, have you registered on the daisy app Log Output Legend ===== Mic Level === DEBUG output skills.log, other voice.log --- 7.86 Input (':' for command, Ctrl+C to quit) ===== > </pre>
The Pairing Code is uttered by the user.	<pre> History ===== >> Hi, have you registered on the daisy app Log Output Legend ===== Mic Level === DEBUG output skills.log, other voice.log --- 0.01 Input (':' for command, Ctrl+C to quit) ===== > </pre>

This transition between using the app and Mycroft is possible because of the API. A researcher can select this user and ask them a question:

		
The researcher selects a user to send questions to.	A question text field as well as response text fields will appear. In the example above How are you is inputted with responses Good and Bad .	Once the researcher clicks <i>Submit Question</i> in the previous screen their question will be added and Daisy will decide which user device to send to and send the question immediately .

This interaction updates the **user details** table:

Data Output							Explain	Messages	Notifications
	id	user_ID	question_ID	ask_question	user_available	device_to_use			
	[PK] character varying (28)	character varying (28)	character varying (28)	boolean	character varying (20)	character varying (20)			
1	10004	9YJSdnL0wUUwqV1ijGu...	6rf3dtw76z	true	True	null			

The *ask_question* field is set to **True** indicating a new question has been asked. *User_available* is set to **True** indicating that the user is available for the home assistant. The field *device_to_use* is initially set to **null** but this will be updated appropriately.

We can also identify the *question_ID* in the **question** table to find what question was asked and pull the possible answers using this ID.

Data Output Explain Messages Notifications			
	id [PK] character varying (28)	question character varying (100)	researcher_ID character varying (28)
1	6rf3dtw76z	How are you?	W9TiLQKbzbvOrnxyCZQi...

The *question_ID* **6rf3dtw76z** is the correct question asked by the researcher.

Data Output Explain Messages Notifications			
	id [PK] character varying (28)	answer character varying (100)	question_ID character varying (28)
1	kl6qbjeenx7x27xkpmx4mety...	Good	6rf3dtw76z
2	d3whpy9od1odsnhcghfye77...	Bad	6rf3dtw76z

We can see that that *question_ID* is linked to two answers in the **answer** table.

A user can answer the question on the app:

<p>Your Profile</p> <p>john</p> <p>john@mail.com</p> <p>LOG OUT</p> <p>User Paired</p> <p>How are you?</p> <p>Location sending</p>	<p>Question: How are you?</p> <p><input type="radio"/> Good</p> <p><input type="radio"/> Bad</p> <p>SUBMIT QUESTION</p> <p>Question asked by: jx3dQFvAu9NvaOM99P1vXw5QQmA2</p>	<p>Question: How are you?</p> <p><input checked="" type="radio"/> Good</p> <p><input type="radio"/> Bad</p> <p>SUBMIT QUESTION</p> <p>Question asked by: jx3dQFvAu9NvaOM99P1vXw5QQmA2</p>
<p>New question to be answered.</p>	<p>Selecting the question will bring the user to this screen.</p>	<p>The user selects their response and clicks the <i>Submit Question</i> button.</p>

The **answer returned** table is updated accordingly:

Data Output

Explain

Messages

Notifications

	id [PK] character varying (28)	answer_time character varying (28)	device_used character varying (20)	user_ID character varying (28)	answer_ID character varying (28)
1	sfh4t9ffkm	2020-05-15 20-08-41	phone	9YJSdnL0wUUwqV1ijGu...	kl6qbjeenx7x27xkpmx4...

We can see the correct user's response with the *answer_ID* that is linked to the answer **Good**. The *device_used* is logged correctly as **phone** as is the date and time responded.

Alternatively, if the system decides that home assistant is to be sent the question, it will update the **user details** table accordingly:

Data Output							Explain	Messages	Notifications
	id	user_ID	question_ID	ask_question	user_available	device_to_use			
	[PK] character varying (28)	character varying (28)	character varying (28)	boolean	character varying (20)	character varying (20)			
1	10004	9YJSdnL0wUUwqV1ijGu...	6rf3dtw76z	false	False	home-assistant			

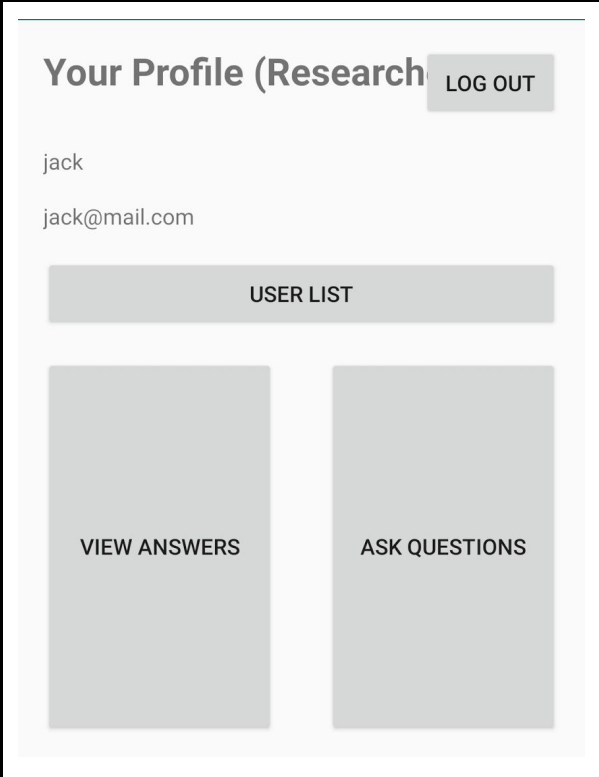
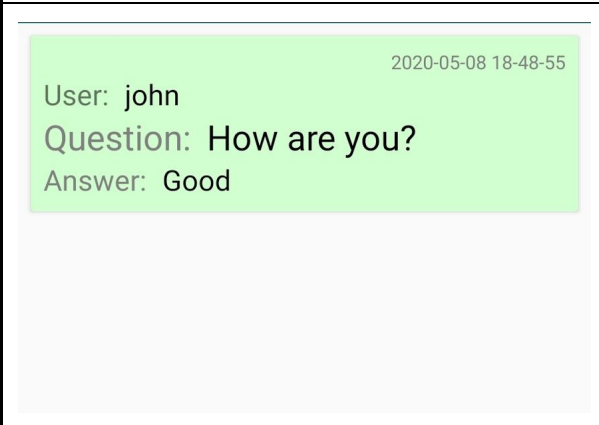
The *device_to_use* field has been updated accordingly and *ask_question* and *user_available* has been reset to **False**.

Mycroft will prompt the user that they have a new question or user can utter the *Hi Daisy* utterance to check if they have any questions.

<pre>History ===== hi daisy >> Welcome john >> You have new questions would you like to answer Input (':' for command, Ctrl+C to quit) ===== ></pre>	<p>Starting the Daisy as usual will tell the user if they have any new questions.</p>
<pre>History ===== yes >> Ok here are your questions >> Question 1 How are you? >> Your responses are >> Good >> Bad >> Which response do you pick Input (':' for command, Ctrl+C to quit) ===== ></pre>	<p>Responding with yes allows Mycroft to issue the user the questions with response options.</p>
<pre>History ===== >> Your responses are >> Good >> Bad >> Which response do you pick Good >> Your response Good has been logged >> Responses recorded Input (':' for command, Ctrl+C to quit) ===== > </pre>	<p>The response Good has been uttered and saved.</p>

Once an answer has been returned to the **answer returned** table, the researchers account on the app will pick this up and display it for the researcher:

With this example used, the following appears in the app once the researcher clicks the *view answers* button:

	<p>The researchers profile screen. Once they click <i>view answers</i> they can see any answers returned by the user.</p>
	<p>We can see the following information in this response box:</p> <ul style="list-style-type: none">• The date and time the user responded at is displayed in the top right.• The user's name, in this case: john.• The question the researcher asked: How are you?• The answer the user responded with: Good.

Future Work

There are many potential ways to further develop our system. With the amount of time we had, we decided to focus on absolute core functionality which meant having a researcher ask a question immediately, having context detection select a device, having a user answer and storing their response.

We initially wanted to have these questions scheduled at times set by the researcher but this was outside the scope of what he had planned. We also believe the context detection should be performed using a server instead of the home assistant.

Machine Learning

An idea for future uses of this system would be in the area of machine learning. We had originally hoped to build some sort of behaviour analysis tool for researchers. This system would ask specific questions to users at particular times in order to build a mood profile. This mood profile could help aid medical professionals in predicting what mood a user could be in at a particular time e.g. depressed.

We quickly realised the complexity of building the initial system to ask users questions and save the responses. Thus we decided to build a system that would do that and record answers in a way that would make the data as useful as possible. We added additional columns in the database such as **answer_time** and **device_used**. We also made sure to remove as many redundancies as possible in the database and to use closed questions so that users' answers could be sorted for better analysis.

These decisions could allow for more comprehensive machine learning to be conducted for whatever purpose researchers would wish.