SmartBell

Intelligent Home Receptionist

AQA A Level NEA

Orlando Alexander
Berkhamsted School

Contents

Analysis

Primary Client

Name: Maria Kramer

Profession: Architect

Background to the Problem

Maria Kramer is an architect and 1 of around 15 million adults who has worked from home throughout the COVID-19 pandemic (HighSpeed Office, 2020). With estimates that 25-30% of the workforce working from home for multiple days a week by the end of 2021 (Global Workplace Analytics, 2020), employees across the country will continue to be faced with embarrassing, frustrating and potentially dangerous interruptions to their working-day. Without knowledge of who is at the door, people who work from home are presented with 3 key issues: the person at the door may be delivering a parcel and so opening the door is not essential, the person at the door may be a friend or family member whom the homeowner would likely have an extended conversation with, delaying their return to their work call, or the person at the door may have malicious intents. A fourth issue also exists for those, such as Kramer, who meet clients at their house: despite working from home, it is important that interactions with clients are still professional. However, at present, it is impossible for Kramer to be aware that the person at the doorstep is a client, jeopardizing the opportunity for a formal greeting. Moreover, if a client arrives early or a business call overruns, those who work from home may not hear the doorbell ring, leaving clients stuck on the doorstep. Even if they do hear the doorbell, they still find themselves torn between 2 undesirable options: un-professionally abandoning their work call to welcome the client or leaving the client waiting until their call finishes. Another issue arises when homeowners, such as Kramer, leave their house; visitors often come to their door whilst they are out, but, at present, homeowners don't know who has come by and so cannot follow up on these visits.

Interview with Maria Kramer (the primary client)

Could you explain what your business is and what you do?

Maria: I'm an architect and I run my own architectural practice, which focuses on residential and community projects. My day-to-day work involves designing, negotiating, developing spatial proposals with various different people. There are different design stages, so you go from outline design all the way up to do a planning application, and then it goes to detailed design where you have to do tendering to various contractors. And then, there's often work on site.

What are the benefits of a traditional doorbell that you wouldn't want to lose with the SmartBell?

Maria: It's low tech so you don't have any data or privacy issues. Because it's low tech, it's less hard to maintain. It's also old fashioned; we live in an old house and have an old door, so the traditional design is important.

What are the drawbacks of a traditional doorbell?

Maria: Sometimes I don't hear it. Sometimes it actually happens that I am in a meeting, and I get interrupted in the meeting, and I have walk with my laptop to the door to the door or shout out so that is an issue with home working. Also, it's quite a long way; not that it's a big house but it's kind of the inconvenience, I suppose. And then, of course, when I'm not in physically here I can't really respond.

How many clients/colleagues come to your home on average day?

Maria: I obviously work with specialist consultants, so it's a different range of professional contacts I suppose. It kind of goes up and down; it depends on the project. So, for example, I did have a freelancer, working before COVID who I would see on a daily basis. So, I would say between 1 and 3 people per day, but it can be 3 people 1 day, and then nobody for 3 days, if that makes sense.

How many uninvited people come to your home on an average day? For example, food or parcel deliveries.

Maria: I would say 2 to 3 times a day. We order quite a lot online, and then there's the postman, and then you have people knocking on the door because they want something; there's quite a lot!

What are some issues that you face when you work at home that you don't face when you work in an office with a receptionist?

Maria: In an office, there is more control; you've got more controlled access because there's a buffer between your work zone and the public.

What features would you like to see in the SmartBell? These could range from playing a personalised audio message to a client/colleague identified by the doorbell's facial recognition system to sending a notification to you via a mobile app to alert you of an unidentified person at the door with an image of the person or notifying you when someone is at the doorstep whilst you are out.

Maria: I really think the personalised message is fantastic; something like: "Hi, could you wait 5 minutes, I'm just finishing up a meeting.". It may be quite helpful when you have an Amazon delivery, because they put it on the doorstep and then they wait for you to come, so the doorbell could maybe say something like: "Thanks for the delivery, I'll pick it up a bit later." So, they can go off and don't have to wait. We also have a Waitrose delivery, so maybe it could say: "Please unload already I will be there shortly." Unfortunately, there are sometimes those that are unwelcome; nearly on a weekly basis we have somebody knocking on the door trying to sell household items completely overpriced. So, a polite message could be played to say that we don't take any sales at the door. So, I think the personalization fantastic because if you are in the middle of something it's really helpful if it's pre-recorded.

Would you prefer for the personalised messages to be spoken by a computer-generated voice or would you rather record them yourself?

Maria: I think it may be nice to have the flexibility. I think it's a nice idea to have a default that you can just choose from. But also, maybe there's quite a specific message that you would like to leave sometimes.

If the doorbell was unable to recognise the person at the door, the app would require you to select the appropriate audio message to be played based on an image of the person at the door. Would this extra step be an inconvenience for you?

Maria: I think it's okay to get a picture of the person and then you can try to identify it and then send a pre-recorded audio message. You'll have to make sure that the camera has a wide enough angle so that you can see clearly who the person is.

Would your clients/colleagues be happy to give consent for their photo and name to be stored on a private database? This may include giving the app access to your mobile phone contacts' names and profile pictures and/or allowing the user to assign a name to particular faces photographed by the doorbell so that it can identify the face next time.

Maria: Sure. I mean, it's a security thing; funnily enough, I don't think people are so worried about privacy and data, even though maybe they should be, but I think it's more the hassle of having to ask. But then again, Facebook does all that stuff already!

Expected features to satisfy primary client's requirements

1. Doorbell is connected to the Internet
2. Button wired to the doorbell's microprocessor which can be pressed to ring the doorbell
3. User is notified through mobile app when doorbell is rung
4. Doorbell has wide-angle camera which captures image of visitor
5. Doorbell uses facial detection/recognition algorithm
6. Image of visitor is displayed to user in the mobile app
7. Mobile app has audio recording capabilities so user can create personalised audio responses
8. Personalised audio responses can be played by the doorbell from the mobile app
9. Doorbell has text-to-speech capabilities
10. Doorbell has speaker to play audio responses to visitor
11. Data protection is adhered to throughout the system

Breakdown of Existing Systems

Skybell HD Doorbell Camera

Good Features:

- **High quality video camera** – 1080p HD colour camera, 180-degree field of view and colour night vision (not infrared).
- **Live streaming** – the view from the doorbell's camera can be livestreamed via an app.
- **Video and audio recordings saved** – all recordings are stored on the cloud for 7 days for the user to access for free.
- **Motion detection** – the homeowner is alerted even if the visitor doesn't press the doorbell button.
- **Two-way audio** – omnidirectional microphones allow the homeowner to communicate with the person at the door via the doorbell.
- **Integration with home automation** – allows the doorbell to be remotely voice controlled with systems such as Alexa.
- **No monthly fees** – once purchased, the doorbell can be used without requiring further payments from the homeowner.
- **Quiet mode** – doorbell's chime can be silenced so it doesn't disturb the homeowner.

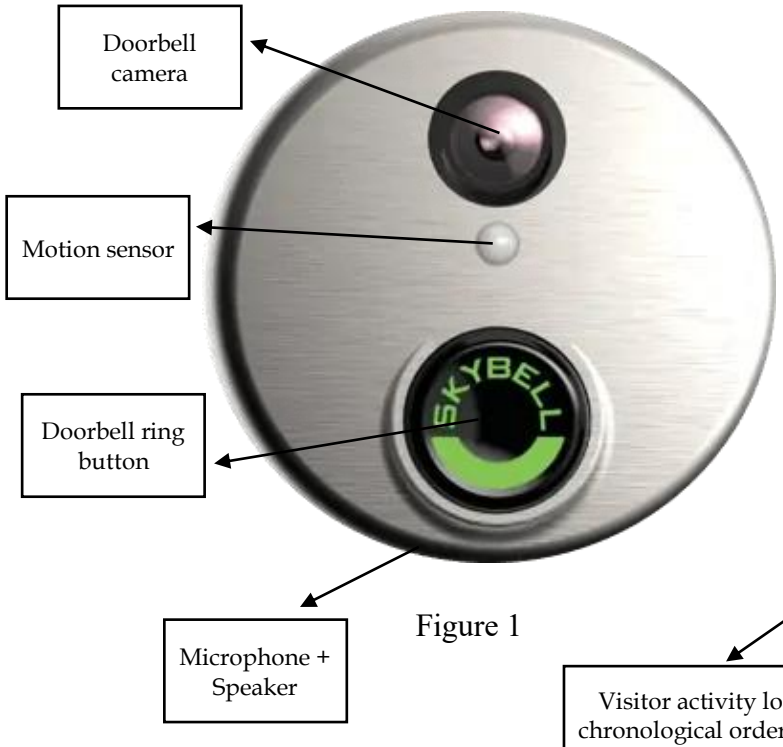


Figure 1

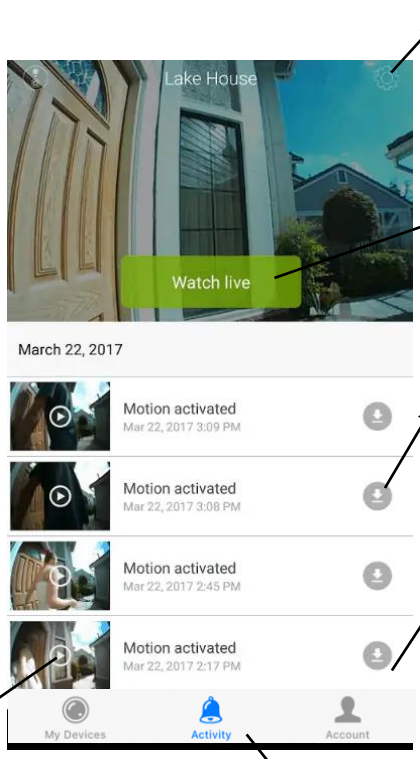


Figure 2

Less Good Features:

- **No face/person detection** – people who work from home may receive many unnecessary notifications which interrupt their work calls as the doorbell's camera will be activated by a passing leaf or animal.
- **No facial recognition** – personalised audio messages for a client at the doorstep and notifications with the name of the client at the door are not possible.
- **No automated audio messages for visitors** – any communication with the visitor must be live audio from the homeowner, which may not be convenient if they are on a work call.
- **Fairly expensive** – the doorbell shown in the above image costs £155.
- **Videos require lots of storage** – storing videos for 7 days of length 30-45 seconds each time motion is detected is costly in terms of storage.
- **Videos are inefficient to identify visitor –** people who work from home may be in the middle of a conference call, so a video is a time-consuming method of identifying who is at the door.
    - Sending a still image of the visitor to the homeowner's phone is a much more efficient method of identifying them.
- **Visitor logs aren't user friendly** – as the doorbell will record video each time the motion detector is activated (which could be done by a passing animal), the visitor activity log is crowded with irrelevant videos.
    - An activity log which only stored images of visitors' faces would be much more effective as it would give a clear overview of who has visited.
    - By foregoing the motion detector and only capturing images when the doorbell is rung, the risk of cluttering the activity log with meaningless videos/images is avoided.
- **Poor reviews** – on Amazon, 46% of the reviews for the device are 1-star. This is largely due to the lengthy delay between the bell being rung and a notification being sent to the home-owner's phone (several minutes), an ineffective motion sensor, poor camera quality and the jumbled visitor activity log.
    - I have theorised that using still camera images rather than videos may reduce the latency between the doorbell being rung and a notification being sent to the homeowner.

Ring Video Doorbell 4

Good Features:

- **High quality video camera** – 1080p HD colour camera, 160-degree field of view and good quality night vision.
- **Live streaming** – the view from the doorbell's camera can be livestreamed via an app.
- **Video and audio recordings saved** – all recordings are stored on the cloud for 7 days for the user to access.
- **Motion detection** – the homeowner is alerted even if the visitor doesn't press the doorbell button.
- **Face/person detection** – notifications only sent when a person is detected, so user doesn't receive unnecessary notifications.
- **Motion zones** – allows homeowner to draw lines around the image seen by the camera so that they are only notified when a person enters their property.
- **Two-way audio** – omnidirectional microphones allow the homeowner to communicate with the person at the door via the doorbell.
- **Integration with home automation** – allows the doorbell to be remotely voice controlled with systems such as Alexa.
- **Pre-roll video previews** – camera records video up to 4 seconds before motion is detected.
- **Photo preview notification** – photo of the person at the door is shown as a preview notification so the user doesn't have to open the app. Useful for those who work from home and may be in the middle of a conference call.
- **Neighbours feed** – allows users to connect to their neighbours' Ring doorbell to aid crime detection and prevention.
- **Rechargeable battery** – as the doorbell is powered by battery, it is easy to install and causes minimal visual disruption.

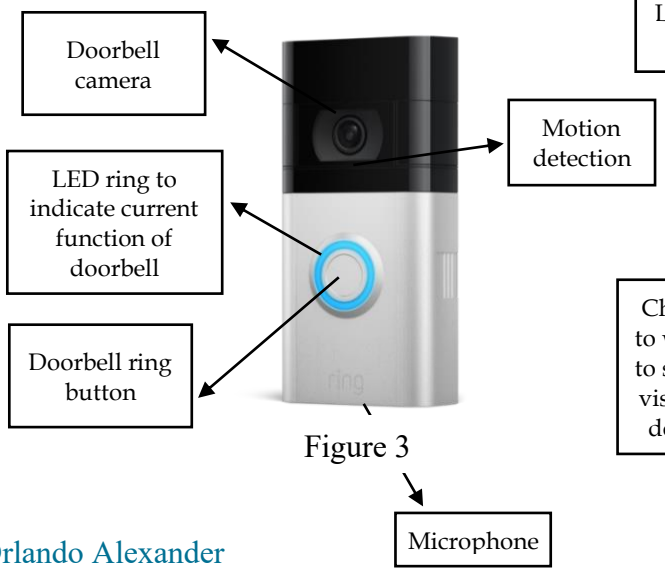


Figure 3

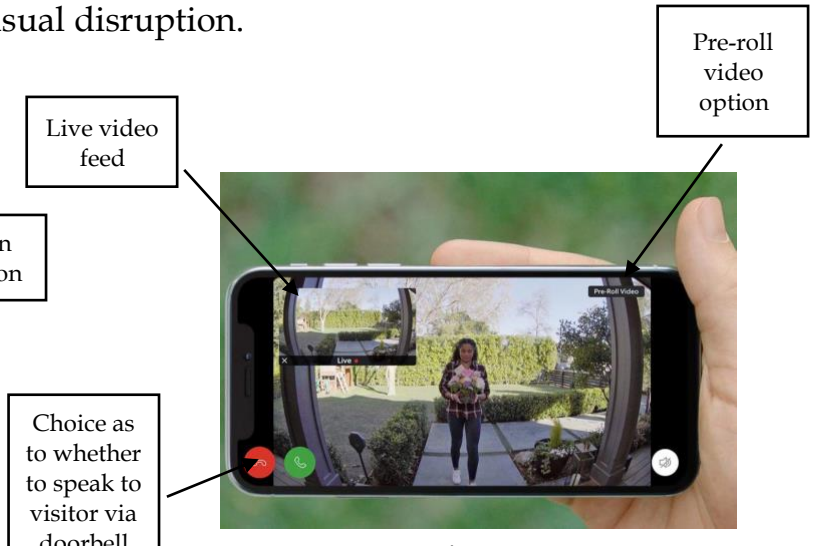


Figure 4

Less Good Features:

- **No facial recognition** – personalised audio messages for a client at the doorstep and notifications with the name of the client at the door are not possible.
- **No automated audio messages for visitors** – any communication with the visitor must be live audio from the homeowner, which may not be convenient if they are on a work call.
- **Fairly expensive** – the doorbell shown in *figure 3* costs £179.
- **Videos require lots of storage** – storing videos for 7 days of length 30-45 seconds each time motion is detected is costly in terms of storage.
- **Paid subscription required to store videos** – users must pay at least £2.50 per month per device to store video footage from the doorbell in the cloud for 30 days.
- **Videos are inefficient to identify visitor –** people who work from home may be in the middle of a conference call, so a video is a time-consuming method of identifying who is at the door (*figure 4*).
    - Sending a still image of the visitor to the homeowner's phone is a much more efficient method to allow the homeowner to identify them.
- **Some poor reviews** – several Amazon reviews identified connectivity issues, poor video quality, problems viewing the camera live and difficulties with having two-way communication.
    - Providing high quality still images may be more valuable than poor quality, live-streamed videos.
    - Automated and pre-recorded audio messages may be a more effective way to communicate with the visitor.

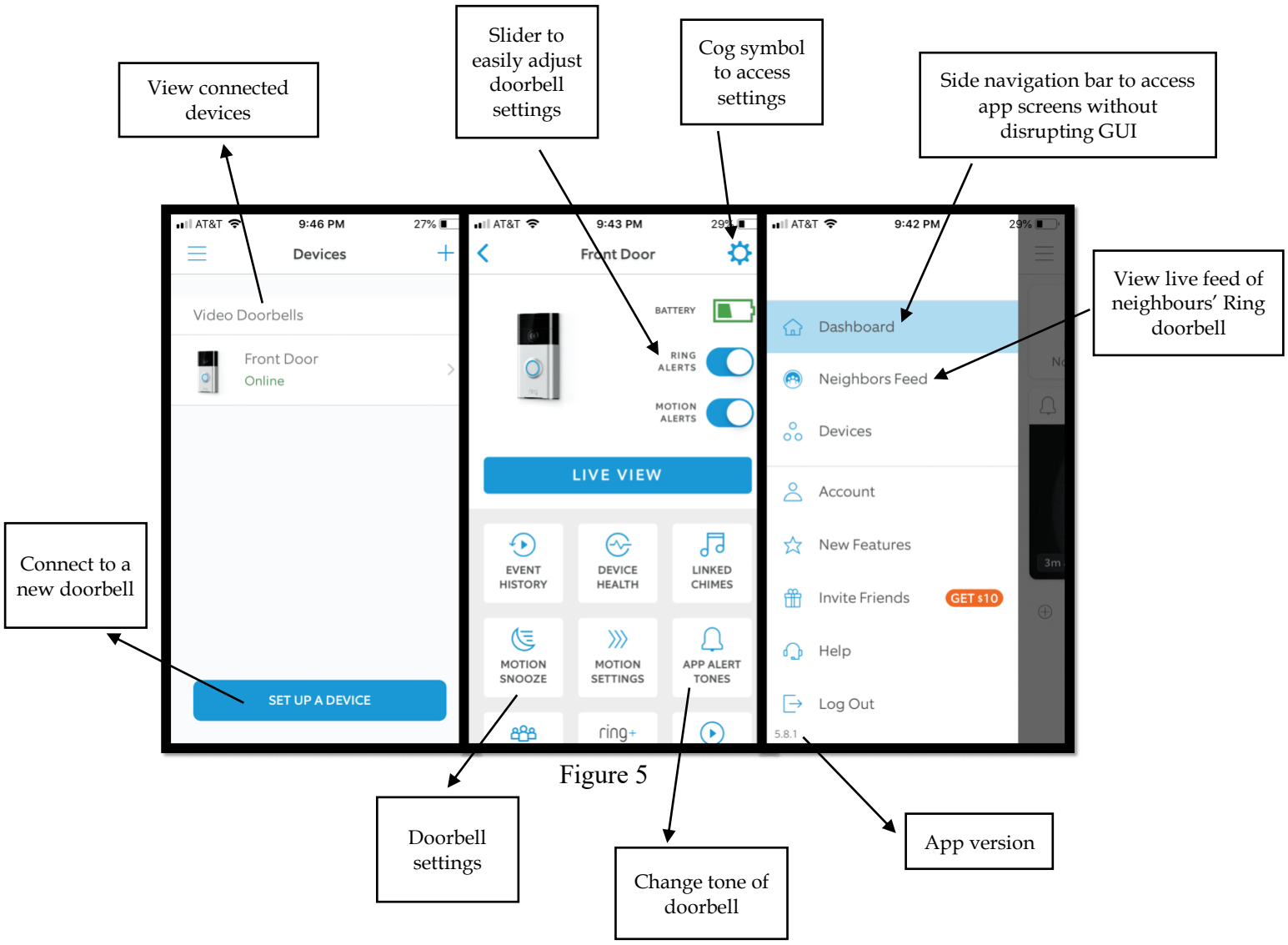


Figure 5

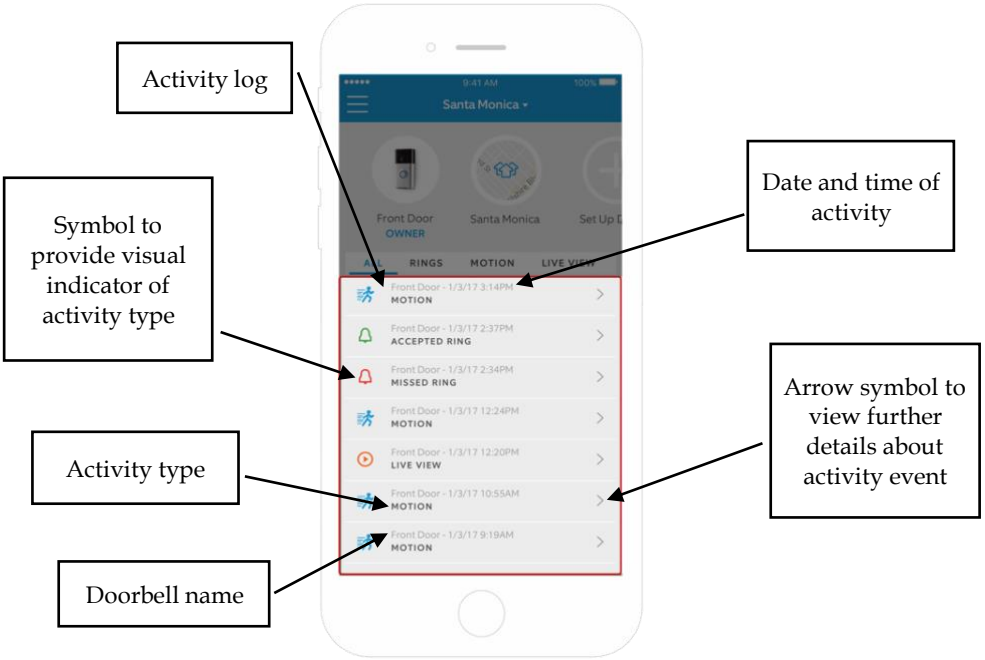


Figure 6

Nest Hello Doorbell

Good Features:

- **High quality video camera** – 1600 x 1200 HD colour camera, 160-degree field of view and good quality night vision.
- **Live streaming** – the view from the doorbell's camera can be livestreamed via an app.
- **Video and audio recordings saved** – up to 60 days' worth of event recordings are stored on the cloud for the user to access and users can record 24/7 for up to 10 days continuously.
- **Motion detection** – the homeowner is alerted even if the visitor doesn't press the doorbell button.
- **Face/person detection** – notifications only sent when a person is detected, so user doesn't receive unnecessary notifications.
- **Facial recognition** – the doorbell allows the user to teach it the names of faces it has already captured so that it can tell the user who is at the door the next time they come to the house.
- **Package tracking** – the doorbell recognises if a package has been dropped off or picked up from the doorstep, allowing the user to collect packages immediately or respond to package thefts.
- **Motion zones** – allows homeowner to draw lines around the image seen by the camera so that they are only notified when a person enters their property.
- **Two-way audio** – omnidirectional microphones allow the homeowner to communicate with the person at the door via the doorbell.
- **Integration with home automation** – allows the doorbell to be remotely voice controlled with systems such as Alexa.
- **Pre-Recorded Quick Response** – the user is able to record a selection of quick responses when setting up the doorbell which can be played.
- **Quiet mode** – doorbell's chime can be silenced so it doesn't disturb the homeowner.

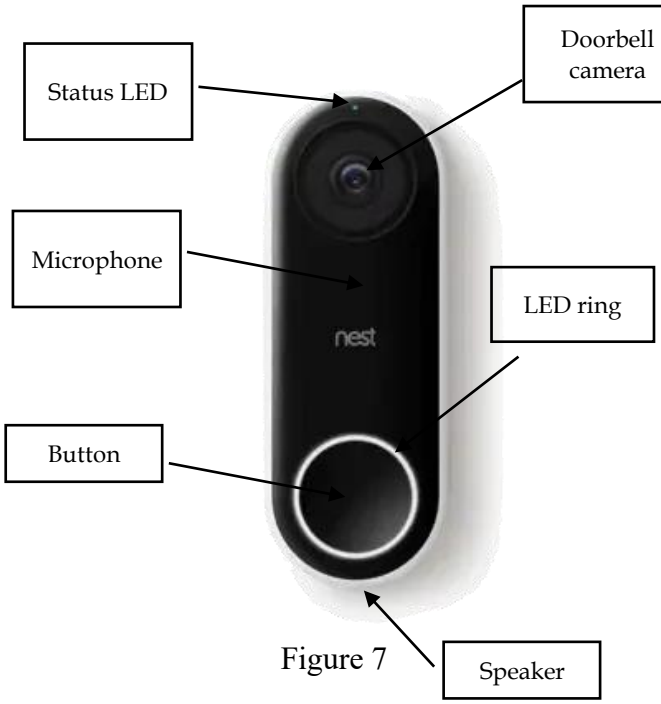


Figure 7

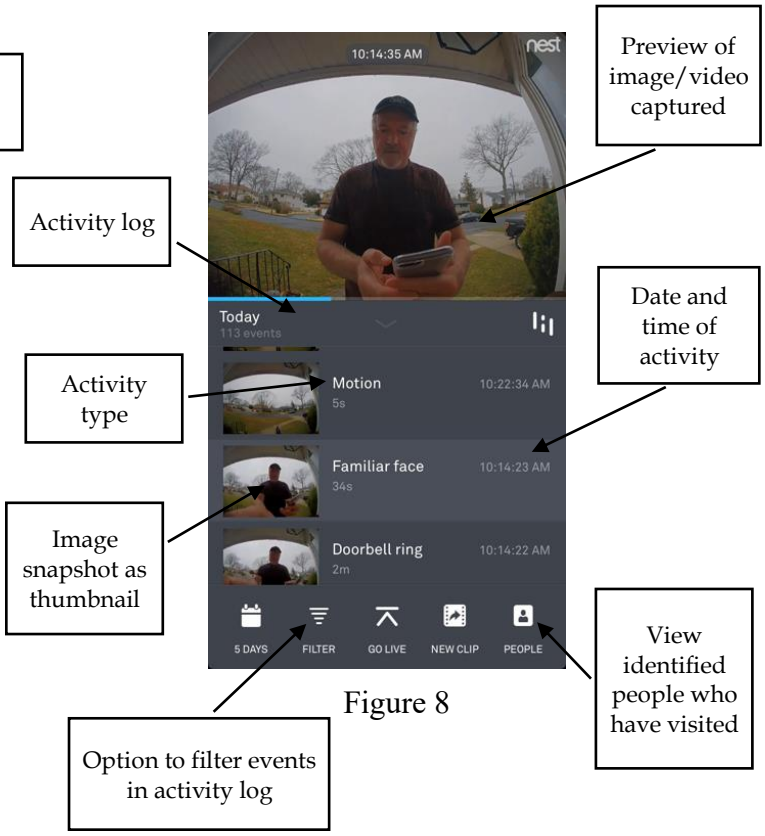


Figure 8

Less Good Features:

- **Fairly expensive** – the doorbell shown in *figure 7* costs £229.
- **Videos require lots of storage** – storing event recordings of up to 60 days and up to 10 days of 24/7 continuous recordings is costly in terms of storage.
- **Uploading 24/7 puts strain on bandwidth** – uploading video to the cloud 24/7 could slow down the internet speeds in your home, which is especially an issue for those who work from home.
- **Paid subscription required to store videos and use facial recognition** – users must pay up to £8.50 per month to store video footage from the doorbell and to use the doorbell's facial recognition feature.
- **Hardwired** – the doorbell requires wiring and drilling to set up, which generally induces extra costs for the homeowner and make the device less easy to use.
- **Videos are inefficient to identify visitor –** those who work from home may be in the middle of a conference call, so a video is a time-consuming method of identifying who is at the door (*figure 8*).
    - Sending a still image of the visitor to the homeowner's phone is a much more efficient method to allow the homeowner to identify them.

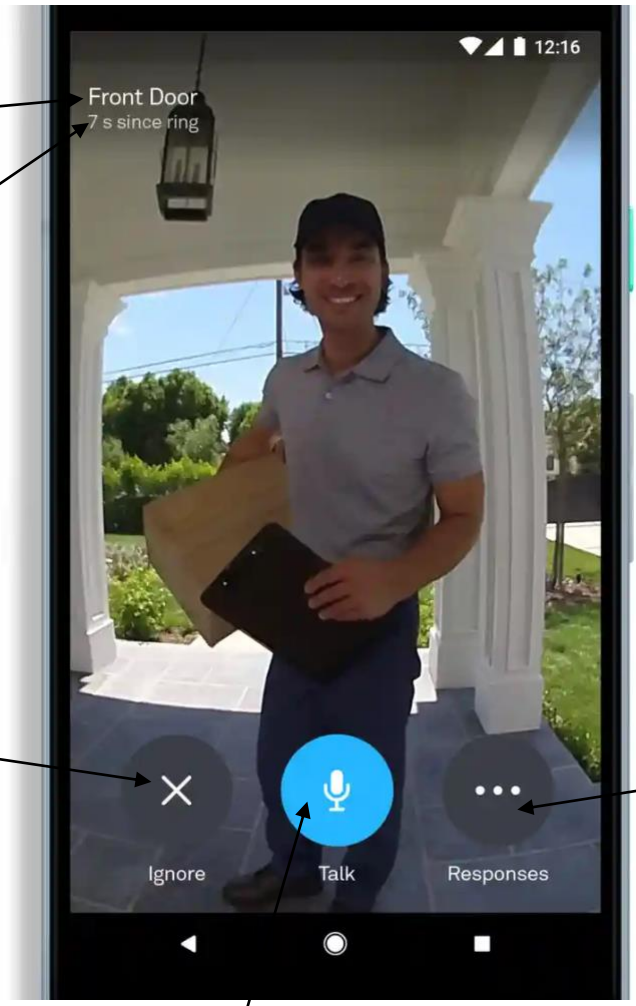

Doorbell name
Time since doorbell ring
Close live video from doorbell
Option to select from pre-recorded audio responses
Live audio transmission between app and doorbell
12:16
Front Door
7 s since ring
Ignore
Talk
Responses

Figure 9

Objectives and Technical Details of Proposed System

Considering my analysis of the features of existing video doorbell systems (*page 8*) in conjunction with the expected features (*page 7*) proposed by the primary client, Maria Kramer, I have determined that successful outcome of the project will be determined by satisfying the following objectives:

Objective	Technical Aspects
Users must be able to **create individual accounts** on the mobile app which will allow them to access and play their personalised audio messages through personalised shortcut buttons.	Users' details and personal data to be stored securely in a cloud-based database, hosted by AWS. They will only be able to access this data through correctly entering their email and password, which will be stored in a hashed format on the database.
User must be able to **pair their doorbell with their account** via a simple GUI (Graphical User Interface) on the mobile app.	User must be within Bluetooth range (around 10 metres) of the Raspberry Pi to link their account to their doorbell, as this ensures the user can only connect to their own doorbell. The doorbell will be assigned an ID that will be stored securely in a database alongside the user's account ID. In the future, the user will be able to access their doorbell's database from anywhere if they are signed into their account.
The user must be able to **record or type personalised audio message responses** (to be played through the doorbell) and create shortcuts to access them quickly through the mobile app.	To record audio messages, the *audiostream* module will be used. If the user chooses to type the audio messages, a text-to-speech module will be used to create audio files. These audio files will be stored on Amazon S3, with their unique names stored securely alongside the user's account ID in a MySQL table in the doorbell database.

The doorbell will attempt to **identify every individual** who presses the button.	When the button (input) connected to the Raspberry Pi via its GPIO pins is pressed, the camera (output) will capture an image. Each image will be stored securely on Amazon S3 under a unique identifier. I will use OpenCV's *Cascade Classifier* module and a locally-stored face Haar Cascade .xml file to determine whether a face can be identified in the image. If a face can be identified, I will use my facial recognition algorithm and OpenCV's *face* module to compare the visitor's face with faces in the trained data set. If a match is found, the name of the recognised face will be retrieved by using a MySQL command to query the cloud-based doorbell database. If the visitor can be identified, their name, along with the unique ID needed to download the captured image, will be sent to the mobile app by creating a socket between the Raspberry Pi and mobile phone. If the visitor cannot be identified, just the unique image ID will be sent to the mobile app.
When the doorbell is pressed, the user will receive a **notification through their phone**, along with the name of the visitor (if identified). When this notification is opened, the user will be shown an image of the visitor in the app.	Notifications will be automated and sent to the user's mobile app using OneSignal's notifications framework. The app GUI will be created using *Python*'s *Kivy* framework.
When visitor image is shown to user in the mobile app, the GUI will also have several large buttons to enable user to **select the desired audio message to be played through the doorbell.**	The message IDs for the selected audio message will be found by sending a query to the AWS environment, which will access the user's record in the cloud-based database. These message IDs will then be sent from the mobile app to the Raspberry Pi via a secure socket which the Raspberry Pi will be listening to continuously. The Raspberry Pi will output the audio messages via the speaker connected to its GPIO pins.
When the doorbell is pressed, details about the visit will be added to the **visitor log**, which will show the name of the visitor (if identifiable), the time of the visitor, the image of the visitor and an option to download the image to their camera role. This visitor log will be accessible via the mobile app, with the option to sort/filter which visits are displayed. Various summary statistics about the visits will also be displayed.	Using a REST API that I will create, the data about the user's visit will be sent securely to an environment running on AWS and stored in the doorbell database alongside the user's account ID. Moreover, the image associated with each visit will be stored in AWS S3 storage. This data will be formatted and displayed in a GUI in the app. I will also use aggregate SQL functions to display a variety of summary statistics about the visits and I will write a merge sort algorithm so that the visits can be viewed in different orders depending on the user's preference.

Following each doorbell ring where the visitor was not identifiable, the user will be given the option to **enter the name of the visitor via the mobile app**. The user will be required to ask the visitor for consent to store their tagged image. This will ensure that the correct name is assigned to the visitor's face the next time that it is detected.	The name of the visitor, as inputted by the user through the mobile app, and the unique ID for the visitor's face will be stored securely alongside the user's account ID in a MySQL table in the doorbell database.

Development plan

There are two primary development methodologies that I will consider:

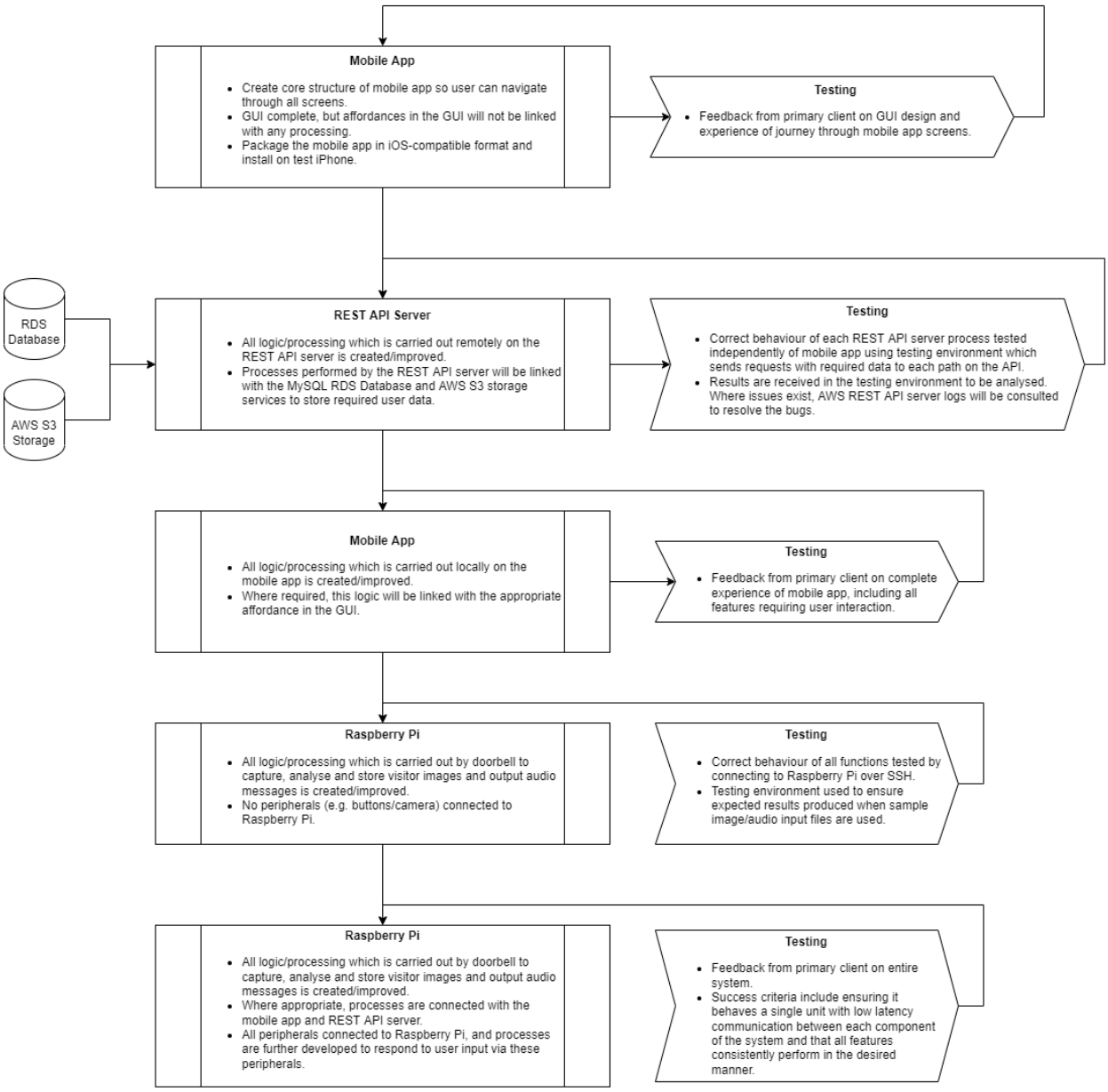
1. **Agile methodology**
2. **Waterflow methodology**

Agile	Waterflow
Iterative approach – development and testing occur simultaneously throughout development	**Sequential approach** – each stage of development is distinct and only progress to next stage once current stage is complete
Objectives can be altered during development	**Objectives** cannot be altered once development has begun
Flexible – adaptability of the project based on discoveries during development/changing client aims is key	**Structured** – all details of the development process are clearly set out at the start and followed precisely

Given the above comparisons, it is clear the **agile methodology** is appropriate where the specific design and structure of key objectives are not well understood prior to development. This is particularly the case where the project involves the development of a new product. As the SmartBell system is both a new product, and this is my first experience of the development process for a software project, I have decided to use the agile methodology throughout the development process.

Using the **agile methodology**, will allow me to easily incorporate new programmatic styles and techniques that I learn throughout the *Development* phase, ensuring the project is completed to the highest possible standard. Moreover, given my current lack of experience with client-focused development, continually requesting feedback from the primary client, and other possible users, and incorporating this feedback into future development (as is characteristics of the agile methodology) will be key to the success of the project.

Indicative of the agile methodology, the development process will include regular testing, both by the primary client and myself using testing environments. The proposed development process for the SmartBell video doorbell, including all in-development testing, is shown below:



Prospective Users and Acceptable Limitations

At present, there would be at most 4 users of the system: Maria Kramer, the primary client, her husband and her 2 sons. Everyone in Kramer's family is comfortable using technology and they already have several smart home devices, so their technical abilities shouldn't impose a limit to the design of the doorbell and mobile app. However, the purpose of the system is to reduce the interruption to those who work from home, so the app must be reliable and the graphical user interface must be easy and clear to follow.

The accepted limitations to the system are as follows:

- **Hardware and software constraints:**
  - Each user must have an iPhone to install the mobile app which controls the doorbell as I will be using Python to compile the app for iOS only. The Kramer family all us iPhones so this is not an issue:
    - I have chosen to only design the app for iOS as I my experience is with Xcode (the software used to compile and deploy apps for iOS devices) and I have previously deployed a *Kivy* app for iOS.
  - The iPhone must still be supported by Apple and so must be newer than the iPhone SE.
  - The doorbell can only be controlled by 1 device/user at a time. Connectivity issues may arise if 2 users attempt to connect to the doorbell at the same time.
  - There must be a strong enough WiFi signal on the doorstep to enable the Raspberry Pi in the doorbell to communicate with the mobile app.
  - The user must have the technical abilities to wire up the doorbell to a power supply, or have access to someone who can do it for them
  - The Raspberry Pi must have WiFi, Bluetooth camera capabilities and must be small enough to fit inside a doorbell. The Raspberry Pi Zero W matches this description.
  - The Raspberry Pi will be running with Raspbian OS with at least 4GB RAM.

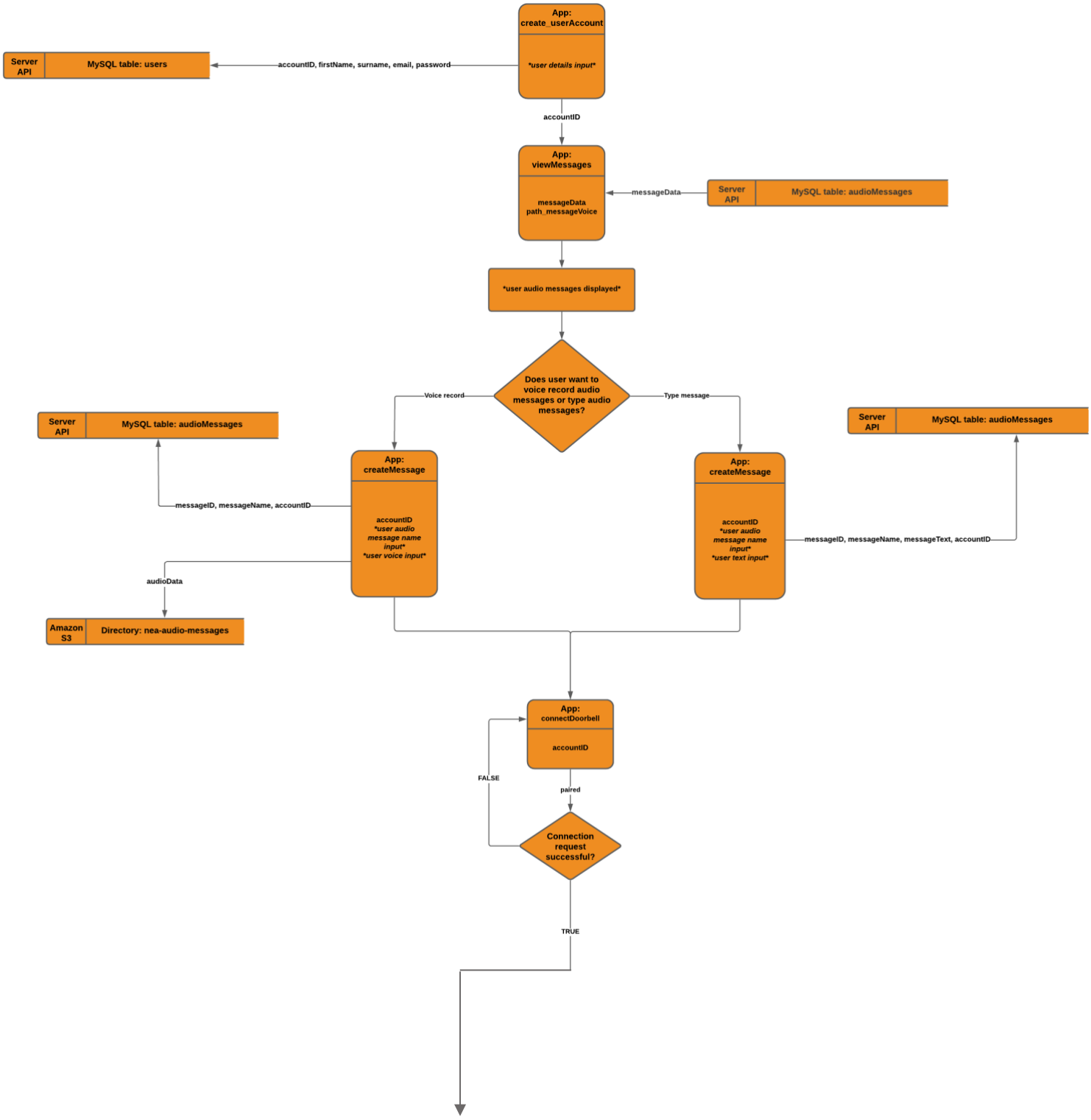
- **Accessibility constraints:**
  - The user must have the ability to type personalised text phrases *or* record personalised audio phrases which the doorbell will play on command. If the user opts for typing personalised text phrases, text-to-speech software will be used to output the audio message. This feature will make the doorbell system accessible for users who have speech impairments.
  - When the doorbell is unable to recognise the visitor, the user must have the visual capability to identify an unknown visitor without hearing their voice.
  - The mobile app will be made accessible for users with moderate visual impairments by using contrasting colours, large font sizes and minimalistic designs.

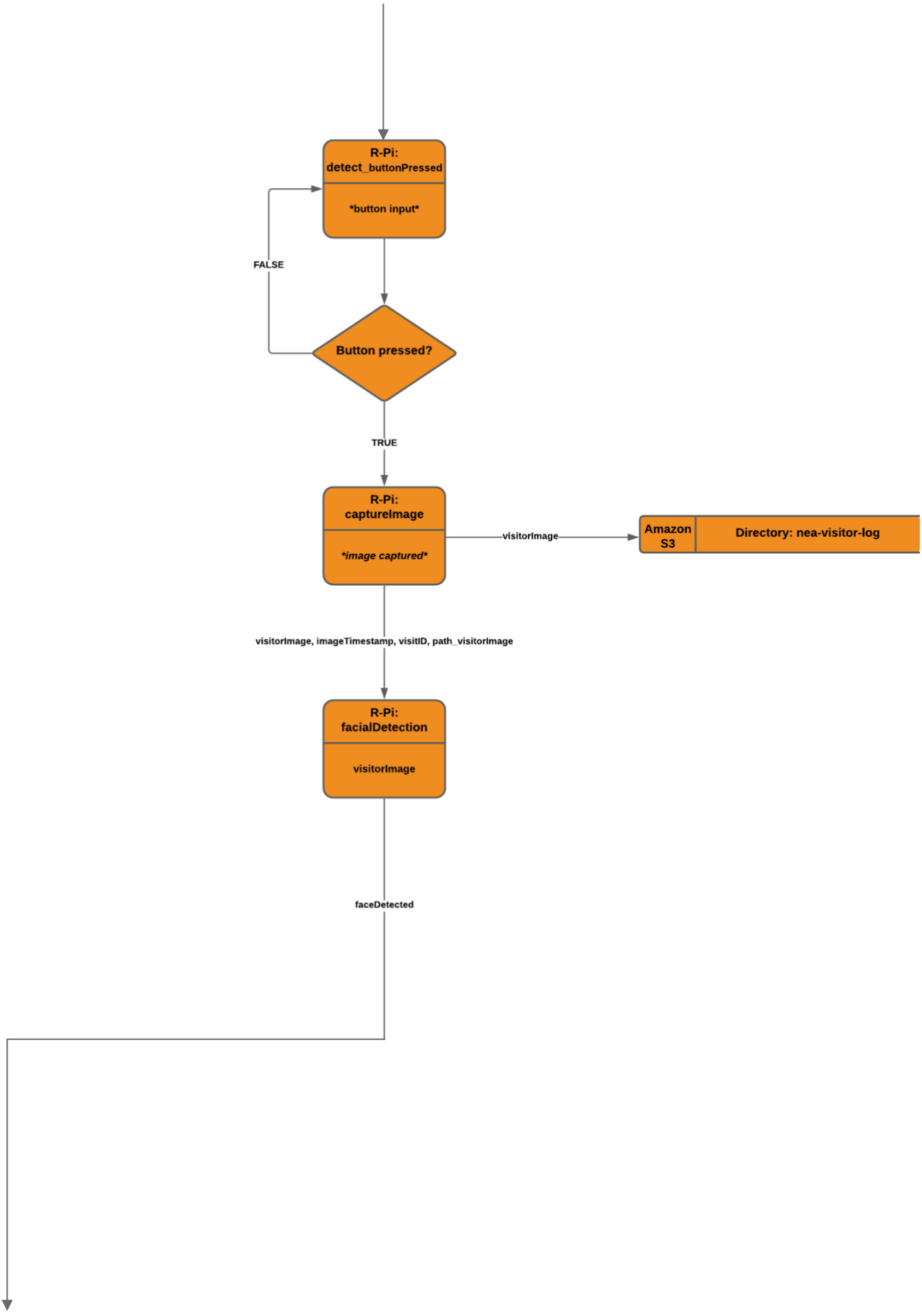
- **Functionality constraints:**
  - The doorbell system is designed to behave as an intelligent and digital home receptionist. Therefore, its functionalities will be optimised to allow people who work from home to easily and quickly communicate with someone at the door without interrupting their work. For example, the doorbell won't have video functionality as it is time consuming to watch a video, and a good quality photograph is easier to access than a poor-quality video. Instead, the user will simply be sent an image of the person at the door and will be given several automated and pre-recorded audio responses to select. Moreover, the user will only be notified when the doorbell is rung (there will not be a motion sensor), as the doorbell serves to identify who is at the door; it is not a security measure.
  - The doorbell system is designed to support those who work from home, so it will only be designed to work during daylight hours, and it will not be equipped with night-vision technology.

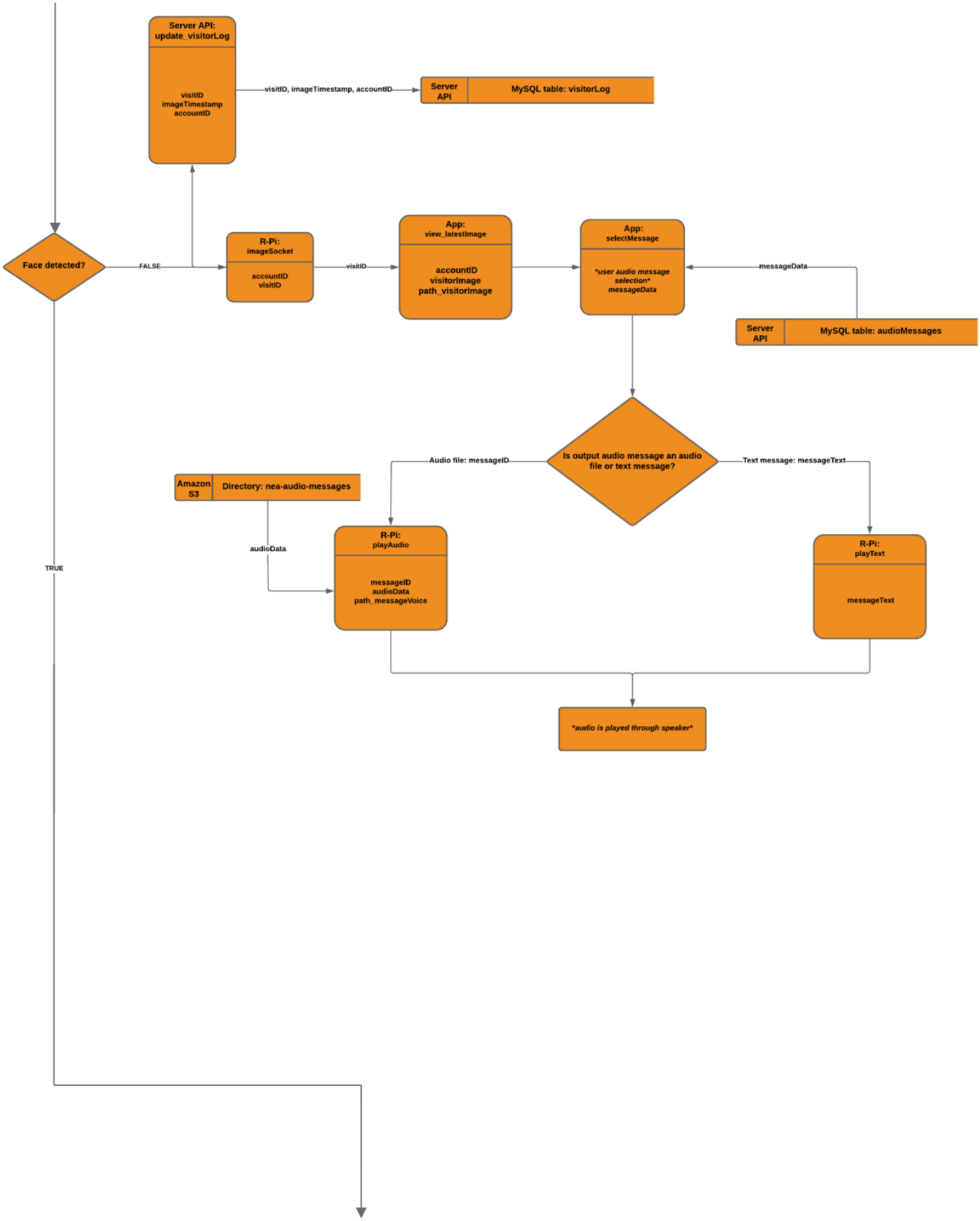
- **Legal constraints:**
  - The data in the private database which stores images of individuals' faces and their name is classed as personal data according to the Data Protection Act 2018.
  - The Data Protection Act states that storing personal details is acceptable so long as these details are solely for personal use and no members of the public can access them. A database which stores and tags images of people is very much like a contacts list on your mobile phone, which doesn't require consent from the individual.
  - However, the primary client, Maria Kramer, mentioned she would be willing to ask for consent to store and tag images of her clients/colleagues/friends.
  - Design decisions will be in line with the Data Protection Act 2018.

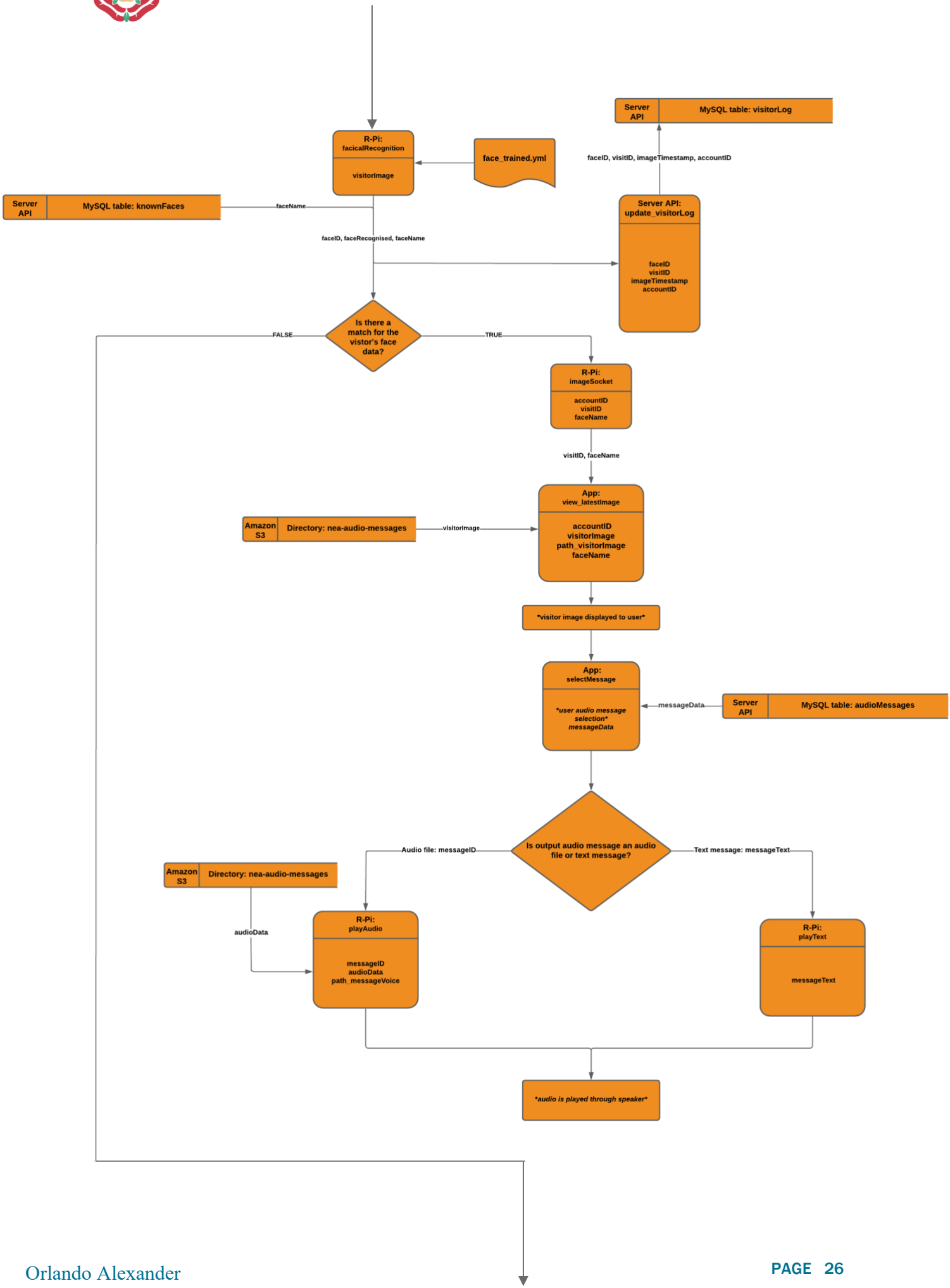
Data Flow Diagram

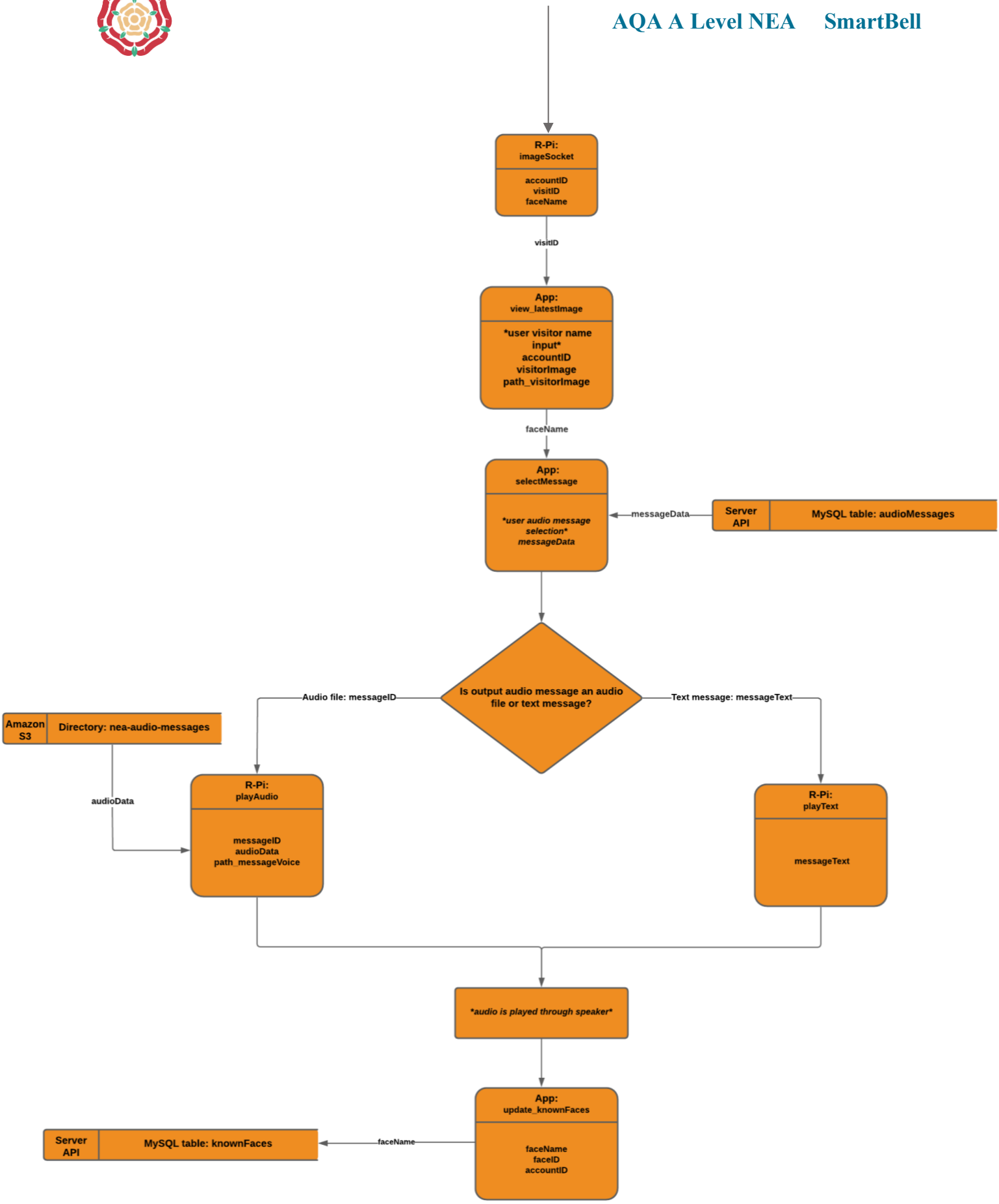
This data flow diagram shows the most direct route through the key functions/methods of the mobile app, Raspberry Pi, cloud-based Server REST API and Amazon S3 upon initial use of the app and doorbell. I will use this diagram throughout the *Development* phase to ensure the iterative development process remains focused around the key features of the system throughout.




R-Pi:
detect_buttonPressed
button input
FALSE
Button pressed?
TRUE
R-Pi:
captureImage
image captured
visitorImage
Amazon S3
Directory: nea-visitor-log
visitorImage, imageTimestamp, visitID, path_visitorImage
R-Pi:
facialDetection
visitorImage
faceDetected


Server API: update_visitorLog
visitID
imageTimestamp
accountID
visitID, imageTimestamp, accountID
Server API
MySQL table: visitorLog
Face detected?
FALSE
R-Pi: imageSocket
accountID
visitID
visitID
App: view_latestImage
accountID
visitorImage
path_visitorImage
App: selectMessage
user audio message selection
messageData
messageData
Server API
MySQL table: audioMessages
Is output audio message an audio file or text message?
Audio file: messageID
Text message: messageText
Amazon S3
Directory: nea-audio-messages
audioData
R-Pi: playAudio
messageID
audioData
path_messageVoice
R-Pi: playText
messageText
TRUE
audio is played through speaker

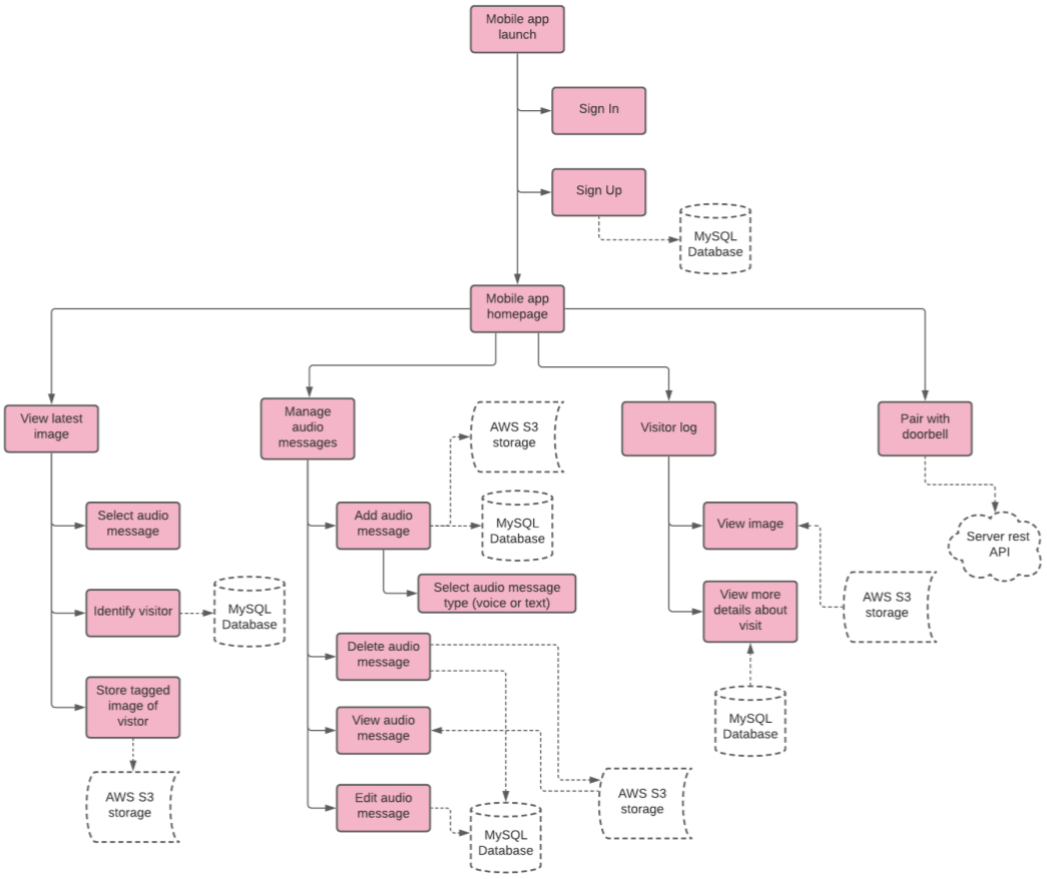

Server API
MySQL table: visitorLog
R-Pi: facicalRecognition
visitorImage
face_trained.yml
faceID, visitID, imageTimestamp, accountID
Server API
MySQL table: knownFaces
faceName
Server API: update_visitorLog
faceID
visitID
imageTimestamp
accountID
faceID, faceRecognised, faceName
Is there a match for the vistor's face data?
FALSE
TRUE
R-Pi: imageSocket
accountID
visitID
faceName
visitID, faceName
App: view_latestImage
Amazon S3
Directory: nea-audio-messages
visitorImage
accountID
visitorImage
path_visitorImage
faceName
visitor image displayed to user
App: selectMessage
user audio message selection
messageData
Server API
MySQL table: audioMessages
messageData
Is output audio message an audio file or text message?
Audio file: messageID
Text message: messageText
Amazon S3
Directory: nea-audio-messages
audioData
R-Pi: playAudio
messageID
audioData
path_messageVoice
R-Pi: playText
messageText
audio is played through speaker


R-Pi:
imageSocket
accountID
visitID
faceName
visitID
App:
view_latestImage
user visitor name input
accountID
visitorImage
path_visitorImage
faceName
App:
selectMessage
user audio message selection
messageData
messageData
Server API
MySQL table: audioMessages
Is output audio message an audio file or text message?
Audio file: messageID
Text message: messageText
Amazon S3
Directory: nea-audio-messages
audioData
R-Pi:
playAudio
messageID
audioData
path_messageVoice
R-Pi:
playText
messageText
audio is played through speaker
App:
update_knownFaces
faceName
faceID
accountID
faceName
Server API
MySQL table: knownFaces

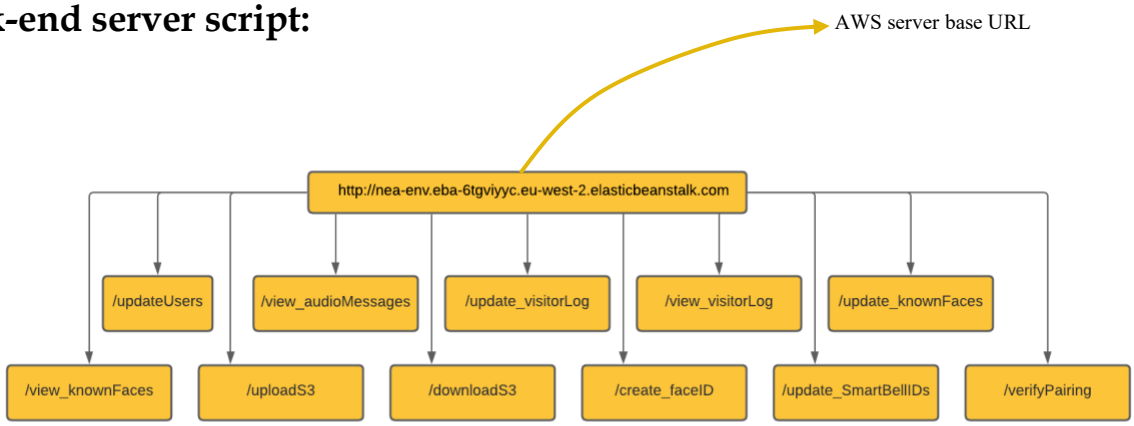
Design

System Hierarchy

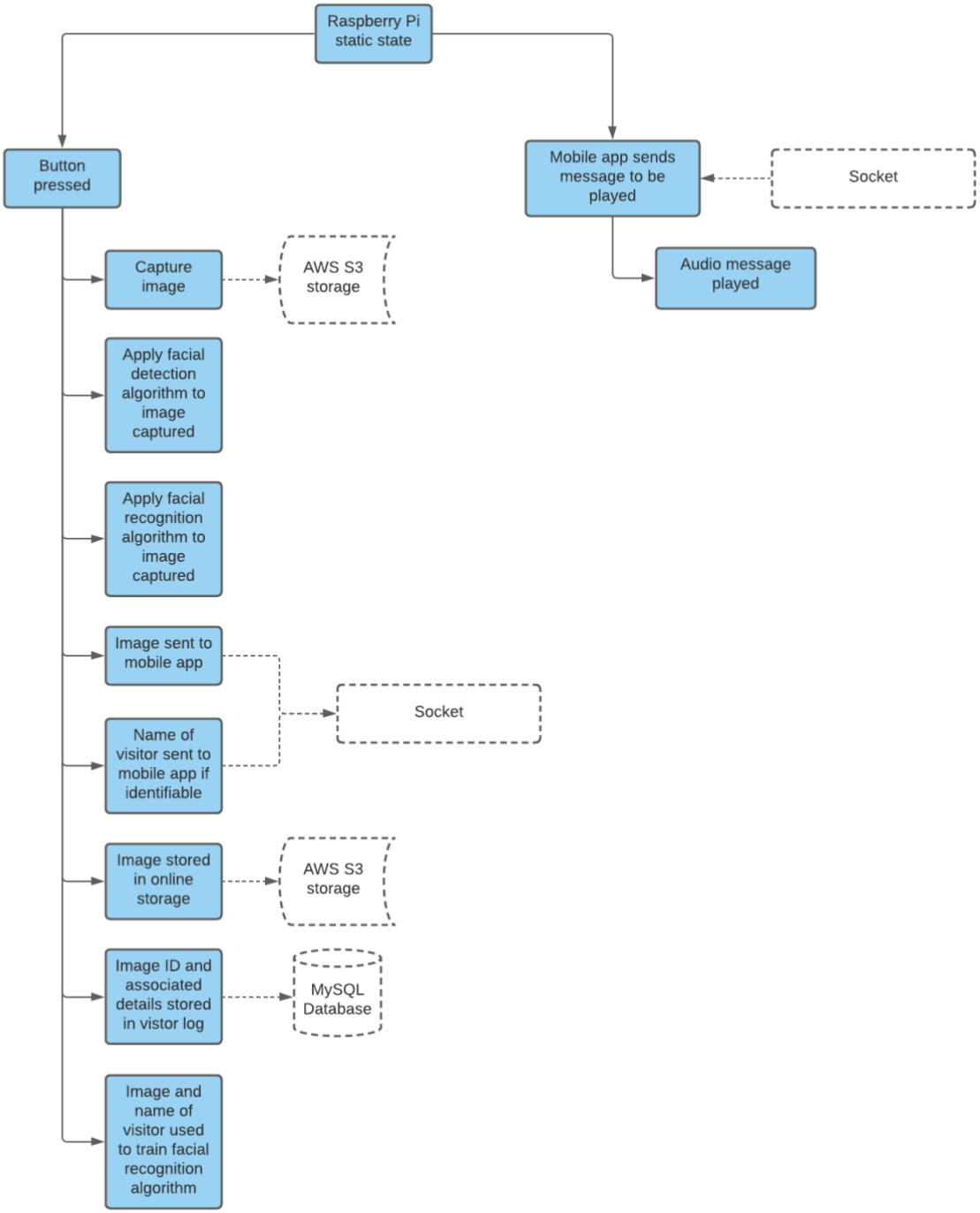
Mobile app script:



Back-end server script:



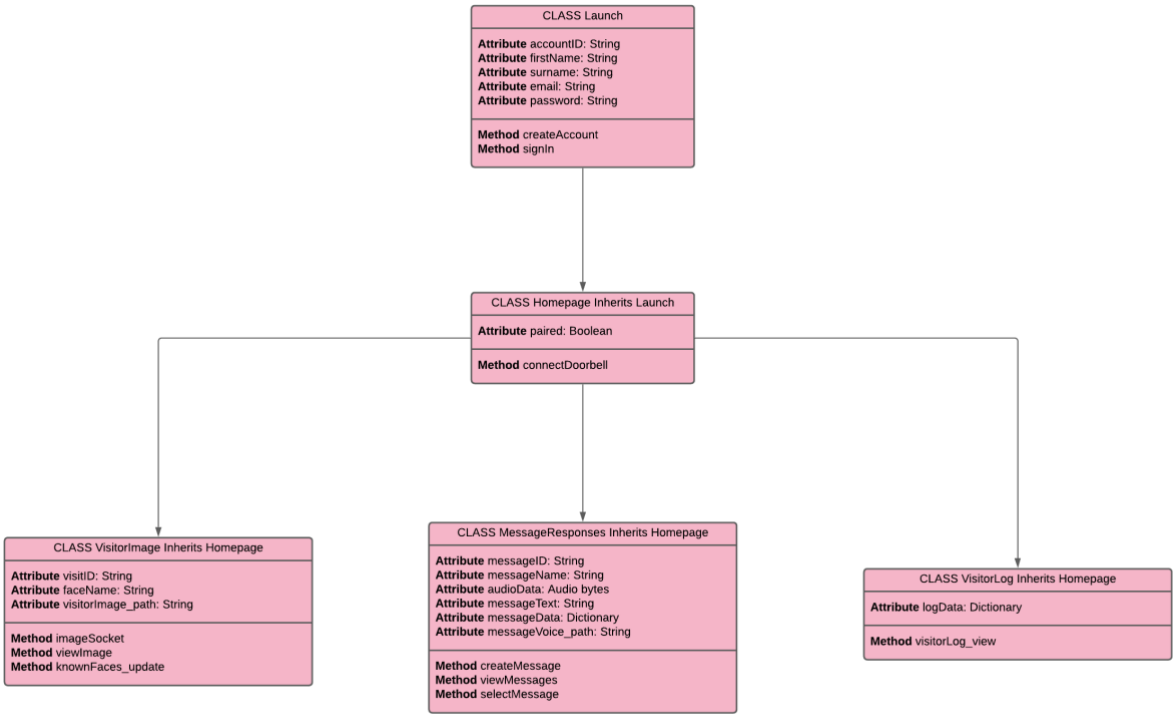
Raspberry Pi script:



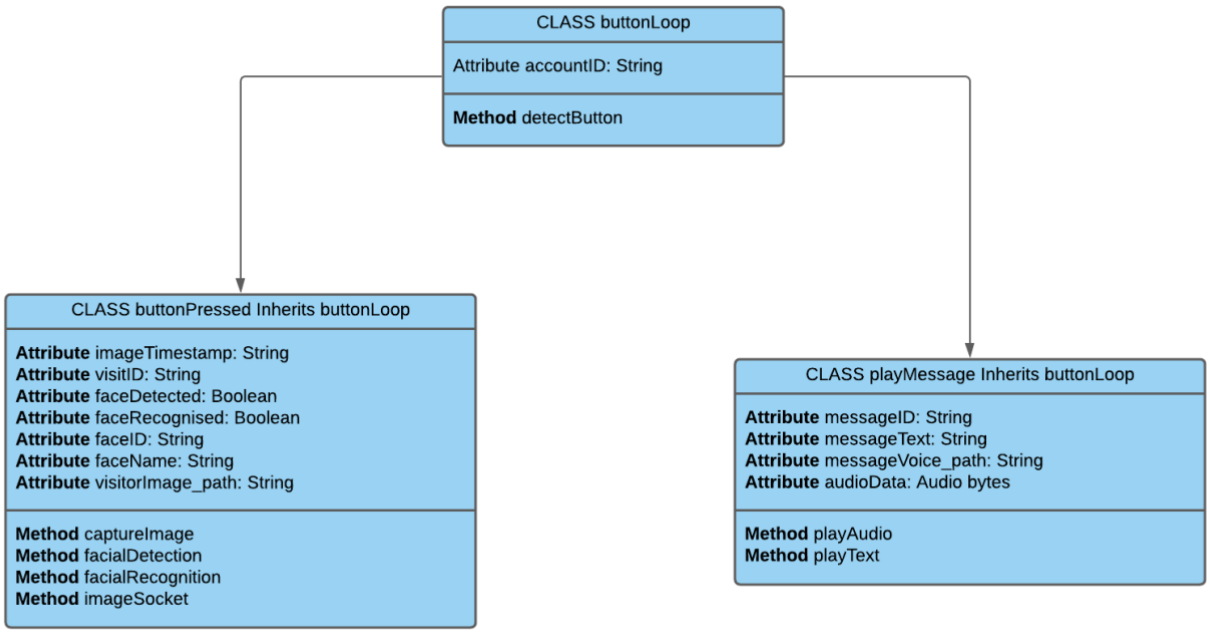
UML Class Diagrams

All classes have been named used camel case notation.
All methods have been named using the following notation: *myMethod* or *myMethod_name* or *myMethod_nameConvention*.

Mobile app:



Raspberry Pi:



Data Dictionary - Database

Table Name	Primary key	Foreign key	Fields	Justification
users	accountID	n/a	firstName surname email password	Stores the personal details of each user (*firstName, surname, email* and *password*) alongside their unique *accountID*. These details are the only means of accessing the user's *accountID*, and therefore the data in the other 4 tables which have *accountID* as a foreign key.
knownFaces	faceID	accountID	faceName	Stores the *faceName* (if identified) of each visitors' *faceID*, and the associated *accountID*.
audioMessages	messageID	accountID	messageName messageText	Stores the *messageID* of each audio message alongside the *messageName* and associated *accountID*. If the audio message is a text message, the message text will be stored in the field *messageText*. Otherwise, the *messageID* can be used to access the audio message file stored on Amazon S3.
visitorLog	visitID	accountID faceID	imageTimestamp confidence	Stores the *visitID* of each visit alongside the data about the visit (*imageTimestamp, faceID* and *confidence*) and the associated *accountID*. This data allows the visitor log to be displayed to the user through the mobile app.
SmartBellIDs	id	accountID	n/a	Stores the unique *id* of each Raspberry Pi doorbell alongside the *accountID* currently paired with that doorbell.

AccountID primary key

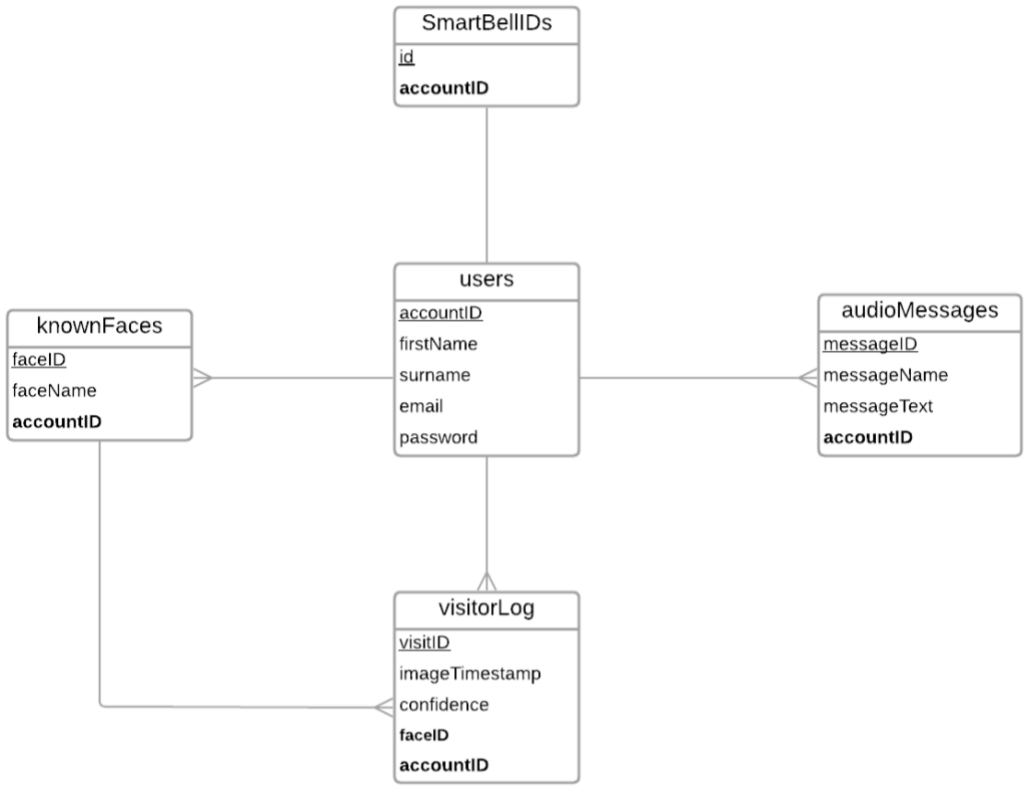
The identifier a*ccountID* is assigned to each user and is used as a primary/foreign key in each table of the MySQL database, as they all store personal data relating to the user. This ensures that the doorbell is customised for each user's needs and prevents breaches of the Data Protection Act 2018.
A hashing algorithm will be used to create a unique *accountID* value for each user which can only be accessed if the user enters the correct password and email, meaning that only authenticated users can access their personal data in the databases.

Hashing algorithm

I will use SHA3-256 to create a hash of each user's plaintext password before it is stored in the table **users**, as SHA3-256 is 1 of the most secure hashing algorithms. Using a hashing algorithm to create a hash of the user's password ensures users' data is secure and cannot be accessed by unauthorised users, avoiding the risk of breaches of the Data Protection Act 2018.

When the user initially enters their password through the mobile app, the SHA3-256 algorithm will be applied to the string entered and the hashed output from this algorithm will be stored in the MySQL database in the field *password* in the **users** table. The plaintext password value entered by the user will not be stored anywhere. When the user logs into the mobile app, the same hashing algorithm will be re-applied to their password input and the hashed output will be compared to the hashed value stored in the *password* field in the **users** table. If the values match, the value of *accountID* will be sent to and stored on the mobile app, allowing the user to access their personal data from the MySQL database. I have chosen to use hashing, rather than encryption, to store the user's password, as a hash cannot be decrypted back to its original format, whereas an encrypted value can be. This makes hashing a much more secure means of store sensitive data.

Entity Relationship Diagram – 3rd Normal Form



All the tables in the MySQL database are linked together using the primary key of the **users** table (*accountID*) as a foreign key in all the other tables (**SmartBellIDs**, **audioMessages,** **visitorLog** and **knownFaces**). This link means that all the data stored in the *SmartBell* database is specific for each user, ensuring that the **user's experience is personalised** (for example, they are able to store and access only their own audio responses) and that **data protection regulations are followed** (for example, the names associated with visitors' face encodings can only be accessed by the user who owns the doorbell that they visitor rung).

My **normalisation process** was as follows:

1st Normal Form
- Each table has sufficient fields such that all data stored is **atomic**
  - For example, the details about the user's name are separated into their *firstName* and *surname*
- All primary keys will be unique ID strings, ensuring that each record is unique

- **2nd Normal Form**
  - All the non-primary key attributes in each table are **entirely dependent on the whole primary key**
  - As each table relates to only one data group, all the record attributes will be entirely dependent on the unique primary key for their data group
    - For example, consider a record in the table **audioMessages**: *messageName, messageText* and *accountID* (the non-key attributes) are specific to the *messageID* (the primary key) they relate to

- **3rd Normal Form**
  - There are **no transitive functional dependencies** as all of the non-primary key attributes determine the value of another non-primary key attribute
  - I can be certain of this because the attributes of each record are neither factual nor immutable. The data stored in each cell is verifiable 'correct' (within the bounds of the system) on the basis that it exists, so all the non-key attributes are solely dependent on their indicator of existence: their primary key.
    - For example, consider a record in the **visitorLog** table: the *confidence* value for the *faceID* of the visitor as determined by the Raspberry Pi is absolute as it is a fact about an event that has occurred.

SQL Queries

To interact with the tables in the MySQL database, **SQL queries** must be executed. Typically, there are four types of operations which can be performed on a database:

1. **Create**
   - Insert new data into database
2. **Read**
   - Select/query specific data values from database
3. **Update**
   - Change the value of a specific data value in the database
4. **Delete**
   - Remove a specific data value from the database

All four of these types of operations will be used throughout my project to store and access the required data values for each user. Below are examples of each type of query:

Create

The **Create** operation uses the *CREATE* statement to insert a new record with a unique primary key into the MySQL database:

```
INSERT INTO users(accountID, firstName, surname, email, password) VALUES
('myAccountID', 'Maria', 'Kramer', 'email@address.com', 'myPassword')
```

Database output

```
Query 1 OK: 1 row affected
```

The above SQL creates a new record in the *users* table with primary key *accountID* value **myAccountID**. This will be used to create and store the details for new user accounts in the MySQL database, so that users can log into their account from any device anywhere in the world.

Read

SELECT

The most basic type of **Read** operation uses the *SELECT* command to select the data values from a specified selection of fields. As the required data will generally be user-specific, the *WHERE* command can be used to filter the data values returned to only include those from records which a certain value for a specified field:

```
SELECT faceID FROM knownFaces WHERE faceName = 'Maria' AND accountID =
'myAccount'
```

Database output

faceID
myFaceID

The above SQL query returns the data value stored in the field *faceID* in the table *knownFaces* for the record with the following field values:

faceName = **Maria**
accountID = **myaccount**

SELECT EXISTS

Another permutation on the standard *SELECT* command that I will be using during my project is the *SELECT EXISTS* command, which returns *'1'* if at least one record exists which satisfies the selection criteria specified by the *WHERE* clause, and *'0'* if not. This is useful to verify whether a specific record exists for a specific user's account before a query is made to retrieve data from that query. This will serve as a means of **error prevention**, as this allows selection statements to implemented which ensure a query is only made if a piece of data exists within the database, avoiding any errors from arising.

```
SELECT EXISTS(SELECT * FROM SmartBellIDs WHERE accountID = 'myAccount')
```

Database output if record **exists**

EXISTS(SELECT * from SmartBellIDs WHERE accountID = 'myAccount')
1

Database output if record **doesn't exist**

EXISTS(SELECT * from SmartBellIDs WHERE accountID = 'myAccount')
0

The above SQL query returns *'1'* if there exists a record in the *SmartBellIDs* table with following field value:

accountID = **myAccount**

If this record does not exist, then the SQL query returns *'0'*.

SELECT […] GROUP BY […] HAVING COUNT […]

I will also use **aggregate SQL** functions to retrieve summary data from my MySQL database. Aggregate functions perform a calculation on multiple values in the database and return a single result. I will use the *GROUP BY* function in conjunction with the aggregate function *COUNT* to retrieve any duplicated face name values associated with one user's account from the database. This will provide the facial recognition training algorithm with the data required to improve the training data set whereby a user indicates that a particular visitor's face has been assigned the incorrect name by the facial recognition algorithm:

```
SELECT faceName FROM knownFaces WERE accountID = 'myAccount' GROUP BY faceName
HAVING COUNT (faceNAME) > 1
```

Database output

faceName
duplicateFaceName

The above SQL query returns all values of *faceName* from the records which satisfy the following conditions (note the hierarchical structure of the conditions):

accountID = **myAccount**
↳ The value of *faceName* appears in more than 1 record (in which *accountID* = **myAccount**)

The hierarchical structure of the selection conditions ensures that *GROUP BY […] HAVING COUNT […]* aggregate function is only applied to records associated with the user's account.

Explanation of the selection criteria:

- *GROUP BY*
  - Returns values (i.e. *faceName*) based on the field *faceName*. As such, each unique value of *faceName* associated with a user's account is returned, rather than all the values of *faceName* associated with a user's account
- *HAVING*
  - Much like the *WHERE* clause, the *HAVING* clause applies a selection criteria to the *GROUP BY* clause, restricting the rows affected by the *GROUP BY* clause
- *COUNT*
  - This aggregate SQL function returns the total number of records with the same value of *faceName* for a particular value of *faceName* associated with the user's account

Used together, these three statements return a single value of *faceName* for each *faceName* associated with a user's account which appear more than once in the database.

SELECT AVG

Another **aggregate SQL** function that I will use is the *AVG* function, which returns the average value in a field storing numerical data values. I will use the *AVG* function to find the average time of day at which visitors ring the user's doorbell:

```
SELECT AVG(SUBSTRING(imageTimestamp,1,5)) FROM visitorLog WHERE accountID =
'myAccount'
```

Database output

AVG(substring(visitID,1,5))
8.833333333333334

The above SQL query returns a float of the average value stored in the first five characters of each data item in the *imageTimestamp* field which satisfy the following condition:

accountID = **myAccount**

The SQL function *SUBSTRING* is used to extract the first five characters stored in the *imageTimestamp* field, as these characters store the time in 24hr format at which the visitor rang the doorbell.

SELECT COUNT

A further **aggregate SQL** function that I will use is the *COUNT* function, which returns the total number of occurrences of each data value in a particular field. I will use the *COUNT* function to find the total number of times a user's doorbell has been rung (i.e. total number of visits), so that the average number of visits per day can be determined.

```
SELECT COUNT(*) FROM visitorLog WHERE accountID = 'myAccount'
```

Database output

COUNT(visitID)
3

The above SQL query uses the *COUNT* function to return an integer representing the total number of records in the table *visitorLog* which satisfy the following condition:

accountID = **myAccount**

SELECT MIN

The final **aggregate SQL** function that I plan to use is the *MIN* function, which returns the minimum value stored in a particular field. I will use the *MIN* function to find the date/time at which the doorbell was first rung. Using this data, and the present date/time, along with the total number of visits (as determined using the *COUNT* function above), it is possible to determine the average number of times a user's doorbell is rung per day.

```
SELECT MIN(SUBSTRING(imageTimestamp,7)) FROM visitorLog WHERE accountID =
'myAccount'
```

Database output

MIN(substring(imageTimestamp, 7))
1645328756.7732236

The SQL function *SUBSTRING* is used to extract the characters, starting at the seventh character, stored in the *imageTimestamp* field, as these characters store the number of seconds that have passed since January 1, 1970, 00:00:00 (UTC) at the instance the doorbell is rung. This data can be used to extract the time and date at which the doorbell was rung.

The above SQL query uses the *MIN* function to return a float which stores the minimum substring value in the field *imageTimestamp* (as specified by the SQL *SUBSTRING* function) which satisfies the following condition:

accountID = **myAccount**

Update

The **Update** operation uses the *UPDATE* statement to alter the value of an existing data entry in the MySQL database. As these alterations must be made to a specific data cell, the *SET* clause is used to specify which field is being altered and provide the new value, and the *WHERE* clause is used to specify which record the change should be applied to:

```
UPDATE SmartBellIDs SET accountID = 'myAccount' WHERE id = 'myDoorbell'
```

Database output

id	accountID
myDoorbell	myAccount

The above query sets the value of the field *accountID* to **myAccount** for the record with *id* field value **myDoorbell** in the table *SmartBellIDs*. This will be used to store the pairings between user accounts and doorbell IDs in the MySQL database.

Delete

The **Delete** operation uses the *DELETE* statement to remove a particular record from the MySQL database. To specify which record is to be removed, the *WHERE* clause is used:

```
DELETE FROM audioMessages WHERE messageID = 'myMessageID'
```

Database output

```
Query 1 OK: 1 row affected
```

The above SQL query removes the record from the table *audioMessages* which has field *messageID* value **myMessageID**. This will be used to delete user's audio messages from the database.

Data Structures

I intend to use two key data structures throughout the project:

Json files to store required user data and status indicators locally on the mobile app and on the Raspberry Pi doorbell. This cached data on the mobile app and doorbell will ensure that the *SmartBell* system continues as normal after the user closes the app/power off the doorbell. For example, the *json* file stored on the mobile app will ensure that the user will not be logged out of their account or lose pairing connections. For the doorbell, the *json* file will store several details relating to the doorbell's status, as well as data about the paired user account and the face IDs which the facial recognition algorithm has been trained on.

```
{
   "training": "False",
   "id": "NEA",
   "accountID": "myAccount",
   "myAccount": {
      "faceIDs": [
            "faceID1", "faceID2", "faceID3", "faceID4"
       ]
    }
}
```

The example *json* file above shows the proposed design for the *json* file stored by the Raspberry Pi. There are several key features of this *json file*:

i. Each **json object** is enclosed by *curly brackets* ({})
ii. Within each json object, there are several related **name-value pairs** separated by commas (,). The names must be of *string* type, while the values, which are separated from the names by a colon (:) can be of any data valid data type.
iii. The *json file* is **three dimensional**:
    a. **First dimension** stores several values relating to properties of the doorbell at any moment in time (i.e. the training status, the doorbell's id and the account ID which the doorbell is paired with)
    b. **Second dimension** stores data which is specific to a particular account ID which the doorbell is/was paired with. This second dimension is created by assigning the value to a new json object. This data (the face IDs) must be stored within a separate json object as it is not a property of the doorbell; the data is only associated with one account ID, and therefore it must be possible to it alongside the same type of data belonging to a separate account ID without overriding it
    c. **Third dimension** is an array enclosed in square brackets ([]) which stores the face IDs associated with each account ID that the doorbell has been paired with during its existence.
iv. The multi-dimensional nature of the *json file* means that it has a tree-like structure.

Json objects are used to transmit data between the mobile app/Raspberry Pi and the REST API server. This will be necessary to enable the low latency communication between each component of the network.

```
{
   "bucketName": "nea-audio-messages",
   "s3File": "messageID"
}
```

The example *json object* above shows the proposed design for the *json objects* which transmit data to/from the API server. The structure is very much like that of the *json* file, however, unlike a *json* file, the data is stored as an object and is never converted into a file format.

Data Dictionary - Variables

The table below explains the fundamental variables that will be used in the project.
All variables have been named using the following notation: *myVariable* or *myVariable_name* or *myVariable_nameConvention*.

Data variable	Data type	Validation	Sample	Justification
firstName	String	n/a	"Maria"	Allows multiple users to create an account which is connected to the same doorbell.
surname	String	n/a	"Kramer"	Allows multiple users to create an account which is connected to the same doorbell.
email	String	Contains '@' and a period Must be unique	"maria@kramer.com"	Allows multiple users to create an account which is connected to the same doorbell.
password	String	=>8 digits, =>1 number, => 1 lowercase letter, => 1 uppercase letter, => 1 special character	"P@55w0rd"	Allows multiple users to create an account which is connected to the same doorbell. This plaintext value of password entered in the mobile app will be deleted once its hashed value is created and stored in the MySQL database.
accountID	String	43 characters (uppercase and lowercase letters and digits) Must be unique	"7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8FbO="	A unique identifier is assigned to each user so that each table in the MySQL database can store data (such as known faces, audio messages and a visitor log) which is specific for each user. This ensures that the doorbell is customised for each user's needs and prevents breaches of the Data Protection Act 2018. The user's *accountID* will also be used with Fernet encryption to symmetrically encrypt keys during transmission between the server and

				Raspberry Pi/mobile app. Therefore, to comply with the standard format for Fernet encryption keys, the *accountID* must be 43 characters in length (uppercase and lowercase letters and digits).
audioData	Audio bytes list	n/a	[b'\x00\x00\x00\x00\xfe\xffo\x00\xb6\x00g\x00d\x00q\x00\x89\x00s\x00R\x00?\x00\x1c\x00\x0e\x00%\x00\x1c\x00\x1c\x00\xe7\xff\xbf\xff\xa7\xff\xbf\xff\xaa\xff\xc4\xff\x9f\xffs\xff\x8e\xff\xac\xff\x9c\xffm\xffa\xffC\xff….]	Allows user to record custom audio responses on mobile app, which can be saved and outputted by the Raspberry Pi doorbell on command via the mobile app.
messageText	String	n/a	"Sorry, I'm on a call right now. Could you leave the parcel behind the green bin?"	Allows user to input and store custom responses which can be played through the doorbell's speaker using a text-to-speech module when a visitor comes to the door.
messageName	String	0<length<14	"Delivery"	Allows user to create shortcuts which they can use to select certain audio messages through the mobile app to be played through the doorbell when a visitor rings the doorbell.
messageVoice_path	String	n/a	"/private/var/mobile/Containers/Data/Application/428829E4-07B3-4A33-B6BE-2A499EA875A4/Documents/audioMessage_tmp.pkl"	Stores the path to the audio message file so that the user can preview it on their mobile phone, or the Raspberry Pi can output it via its speaker.
messageID	String	16 characters in length Must be unique	"0iuymfFuNlDp2f1L"	Allows MySQL table **audioMessages** to store unique ID of each audio message file alongside the *messageName* and the *accountID* who the audio message belongs to. Where the audio message is a voice recording, this *messageID* can be used to

				access the audio file from Amazon S3. Where the audio message is a text message, the *messageText* will be stored in the MySQL table.
messageData	Dictionary	Format: {message number: [*messageID*, *messageName*, *messageText*], 'length': number of messages}	{'0': ['0EUyMFFuNlDp2f1L', 'message1', 'test1'], '1': ['hgSRCNcOOF8KREnv', 'Driver', 'Null'], '2': ['IFvbdpa67xXmKBes', 'Hi', 'Null'], '3': ['s9lpBIP1Kb9n5w9I', 'Groceries', 'Null'], '4': ['3KoJtdNjkQYMM7G9', 'new message', 'What's up?'], '5': ['uW3rabpshVHVtsCH', 'test', 'Null'], 'length': 6}	Dictionary stores data relating to all audio messages associated with user's *accountID* as retrieved from MySQL table **audioMessages.** Using this data, the mobile app can display all the audio messages in a GUI, allowing the user to view, edit and delete them.
paired	Boolean	True/False	True/False	Determines whether the user has successfully linked the mobile app to their doorbell. This linking process will be attempted until a successful connection is established.
visitorImage	.png	n/a	n/a	Image captured by Raspberry Pi camera when visitor rings the doorbell. If a *faceDetected* is True, facial recognition will be attempted. If not, the user will be sent an image of the visitor without a name tag.
visitorImage_path	String	n/a	"/home/pi/Desktop/NEA/ComputerScience-NEA-RPi/Photos/Visitor/visitorImage.png"	Allows Raspberry Pi to store image of most recent visitor locally so that it can be formatted and uploaded to Amazon S3 for display to the user through the mobile app. Also used to store visitor image on mobile phone so it can be displayed to user after visitor rings the doorbell.
faceID	String	43 characters (uppercase and	"QfiPWNu5WPcSxXC2wpySo	Allows MySQL table **knownFaces** to store

		lowercase letters and digits) Must be unique	wkRidDmmFrEtZXLcsGDs7v="	unique ID of each visitors' face and the visitor's name (*faceName*), to support the facial recognition algorithm, as each abstract *faceID* is associated with a name defined by the user through the mobile app.
faceName	String	n/a	"Orlando Alexander"	Allows for name of tagged faces to be stored in the **knownFaces** table in the MySQL database so that if a visitor's face can be matched to face data stored in the file *faces_trained.yml*, then the user can be notified of the visitor's name via the mobile app.
imageTimestamp	String	Include date and time	"1640797696.1842763"	Records time of visitor's visit so that an informative log of visits to user's house can be stored in MySQL table **visitorLog** and displayed to user in mobile app.
faceDetected	Boolean	True/False	True/False	When *faceDetected* returns True, this indicates that a face has been detected in the image of the visitor at the doorstep captured by the Raspberry Pi's camera. If this is the case, then the function *facialRecogniton* will be initiated on the captured photo. If *faceDetected* returns False, then no face can be detected in the captured image, and therefore the *facialRecognition* algorithm is not attempted.
faceRecognised	Boolean	True/False	True/False	When *faceRecognised* returns True this indicates that a match has been found for the visitor's face in the data set file *faces_trained.yml*. If *faceRecognised* returns False, then no match has

				been found for the visitor's face.
faces_trained	.yml	n/a	n/a	Allows for facial recognition algorithm (*facialRecognition*) and facial recognition training algorithm (*updateTraining*) to run by storing tagged data about visitors' faces.
visitID	String	43 characters (uppercase and lowercase letters and digits) Must be unique	"hrA8FudoD3AQOWWq0pHO0Bi5tyYbNhuHlkLI4ttUvvA="	Allows each visit to be stored under a unique *visitID* in the MySQL table **visitorLog** alongside the *faceID* of the visitor's face, *confidence* of this assigned *faceID*, the time of the visit (*imageTimestamp*) and the *accountID* associated with the visit. Moreover, each image captured of a visitor will be stored in the **nea-visitor-log** directory on Amazon S3 under the name of the associated *visitID*. As each *visitID* will be stored in the MySQL table **visitorLog**, the image of each visitor's face can be downloaded from Amazon S3 by the mobile app and displayed to the user when the doorbell is rung and in the visitor log.

Data Dictionary – Functions

The table below explains the fundamental functions that will be used in the project.
All functions have been named using the following notation: *myFunction* or *myFunction_name* or *myFunction_nameConvention*.

Function	Inputs	Outputs	Location (mobile app, Raspberry Pi, server REST API or Amazon S3)	Justification
createAccount	***user details input****	accountID firstName surname email password **Updated version of MySQL table:** users	**Mobile app:** User inputs their details. **Server REST API:** User details stored in **users** table in MySQL database via *updateUsers* function on AWS server.	Allows multiple users to create an account which is connected to the same doorbell. Each account has a unique 43 character *accountID*. The password is hashed using SHA3-256 before being stored in MySQL database .
signIn	***user details input****	accountID	**Mobile app:** User inputs details. **Server REST API:** User details validated by comparing them to details stored in **users** table in MySQL database via *verifyUser* function on AWS server. *verifyUser* function will also return the user's *accountID*.	Allows user to sign into their account and access their *accountID* so they can access their personalised audio response messages, view the visitor log from the doorbell they are paired with and access the tagged faces associated with their account.
viewMessages	messageData path_messageVoice_path **MySQL table:** audioMessages	***user audio messages displayed****	**Mobile app:** Mobile app GUI displays names of the audio messages associated with the user's *accountID*. Multiple pages used to display the audio message names to ensure an accessible layout. Tapping on the name of an audio message will allow	As the user can create and store customized audio messages with meaningful names, when a visitor rings their doorbell, they are able to respond quickly and effectively without interrupting their work.

			the user to view, edit or delete the message – if the message is a voice recording, it will be stored at *path_messageVoice_path*.	
createMessage	accountID ***user audio message name input*** ***user voice input*** *or* ***user text input***	messageID messageName audioData *or* messageText **Updated version of MySQL table:** audioMessages **Updated Amazon S3 directory:** nea-audio-messages	**Mobile app:** Audio messages recorded *or* audio message text inputted by user and audio message names inputted by user. **Amazon S3:** If audio message inputted as voice recording, audio message file stored in **nea-audio-messages** directory in Amazon S3. **Server REST API:** Name (*messageID*) of audio file in **nea-audio-messages** directory on Amazon S3 is stored alongside *accountID* of user who created audio message in **audioMessages** table in MySQL database on server. If audio message is inputted as text, then text of audio message stored alongside account ID of user who created audio message stored in **audioMessages** table in MySQL database.	If user wants to use their own voice, they will record a set of default audio messages which can be quickly and easily played by the doorbell with a single tap on the mobile app. The user could instead choose to input a text message which will be outputted as an audio message by the doorbell using a generic computer-generated voice. If audio message is already created, this function is used to allow user to review audio message and give option to delete it.
connectDoorbell	accountID	paired	**Mobile app:** Connection request initiated by user. **Server REST API:**	When a connection attempt is made with the doorbell, the function will assign *paired* to True if the

			Connection request and user account ID received and sent to Raspberry Pi. If the connection is successful, it will be stored in the MySQL table **SmartBellIDs.** **Raspberry Pi:** Responds to connection request.	connection is successful and False if the connection is unsuccessful .
detectButton	***button input***	captureImage *function is called*	**Raspberry Pi:** Button is connected to Raspberry Pi so button input is detected by Raspberry Pi.	When doorbell button is pressed, the function *captureImage* is called to capture an image of the visitor.
captureImage	****image captured****	visitorImage imageTimestamp visitID visitorImage_path **Updated Amazon S3 directory:** visitorImages	**Raspberry Pi:** Camera connected to Raspberry Pi captures still image of visitor. **Amazon S3:** Captured image is stored in **nea-visitor-log** directory on Amazon S3 with name equal to *visitID*.	The camera connected to the Raspberry Pi takes a photo of the visitor so that *facialDetection* function can be carried out. The image captured is assigned a *visitID*, under which the visitor image is stored on Amazon S3 so that the image can be displayed to the user through the mobile app.
facialDetection	visitorImage	faceDetected	**Raspberry Pi:** OpenCV's *CascadeClassifier* module and a locally stored face Haar Cascade .xml file will be used to check if a face is present in the captured image. If a face is detected, *faceDetected* is set to True and the function *facialRecognition* will be called and facial	Running the facial detection algorithm prior to the facial recognition algorithm, ensures that time is not wasted if no face can be detected in the image. Moreover, if *faceDetected* is set to True, the *visitorImage* can be cropped to only

			recognition will be attempted. If not, 1 more photo will be captured and facial detection performed again. If a face is still not detected, *faceDetected* is set to False and the facial recognition algorithm is not applied to the captured image. Instead, 2 functions will be called: *imageSocket* and *visitorLog_update*	include the detected face, which will improve the accuracy of the facial recognition algorithm. If *faceDetected* is set to False, only the *visitID* will be sent to the function *imageSocket*, and only *visitID*, *imageTimestamp* and *accountID* will be sent to *visitorLog_update* as there is no data about the visitor's face.
facicalRecognition	visitorImage faces_trained.yml **MySQL table:** knownFaces	faceID faceRecognised faceName *if faceRecognised is True*	**Raspberry Pi:** My facial recognition algorithm and OpenCV's *face* module will be used to read in the trained data set *faces_trained.yml* and predict whether the visitor's face matches any tagged faces. If there is a match, *faceRecognised* will be set to True and *faceID* will be set to that of the matching face. If there is no match, *faceRecognised* will be set to False and a new *faceID* will be generated. Finally, the functions *imageSocket* and *visitorLog_update* will be called. **Server REST API:** If *faceRecognised* is set to True, a query will be made to the MySQL table **knownFaces** to determine the *faceName* associated	Analysis of the individual's face will be performed locally on the Raspberry Pi to reduce latency, as performing this analysis on a remote server would require the image to be transmitted over the Internet initially. Using the data about the visitor's face, this function check for a match with faces in the data set *faces_trained.yml*. If *faceRecognised* is True, both the *faceID* and the *faceName* will be passed to the function *imageSocket* to maximise the amount of personalised information that can presented to

			with the detected *faceID* of the visitor.	the user through the mobile app.
updateTraining	visitorImage faceID faceRecognised faces_trained.yml	**Updated data set:** faces_trained.yml	**Raspberry Pi:** If *faceRecognised* is set to True, using OpenCV's *face* module, the *visitorImage* will be added to the face image data for the detected *faceID* in the data set *faces_trained.yml*. If *faceRecognised* is set to False, the *visitorImage* will be added to the data set *faces_trained.yml* as a new face with a new *faceID*.	By training the data set each time an image of a visitor's face is captured, the Raspberry Pi doorbell will become more accurate at assigning the correct name to visitors, improving the experience for users.
imageSocket	accountID visitID faceName *if faceRecognised is True*	***data sent to mobile app****	**Raspberry Pi:** Socket created between Raspberry Pi and the mobile app which is used to send *visitID* and *faceName* (if visitor's face is identifiable) to the mobile app when the doorbell button is pressed. Using the *visitID*, the mobile app can download the image of the visitor from the **nea-visitor-log** on Amazon S3. **Mobile app:** When *visitID* received by the mobile app, the visitor image can be downloaded from Amazon S3 and displayed to the user alongside the name of the visitor (if identified).	A socket is used here to reduce the latency between the visitor pressing the doorbell and the user receiving a notification on the mobile app.
viewImage	accountID visitorImage	***visitor image displayed to user**** faceName	**Mobile app**: Image of latest visitor shown to user, alongside name tag of	Allows user to see who is at the door so they can choose an appropriate

	visitorImage_path faceName *if faceRecognised is True* ***user visitor name input**** *if faceDetected is True but faceRecognised is false* **Amazon S3 directory:** nea-visitor-log	*if faceDetected is True but faceRecognised is false*	the visitor if the *faceRecognised* is True. If *faceRecognised* is False, then the user will have the opportunity to input the name of the visitor.	audio response. If the visitor's face has not been identified by the algorithm *facialRecognition*, then the function *knownFaces_update* is called to improve the chances of the detected face being assigned the correct name by the facial recognition algorithm in the future.
selectMessage	***user audio message selection**** messageData **MySQL table:** audioMessages	messageID *or* messageText	**Mobile app:** User uses app GUI to select which audio message they would like to be played by doorbell.	Each time the doorbell is pressed, the user will be able to choose a personalised audio message response. For example, the button selected may be: "Food delivery".
playAudio	messageID audioData path_messageVoice_path **Amazon S3 directory:** nea-audio-messages	***audio is played through speaker****	**Raspberry Pi:** Audio message with name *messageID* is downloaded from the **nea-audio-messages** directory on Amazon S3 and played through connected speaker.	During the initial set-up of the doorbell, through the mobile app, the user can record and name several audio messages. Through playing pre-existing audio messages through the Raspberry Pi, the user can communicate with visitors quickly and without disturbance to their work.
playText	messageText	***physical output – audio is played through speaker****	Text-to-speech module used to output audio message with text *messageText*.	During the initial set-up of the doorbell, through the mobile app, the user can type and name several audio

				messages. Through playing pre-existing audio messages through the Raspberry Pi, the user can communicate with visitors quickly and without disturbance to their work.
visitorLog_update	visitID imageTimestamp accountID faceID *if faceDetected is True*	**Updated version of MySQL table:** visitorLog	**Server REST API:** Unique *visitID* (which is the name under which the image of the visitor is stored in the **nea-visitor-log** directory on Amazon S3) is stored alongside the user's *accountID*, *imageTimestamp* and the *faceID* if a face is detected in the image.	Updates MySQL table **visitorLog** each time a visitor rings the doorbell to ensure the user can access data about all visits to their house from the mobile app.
knownFaces_update	faceName faceID accountID	**Updated version of MySQL table:** knownFaces	**Server REST API:** *faceName* for previously not recognised visitor is stored alongside the *faceID* assigned to the visitor and the *accountID* associated with the doorbell which captured the image of the visitor.	User will be given the option to update the **knownFaces** MySQL table which stores data about each visitor's *faceID* and *faceName*. As the *visitorImage* captured of the visitor and the newly generated *faceID* will already have been used to train the data set in the Raspberry Pi function *updateTraining*, the next time that the facial recognition algorithm is run on the same visitor's face, it will be possible to name the vistor. Note that this function is

				only called if *faceRecognised* is False (i.e. data about the visitor's face was not already stored in the facial data set).
visitorLog_view	accountID logData **MySQL table:** visitorLog	***visitor log images displayed to user in in mobile app**** imageTimestamp faceName *if faceRecognised is True*	**Mobile app:** Data about visitor's visits is stored in **visitorLog** table in the MySQL database on the server, so that user can view details of previous visits from the app in a GUI.	Allows user to access a log of visitors so that they can follow up on any visits if they were out of the house at the time of the visit.

Pseudocode

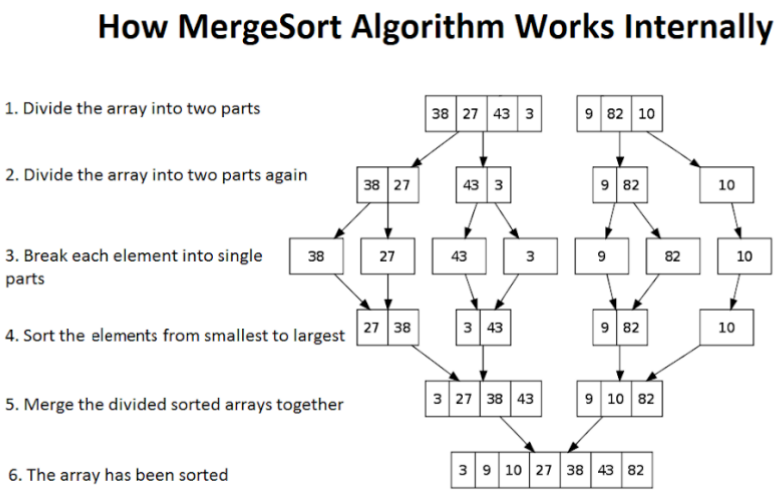
I designed the main algorithms in my project using pseudocode to help me plan and understand my code better before I began implementing it in *Python*. This reduced the time it took to code these algorithms during the *Development* phase.

Merge Sort

```
DEFINE FUNCTION mergeSort(array):

    IF len(array) > 1:
        SET mid TO len(array) FLOOR DIV 2
        SET left TO array[:mid]
        SET right TO array[mid:]

        mergeSort(left)
        mergeSort(right)

        SET i TO 0
        SET j TO 0
        SET k TO 0

        WHILE i < len(left) and j < len(right):
            IF left[i][index] < right[j][index]:
                SET array[k] TO left[i]
                INCREMENT i BY 1
            ELSE:
                SET array[k] TO right[j]
                INCREMENT j BY 1
            ENDIF
            INCREMENT k BY 1

        WHILE i < len(left):
            SET array[k] TO left[i]
            INCREMENT i BY 1
            INCREMENT k BY 1

        WHILE j < len(right):
            SET array[k] TO right[j]
            INCREMENT j BY 1
            INCREMENT k BY 1
    ENDIF
```

The **merge sort** algorithm will be used to sort the visitor log by various conditions specified by the user through the mobile app's GUI. Using a merge sort algorithm to sort lists is one of the mist efficient approaches, as it has a Big-O notation of *O(n log(n))*.

My merge sort algorithm uses recursion to split the array into two halves of sub-arrays until each sub-array contains only one element. When each half of array has been split into individual elements, the values of the elements in left sub-array and right sub-array are compared and swapped where necessary to put the elements in the correct order, and the ordered elements of the sub-arrays are stored as a merged array. When the sub-arrays have been sorted, this merged array is returned to the point where the recursive function call was made. This returned and merged array will be stored as a left sub-array or a right sub-array, depending on whether the recursive function was called on a left or right half of the array, allowing the process of comparing and merging sorted arrays to continue until the entire array is sorted.



Facial Recognition

```
DEFINE FUNCTION recognise(self, faceRGB):
    SET fileName TO openRead("pathToFile/trainingData")

    IF fileName NOT EXISTS:
        RETURN 'Unknown', False
    ENDIF

    SET encoding TO face encodings of faceRGB
    SET matches TO face encoding matches between training data in
data["encodings"] and encoding of faceRGB

    IF True IN matches:
        SET matchedIndexes TO LIST of indexes at which match between encoding
of faceRGB and trained data

        SET labelCount TO DICT
        FOR index IN matchedIndexes:
            SET label TO data["labels"][index]
            INCREMENT labelCount[label] BY 1

        SET label TO label in labelCount with maximum value
        SET matchCount TO labelCount[label]
        SET actualCount TO 0

        FOR i IN data['labels']:
            IF i EQUALS label:
                INCREMENT actualCount BY 1

        IF matchCount DIV actualCount > 0.5:
            RETURN label, True
        ELSE:
            RETURN 'Unknown', False
        ENDIF

    ELSE:
        RETURN 'Unknown', False
    ENDIF

    RETURN 'Unknown', False
```

My proposed design for the **facial recognition algorithm** uses *deep metric learning* (implemented by a *Python* module) to calculate the face encodings for each visitor's face. These face encodings are then compared with the face encodings of the tagged images stored in the trained data set to find any matches. If the face encodings of the visitor match several different faces in the trained data set, the face with the most matches is considered. To ensure that the result is reliable, the algorithm also ensures that the visitor image match at least 50% of the images in the trained data set for the face it has been identified as. If a match is found, the algorithm will return the name of the identified face.

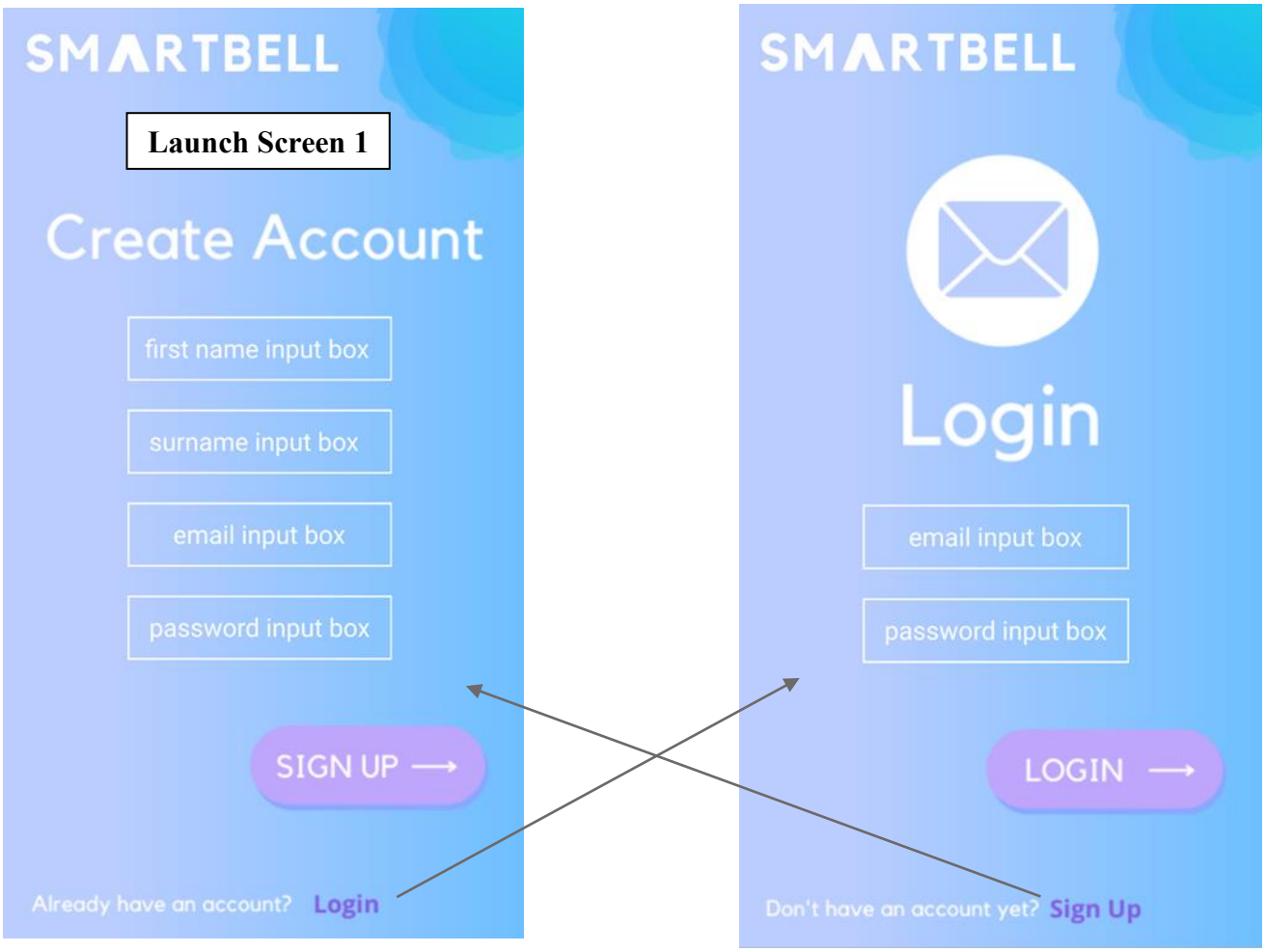
Wireframes

The Wireframes below show the UI design for the mobile app and the hierarchy between the windows/Classes within the app, as outlined on page 19.

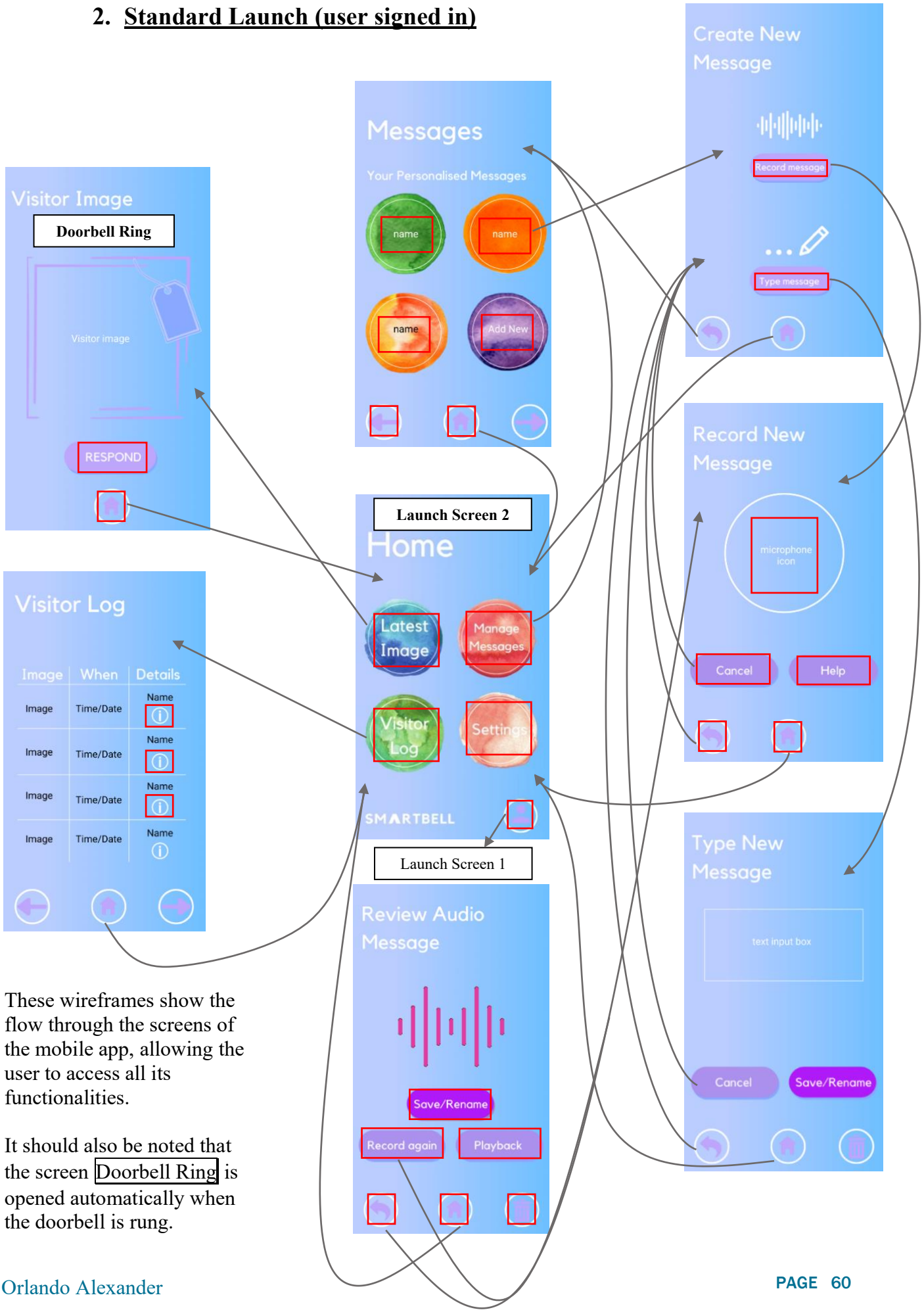
I designed the graphics for each window using *Canva*, and I have also indicated the location of text fields (white rectangles) and buttons (red rectangles).

As the mobile is designed to make it easy for home-workers to quickly respond to visitors at their door, the interface is minimalistic and visually appealing to ensure the user has an intuitive experience when using the mobile app. Explained further in the *Development* section, these design features also support the **accessible** design of the mobile app.

1. Initial Launch



2. Standard Launch (user signed in)



These wireframes show the flow through the screens of the mobile app, allowing the user to access all its functionalities.

It should also be noted that the screen Doorbell Ring is opened automatically when the doorbell is rung.

Development Report

In the *Development* report, I will walk-through the programs I have created for the *mobile app (front-end)*, the *Raspberry Pi (front-end)* and the *server (back-end)*. For each program, I will breakdown the code into sections corresponding to each Class within the program.

As I used an iterative approach during the *Development* phase, in each section I will cover the following:

- The **file structure** for the section
- To what extent the **System Hierarchy** and **UML Class** diagrams relating to the section changed from the System Hierarchy and UML Class diagrams proposed earlier during the *Design* phase. If changes *were* made, I will also justify these modifications.
- **Explanations and justifications** of the final **code** for the section, accompanied by screenshots of the output.
- Any **errors** that arose during the development of the section and how I resolved them.

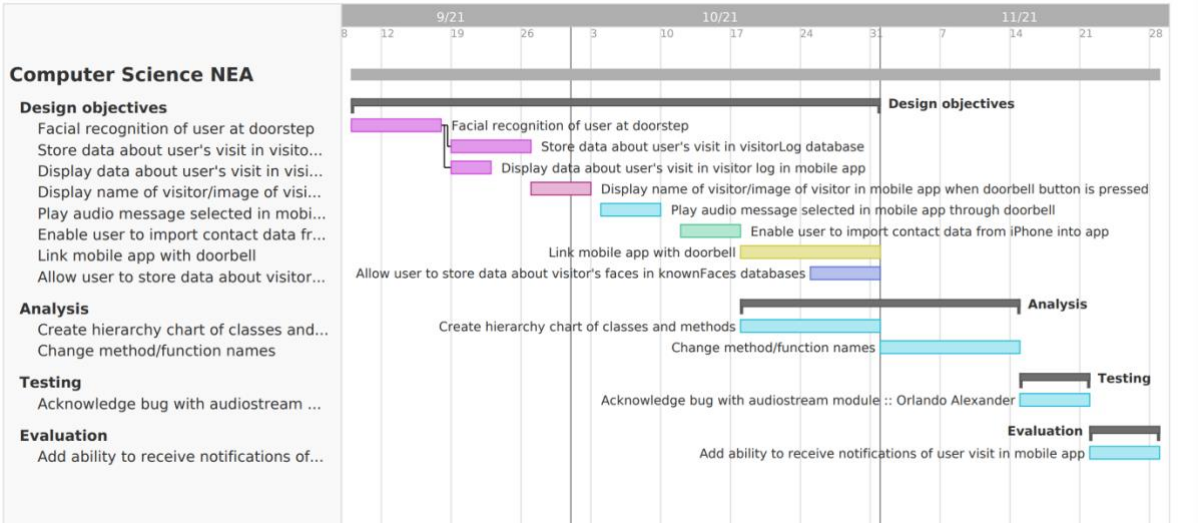
September 2021 Progress Update

Project objectives status

Users must be able to create **individual accounts** on the mobile app which will allow them to access and play their personalised audio messages through personalised shortcut buttons.
User must be able to **pair their doorbell with their account** via a simple GUI on the mobile app.
The user must be able to **record or type personalised audio message responses** (to be played through the doorbell) and create shortcuts to access them quickly through the mobile app.
The doorbell will attempt to **identify every individual** who presses the button using a facial recognition algorithm.
When the doorbell is pressed, the user will receive a **notification through their phone**, along with the name of the visitor (if identified). When this notification is opened, the user will be shown an image of the visitor in the app.
When visitor image is shown to user in the mobile app, the GUI will also have several large buttons to enable user to **select the desired audio message to be played through the doorbell.**
When the doorbell is pressed, details about the visit will be added to the **visitor log**, which will show the name of the visitor (if identifiable), the time of the visitor, the image of the visitor and an option to download the image to their camera role. This visitor log will be accessible via the mobile app, with the option to sort/filter which visits are displayed. Various summary statistics about the visits will also be displayed.
Following each doorbell ring where the visitor was not identifiable, the user will be given the option to **enter the name of the visitor via the mobile app**. The user will be required to ask the visitor for consent to store their tagged image. This will ensure that the correct name is assigned to the visitor's face the next time that it is detected.

- Green objective points have been entirely completed
- Orange objective points have been partially completed
- Red objective points have not been attempted

Gantt Chart



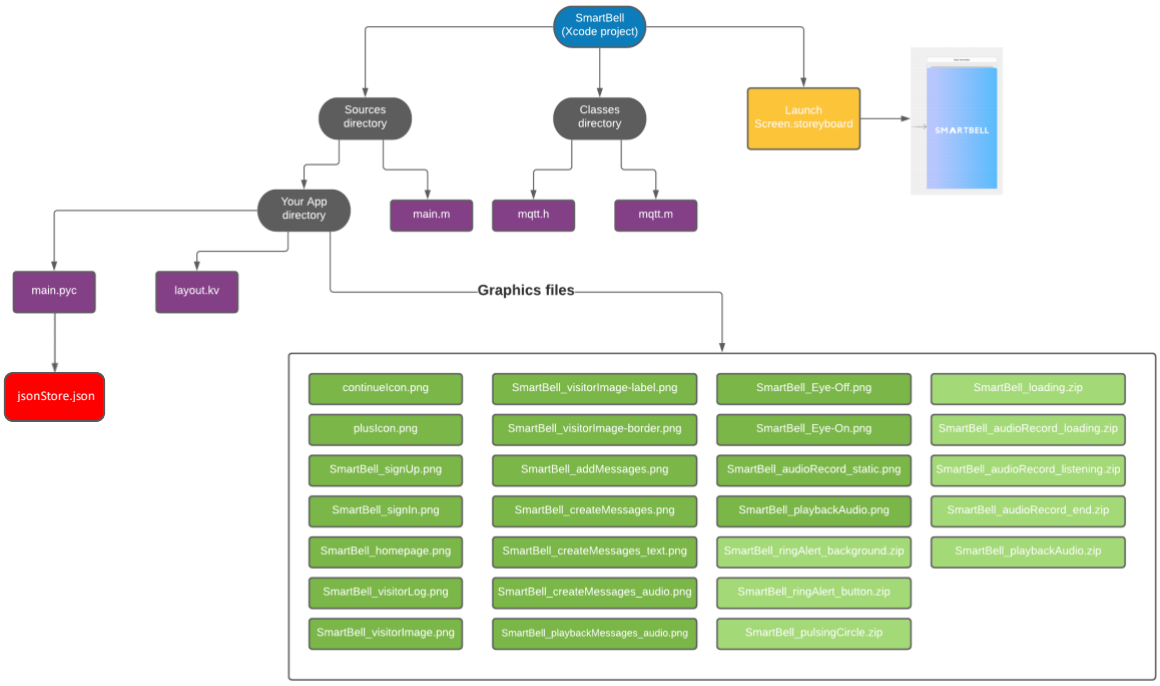
January 2022 Progress Update

Users must be able to create individual accounts on the mobile app which will allow them to access and play their personalised audio messages through personalised shortcut buttons.
User must be able to pair their doorbell with their account via a simple GUI on the mobile app.
The user must be able to record or type personalised audio message responses (to be played through the doorbell) and create shortcuts to access them quickly through the mobile app.
The doorbell will attempt to identify every individual who presses the button using a facial recognition algorithm. **If the individual's face is recognised to already exist in the data set, the newly captured images will be used to improve the data for their face in the data set. If not, the images will be added to the data set as a new face.**
When the doorbell is pressed, the user will (~~receive a notification through their phone, along with the name of the visitor (if identified). When this notification is opened, the user will be shown an image of the visitor in the app.~~) **be shown an image of the visitor, along with their name (if identified) in the mobile app.**
When visitor image is shown to user in the mobile app, the GUI will also have several large buttons to enable user to select the desired audio message to be played through the doorbell.
When the doorbell is pressed, details about the visit will be added to the visitor log, which will show the name of the visitor (if identifiable), the time of the visitor, the image of the visitor and an option to download the image to their camera role. This visitor log will be accessible via the mobile app, with the option to sort/filter which visits are displayed.
Following each doorbell ring where the visitor was not identifiable, the user will be given the option to enter the name of the visitor via the mobile app. **If the name entered by the user matches an existing face associated with their account, the newly captured images will be associated with the existing face in the data set.** ~~The user will be required to ask the visitor for consent to store their tagged image.~~ This will ensure that the correct name is assigned to the visitor's face the next time that it is detected.

- Where objective sub-points have been ~~crossed out~~, this indicates that, through completion of the *Development* process, I have deemed these sub-points to no longer be useful features.
- Where objective sub-points have been written in **bold,** this indicates that these points have been added throughout the *Development* process, as I found them to be necessary or useful in the design of the final project.
- Throughout the *Development* section, I will address both sub-points which I have decided *not* to include in the final project and sub-points which I have decided to include *additionally* in the final project.

Mobile App

Mobile App – File Structure



- SmartBell is a compiled Python package which can be installed as a mobile application on an iPhone. With the exception of the files within the *Classes directory* and the file *LaunchScreen.storyboard*, this Python project was created by compiling the *Kivy* distribution and using this to create the Python project from the directory containing all the *Kivy*-based components for my mobile app (included in the Python project directory *Your App*). The steps were as follows:
    - Install the required dependencies as outlined at *Kivy.org*
    - Ran the following commands in the terminal on MacOS to compile the *Kivy* distribution:
        - `pip install Kivy-ios`
        - `toolchain build Kivy`
    - Created the Python project *SmartBell* with the following command in the terminal on MacOS:
        - `toolchain create SmartBell /Users/orlandoalexander/Documents/Berkhamsted/ComputerScience/NEA/ComputerScience-NEA-App`

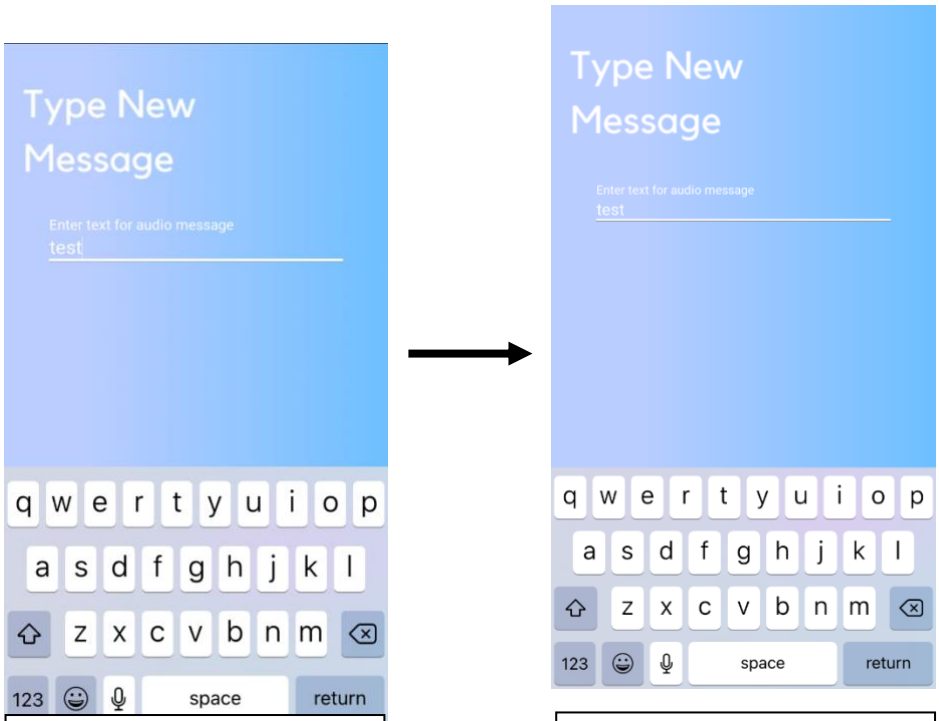
- Sources is a directory which contains the sub-directory *Your App* and the file *main.m*.

    - main.m file is created automatically when *toolchain* is used to compile the Python project, and it serves to initialise the correct setup such that the iPhone can correctly run the *Kivy* application as a mobile app.

- Your App is a directory which stores all the files associated with my *Kivy* application: *main.pyc, layout.kv* and all the *graphics files* required for the intuitive and accessible interface.
    - main.pyc is the compiled bytecode form of the *main.py* file in which the logic code for the mobile app is written in *Python*. This logic includes communication with the Raspberry doorbell and the server, interfacing with the *layout.kv* file to control the graphics and recording and storing audio input and outputting audio files.
        - jsonStore.json is a local *json* file which stores data about the user and status of the mobile app, with the following keys:
            - initialUse
            - loggedIn
            - accountID
            - paired
    - layout.kv is the file which stores the code to display the graphics for the mobile app and is written using the markup language *kivy*. The graphics in the mobile app include both static images and gifs, buttons and text input boxes.
    - The **graphics files** used in the mobile app are all stored in 1 of 2 formats: *.png* files or *.zip* files. The *.png* format is used for static images (e.g. the homepage background), whereas the *.zip* format is used for gifs. Initially, I faced issues with loading the *.gif* format into the mobile app, as the *kivy* markup language does not support the *.gif* format. However, the *kivy* markup language *does* support the *.zip* format. As such, to resolve this issue I used the online frame-splitter tool on *ezgif.com* to split the *.gif* files into a compressed *.zip* folder containing the individual *.png* used to create the gifs.

- Classes is a directory which stores the file *mqtt.h* and *mqtt.m* which together make-up the *MQTT* class. The *MQTT* class is used to integrate code written in objective-c into the *Python* file *main.py*. As explained in further detail later in the *Development* report, this structure (which is facilitated using the *Python* module *pyobjus*) is necessary to enable quick and light-weight communication between the mobile app, server and Raspberry Pi doorbell via the MQTT (MQ Telemetry Transport) framework.
    - mqtt.h is the header file for the *MQTT* class. This file is used to declare the public properties (properties are much like variables) of the *MQTT* class. As these properties are public, they can be accessed from outside the *MQTT* class. In my application, these properties are accessed from within the *main.py* file using the *autoclass* class imported from the *pyobjus* module. The *mqtt.h* file must import any external header files which are required to declare class properties which extend the functionality of the *MQTT* class. Finally, the public interface for any custom classes whose implementation (largely comprising of the class methods) is written in the *mqtt.m* file must be declared in the *mqtt.h* file. If the custom class must inherit from a parent class or protocol (a protocol is used to declare methods and properties which are independent of a class), these must also be specified when declaring the custom class interface.
    - mqtt.m is the implementation file for the *MQTT* class. This file is used to write the methods (in objective-c) for each interface declared in the *mqtt.h* file, using *self.propertyName* when referencing any properties declared in the *mqtt.h* file. As such, the *mqtt.h* file must be imported into the *mqtt.m* file. Moreover, the *mqtt.m* file must also import any any external header files which are required in the *mqtt.m* file for any functions which are to be called or classes which are to be intitialised.

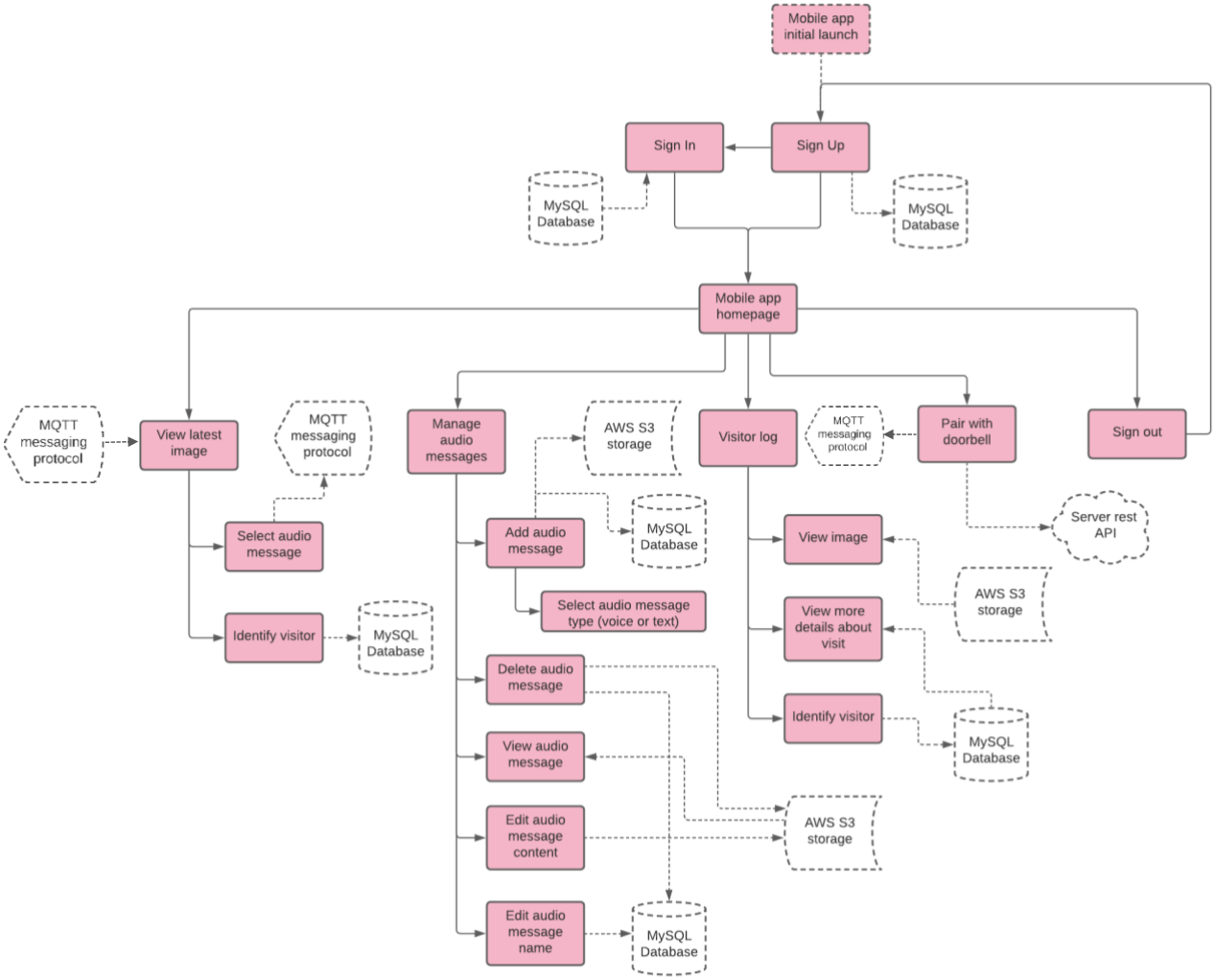
- LaunchScreen.storyboard is a *Python*-specific means of creating graphics for the mobile app. Where an iPhone mobile app is written in Swift (a programming language created by Apple), *storyboards* might be used to create in-app graphics. However, as I used *Kivy* to create my mobile app, I am only using a single *storyboard* to create the launch screen for my mobile app (as shown in the diagram above). This launch screen is shown while the mobile app is opening and initialising.
    - Interestingly, the addition of the *Launch Screen.storyboard* file also resolved a separate, apparently unrelated, issue which occurred when the mobile app was run on my iPhone. Without the *Launch Screen.storyboard* file, the on-screen keyboard that appeared when the user inputted text through the mobile app interface was larger than the standard size of the on-screen keyboard. However, this glitch was resolved when I added the launch screen *storyboard*.



On-screen keyboard **without** *Launch Screen.storyboard* file

On-screen keyboard **with** *Launch Screen.storyboard* file

Mobile App – System Hierarchy Diagram



Note that references to **MySQL Database** and **AWS S3 Storage** should all be read to include the **REST API** as an intermediary, since all communications with the MySQL Database and S3 Storage are executed by the REST API on behalf of the mobile application.

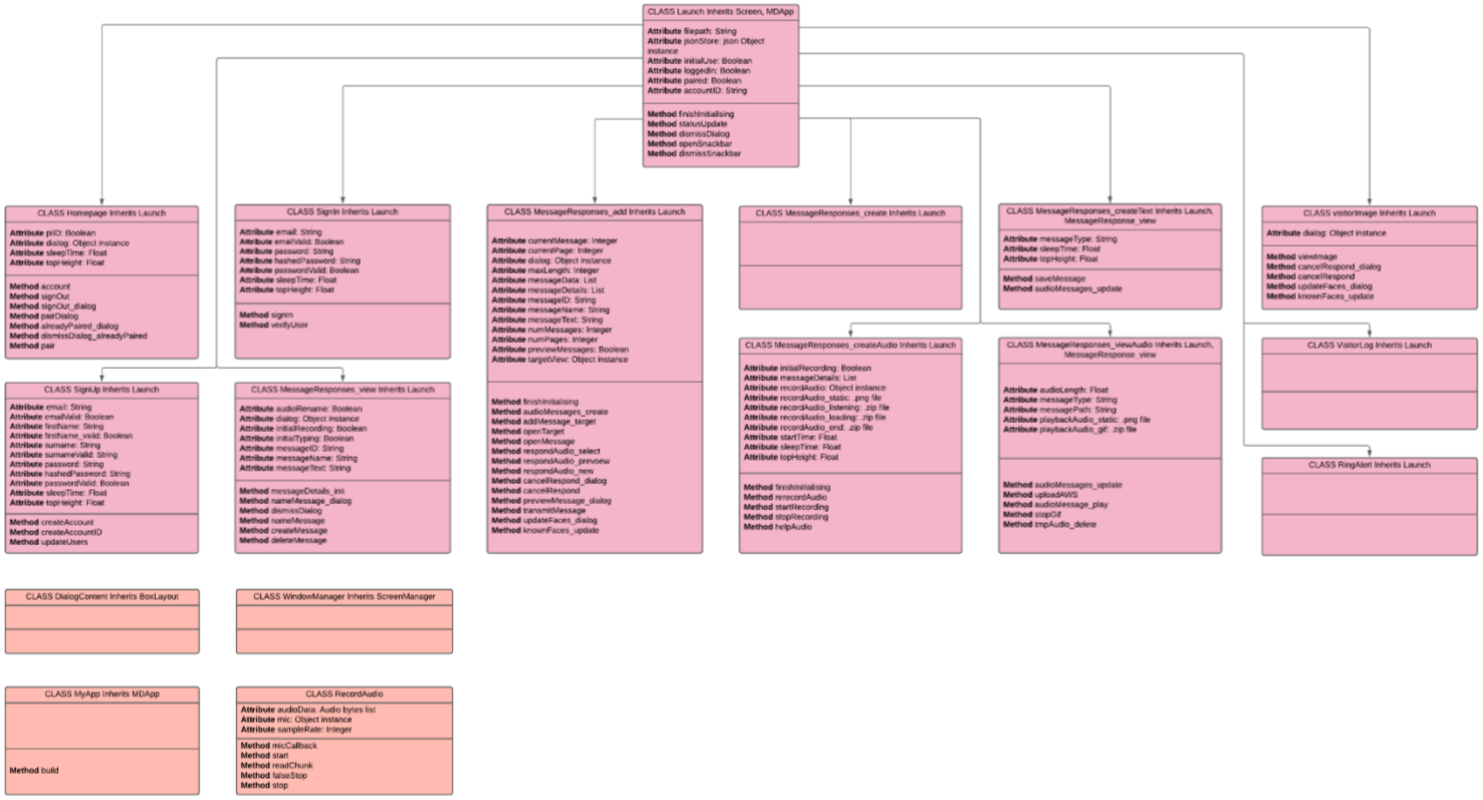
During *Development*, I largely followed the **System Hierarchy structure** for the mobile app which I proposed during the *Design* phase. This approach enabled me to focus on the programmatic implementation of my designs during *Development*, rather than the program design itself. However, I did make a few minor tweaks to the overall program structure as follows:

Feature change	Justification
Removed function *Store tagged image of visitor* from *View latest image*	As the facial recognition data set is stored locally on the Raspberry Pi, this process will be performed directly on the Raspberry Pi. If the user inputs a name for a visitor image where a face has not been recognised by the Raspberry Pi facial recognition algorithm via the mobile app, the Raspberry Pi paired with

	the mobile app will automatically update the tagged image data set.
Sign In and *Sign Up* no longer only called on launch. Instead, they will both be accessible from *Homepage* when the user is not signed in. During the initial launch of the mobile, *Sign Up* will automatically be called, with the option of navigating to *Sign In*.	To ensure the purpose of user accounts is fulfilled, the user should be able to access the *Sign Out* function from the homepage, and then sign into a new account via *Sign In*/*Sign Up*. Moreover, *Sign Up* is automatically called on initial launch for the following reasons: 1. Most users will not have an existing account when they first run the app, so it is appropriate to call *Sign Up* rather than *Sign In*. 2. Users must be signed into an account to pair with their *SmartBell* device and access the features of the mobile app, so directing the user to the *Sign Up*/*Sign In* page upon launch is required.
Added the *Identify visitor* function to *Visitor log.*	Allows user to retrospectively input the name for a visitor image where a face has not been recognised by the Raspberry Pi facial recognition algorithm, improving the facial recognition capabilities of the *SmartBell.*

Mobile App – UML Class Diagram

main.pyc file



During *Development*, I significantly expanded and improved the class structure of my *Python* code to realise the objectives of my project. Throughout the process, I followed 3 fundamental concepts of **Object-Orientated Programming**:

1. **Inheritance** – All classes which correspond with a window/GUI in the *Kivy* code (shaded in pink) inherit the parent class `Launch`, as `Launch` contains methods and attributes required by all subclasses. For example, the variable `accountID` is used by most classes to enable various features of the mobile app, such as accessing the user's audio messages. The methods `openSnackbar` and `dismissSnackbar` are also used by many classes to create an animated snackbar graphic. Through this inheritance structure, I have avoided the repetition of the core code to generate a snackbar; however, the specific features of each snackbar (such as the text) can be defined when the class intialises a snackbar object. Moreover, using **multiple inheritance**, `Launch` class also inherits parent classes `screen` and `MDApp`. Under the principles of **multilevel inheritance**, all subclasses of `Launch` also inherit `screen` and `MDApp`. This is a useful structural design, as the `screen` class is required by all classes which inherit `Launch` to enable interfacing with the *Kivy* code, and `MDApp` is required in most classes to access properties of the mobile app during run-time. For example, `MDApp` is regularly used to change the window through the *Python* code, rather than the *Kivy* code.

2. **Polymorphism** – Where child classes have a method with the same name as a method of their parent class, the child class redefines the method in question. This is polymorphism.

In my program, this occurs with the `dismissDialog` method; the `dismissDialog` belongs to the `Launch` class as many classes which inherit the `Launch` class require access to this standard method to close a dialog box. However, in the `messageResponses_view` class, the functionality of the `dismissDialog` method is extended. To achieve this, `super().dismissDialog()` is called before the new code is run, ensuring that the original functionality of the `dismissDialog` in the `Launch` class is maintained.

3. **Encapsulation** – During *Development*, I created many more classes than I had anticipated *Design* stage, as I ensured that the principles of **encapsulation** were followed throughout my code. Encapsulation refers to the grouping of related methods and attributes in Object Orientated Programming into a single unit (class). As such, the number of classes in my final code for the mobile app reflect the number of different behaviours of the mobile app, each of which require their own class to contain their associated methods and attributes.

The **attributes** shown in the above UML class diagram are not an exhaustive representation of all the variables used in the code for the mobile app; class attributes refer only to the instance variables. Throughout my code, I have made deliberate distinctions between instance variables and normal variables; instance variables are only used where multiple methods are required to access that variable. For example, in the `RecordAudio` class, the attribute `audioData` must be accessed by both the `micCallback` and `stop` methods so that the audio bytes can be appended to `audioData` during recording, and then `audioData` must be returned when `RecordAudio.stop()` is called.

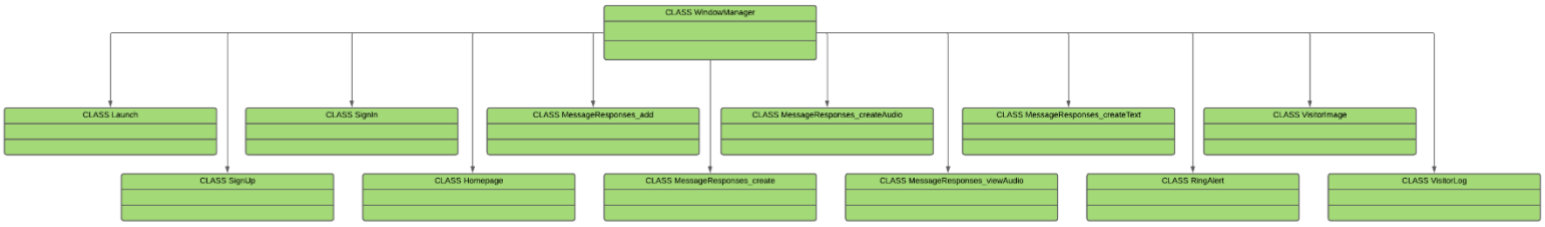
All classes which do not inherit `Launch` (shaded in orange) do not correspond to a window in the mobile app. Their functions will be explained later in the *Development* report.

There are also several **functions** in the mobile app which are not associated with a class, as follows:

`visitorImage_thread`
`createThread_ring`
`createThread_visit`
`ringThread`
`visitThread`
`pairThread`

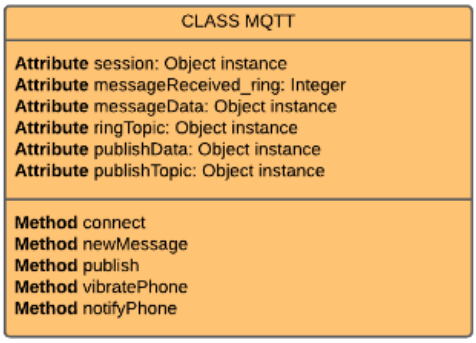
All these functions are used to create **threads** which run in the background in pseudo-parallel to the main program code. This allows processes to completed without affecting the smooth running of the app and ensuring a seamless user experience is maintained. As threading is a means of hiding the complexities of the mobile app from the user, this is a form of **abstraction.** In the mobile app, threads are either used to poll the MySQL database until a specific record is detected, or to check the status of a variable in the *mqtt.m* file which receives messages transmitted via MQTT from a paired doorbell.

layout.kv file



Each **class** in the *Kivy* markup language corresponds to a window in the mobile app, as well as a class with the same name in the *main.pyc* file. These classes contain code (written in the *Kivy* language) which create the graphics for each window. The logic for each window is written in the corresponding class in *main.pyc*. Using *Kivy*, I am able to interface between the graphics in the *layout.kv* and the logic in the *main.pyc* file. In the *main.pyc* file, the WindowManager class inherits the *ScreenManager* widget. As such, WindowManager is used in the *layout.kv* file to initialise all the windows and determine the default running order of windows.

mqtt.h & mqtt.m file



The MQTT class is written in **Objective-C** to enable the mobile app to send and receive messages using the MQTT protocol. The **attributes**, which can be accessed and edited through *main.pyc* file using the *pyobjus* module, are declared in the header file *mqtt.h*. The **methods** are all defined in the following way:

```
-(void) methodName
```

The *minus* sign indicates that the methods are instance methods; they are performed by an individual instance of the class. Therefore, the methods of the MQTT class can only be accessed through an object of the class.

The *(void)* type indicates that the methods do not return any data. This is appropriate, as the methods either have a finite run time during which they connect to the MQTT broker or publish an MQTT message to a topic, or they run infinitely, polling the MQTT broker for messages. As such, they will never terminate and return a value. Instead, when a message is received, the status variable messageReceived_ring is set to '1'.

Mobile App – Modules

```
import os
from Kivy.uix.image import AsyncImage
from Kivy.uix.boxlayout import BoxLayout
from Kivy.core.audio import SoundLoader
from Kivy.uix.screenmanager import ScreenManager, Screen, SlideTransition,
NoTransition
from Kivy.lang import Builder
from Kivy.clock import Clock
from Kivy.core.window import Window
from Kivy.animation import Animation
from Kivy.storage.jsonstore import JsonStore
from Kivymd.app import MDApp
from Kivymd.uix.textfield import MDTextField
from Kivymd.uix.dialog import MDDialog
from Kivymd.uix.taptargetview import MDTapTargetView
from Kivymd.uix.button import MDFlatButton, MDRaisedButton
from os.path import join
import os
import re
import random
import string
import requests
import hashlib
from threading import Thread
import time
import math
import wave
from pyobjus import autoclass
from audiostream import get_input
import pickle
```

The purpose and general application of each imported class/module/library is as follows:

- **os** *(Module)*
  - Provides functions for interacting with the operating system (renaming/moving/deleting files/directories)
  - The os environment variable *KIVY_AUDIO* is set to *avplayer* as this is required to ensure the app is able to record audio (using *audiostream*) following audio output (using *SoundLoader*). Without this adjustment,
- ***kivy*** *(Library)*
  - Framework to create cross-platform applications with user interfaces
  - The UI element of the mobile app is written in the *kivy* markup language and stored in *kv* files (*Kivy*.org, 2019)
    - ***AsyncImage*** *(Class imported from kivy.uix.image module)*
      - Allows images to be loaded in the background so as to avoid impact on the performance of the app
    - ***BoxLayout*** *(Class imported from kivy.uix.boxlayout module)*
      - Allows organisation of widgets into a table structure on the screen
    - ***SoundLoader*** *(Class imported from kivy.core.audio module)*
      - Load and play audio files through the mobile app
    - ***ScreenManager*** *(Class imported from kivy.uix.screenmanager module)*

        - Manages the navigation between multiple screens for the mobile app
    - ***Screen*** *(Class imported from kivy.uix.screenmanager module)*
        - Base Class which must be inherited by each Class which has an associated screen/window
    - ***SlideTransition*** *(class imported from kivy.uix.screenmanager module)*
        - Allows a slide-style transition between separate screens in the mobile app
    - ***NoTransition*** *(class imported from kivy.uix.screenmanager module)*
        - Allows a cut-style transition between separate screens in the mobile app
    - ***Builder*** *(class imported from kivy.lang module)*
        - Used to load *kv* graphics file and build the *kivy* widgets
    - ***Clock*** *(class imported from kivy.clock module)*
        - Allows a function call to be scheduled in the future or to be executed multiple times with a certain interval
    - ***Animation*** *(class imported from kivy.animation module)*
        - Creates motion between 2 static states of a widget
    - ***jsonStore*** *(class imported from kivy.storage.jsonstore module)*
        - Save and load standard key-value data pairs from a local json file

- ***kivymd*** *(library)*
    - Framework to create dynamic widgets that are compatible with *kivy* and replicate those of Google's Material Design specification (Rodríguez et al.)

        - ***MDApp*** *(Class imported from kivymd.app module)*
            - Base Class which is required to control properties (colour/style/font) of *kivyMD* widgets (Rodríguez et al., 2021) and used to access properties of mobile app during run-time
        - ***MDTextField*** *(Class imported from kivymd.uix.textfield module)*
            - Create dynamic and visually appealing text fields in which the user can enter text, such as login details
            - The clarity of design of these text fields ensures ease of use for users with special accessibility requirements, such as a visual impairment
        - ***MDDialog*** *(Class imported from kivymd.uix.dialog module)*
            - Create pop-up box which allows user to make a decision, such as select an option or confirm they want to save an audio recording
        - ***MDTapTargetView*** *(Class imported from kivymd.uix.taptargetview module)*
            - Creates an icon that can easily expanded to reveal further details when it is tapped
        - ***MDFlatButton*** *(Class imported from kivymd.uix.button module)*
            - Create a simple button which the user can tap to progress through the mobile app
            - The button widget becomes grey upon being tapped; this acknowledgement of the user's input reaffirms the responsiveness of the app
        - ***MDRaisedButton*** *(Class imported from kivymd.uix.button module)*

- Create a button with a shadow below which the user can tap to progress through the mobile app
- The button widget becomes grey upon being tapped; this acknowledgement of the user's input reaffirms the responsiveness of the app

- ***join*** *(Method imported from os.path class)*
  - Join multiple variables/strings together to create a complete path to a file/directory
- ***re*** *(Module)*
  - Provides ability to perform regular expression operations
- ***random*** *(Module)*
  - Perform a range of tasks which require randomised selections
- ***string*** *(Module)*
  - Provides functionalities related to strings, such as creating a string of all the letters in the alphabet
- ***requests*** *(Module)*
  - Enables HTTP-based interaction with my back-end APIs (running on a server in my AWS Elastic Beanstalk environment)
- ***hashlib*** *(Module)*
  - Create a hashed string using a specified hashing algorithm (I used *SHA3-256*) so that passwords can be stored securely
- ***Thread*** *(Class imported from threading module)*
  - Create and run multiple threads simultaneously within the *Python* program, so that multiple functions/tasks can be executed in pseudo-parallel
- ***time*** *(Module)*
  - Provides functionalities related to time, such as returning the current time
- ***math*** *(Module)*
  - Access a range of mathematical functions
- ***wave*** *(Module)*
  - Save audio files recorded in the mobile app by writing bytes of audio data to a *wav* file
- ***autoclass*** *(Class imported from pyobjus module)*
  - Provides an interface between *Python* files and Objective-C files, so classes written in Objective-C can be called and run from a *Python* file (pyobjus.readthedocs.io, n.d.). This is used to implement MQTT in the iOS application.
- ***get_input*** *(Class imported from audiostream module)*
  - Record audio from the user through the mobile app by *polling* the microphone for byte data (audiostream.readthedocs.io, n.d.)
- ***pickle*** *(Module)*
  - Serializes a *Python* object into a byte stream which can then be written to a *pkl* file and stored (datacamp, 2018)

The above modules and classes are imported to extend the functionality of my program, enabling me to create an app which has the **following features**:

1. *Communicate with the Raspberry Pi doorbell and server*
2. *Display dynamic graphics and an accessible GUI*
3. *Process sensitive and non-sensitive data input*
4. *Record and store audio input and output audio files*

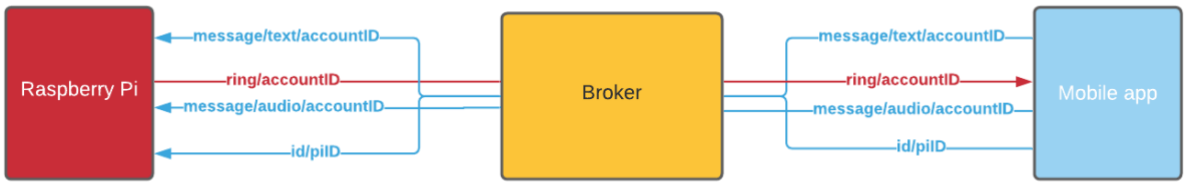
I will use the 4 categories above to explain my technical solution for the mobile app.

Mobile App – Communicate with Raspberry Pi doorbell and server

MQTT Communication

MQTT is a lightweight messaging protocol which I decided to use throughout my program (in place of the *socket* indicated in the *Design* stage) to enable low latency communication between the Raspberry Pi doorbell and the mobile app. This would allow the following 3 behaviours:

1. User receives an alert on the mobile app when the doorbell is rung
2. User can pair with/unpair from a Raspberry Pi doorbell (*SmartBell*)
3. User can send an audio message via the mobile app to be played through the Raspberry Pi's speaker



The MQTT protocol uses a central broker to receive messages, filter them by topic and distributes these messages to clients which are subscribed to that topic. Throughout the *Development* phase, I explored a variety of possible brokers:

- Eclipse Mosquito
- HiveMQ
- CloudMQTT

I decided to use **CloudMQTT** as it offers a basic plan for $5 per month with up to 25 connections and no cap on the number of messages that can be sent. Moreover, the CloudMQTT broker offers a *websocket UI* which allowed me to view all the messages passing through the broker in real-time, as well as distribute messages quickly and easily. This feature was incredibly useful during development and debugging, as it enabled me to understand clearly how the data was flowing within my system and, therefore, to easily identify where issues were arising. Primarily, I was able to implement MQTT using the CloudMQTT broker without any issues, which was not the case for the other 2 brokers I considered.

MQTT Implementation – Mobile app

During the early stages of *Development*, I used *Pycharm* installed on a Macbook Air to code and test the app, using *Kivy* to create the GUI and the *Python* launcher to display the GUI. This was possible as *Kivy* is a *cross-platform framework*. As such, during this stage of *Development*, implementing MQTT was relatively straightforward – I used the `Client` class from the `paho-mqtt-client` module as follows:

```
1  import paho.mqtt.client as mqtt # import paho-mqtt-client module
2
3  def on_message(client, userdata, msg): # function called when a message is
4  received
5      messageData = msg.payload.decode() # decode the message data received
6  ('msg' variabvel) into utf-8 and assign this message to variable 'messageData'
7
8
9  def on_connect(client, userdata, flags, rc): # function called when the client
10 connects to the broker
11     if rc == 0: # if connection is successful
12         client.publish("myTopic", "myMessage")# publishes message 'myMessage'
13 to 'myTopic'
14         client.subscribe("subscribeTopic") # subscribe to the topic
15 'subscribeTopic' so that messages published to this topic are received
16         client.message_callback_add("subscribeTopic", on_message)
17     else:
18         # if connection is unsuccessful, attempts to reconnect
19         client = mqtt.Client()
20         client.username_pw_set(username="myUsername", password="myPassword")
21         client.on_connect = on_connect
22
23 client = mqtt.Client() # create instance of the Client class
24 client.username_pw_set(username="myUsername", password="myPassword") # details
25 to connect to MQTT broker
26 client.on_connect = on_connect # create callback function which is run when
27 the client connects to the broker
28
29 client.connect("hairdresser.cloudmqtt.com", 18973) # attempt to connect to the
30 MQTT broker
31 client.loop_forever() # loops the above code forever so messages published to
32 'myTopic' are received instantly
```

However, issues arose on when I attempted to compile this MQTT implementation into an *Xcode* project and run it on my mobile app. Running the mobile app, the *Xcode* logs gave the following error on *line 1*:

```
ModuleNotFoundError: No module named 'paho'
```

To resolve this, I attempted to install the `paho-mqtt-client` module using **Toolchain**, which is used to compile *Python* libraries so they are compatible with iOS:

```
toolchain pip install paho.mqtt.client
```

However, in doing so, I was met with a new error:

```
ERROR: Could not find a version that satisfies the requirement paho.mqtt.client (from versions: none)
ERROR: No matching distribution found for paho.mqtt.client
```

After carrying out some research into these issues, I concluded that using a *Python* module to setup the iOS mobile app as an **MQTT client** was most likely impossible due to **Apple Sandbox**, which imposes restrictions on iOS mobile apps. Considering that 'Apple does not support the [*Python*] subprocess module' (Priegue, 2020), which controls processes running in the background, it didn't seem unlikely that Apple would bar the use of an MQTT client (which persistently polls the associated broker) on an iOS mobile app.

However, during my research I discovered the **MQTT-Client-Framework** which is a 'native Objective-C iOS library' that serves as a 'complete implementation of MQTT' (Krey, 2015). Although Apple is transitioning to *Swift* as its primary coding language for their OS and mobile apps, *Objective-C* is still widely used and accepted. As such, I now realised it was, in theory at least, possible to implement MQTT into my mobile app, using *Objective-C*. One key issue remained: my mobile app was written in *Python*, not *Objective-C*. To overcome this, I needed a means of creating a bridge between code written in *Objective-C* and my mobile app, written in *Python*.

My research led me to the **Pyobjus** module, which enables you to load *Objective-C* classes into *Python* code by constructing a *Python wrapper* around them (pyobjus.readthedocs.io, n.d.). The first step was to create the class in *Objective-C* which would enable me to create an MQTT client on the mobile app.

MQTT class – Objective-C

As detailed in the **MQTT-Client-Framework** docs, the framework must be installed using **CocoaPods**, which is a dependency manager for *Objective-C*. To do so, I navigated to the directory of my *Xcode* project in the terminal and ran `sudo nano podfile`.

This opened the *Podfile* for my project, where I added `pod 'MQTT Client'` to the target '*nea*', which is product name of the *Xcode* project for my mobile app:

```
  GNU nano 2.0.6                File: podfile

platform :ios, '14.7'

target 'nea' do
  # Comment the next line if you don't want to use dynamic frameworks
  use_frameworks!
  pod 'MQTTClient'

  # Pods for nea

end
```

To install **MQTTClient** into my *Xcode* project, I ran `pod update` through the command line, from within the *Xcode* project directory.

All classes in *Objective-C* consist of 2 files: the *header file* (.h) and the *implementation file* (.m). To create an instance of the MQTT class in the implementation file, it is necessary to define the *interface* for the class in the header file, specifying any protocols which the MQTT class conforms to. The ***mqtt.h*** header file is shown below:

mqtt.h

```
1  /** MQTTClient header file must be imported to access associated protocols required to
2  implement MQTT
3   */
4  #import "MQTTClient.h"
5
6
7  /** 'MQTT' is name of custom class which conforms to protocol 'MQTTSessionDelegate' (a
8  protocol is used to declare methods and properties which are independent of a class).
9  Therefore, the 'MQTT' class implements the methods associated with the
10 'MQTTSessionDelegate' protocol.
11  */
12 @interface MQTT: NSObject <MQTTSessionDelegate>
13
14
15 /** The following properties are properties of the 'MQTT' class
16  */
17 @property (nonatomic) int messageReceived_ring;
18 @property (nonatomic, assign) NSString *messageData;
19 @property (nonatomic, assign) NSString *ringTopic;
20 @property (nonatomic, assign) NSString *publishData;
21 @property (nonatomic, assign) NSString *publishTopic;
22
23 @end
```

Explanation

The `MQTTClient.h` header file must be imported (**Line 4**) so that the custom `MQTT` class can access the required protocols from the **MQTT-Client-Framework**. When defining the *interface* for the custom `MQTT` class (**Line 11**), a colon is used to indicate that the `MQTT` class is *inheriting* from the standard root class `NSObject`. The angled brackets are used to specify that the `MQTT` class conforms to the protocol `MQTTSessionDelegate`, meaning that the `MQTT` class now implements all the class-less methods associated with the `MQTTSessionDelegate` protocol. This is required to give the application control over the instance of `MQTTSession` (initialised in the *implementation file*). The `MQTT` class contains several *properties* (**Lines 16-20**) – these 7 properties are public and can be accessed from outside the class (i.e. using *Pyobjus*).

Property	Type	Justification
messageReceived_ring	Integer	Variable temporarily assigned the value '1' when a message is received indicating that the doorbell has been rung, so that the *Python* code can monitor the status of the MQTT client in the *Objective-C* code and react accordingly.
messageData	String	Variable assigned to the string storing the MQTT message received, so that this string can be accessed from within the *Python* code.
ringTopic	String	Variable assigned to the topic name which the mobile app must subscribe to so that it receives the MQTT message when the doorbell it is paired with is rung. This is assignment takes place within the *Python* code.
publishTopic	String	Variable assigned to the topic subscribed to by the *SmartBell* that the user wishes to pair with/unpair from. This assignment takes place within the *Python* code.
publishData	String	Variable assigned to the message data which is to be published to publishTopic. This assignment takes place within the *Python* code. This message is published to update the *SmartBell* subscribed to the topic publishTopic of the current pairing status

Note that, in *Objective-C*, strings are objects of the NSString class, so they must be defined as such (see **Line 17** for example).

The completed declaration of the MQTT class interface is indicated by the @end keyword on **Line 22**.

The *implementation file* provides the *implementation* of the MQTT class, defining the methods associated with it. The ***mqtt.m*** implementation file is shown below:

mqtt.m

```
#import "mqtt.h"
#import <AudioToolbox/AudioToolbox.h>

/** defines the methods for the 'MQTT' class
 */
@implementation MQTT
- (id) init {

    return self;
}

- (void) connect {

        MQTTCFSocketTransport*transport = [[MQTTCFSocketTransport alloc] init];
        transport.host = @"hairdresser.cloudmqtt.com";
        transport.port = 18973;

        MQTTSession *session = [[MQTTSession alloc] init];
        session.delegate = self;
        session.transport = transport;
        session.userName = @"myUsername";
        session.password = @"myPassword";
        session.protocolLevel = MQTTProtocolVersion31;
        self.messageReceived_ring = 0;



        [session connectAndWaitTimeout:30];

        [session subscribeToTopic:self.ringTopic atLevel:2];
}

- (void)newMessage:(MQTTSession *)session data:(NSData *)data onTopic:(NSString *)topic
qos:(MQTTQosLevel)qos retained:(BOOL)retained mid:(unsigned int)mid {

        NSString* dataStr = [[NSString alloc] initWithData:data
                                    encoding:NSUTF8StringEncoding];

    if (![dataStr isEqualToString:@""] && [topic isEqualToString:self.ringTopic] ){
            self.messageData = dataStr;
            self.messageReceived_ring = 1;
    }
}
```

```
44  - (void) publish {
45
46      MQTTCFSocketTransport*transport = [[MQTTCFSocketTransport alloc] init];
47      transport.host = @"hairdresser.cloudmqtt.com";
48      transport.port = 18973;
49
50      MQTTSession *session = [[MQTTSession alloc] init];
51      session.transport = transport;
52      session.userName = @"yrczhohs";
53      session.password = @"qPSwbxPDQHEI";
54      session.protocolLevel = MQTTProtocolVersion31;
55      [session setDelegate:self];
56      [session connectAndWaitTimeout:30];
57
58      NSData* data = [self.publishData dataUsingEncoding:NSUTF8StringEncoding];
59      [session publishData:data
60                   onTopic:self.publishTopic
61                    retain:NO
62                       qos:1];
63      [session disconnect ];
64  }
65
66  - (void)vibratePhone; {
67
68      AudioServicesPlayAlertSound(4095);
69  }
70
71  - (void)notifyPhone; {
72
73       AudioServicesPlayAlertSound(1022);
74  }
75
76  @end
```

Explanation

The mqtt.h header file must be imported (**Line 1**) as it contains the associated *interface* for the *implementation* of the MQTT class. As *Objective-C* is an OOP language, an object of the MQTT class must be instantiated to access its variables and methods; this is done on **Line 7** using the init method to return an object of the class (**Line 9**). Note that the minus sign used at the start of the method declaration indicates that init is an *instance method* (rather than a *class method*) and the (id) method return type denotes that the init method returns an *Objective-C* object (i.e. self).

There are 5 methods associated with the MQTT class which have the method return type (void), denoting that they do not return any value. These 5 methods are as follows:

1. connect

   To create the initial connection with the CloudMQTT broker, the connect method is called (**Line 12**). This connection is established by first defining the required *properties* for the MQTT session object, which is initialised on **Line 19**. Here, the alloc method (a *class method* from the inherited root class NSObject) allocates a memory location for the session object and returns a pointer to this memory location. The init method is then used to initialise and return the object at the specified memory location. **Lines 19-23** set the desired values for the required parameters:

   delegate
   - The MQTT delegate is responsible for receiving messages published to topics which the associated session is subscribed to. As the MQTT class conforms to the MQTTSessionDelegate protocol, the delegate property is assigned to the object self.

   transport
   - An object initialised on **Line 14** with its own properties: the host and port of the broker.

   username
   - The broker username string (indicated by the @ preceding the string).

   password
   - The broker password associated with the username.

   protocolLevel
   - Defines the MQTT protocol to be used.

   Once the required properties for the session object are defined, the associated method subcribeToTopic is called to subscribe to ringTopic (**Line 30**). 2 parameters are passed to the subcribeToTopic method: topic and qosLevel. Here, the topic property is ringTopic as defined in the *Python* code, with its value

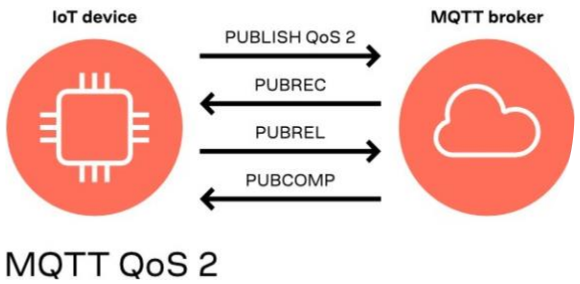
corresponding to the account ID of the user signed into the mobile app. The qosLevel is the *Quality of Service* for the MQTT message delivery.

- **QoS 0** – message sent by the *SmartBell* at most once, so there is no guarantee that the message is received by the mobile app. In other words, a 'fire and forget' approach is used (HiveMQ, 2015).
- **QoS 1** – message delivered to mobile app at least once. As a result, the message may be sent by the *SmartBell* or delivered to the mobile app more than once. I initially used QoS 1, however, this caused issues when the same message sent on the ringTopic topic was delivered to the mobile app multiple times in a short period of time, as the phone would vibrate repeatedly (the mobile phone should only vibrate once when an MQTT message is received).

Topic	Message
ring/7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8FbO=	5dtsSTR0bN6q0W17jfTr4PS6js21r83ng
ring/7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8FbO=	5dtsSTR0bN6q0W17jfTr4PS6js21r83ng
ring/7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8FbO=	5dtsSTR0bN6q0W17jfTr4PS6js21r83ng
ring/7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8FbO=	5dtsSTR0bN6q0W17jfTr4PS6js21r83ng
ring/7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8FbO=	5dtsSTR0bN6q0W17jfTr4PS6js21r83ng
ring/7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8FbO=	5dtsSTR0bN6q0W17jfTr4PS6js21r83ng
ring/7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8FbO=	5dtsSTR0bN6q0W17jfTr4PS6js21r83ng
ring/7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8FbO=	5dtsSTR0bN6q0W17jfTr4PS6js21r83ng

This screen capture from the CloudMQTT *websocket UI* shows that, when QoS 1 used, the mobile app received multiple MQTT messages from the *SmartBell* indicating that the doorbell had been rung. This quick succession of messages caused the desired response (mobile phone vibrates) to occur multiple times, which adversely impacted the intended discrete behaviour of the mobile app.

- **QoS 2** – message delivered to the mobile app exactly once. Although this is the slowest level of service, it guarantees that the message is delivered to the mobile app once, which avoids the aforementioned errors. To achieve this, a *four-part handshake* is used:



PUBLISH – sender transmit message to receiver and await confirmation (**PUBREC**)
PUBREC – receiver send message to confirm receipt of message. If sender does not receive a **PUBREC**, then it will resend the message.
PUBREL – sender send message to confirm that received the **PUBREC** message. When receiver receives this message, original message can be passed to subscribers (this must not be done before the **PUBREC** message is received to avoid duplicates)
PUBCOMP – receiver send message to confirm process is complete.

Finally, the property messageReceived_ring of the MQTT class is set to '0' on **Lines 29-30**. Zero is the base case state for these 2 variables, as they are only temporarily assigned the value '1' when a message received on their associated topic, allowing the *Python* code to monitor the message receival status.

2. newMessage

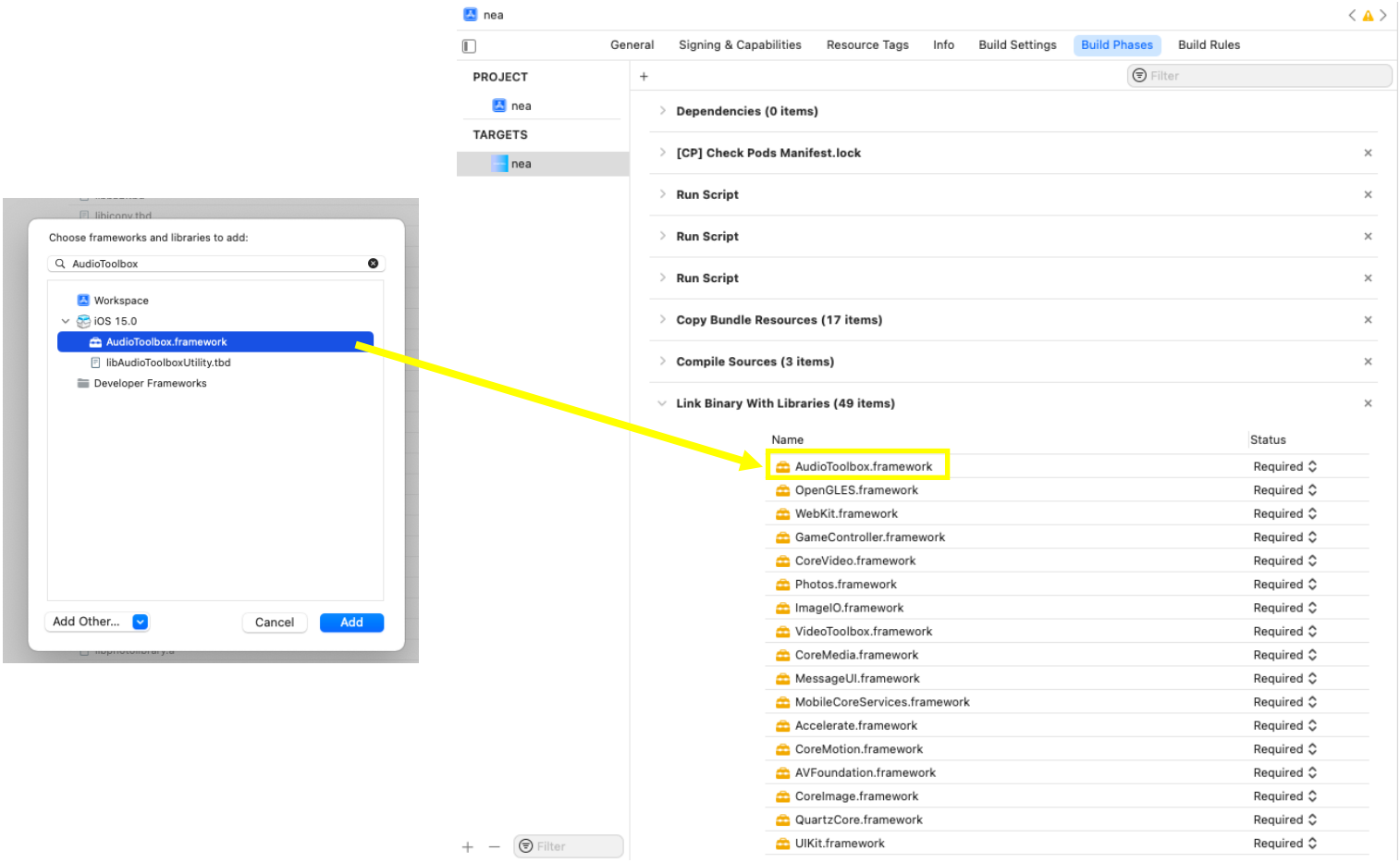
When a new message is received, the newMessage method (belonging to the MQTTSessionDelegate protocol) is called. The parameters listed on **Lines 33-34** show the information sent with each message that is received, including data and topic. As the received data is an NSData object, it is a bytes object, which is not readable by humans or in a useful format for use within the *Python* code, the initWithData method is used in **Line 36** to convert the bytes object into a string with *UTF-8 encoding*. This string is assigned to the variable dataStr. If the dataStr is empty (**Line 39** – checked by using the isEqualToString method) and the message is sent to ringTopic, this indicates that the message received has been sent by the *SmartBell* because the doorbell has been. Following the successful receipt of a message, the corresponding status property messageReceived_ring is set to '1' (**Line 41**) so that this status change can be detected by the *Python* code, triggering the appropriate response.

3. publish

The publish method is used in 2 scenarios: to transmit an MQTT message from the mobile app which the user is signed into to the *SmartBell* that the user wants to pair with/unpair from, and to transmit the audio message data for an audio message selected by the user to be played through the *SmartBell*. As a session object must always be created prior to communicating via MQTT, **Lines 46-56** show the process of establishing a connection with the CloudMQTT broker. The message topic is defined in the *Python* code and corresponds to either the ID of the *SmartBell* which the user wishes to pair with/unpair from or the type of audio message and the account ID of the user. The message topic is assigned to the variable publishTopic, which is a property of the MQTT class. The message data (publishData) to be sent via MQTT is defined in the *Python* code, and is either an empty string, indicating that the user wishes to unpair from the *SmartBell* subscribed to publishTopic, the user's account ID, indicating that the user with that account ID wishes to pair with the *SmartBell* subscribed to publishTopic, or the audio message data to played by the *SmartBell* subscribed to publishTopic. As the MQTT message data must be sent as a bytes object, the method dataUsingEncoding is applied to the property publishData to create a NSData object data (**Line 58**). The publishData method is used to transmit publishData (**Line 59**) to publishTopic (**Line 60**) with the Boolean parameter retain set to 'NO' so that pairing instruction/audio message data is not stored on the broker (doing so could cause issues if the *SmartBell* reconnects to the network and subsequently downloads this message). Finally, the disconnect method is called on the session object to terminate the existing connection with the broker; this is important as my broker subscription is limited to 25 concurrent connections, so stagnant connections must be avoided.

4. vibratePhone

When a message is received on ringTopic and the MQTT class property messageReceived_ring is assigned the value '1' temporarily, a thread running in the *Python* code detects this status change and calls the vibratePhone method using the *Pyobjus* module. To access the *iOS system sounds*, I added AudioToolbox.framework to my Xcode project:



The AudioToolbox/AudioToolbox.h *header file* must also be imported from the *AudioToolbox framework* (**Line 2**). By calling the inbuilt AudioServicesPlayAlertSound method from the custom vibratePhone method (**Line 72**) and passing '4095' as the *SystemSoundID* parameter, I was able to make the iPhone vibrate when the doorbell was rung. This is a crucial **accessibility** feature, as it ensures that users with hearing difficulties are also notified when the doorbell is rung.

5. notifyPhone

When the image of the visitor has been captured, formatted and uploaded to Amazon S3 storage, the thread running in the *Python* code detects this and calls the notifyPhone method using the *Pyobjus* module. The steps required to access the *iOS system sounds* are explained above. When notifyPhone is called, the in-

built method AudioServicesPlayAlertSound is called (**Line 77**) and '1022' is passed as the *SystemSoundID* parameter (corresponds to filename *Calypso*.caf), I was able to make the iPhone play an alert sound when the visitor image was available and displayed in app GUI.

MQTT class – Python

The **Pyobjus** *Python* module must be used to interface the mobile app coded in *Python* with the *Objective-C* coded MQTT class detailed above. The class autoclass, which is imported from the *Pyobjus* module, is used 3 times within the *Python* code to instantiate and create an object of the *Objective-C* MQTT class:

1. User receives an alert on the mobile app when the **doorbell is rung**
   Python function: createThread_ring

2. User can **pair** with/unpair from a Raspberry Pi doorbell (*SmartBell*)
   Python method: pair

3. User can send an **audio message** via the mobile app to be played through the Raspberry Pi's speaker
   Python method: transmitMessage

createThread_ring

While the workaround of running the MQTT code in *Objective-C* and using *Pyobjus* to interface between the *Objective-C* MQTT class and the *Python* code was viable, it required a means of persistently checking the status of messageReceived_ring (a property of the MQTT class) from the *Python* code. Without doing so, it was impossible for the *Python* code to register the receipt of a MQTT message. Initially, I created a function which contained *infinite while loop* that checked the status of messageReceived_ring and reacted accordingly if the status was set to '1':

```
while True:
    if mqtt.messageReceived_ring == 1:
        # SmartBell doorbell has been rung
        # do stuff (i.e. notify user through mobile etc.)
```

However, the use of an *infinite while loop* caused the code to get stuck inside the *while loop* and disabled the running of the mobile app. To overcome this, I contained the *infinite while loop* within a *background thread,* using the imported Thread class. This thread ran in *pseudo-parallel* to the rest of the program, allowing normal execution of the mobile app to continue, whilst simultaneously checking the status of messageReceived_ring:

```
1  def createThread_ring(accountID, filepath):
2      MQTTPython = autoclass(
3          'MQTT') # autoclass used to load Objective-C class 'MQTT' and create a
4  Python wrapper around it
5      mqtt = MQTTPython.alloc().init() # instance of the Objective-C 'MQTT'
6  class created
7      mqtt.ringTopic = f"ring/{accountID}" # the Objective-C property
8  'ringTopic' is assigned
9      mqtt.connect()  # call Objective-C method which connects to the MQTT
10 broker
11     visitorImage_path = join(filepath, 'visitorImage.png') # path to store
```

```
12  visitor image on mobile app
13      thread_ring = Thread(target=ringThread, args=(mqtt, visitorImage_path))
14  # thread which checks status of Objective-C property 'messageReceived_ring'
15      thread_ring.start() # start the thread
```

The createThread_ring method is called once the user is logged into their account, as the MQTT session must subscribe to the topic associated with the user's account ID (**Line 6**) before the background thread threadRing is started. This ensures that the user receive a notification through the mobile app when the *SmartBell* paired with their account is rung.

The class autoclass is used to create a *Python* wrapper around the *Objective-C* class MQTT and assigns this *Python*-wrapped object to MQTTPython (**Line 2**). MQTTPython can now be used to interface with the *Objective-C* class MQTT. All the expected *Objective-C* behaviours can be accessed through this interface using a *Pythonic*-syntax. For example, to create an instance of the *Objective-C* class MQTT through the *Python* interface (**Line 4**), the methods that must be called correspond with those called when creating a class instance in *Objective-C*, but the syntax formatting is unmistakably *Pythonic*:

Objective-C instantiation	**Python** instantiation
`MQTT *mqtt = [[MQTT alloc] init];`	`mqtt = MQTTPython.alloc().init()`

To connect the MQTT session instance to the broker CloudMQTT, the *Objective-C* method connect must be called (**Line 8**). At first, I connected to (and disconnected from) the broker during each loop through the *infinite while loop* in the *background thread* threadRing, as my initial design used autoclass to create a new instance of the *Objective-C* class MQTT on each loop. Not only was this design inefficient, the regularly calling the connect method resulted in the following error log in Xcode:

```
2021-11-16 00:38:11.655066+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:11.770450+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:11.871823+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:11.973181+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:12.074559+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:12.181688+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:12.283035+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:12.384677+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:12.486130+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:12.587458+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:12.688883+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:12.789884+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
2021-11-16 00:38:12.891389+0000 nea[4570:242297] [MQTTSessionSynchron] waiting for connect
```

Without sufficient time to establish a stable connection with CloudMQTT broker, the program stalled.

To overcome this, I created the MQTT session instance (mqtt) and connected to the CloudMQTT broker from outside of the ringThread function. As shown in **Line 13**, I passed this session instance as an argument to the ringThread function. As such, each loop of the *infinite while loop* can check the message status of the same MQTT session instance (mqtt). I also passed the path to store the latest visitor image (visitorImage_path), as created on **Line 11**.

ringThread

The function ringThread checks the status of the *Objective-C* property messageReceived_ring, which indicates whether a message has been published to ringTopic. If a message is received, ringThread also coordinates the appropriate response to notify the user and displays the image of the visitor through the mobile app.

```
1  def ringThread(mqtt, visitorImage_path):
2      while True:
3          if mqtt.messageReceived_ring == 1: # if message received on topic
4  'ring/accountID' by Objective-C MQTT session (i.e. SmartBell doorbell rung)
5              try: # runs successfully if visitor image already exists on app
6                  for visitorImage in
7  MDApp.get_running_app().manager.get_screen(
8                          'VisitorImage').ids.visitorImage.children:
9  # 'visitorImage' is a nested Kivy float layout, whose children are images of
10 the visitor
11                     visitorImage.opacity = 0 # set the opacity of each
12 existing vistor image to 0, so that the previous visitor image is not shown
13             except: # if visitor image doesn't already exist on app
14                 pass
15             mqtt.messageReceived_ring = 0  # value of 'messageReceived_ring'
16 must be reset to 0 so that new messages can be detected in Python code
17             mqtt.vibratePhone() # calls Objective-C method to vibrate mobile
18             MDApp.get_running_app().manager.get_screen('VisitorImage').ids.loa
19 ding.opacity = 1 # reset opacity of image loading gif
20             MDApp.get_running_app().manager.get_screen('VisitorImage').ids.fac
21 eName.text = "Loading..." # reset text of visitor image name label
22             MDApp.get_running_app().manager.current = "RingAlert" # open Kivy
23 screen to notify user that the doorbell has been rung
24
25             visitID = str(mqtt.messageData.UTF8String()) # decode message
26 published to topic 'ring/accountID' by Raspberry Pi doorbell
27             createThread_visit(visitID) # visit thread only called once
28 doorbell is rung to reduce energy usage.
29             downloadData = {"bucketName": "nea-visitor-log",
30                  "s3File": visitID}  # creates the dictionary which stores
31 the metadata required to download the png file of the visitor image from
32 AWS S3 (via the server REST API)
33             responseMessage = b'error'  # message returned by REST API if the
34 visitor image uploaded by the Raspberry Pi is not yet available on AWS S3
35
36             while responseMessage == b'error':  # while loop continue looping
37 if message returned by REST API is b'error', as visitor image uploaded by the
38 Raspberry Pi is not yet available on AWS S3
39                 response = requests.post(serverBaseURL + "/downloadS3",
40                                          downloadData)  # request sent to
41 custom REST API, which uses 'boto3' module to attempt to download the visitor
42 image with name 'visitID' from AWS S3
43                 responseMessage = response.content  # bytes content of message
44 returned by REST API
45                 time.sleep(0.5)  # time delay to reduce number of requests to
46 AWS API, reducing running costs
47
48             visitorImage_data = response.content  # stores visitor image bytes
49             f = open(visitorImage_path,'wb')  # opens file to store image
50 bytes (opens in 'wb' format to enable bytes to be written to this file)
51             f.write(visitorImage_data)  # writes visitor image bytes data to
52 the file
53             f.close()
```

```
54              MDApp.get_running_app().manager.current = "VisitorImage"
55              visitorImage = AsyncImage(source=visitorImage_path,
56                                        pos_hint={"center_x": 0.5,
57                                                  "center_y": 0.53})
58  # AsyncImage loads image as background thread, so doesn't hold up running of
59  program if there is a delay in loading the image
60              visitorImage.reload () # ensures latest visitor image is loaded
61              mqtt.notifyPhone() # calls Objective-C method to play notification
62  sound through mobile phone
63
64  MDApp.get_running_app().manager.get_screen('VisitorImage').ids.visitorImage.ad
65  d_widget(visitorImage)  # accesses screen ids of 'VisitorImage' screen and
66  adds the visitor image as a widget to a nested Kivy float layout
67  MDApp.get_running_app().manager.get_screen('VisitorImage').ids.loading.opacity
68  = 0 # set opacity of image loading gif to zero as image is loaded and
69  displayed
70
71          else:
72              time.sleep(3) # reduce energy usage of mobile as checking status
73  of 'messageReceived_ring' less frequently
```

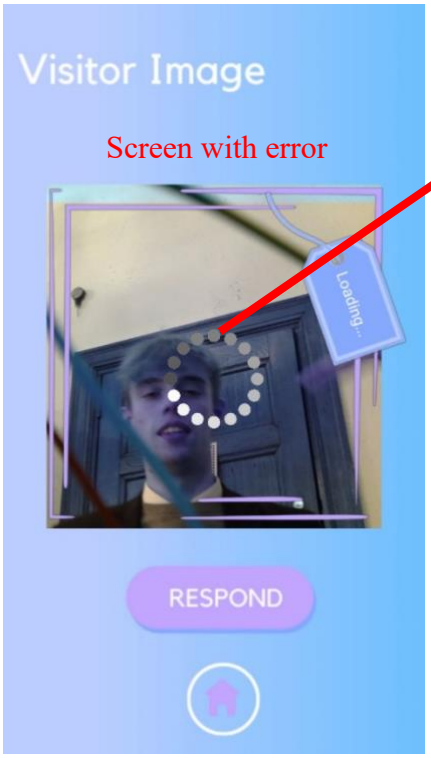
Explanation

On each loop through the *infinite while loop*, 1 of 2 paths are taken:

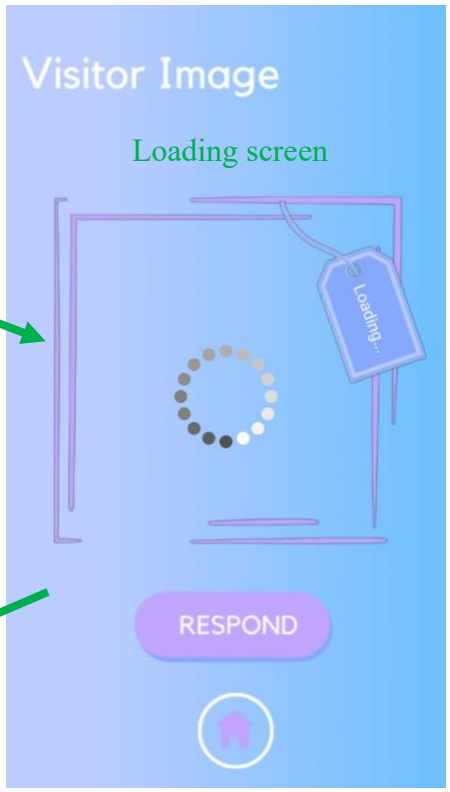
1. *IF* mqtt.messageReceived_ring is equal to *1* (**Line 3**)*:*
    - User is notified that *SmartBell* doorbell has been rung and visitor image is downloaded and displayed to user.
2. *IF* mqtt.messageReceived_ring is equal to *0* (**Line 71**)*:*
    - *Infinite while loop* pauses for 3 seconds, and then loops again, checking the status of mqtt.messageReceived_ring.

Path 1 – *SmartBell* doorbell has been rung

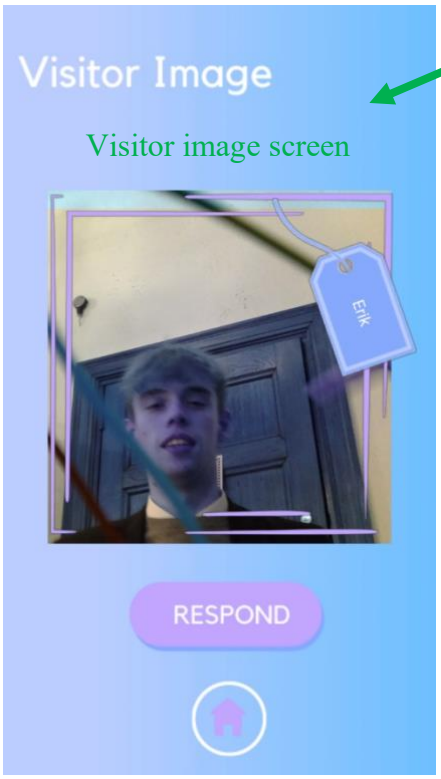
Before downloading the latest visitor image, it is necessary to check whether there is a previous visitor image saved locally on the mobile phone. If so, the *opacity* of this image must be set to 0.0 to avoid showing the incorrect image initially. Avoiding this step results in the following error while the latest visitor image is downloaded:



Although the loading gif (*Kivy image widget* with ID loading) and image label (*Kivy widget* with ID faceName) indicate that the latest visitor image is still being downloaded, the previous visitor image is still visible. As the visitor image widget is added through *Python* as a child of the *Kivy FloatLayout widget* with ID visitorImage (**Line 64**), to check whether a visitor image widget already exists, it is necessary to iterate through the children of the *Kivy FloatLayout widget* visitorImage (**Line 6**). If visitorImage has a child widget (i.e. the previous visitor image), the opacity of the child widget is set to 0.0 (**Line 11**). As a result, while the latest visitor image is downloaded, the following screen is shown.



Once the latest visitor image has been downloaded, it is added as a new *Kivy image widget* with opacity 1.0 to the *Kivy FloatLayout widget* visitorImage (**Line 64**) and the opacity of the *Kivy widget* loading is set to 0.0 (**Line 67**). As a result, the image of the visitor captured by the *SmartBell* is displayed to the user alongside the name of the visitor (if identified), as shown in the following screen:



Once the *Python* code has registered the status change of the *Objective-C* property messageReceived_ring, the value of mqtt.messageReceived_ring must be reset to *0* (**Line 15**). Avoiding this step would cause the ringThread to continue looping regardless of whether a new message had been published to ringTopic.

The name of each visitor image captured by the *SmartBell* and stored on **AWS S3** corresponds to its visit ID. Therefore, to download the image of the visitor captured by the *SmartBell*, the visit ID associated with the image is needed. When the Raspberry Pi publishes the MQTT message to ringTopic indicating that the doorbell has been rung, the visit ID is sent as bytes in the message data and assigned to the property messageData in the *Objective-C* code. On **Line 25**, messageData is decoded using *UTF-8,* converted into a string format using the *Python* str() function and assigned to the *Python* variable visitID.

A second thread is created within the ringThread by calling the function createThread_visit on **Line 27**. The createThread_visit function polls the MySQL table *visitorLog* until further information (including the visitor's name, if identified) about the latest visit has been added to the table as a new *record*. By creating this thread only once messageReceived_ring is set to *1* (indicating the doorbell has been rung), the *Python* code avoids unnecessarily polling the MySQL table *visitorLog*, which reduces energy usage of the mobile app and the running costs of the mobile app by limiting the number of requests to the **AWS Elastic Beanstalk** environment which hosts the REST API.

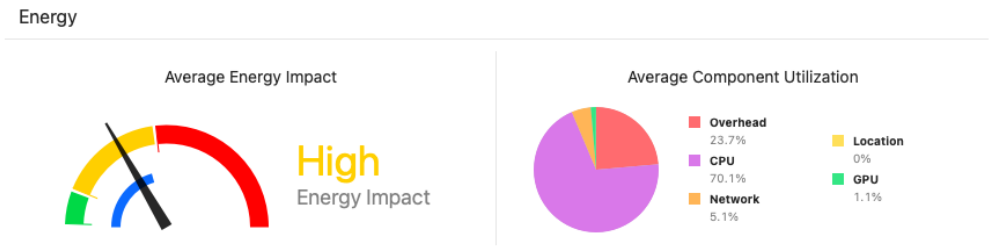
To interact with **AWS S3** (where the visitor images are stored) using *Python,* the module boto3 must be used. However, much like the paho-mqtt-client module, I discovered that boto3 cannot be installed on iOS due to **Apple Sandbox**. To overcome this, I created a REST API on my **AWS Elastic Beanstalk** environment which uses the boto3 module to download the latest visitor image and return the image byte data. **Line 39** shows the requests module being used to send a request to the */downloadS3* path of the REST API with a *dictionary* (downloadData) that contains the required information to download the latest visitor image. As the Raspberry Pi will upload the visitor image with varying speeds depending on the WiFi connection strength, a *while loop* is used to send a request to the */downloadS3* path until the latest visitor image has been uploaded by the Raspberry Pi and the REST API successfully returns the image. When the REST API fails to return latest image, the message '*error*' is returned instead. However, the content of the message returned by the REST API and assigned to the *Python* variable response can be viewed either in *bytes* (response.content) or *Unicode* (response.text). Initially, I assigned responseMessage to response.text so that I could compare responseMessage to the *Unicode string* '*error'*, which would indicate that the latest visitor image was not yet available on **AWS S3** and so the *while loop* should continue looping. However, using this approach I noticed that the mobile app was taking an excessively long time to download the latest visitor image. Upon further investigation, I realised this was because the program was attempting to encode the visitor image bytes in *Unicode*, which is time consuming. The output of this *Unicode* encoding was as follows:

Once I had identified where the issue was arising, it was easy to solve: instead of assigning `responseMessage` to `response.text`, I assigned it to `response.content` (which gives the message data in *bytes* format) on **Line 48** and compared `responseMessage` to the *bytes string b'error'*, rather than the *Unicode string 'error'*, on **Line 36**. This radically reduced the loading time of the latest visitor image, improving the user's experience.

Once the visitor image bytes data is downloaded, it is written to a *.png* file on the mobile app **(Line 51)** and added as a *Kivy image widget* on **Line 64**.

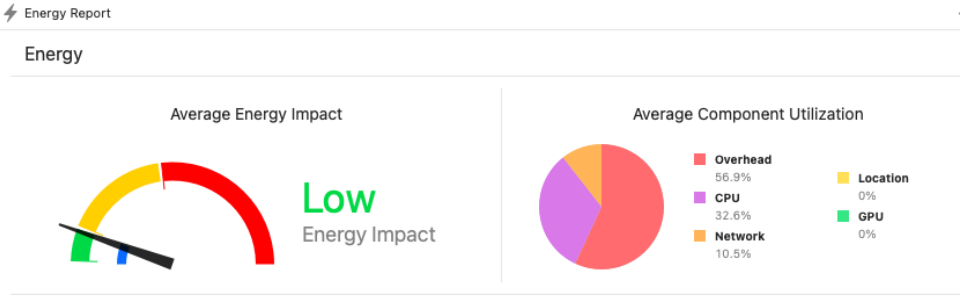
Path 2 – *SmartBell* doorbell has not been rung

Initially, I simply used the `pass` statement to instantly loop through the *while loop* again if no change to the value of `mqtt.messageReceived_ring` was detected on **Line 3**. However, upon inspection of the energy usage statistics for the mobile app (available in Xcode), I noticed that the *energy usage* of the mobile app was incredibly *high*:



This was concerning as it caused the app to drain the battery of the mobile app.

However, by replacing the `pass` statement with `time.sleep(3)`, the frequency at which the thread looped through the *while loop* was significantly reduced. As such, the energy usage of the mobile app was reduced to *low*:



REST API Communication

As explained previously, the mobile app must interact with the **MySQL database** and **AWS S3 storage buckets** during run-time to store personal details and files.

To interact with the MySQL database AWS S3 buckets, the mysql and boto3 module must be used respectively.

However, I faced many issues importing these 2 modules into the mobile app. To resolve this, I created an *AWS Elastic Beanstalk (EB) environment* which hosts server code that can be accessed through a **REST API**. Each route on the REST API navigates to area on the server which performs specific interactions with the MySQL database and S3 storage. Therefore, it should be noted that arrows between the mobile app and the MySQL database/S3 storage in the system hierarchy diagram are not direct links, since all data must pass through a path on the REST API.

Setup

```
serverBaseURL = "http://nea-env.eba-6tgviyyc.eu-west-2.elasticbeanstalk.com/"
# base URL to access REST API server
```

serverBaseURL is the base URL used to access the *EB environment* hosting the REST API. By joining this base URL with specific URL routes, each different functionality of the REST API can be accessed by the mobile app.

Usage – MySQL database

To send requests to specific routes on the REST API, the requests module must be used in the **mobile app**:

```
response = requests.post(serverBaseURL + "/latest_visitorLog",
dbData_accountID)
```

The post class of the requests module is used as the mobile app must *POST* data to the server to be processed. The data is sent as a *json* object so that multiple individual data items can be passed and accesses by the server. In the example above, the *josn* object dbData_accountID assigns the user's account ID to the key *'accountID'*:

```
{'accountID': self.accountID}
```

The **REST API** route *'/latest_visitorLog'* accesses this *json* object passed by the mobile app:

```
data = request.form
```

The form attribute of the imported request class stores the data passed to the REST API server.

Having established a connection with the MySQL database, the **REST API** server executes the following:

```
query = "SELECT visitID, faceID FROM visitorLog WHERE accountID = '%s' ORDER
BY imageTimestamp DESC" % (data["accountID"])
myCursor.execute(query)  # the query is executed in the MySQL database which
the variable 'myCursor' is connected to
result = myCursor.fetchone()
return {'result': result}
```

By accessing the value for the key *'accountID'* in the *json* object dbData_accountID that the user passes to the REST API server, the above SQL statement retrieves the visitID and faceID associated with the visits to the doorbell paired with the user's account. Moreover, the SQL keyword ORDER BY DEC is applied to the field imageTimestamp so that the data retrieved from the database is sorted chronologically.

Using the method fetchone of the class myCursor, the server stores an array result of the visitID and faceID for the latest visit. This array is returned to the client in a *json* object, so that the required data can be accessed by the **mobile app**:

```
response = response.json()['result']
```

Using the json method of the response object, the mobile app can access the *json* data returned by the server.

Not only does the use of a REST API server make it possible to interact with the MySQL database, but it also reduces the processing that must be performed locally on the mobile app. As a result, the code for the mobile app is less verbose and is abstracted where possible.

Usage – S3 storage buckets

To download files store in the AWS S3 buckets, the REST API server is also used. By sending a *POST* request and the required data to the route *'/downloadS3'*, the **mobile app** is able to download the latest visitor image:

```
downloadData = {"bucketName": "nea-visitor-log","s3File": visitID}  # creates
the dictionary which stores the metadata required to download the png file of
the visitor image from AWS S3
response = requests.post(serverBaseURL + "/downloadS3",downloadData) # request
sent to custom REST API, which uses 'boto3' module to attempt to download the
visitor image with name 'visitID' from AWS S3
responseMessage = response.content  # bytes content of message returned by
REST API
```

As with the queries to the MySQL database, the mobile app is abstracted from the processing required to communicate with the AWS S3 storage buckets. By accessing the content attribute of the response object, the mobile app downloads the pickled bytes of the latest visitor image, which can then be written to a *.png* file locally and displayed to the user.

Visitor Log

To help the user monitor who has visited their home, the *Visitor Log* screen of the mobile app displays a scrolling list with the **details of each visit**:

1. Visitor name (if identifiable)
2. Date/time of visit
3. Visitor image

I have also implemented a *merge sort* algorithm to enable the user to **sort the visitor log** by either:

1. Visitor's name
2. Date and time of the visit.

Furthermore, the user is also provided with two **summary statistics** regarding the use of their doorbell:

1. Average number of visits per day
2. Average time of visit

To retrieve the details of all the visits to a user's doorbell, the mobile app interacts with the REST API server path *'get_visitorLog'*, which returns an array visitors:

```
self.visitors = requests.post(serverBaseURL + "/get_visitorLog",
dbData_accountID).json() # get visitor log details associated with user's
account
```

The array visitors contains a list of tuples which store the core details for each visit:

```
[('04.10,1646971806.1329312', 'ScQQv3eWBaiUXMcMmdA36jQJKX5KmAj0P5SDnJFhupp',
'GyBI37QOh6xUifE3YSGdYRkvnFqQrUWFffRfxHVxN8F='),...]
```

0. *Index 0* of each tuple stores the time of the visit in two formats:
    a. *Index 0*-4 = Decimal time
    b. *Index 6*+ = Number of seconds since beginning of epoch (Jan 1st 1970)

    For the visitor log, the date is displayed in the format '*DD-MM-YYYY, HH:MM:SS'*:

```
self.epoch = float(visitDetails[0][6:])
self.date = time.strftime('%d-%m-%Y, %H:%M:%S', time.gmtime(self.epoch))
# convert epoch time in seconds into actual time/date
```

To access the substring with the number of seconds since the beginning of the epoch, a slice of the value at *index 0* (characters at *index 6* of the string onwards) is assigned to epoch as a *float* type variable. Using the class time.gmtime, the seconds stored in epoch are converted into their constituent parts which specify individual properties about the instance in time, including year, month, day, hour, minute, second. This data is then formatted using the class time.strftime into the format '*DD-MM-YYYY, HH:MM:SS*':

```
11-03-2022, 04:10:06
```

1. *Index 1* of each tuple stores the face ID of each visit:

   By sending a request to the REST API path *'get_faceName'*, the name associated with the faceID can be retrieved and assigned to *Index 1* of the tuple (overriding the faceID):

```
result = requests.post(serverBaseURL + "/get_faceName",
dbData_faceID).json() # get face name associated with face ID
faceName = result[0]
self.visitors[index][1] = faceName # replace face ID with face name
```

2. *Index 2* of each tuple stores the visit ID of each visit:

   The visitID is used to download the visitor image associated with the visit from AWS S3 storage by sending a request to the REST API server path *'downloadS3'*. The visitID is also used to construct the path to which each visitor image is saved to, ensuring that each image is saved to a unique path that can be accessed when creating the visitor log:

```
self.visitorImage_path = join(self.filepath, self.visitID + '.png')
```

   The visitorImage_path is assigned to *Index 2* of the tuple (overriding visitID)

To create the final array required to display the visitor log details, *Python's* map function is used alongside the anonymous function lambda:

```
self.visitorsFormatted = list(map(lambda a:(a[0][6:], a[1], a[2], a[3]),
self.visitors)) # convert required values in tuple into list
```

The map function runs the lambda function over the array visitors, applying the operation highlighted in yellow to each element in the array visitors and storing the result in the array visitorsFormated.

As a result of the above formatting, each tuple in visitorsFormated has the following format:

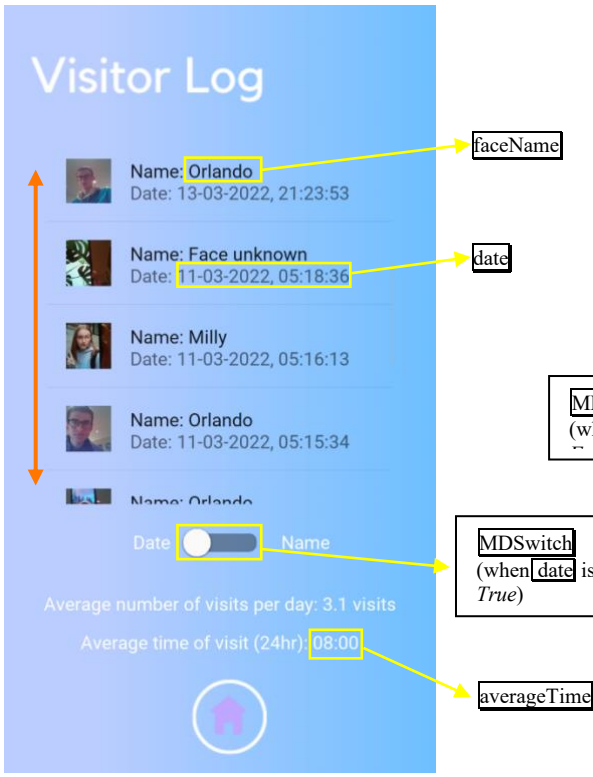
0. Time in seconds since epoch began
1. Face name of visitor
2. Path to visitor image
3. Date/time of visit in '*DD-MM-YYYY, HH:MM:SS*' format

Before the data in visitorLog is displayed to the user in the mobile app GUI, it is sorted using a *merge sort* algorithm based on the criteria selected by the user using the *KivyMD widget* MDSwitch in the *VisitorLog* screen

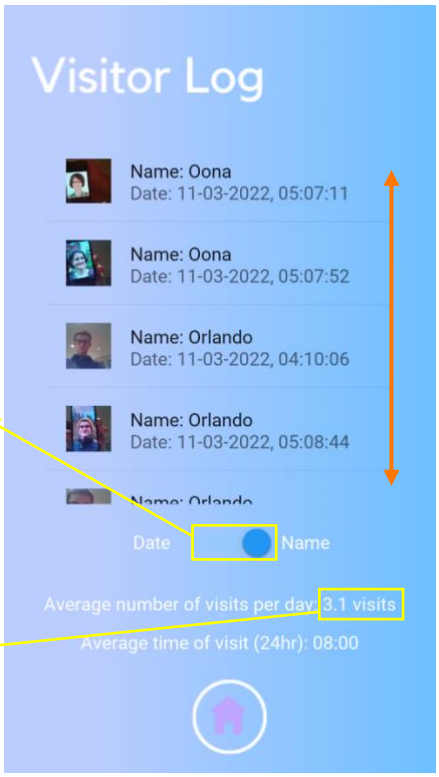
```
MDSwitch:
    pos_hint: {'center_x': .5, 'center_y': .3}
    width: dp(64)
    selected_color: 37/255, 41/255, 88/255, 1
    on_active:
        if root.date == True: root.displayLog('name')
        else: root.displayLog('date')
```

If the status of *Python* attribute date (which belongs to the *Python* class visitorLog that is associated with the *VisitorLog* screen) is set to *True*, this indicates that the visitor log is currently sorted by date. Therefore, when the *KivyMD widget* MDSwitch is activated, the the *Python* method displayLog of the root class is called with the argument '*name*', so that the visitor log is now sorted by name rather than date. If date is set to *False*, displayLog is called with the argument *'date'*.

Sorted by date



Sorted by name



The *scrollable list* above which displays the visitor log data (as sorted by the *merge sort* algorithm) is implemented using the *Kivy widget* ScrollView:

```
ScrollView:
    size_hint: 0.8,0.45
    pos_hint: {"center_x": 0.5, "top": 0.8}
    MDList:
        id: container
```

I chose to use a scrollable list, as the visitor log has an undetermined length, so the scrollable list allows a list of unspecified length to fit on a screen of a specified size. To create new rows within the scrollable list, I used the *KivyMD widget* TwoLineAvatarListItem, as it enabled me to create rows which contained two items of text data (the visitor name and the visit date) *and* an image of the visitor:

```
1  self.visitorsSorted = self.mergeSort(self.visitorsFormatted, dateORname)
2  # sort list storing visit details by visitor name or by date of visit
3  self.ids.container.clear_widgets() # clear existing visitor log scroll list
4  for visit in self.visitorsSorted:
5      rowWidget = TwoLineAvatarListItem(text=f"Name: {visit[1]}",
6      secondary_text = f"Date: {visit[3]}") # create row widget with visitor
7  name and date
8      rowWidget.add_widget(ImageLeftWidget(source = visit[2])) # add associated
9  visit image to row widget
10     self.ids.container.add_widget(rowWidget) # add row widget to scroll list
```

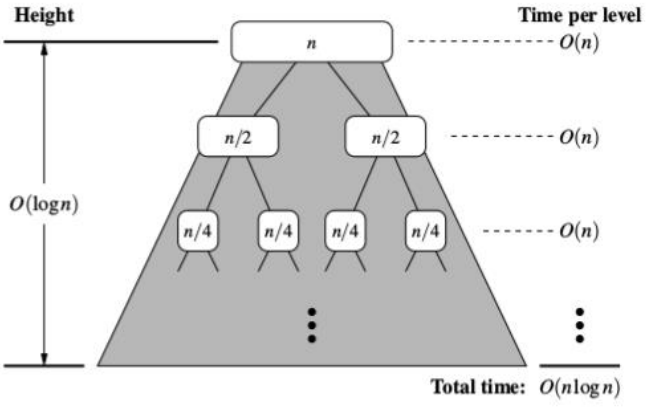
On **Line 1**, the array storing the details for each visit is sorted into the order specified by the user through the mobile app GUI using a *merge sort* algorithm mergeSort (explained below). On **Line 3**, the clear_widgets is called to remove all existing row widgets in the *Kivy widget* MDList with ID *'container'* when the user selects a new sorting criteria for the visitor log. This means that only the visits in the newly ordered visitor log exist as rows in the visitor log list. By iterating through each element (visit) in visitorsSorted (**Line 4**), a text-based row widget rowWidget with the visitor's name and the date of the visit is created using the method TwoLineAvatarListItem. To add the visitor image as an icon to each row, the method ImageLeftWidget is used to create a child widget image which can be added to rowWidget (**Line 8**). Note that the source argument for each child widget image is the path to the visitor image associated with the visit.

To sort the visitor log by either the *name* of the visitor or the *date* of the visit, I decided to use a **merge sort** algorithm as it has the lowest *time complexity* of the sorting algorithms:

O(n log(n))

This time complexity is calculated by analysing the merge sort algorithm process:

1. *Divide* the list of length *n* in halve at each step until you have single elements → **O(log n)**
2. *Compare and merge* elements in sub-lists created during *Divide* stage → **O(N)**



This diagram shows the worst case time complexities for each aspect of the *merge sort* algorithm.

```
1  def mergeSort(self, array, dateORname):
2      if dateORname == 'date':
3          index = 1 # specifies index of value in 'array' to be used to sort
4  array
5          self.date = True
6      else:
7          index = 0 # specifies index of value in 'array' to be used to sort
8  array
9          self.date = False
10     if len(array) > 1: # if array has length greater than 1, continue
11 splitting it
12         mid = len(array) // 2 # find middle index of array
13         left = array[:mid] # store left half of array elements
14         right = array[mid:] # store right half of array elements
15
16         self.mergeSort(left, dateORname) # recursively call the mergesort
17 function on the left of array elements to sort left half
18         self.mergeSort(right, dateORname) # recursively call the mergesort
19 function on the right of array elements to sort right half
20
21         # merge and sort left and right array
22         i = 0 # left array index
23         j = 0 # right array index
24         k = 0 # main array index
25         while i < len(left) and j < len(right):
26             # sort array by comparing and swapping values in tuple at index
27 'index'
28             if left[i][index] < right[j][index]: # value in left array less
29 than value in right array
30                 array[k] = left[i] # save value in left index into merged
31 array
32                 i += 1 # move to next index in left array
33             else: # value in left array greater than or equal to value in
34 right array
35                 array[k] = right[j]
36                 j += 1 # move to next index in right array
37             k += 1 # move to next index in merged array
38
39         # move all remaining values in left array into merged array, as no
40 more values in right array to compare with
41         while i < len(left):
42             array[k] = left[i]
43             i += 1
44             k += 1
45
46         # move all remaining values in right array into merged array, as no
47 more values in left array to compare with
48         while j < len(right):
49             array[k] = right[j]
50             j += 1
51             k += 1
52     return array
```

On **Line 2-9**, the value of index must be determined based on whether the merge sort algorithm is to sort the visitor log (stored here as array) by the visitor's first name or the visit date/time.

The *merge sort* algorithm uses a *divide and conquer* approach as follows:

1. Use recursion (**Line 16-18)** to split array into two halves of sub-arrays until each sub-array contains one element (**Line 10**)
2. When each half of array has been split into individual elements, the values of the elements in left sub-array (left) and right sub-array (right) are compared and swapped where necessary to put the elements in the correct order (**Line 25-51**)
3. Store the ordered elements of the sub-arrays in the merged array array
4. When the sub-arrays have been sorted, return the merged array array to the line at which the recursive function call was made
5. The merged array array will be assigned to the variable left or right, depending on whether the recursive function was called on a left or right half of the array
6. This allows the process of comparing and merging sorted arrays to continue until the entire array is sorted

To display the **average number of visitors** per day, I used *SQL aggregate functions* to retrieve two types of summary data about the user's visitor log:

1. Average number of visits per day

    a. To get the total visits to the user's doorbell, SQL *COUNT* function used to return total number of records associated with user in the SQL table *visitorLog*:

```
query = "SELECT COUNT(*) FROM visitorLog WHERE accountID = '%s'"
%(data["accountID"]) # retrieve total number of visits for a
user's account
```

    b. To get the time (in seconds) since the doorbell was first rung (initialTime), SQL *MIN* function used to return the earliest time when a record associated with the user was stored in the SQL table *visitorLog*:

```
query = "SELECT MIN(SUBSTRING(imageTimestamp, 7)) FROM visitorLog
WHERE accountID = '%s'" % (data["accountID"]) # retrieve time of
first recorded visit to user's doorbell
```

    The above SQL query uses *MIN* to return the minimum value stored in *imageTimestamp* from the seventh character onwards (extracted using the SQL *SUBSTRING* function), as these characters store the number of seconds that have passed since January 1, 1970, 00:00:00 (UTC) at the instance the doorbell is rung. This data can be used to extract the time and date at which the doorbell was rung.

c. To get the total number of days since the doorbell was first rung, the difference in seconds between the current time (currentTime) and the time when the doorbell was first run (initialTime) is found:

```
time = time.time()
totalDays = (currentTime-float(initialTime))/3600/24 # total days
passed since first visit to user's doorbell
```

currentTime and initialTime are stored in seconds, so to convert these values into days, the following operations must be applied:

i. Divide by *3600* to convert to hours
ii. Divide by *24* to convert to days

d. To get the average number of visits per day, the total number of visits is divided by the number of days since the doorbell was first rung:

```
averageRate = count/totalDays # average number of visits to user's
doorbell
```

2. Average time of day when a visitor visited the user's house

a. To get the average time of day at which the user's doorbell is rung (averageTime), the SQL *AVG* function is used to retrieve the average value stored in the first five characters of each data item in the *imageTimestamp* field (i.e. the time in 24hr format) associated with the user's account:

```
query = "SELECT AVG(SUBSTRING(imageTimestamp,1,5)) FROM visitorLog
WHERE accountID = '%s'" % (data["accountID"])
```

b. To ensure the average time at which the doorbell is rung is displayed in the 24hr format, rather than as a *FLOAT*, averageTime must be processed as follows:

```
hours = int(self.averageTime) # total number of complete hours
minutes = (self.averageTime * 60) % 60 # total number of minutes
remaining from complete hours
self.averageTime = str("%02d:%02d" % (hours, minutes)) # zero
padding and two decimal places for each number
```

To calculate the hour component of the average time (hours), averageTime is converted to an *INTEGER* which stores only the integer part of the average time (i.e. any decimal values are truncated).
To calculate the minutes component of the average time (minutes), averageTime is multiplied by 60 to convert it into minutes and then *modded* by 60 using the modulus operator %. This returns the remaining number of minutes on top of the complete hours (hours).

The values of hours and minutes are both formatted using *'02d'*, which necessitates that they are both at least two characters in length and that, where their values are not two characters in length, zeros are inserted from the left until they are two characters in length. This ensures that the time displayed is correctly formatted.

Standard graphical widgets

The mobile app uses several standard widgets to create an accessible Graphical User Interface. These **widgets** include (but are not limited to):

1. Text field
2. Button
3. Image
4. Label

These widgets are designed using the *Kivy* framework to create the GUI for the mobile app in such a way that the *Python* (logic) code can interface with these graphics.

Throughout the graphics design process, I made sure to comply with the following **accessibility criteria**:

1. Clear and simple layout
2. Consistency throughout all screens
3. Colour coding and contrast

These accessibility features ensure that all users can easily navigate the mobile app without giving it their full attention; this is essential, since the mobile app is designed for users who are likely to be on a conference call at the time when the doorbell is rung.

The **user experience** was also a key focus when designing the graphics, as 1 aim of the mobile app and *SmartBell* was to make the process of answering the door slightly less tedious.

Moreover, I have used a range of **signifiers** to indicate clearly to the user how they can interact with the mobile app through **affordances**. These signifiers are both **colour-coded** *and* **shape-coded**, ensuring that users who are colour-blind are still able to understand the signifiers.

Instantiating mobile app screens (Kivy)

```
WindowManager:
#initiates and orders all screens
    Launch:
    SignUp:
    SignIn:
    Homepage:
    MessageResponses_add:
    MessageResponses_create:
    MessageResponses_createAudio:
    MessageResponses_viewAudio:
    MessageResponses_createText:
    RingAlert:
    VisitorImage:
    VisitorLog:
```

`WindowManager` is a widget which initialises all the screens for the mobile app:

Launch:
SignUp:
SignIn:
Homepage:
MessageResponses_add:
MessageResponses_create:
MessageResponses_createAudio:
MessageResponses_viewAudio:
MessageResponses_createText:
RingAlert:
VisitorImage:
VisitorLog:

Each screen corresponds to a Class in the *Python* code, as this is the required structure for a *Kivy* application which allows for interaction between the *Kivy* widgets and the *Python* logic.

Text field (Kivy)

```
<MDTextField>
#creates default settings for MDTextField objects
    mode: "line"
    size_hint_x: 0.5
    color_mode: "custom"
    use_bubble: False
    line_color_normal: 128/255,128/255,128/255,1
    line_color_focus: 1,1,1,1
```

Initially, I used the *Kivy TextInput* widget to create the text boxes. However, much like the base case *Kivy Button*, the *Kivy TextInput* was clunky. Following some research into *Kivy* graphics, I discovered **KivyMD** (kivymd.readthedocs.io, n.d.), which is a library designed for the *Kivy* framework to replicate **Google's Material Design** widgets. Using this library, I was able to create intuitive and **simple** text boxes with the MDTextField widget.

To create **consistency** throughout the mobile, I decided to use a constant design for text fields. Therefore, I created global properties for the MDTextField widget; my default properties for MDTextField are as follows:

mode: "line" sets the text field type to 'line' style
size_hint_x: 0.5 sets width of the text field to half the screen width
color_mode: "custom" allows colour of text field lines to be customised
use_bubble: False prevents the cut/copy/paste bubble from appearing when text in a text field is selected in mobile app
line_color_normal: 128/255,128/255,128/255,1 sets the normal line colour of the text field to grey (as shown in *figure 10*). The line colour is given in an *rgba* format, where *r* = red (0-1), *g* = green (0-1), *b* = blue (0-1) and *a* = alpha (transparency of the colour)
line_color_focus: 1,1,1,1 sets the line colour of the text field to white (as shown in *figure 11*) when it is tapped by the user to enter text

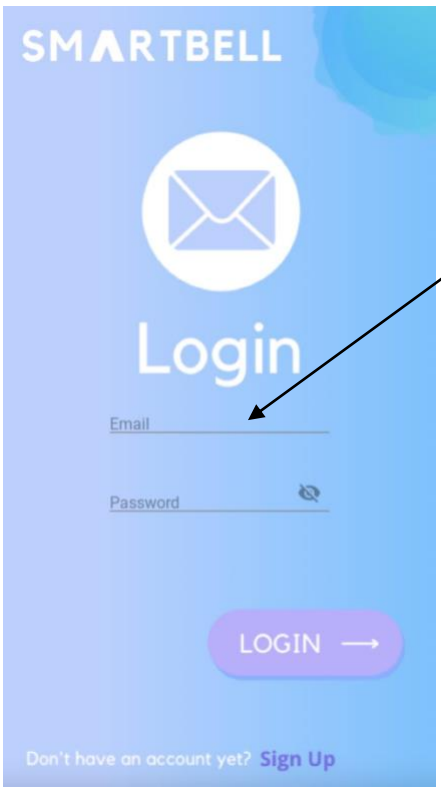


Figure 10

Example of *TextField*

Example of *TextField* when tapped (*line_color_focus*)

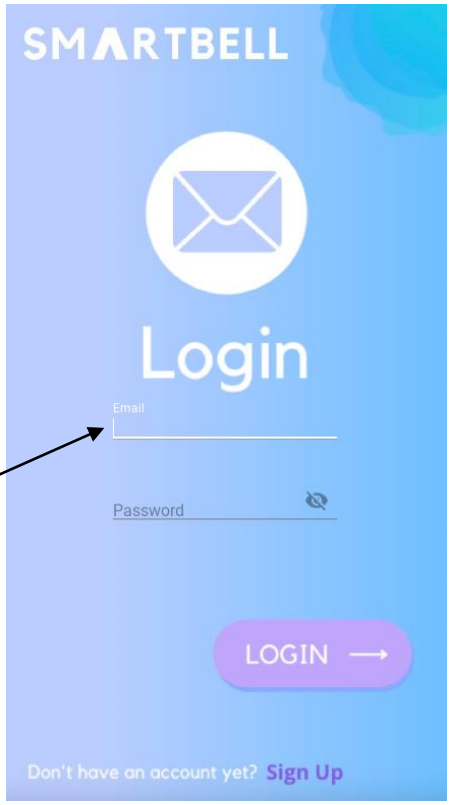
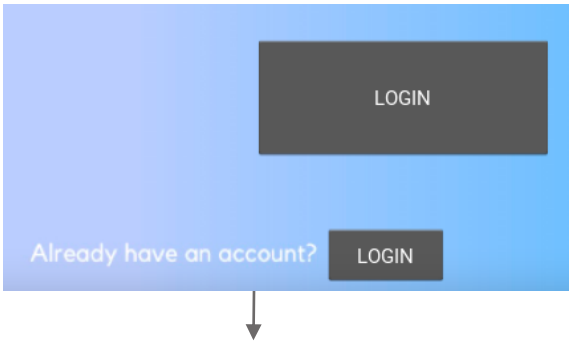


Figure 11

Button (Kivy)

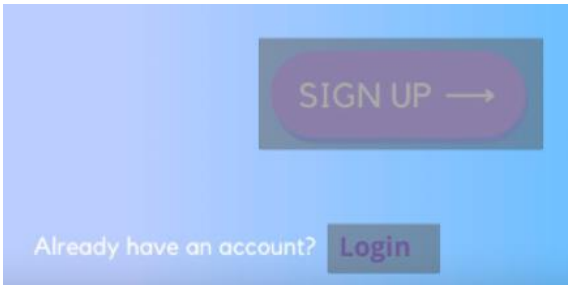
Buttons are an essential part of the UI of the mobile app as they allow the user to navigate through the app and select their desired options. As the app is designed to make it easy for home-workers to respond to visitors at their doorstep remotely, it was important to make the UI clear and visually appealing for the user.

Therefore, as the standard buttons in *Kivy* are plain and crude, during *Development*, my design of the *Kivy* buttons evolved. Below I have shown the stages of development from the standard *Kivy* button to the buttons I used in the mobile app:

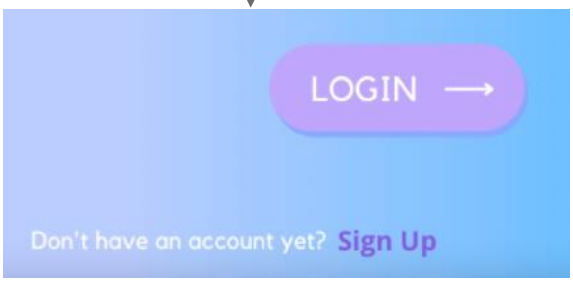


The original buttons I created using the standard *kv* markup code for a button:

```
Button:
    size_hint: 0.5, 0.1
    pos_hint: {"center_x": 0.7, "center_y": 0.2}
    text: "LOGIN"
```



A *Kivy* button placed over a graphic of a button created in Canva. Identifying the correct dimensions and coordinates of the *Kivy* button was often quite fiddly, so I always set the *alpha* value of the buttons to 0.5 to make it visible while I was adjusting these properties.



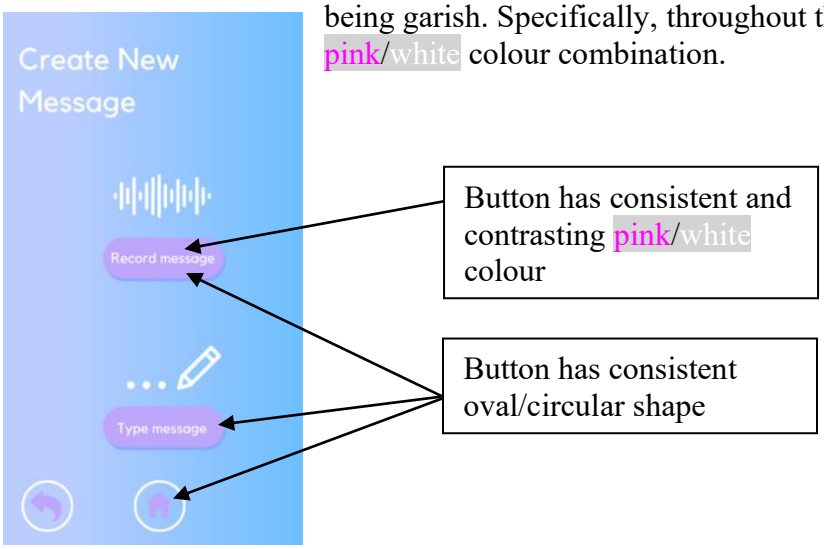
The final design of the buttons in the mobile app.
The graphic of the 'LOGIN' button is now clickable by the user but remains **visually appealing**. I created global properties for the widget Button so that all buttons were transparent:

```
<Button>
#creates default settings for buttons
    background_color: 1,1,1,0
```

Image (Kivy)

```
Image:
    id: createMessages #unique identifier for this specific image
    source: "SmartBell_createMessages.png"
    size: self.size #sets image to same size as screen
    allow_stretch: True #allows image to be stretched so that it fills the
screen
    keep_ratio: False #allows image proportions to be altered so that it fills
the screen
```

To create an accessible and **visually appealing** user interface, I designed the graphics for the mobile app using *Canva.* As a form of **signifier**, each screen is consistently **colour-coded** *and* **shape-coded** to make the app's **affordances** clear to all users (including those who are colour blind). Moreover, I have chosen colour combinations which create sufficient **contrast**, without being garish. Specifically, throughout the mobile app, I have chosen a pink/white colour combination.



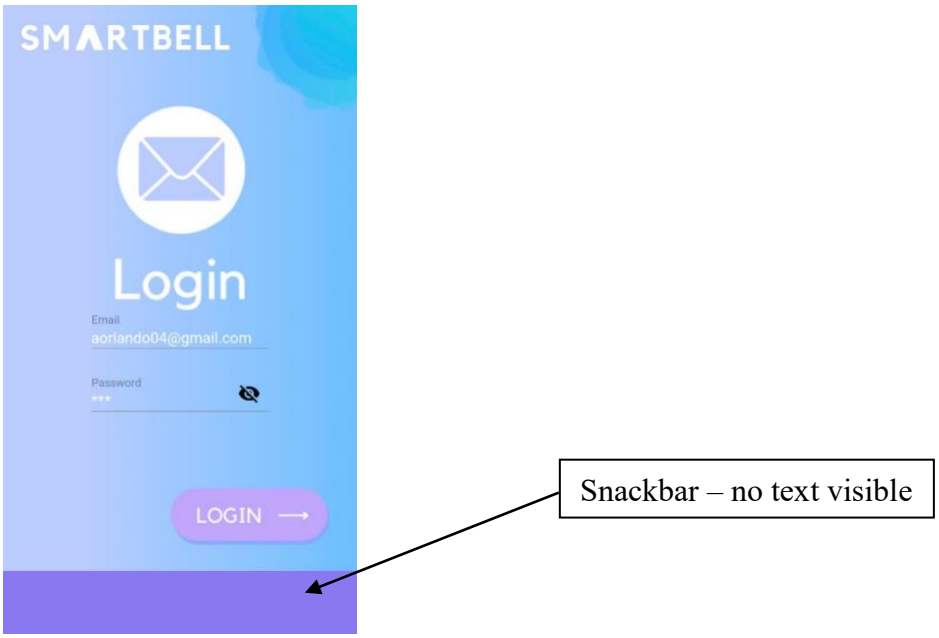
Label (Kivy)

Labels are used throughout the mobile app to communicate information to the user; usually this is feedback to the user's actions. Critically, labels should be clearly visible without obstructing the user's experience. To achieve this, I used the *Kivy Animation* widget to create **snackbars** which automatically open and close at the bottom of the mobile app, displaying the required message to the user.

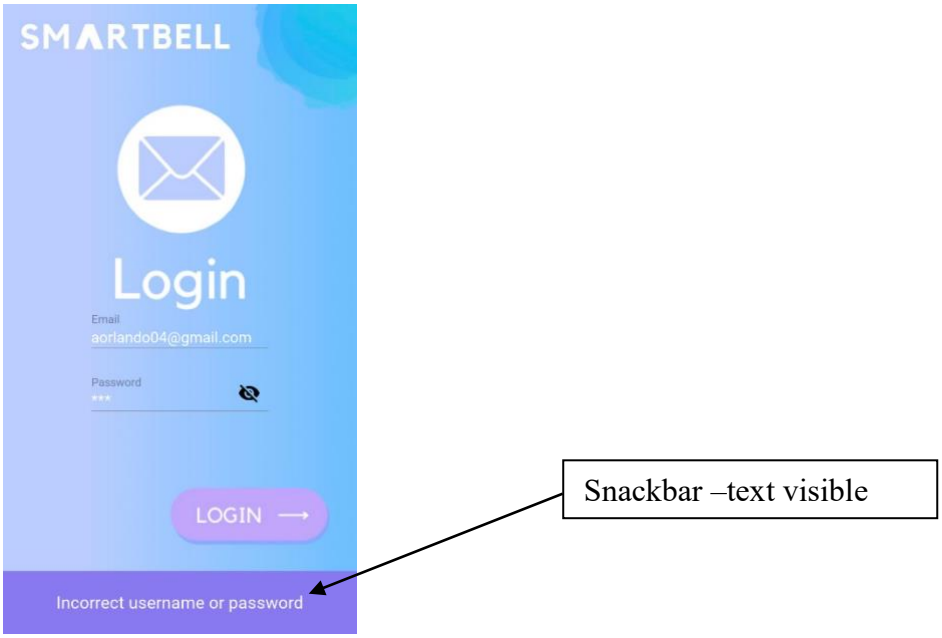
Using *Kivy*, I created the graphics for each snackbar instance:

```
1  Label:
2      id: snackbar #unique identifier for this specific label
3      canvas.before: #these graphics are created first, so appear at the back
4          Color:
5              rgba: (136/255,122/255,239/255,1)
6          Rectangle:
7              pos: self.pos #rectangle has same position as label
8              size: self.size #rectangle has same size as label
9      text_color: (1,1,1,1) #white text
10     valign: "middle" #text is aligned in the middle vertically
11     halign: "center" #text is aligned in the centre horizontally
12     size_hint: 1, 0.1 #label size
13     pos_hint: {"center_x": 0.5, "top": 0.1} #label position with respect to
14     screen
```

On **Line 3**, I initially used the root *Kivy* widget `canvas` to create the background graphics for the snackbar. However, doing so made the snackbar text invisible:



Upon investigation into the *Kivy* `canvas` widget, I realised this error arose due to the order in which the snackbar graphics were drawn. Instead, it was necessary to contain the background graphics instructions under the instruction group `canvas.before` to ensure they were the first aspect of the label to be created (i.e. behind the text of the label). The correct output is shown below:



The snackbar shown above is *static* and *permanent*, which contradicts the desired discrete nature of the snackbar. At first, I simply set the opacity property of the label to '0' to hide and reveal the snackbar. However, this motion was very abrupt, and, from feedback I received, this was detrimental to the user experience.

Using the *Kivy Animation* widget, I was able to make the snackbar label slide up from the bottom of the screen to the position shown above. To ensure that the snackbar was initially invisible to the user, I altered the pos_hint dictionary shown in **Line 13** above: I changed value of the top key from '0.1' to '0'.

As the snackbar is used in 5 separate screens/classes in the mobile app, the *Python* code to implement the Animation class is contained within the Launch class, which is inherited by all screens which use a snackbar. Doing so avoids unnecessary repetition of code. The Launch class methods openSnackbar and dismissSnackbar are below:

openSnackbar *and* **dismissSnackbar**

```
1   def openSnackbar(self):
2       # method which controls the opening animation of the snackbar
3       animation = Animation(pos_hint={"center_x": 0.5, "top": self.topHeight},
4                             d=0.03)  # end properties of snackbar's opening motion
5       animation.start(self.ids.snackbar)  # executes the opening animation
6
7   def dismissSnackbar(self):
8       # method which controls the closing animation of the snackbar
9       time.sleep(self.sleepTime)  # delay before snackbar is closed
10      animation = Animation(pos_hint={"center_x": 0.5, "top": 0}, d=0.03)
11      # end properties of the snackbar animation's closing motion
12      animation.start(self.ids.snackbar)  # executes the closing animation
```

Explanation

In the initialisation of the Animation class to create the object animation on **Line 3**, the attribute topHeight is used, rather than a constant value for the height of the snackbar. This allows the same method openSnackbar to be used to control the snackbar motion in different screens where the required height of the snackbar varies – only the attribute topHeight must be altered before calling openSnackbar. To invoke the **opening motion** of the snackbar, the start method of the Animation class is called on the *Kivy Label* widget with id 'snackbar' (**Line 5**) – this animation changes the value of the key top in the dictionary pos_hint to topHeight over the duration (key d) of 0.03 seconds.

To invoke the **closing motion** of the snackbar, the dismissSnackbar method is called. To begin with, I simply called the method as follows:

```
self.dismissSnackbar()
```

However, as the dismissSnackbar method begins by adding a delay of length sleepTime seconds (**Line 9**), calling this method in the *main thread* caused the other processes running in the *main thread* to be paused for the duration of the delay. This caused an array of issues; for example, snackbar animation didn't run and any gifs running on the mobile app screen were halted.

To solve this, I called the dismissSnackbar method inside a *background thread*, allowing all the processes in the main thread to continue running. After the delay of length sleepTime seconds, a new instance of the Animation class is instantiated to close the snackbar (**Line 10**). This object is then called on the *Kivy Label* widget with id 'snackbar' (**Line 12**) to initiate the closing motion of the snackbar.

Launch

The Launch class is called when the mobile app is opened, triggering 1 of 3 screens to be called depending on the app's status.

```
1  class Launch(Screen, MDApp):
2      # Coordinate the correct launch screen for the mobile app depending on the
3  current status of the app
4
5      def __init__(self, **kw):
6          super().__init__(**kw)
7          self.statusUpdate()
8          Clock.schedule_once(self.finishInitialising)  # Kivy rules are not
9  applied until the original Widget (Launch) has finished instantiating, so must
10 delay initialisation
11
12     def finishInitialising(self, dt):
13         # applies launch processes which require access to Kivy ids
14         self.manager.transition = NoTransition() # set transition type
15         if self.initialUse == True:  # initial launch of mobile app
16             self.manager.current = "SignUp"
17         elif self.loggedIn == True: # user already logged in
18             createThread_ring(self.accountID, self.filepath) # connect to MQTT
19 broker to receive messages when visitor presses doorbell
20             self.manager.current = "Homepage"  # if the user is already logged
21 in, screen 'Homepage' is called to allow the user to navigate the app
22         else: # not initial launch of app, but user not logged in
23             self.manager.current = "SignIn"
```

Explanation

Initially, the method statusUpdate is called (**Line 7**) to set the value of 4 *json* file keys relating to the user and the status of the mobile app:

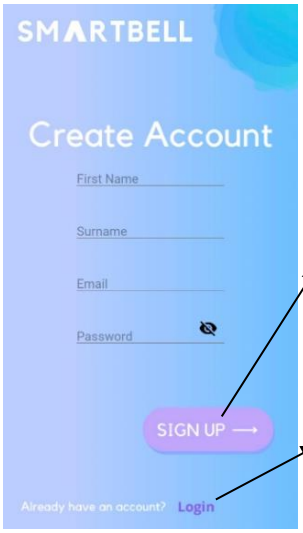
- initalUse
- loggedIn
- paired
- accountID

Depending on the value the method statusUpdate assigns to the attribute initialUse and loggedIn (see below for a further explanation about operation of the statusUpdate method), 1 of 3 possible screens are called. These screens must be initialised from within a separate method finishInitialising (**Line 12**), which is called with a slight delay using the schedule_once method of the *Kivy* Clock class. This delay allows the *Kivy* properties (e.g. the manager property as accessed on **Line 14**) of the Launch class can be successfully initialised before they are called. Failing to include this delay initially resulted in the following error on **Line 14**:

```
AttributeError: 'NoneType' object has no attribute 'transition'
```

The 3 **screens** which can be called on launch are as follows:

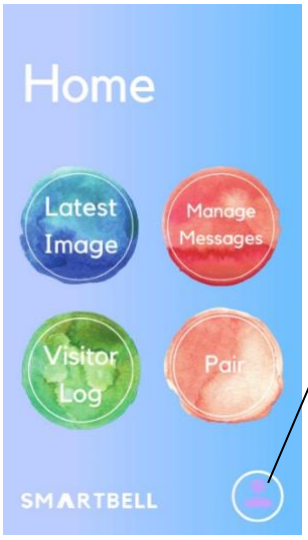
1. Initial launch of mobile app → **SignUp**




```
Button:
    id: signUp_button #unique identifier for this specific button
    size_hint: 0.5, 0.11 #size of button relative to screen
    pos_hint:{"center_x":0.7, "center_y":0.19} #button position relative to screen
    on_press: root.createAccount() #calls Python method 'createAccount'
```

```
Button:
    id: signIn_button #unique identifier for this specific button
    size_hint: 0.2, 0.05 #size of button relative to screen
    pos_hint:{"center_x":0.67,"center_y":0.04} #button position relative to screen
    on_press: root.manager.current = "SignIn"  # switches to 'SignIn' screen
```

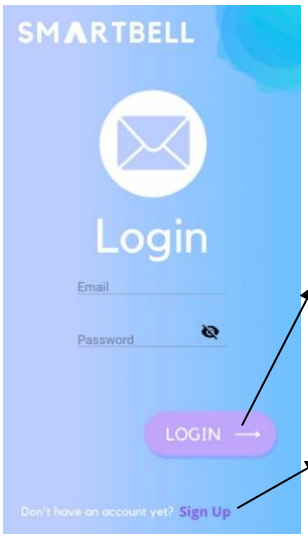
2. User already logged in → **Homepage**




```
Button:
    id: button_account #unique identifier for this specific button
    pos_hint: {"center_x": 0.85, "center_y": 0.07}
    size_hint: 0.2,0.1
    on_press: root.account() #calls Python method 'account'
```

```
def account(self):
    self.statusUpdate()
    self.signOut_dialog() # open dialog which gives user the option to sign out
of their account
```

3. Not initial launch of mobile app and user logged out → **SignIn**




```
Button:
    id: signUp_button #unique identifier for this specific button
    size_hint: 0.2, 0.05 #size of button relative to screen
    pos_hint:{"center_x":0.67, "center_y":0.04} #button position relative to screen
    on_press: root.manager.current = "SignUp"  # switches to 'SignUp' screen
```

```
Button:
    id: signIn_button #unique identifier for this specific button
    size_hint: 0.5, 0.11 #size of button relative to screen
    pos_hint:{"center_x":0.7, "center_y":0.19} #button position relative to screen
    on_press: root.signIn() #calls Python method 'signIn'
```

statusUpdate

As mentioned above, the statusUpdate method is the first procedure called when the mobile app is launched. Its purpose is to store the value of 4 *json* file keys relating to the user and the status of the mobile app permanently in the *local storage*. Doing so ensures that local data relating to the user's use of the mobile app can be carried between sessions (which cannot be achieved if these values are only stored temporarily in the *RAM* assigned to the mobile app during execution). For example, the user will remain logged in after they close the app.

```
1  def statusUpdate(self):
2      self.filepath = MDApp.get_running_app().user_data_dir # path to
3  readable/writeable directory to store local data
4      jsonFilename = join(self.filepath,"jsonStore.json")  # if file name
5  already exists, it is assigned to 'self.filename'. If filename doesn't already
6  exist, file is created locally on the mobile phone
7      self.jsonStore = JsonStore(jsonFilename)  # wraps the json file as a json
8  object
9      if not self.jsonStore.exists("localData"): # if the mobile app is running
10 for the first time, the key 'localData' will not exist
11         self.jsonStore.put("localData", initialUse=True, loggedIn=False,
12 accountID="", paired=False) # sets launch properties
13     self.initialUse = self.jsonStore.get("localData")["initialUse"]
14     self.loggedIn = self.jsonStore.get("localData")["loggedIn"]
15     self.paired = self.jsonStore.get("localData")["paired"]
16     self.accountID = self.jsonStore.get("localData")["accountID"]
```

Explanation

Initially, I simply gave the *json* file the name '*jsonStore.json*', which caused issues when I attempted to write data to it:

```
PermissionError: [Errno 1] Operation not permitted:
    'jsonStore.json'
```

This error arose because the file path '*jsonStore.json*' was in the main *read-only* directory. To overcome this, I assigned the attribute filepath to the *writeable* directory reserved for the mobile app (**Line 1**), and used the join method to create a complete path to the now *writeable* file '*jsonStore.json*' (**Line 4**). If the mobile app is running for the first time, the file with filename jsonFilename will be empty, so the *selection statement* on **Line 9** will be satisfied, and the values for each key in the *json* file will be set to their standard values for the initial launch. On **Lines 13-16**, the value of each key stored permanently in *local storage* is assigned to the corresponding attributes stored temporarily in *RAM*. As each screen in the mobile app inherits from Launch class, they all have access to these attributes, so they do not need to read from the *json* file each time they are required to check the value of 1 of the 4 *json* keys. However, during execution of the mobile app, the value of these *json* keys can be altered by the user (e.g. they may pair sign into a new account). When this occurs, the value of the altered *json* key is written to the *json* file and the method statusUpdate is called to update the value of each attribute (**Lines 13-16**). As it is not possible to directly amend the value of a single key in the *json* file when it is wrapped using the *Kivy* JsonStore class (**Line 7**), the keys which are to remain unchanged are assigned to their corresponding attribute value and the key which is to be changed is assigned its new value:

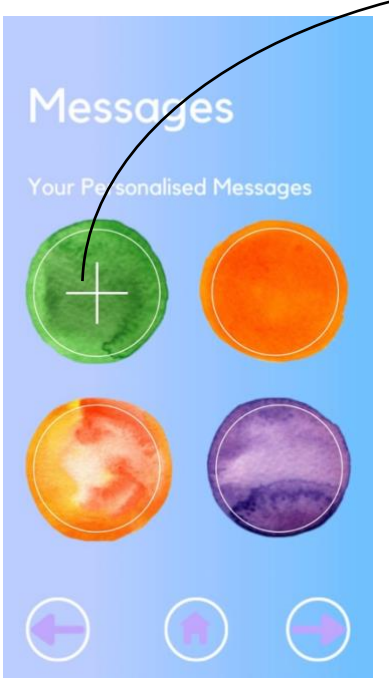
```
self.jsonStore.put("localData", initialUse=self.initialUse,
loggedIn=self.loggedIn, accountID=self.accountID,
              paired=False) # write updated data to the json file
self.statusUpdate()  # update launch variables
```

It is important to call the method statusUpdate after the *json* file is altered to ensure the values of each attribute (**Lines 13-16**) are updated.

Walkthrough

In line with the **accessible** design of the mobile app, when the mobile is run for the first time and the user is signed into a new account, a **walkthrough** processes is triggered which guides the user to create their first audio message. Doing so introduces the users to the **affordances** of the mobile app, enabling them to use the mobile app correctly.

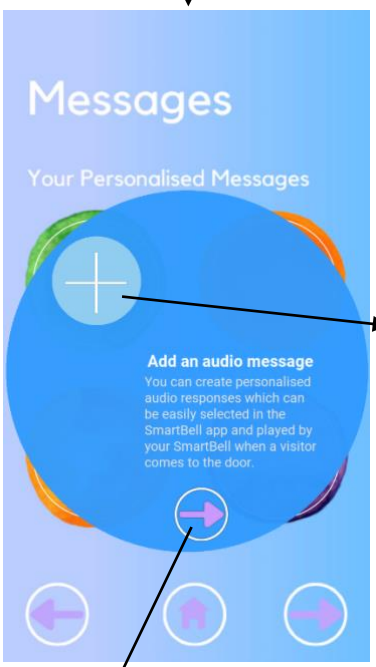
Upon initial launch, the value of initialUse is set to *True,* causing the screen '*MessageResponses_add'* to be called directly after the user signs up:



As part of the walkthrough, the image of a **cross** (*Kivy* id '*plusIcon*') flashes continually to indicate to the user that they are to tap it to create a new audio message. The flashing behaviour is created in the finishInitialising method of the messageResponses_add class using *Kivy*'s Animation class:

```
def finishInitialising(self, dt):
    # applies initialisation processes which require access to Kivy ids
    if self.initialUse == True and self.numMessages == 0: # if the mobile app is
running for the first time and the user has zero audio messages
        animation  = Animation(color=[1, 1, 1, 0], duration=0.1) # become invisible
        animation += Animation(color=[1, 1, 1, 0], duration=1) # invisible
        animation += Animation(color=[1, 1, 1, 1], duration=0.1) # become visible
        animation += Animation(color=[1, 1, 1, 1], duration=1) # visible
        animation.repeat = True # animation loops forever
        animation.start(self.ids.plusIcon) # apply animation to image with id
'plusIcon'
    self.audioMessage_create(1, 3)  # calls method to display user's current audio
messages
```

I decided to require the user to tap the flashing cross themselves (rather than simulate this process programmatically), as a walkthrough should proactively engage the user in the processes of the app.



Takes user to screen to create new message (*'MessageResponse_create'*)

This **target view** is opened when the user taps the cross. The target view provides the user with information about the mobile app and *SmartBell* – specifically, how the user should utilise the pre-recorded audio message feature. The target view object (targetView) is an instance of the *KivyMD* class MDTapTargetView and it is opened in the openTarget method of the messageResponses_add class using the object method start.

```
def addMessage_target(self):
    # instantiates a target view widget which explains how to utilise audio messages
    titleSpaces = '            ' # spaces required to align title text correctly
    descriptionSpaces = '              ' # spaces required to align description text
correctly
    # create target view widget with required properties:
    self.targetView = MDTapTargetView(
        widget=self.ids.button_audioMessage_1,
        title_text="{}Add an audio message".format(titleSpaces),
        description_text="{0}You can create personalised\n{0}audio responses which
can\n{0}be easily selected in the\n{0}SmartBell app and played by\n{0}"
                         "your SmartBell when a visitor\n{0}comes to the
door.".format(descriptionSpaces),
        widget_position="left_top",
        outer_circle_color=(49 / 255, 155 / 255, 254 / 255),
        target_circle_color=(145 / 255, 205 / 255, 241 / 255),
        outer_radius=370,
        title_text_size=33,
        description_text_size=27,
        cancelable=False)
```

Mobile App – Process sensitive and non-sensitive data input

When the user creates a new account or signs into an existing account, they input 2 types of data:

1. **Sensitive data** → *password*
2. **Non-sensitive data** → *all other data items*

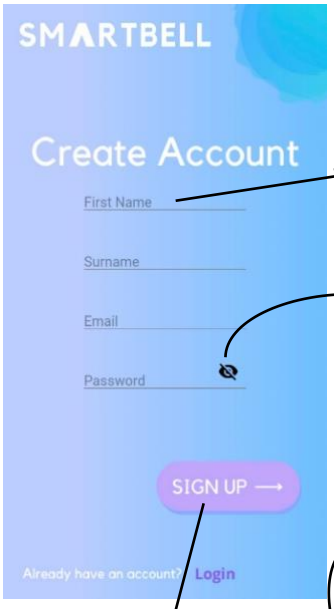
Depending on whether the data is sensitive or not, the approaches taken to process it varies. As outlined in the *Design* section, the SHA3-256 hashing algorithm will be used to create a hash of sensitive data inputted by the user.

Sign Up

When the user creates a new account, they must input the following data items:

1. First name
2. Surname
3. Email
4. Password

The screen (*'SignUp'*) in which this data is inputted is below:



The *MDTextField* widget is used to create the 4 text input boxes – each 1 has the property `hint_text` which contains a string indicating what data the user should input.

```
MDTextField:
    id: firstName #unique identifier for this specific textfield
    hint_text: "First Name" #hint text indicates what data the
user should input into this text field
    pos_hint: {"center_x": 0.5, "center_y": 0.67} #position of
text field relative to screen
```

The **eye icon** is a property of the *KivyMDTextField* widget with id *'password'*, and it enables uses to select whether the character they input in the *Password field* are displayed, or whether they are replaced by bullet points. By default, characters entered into the *Password field* are replaced by bullet points, so the *MDTextField* property `password` defaults to *True*. This is an important feature, as data inputted into the ***Password field*** is **sensitive**, so, where possible, it should be hidden so passers-by cannot see what password the user has inputted.

Python method `createAccount` is called.

If user details are valid, a new account is created.
If user details are invalid, the user is alerted of the issues.

```
MDTextField:
    id: password #unique identifier for this specific textfield
    hint_text: "Password" #hint text indicates what data the user
should input into this text field
    pos_hint: {"center_x": 0.5, "center_y": 0.37} #position of txt
field relative to screen
    password: True #inputted text is replaced by bullet points to
conceal the password as it is sensitive information
    icon_right: "eye-off" #by default, the inputted text is replaced by
bullet points so the eye icon status default to 'eye-off'
```

Password

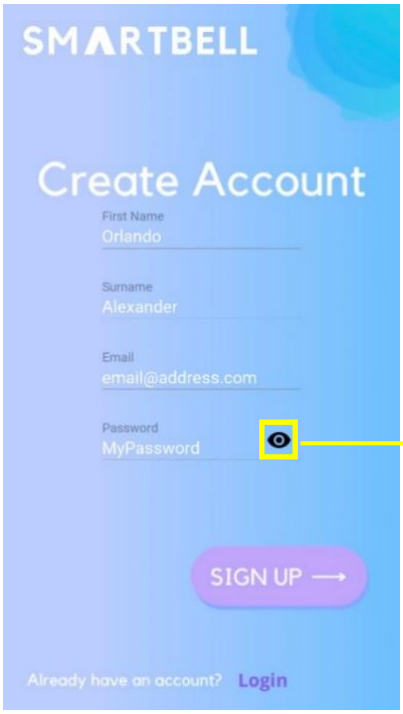
While, by default, the user's input into the P*assword field* is substituted with bullet points, the user may still wish to temporarily view their text input to ensure the characters they have entered are correct. To accommodate this, I used *Kivy* logic:

```
1  Button:
2      id: passwordIcon_button #unique identifier for this specific button
3      size_hint: 0.09, 0.04
4      pos_hint: {"center_x": 0.7, "center_y": 0.375}
5      on_press:
6          password.icon_right = "eye" if password.icon_right == "eye-off" else
7  "eye-off"
8  #if the icon was previously "eye-off", then it is changed to "eye". If the
9  icon was previously "eye", then it is changed to "eye-off"
10         password.password = False if password.password is True else True
11 #the property 'password' of the text field 'password' is inverted when the
12 icon is tapped by the user
```

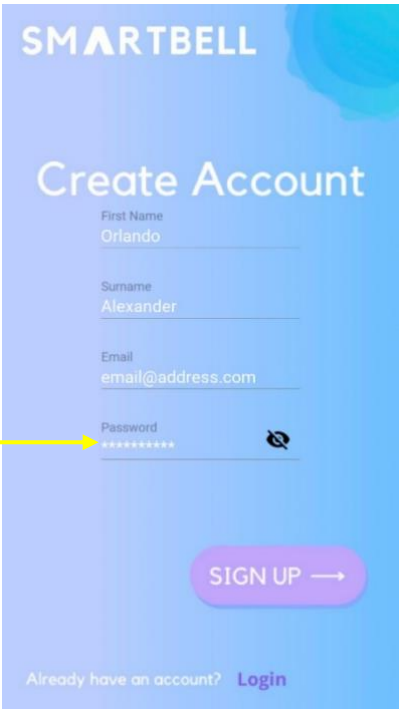
The button `passwordIcon_button` is placed over the eye icon (*KivyMDTextField* property `icon_right`) to invert the status of the password icon and password input text display format when the eye icon is tapped. To achieve this directly within the *.kv* file (removing the need to include a separate method in the *Python* file), *Pythonic*-style *Kivy* logic is used in **Line 6** and **Line 10**. Unlike in *Python*, *Kivy* logic statements are written on 1 line in the following format:

DO ACTION 1 if SELECTION STATEMENT SATISFIED otherwise DO ACTION 2

The output of the above *Kivy* logic is as follows:




```
password: True
icon_right: "eye-off"
```




```
password: False
icon_right: "eye-on"
```

When the *SIGN UP* button on the *'SignUp'* screen is pressed, the method createAccount is called to check the validity of the details inputted by the user. If the details are valid, the procedures are executed to create the user's new account. If the details are invalid, the user is alerted of the issues.

createAccount

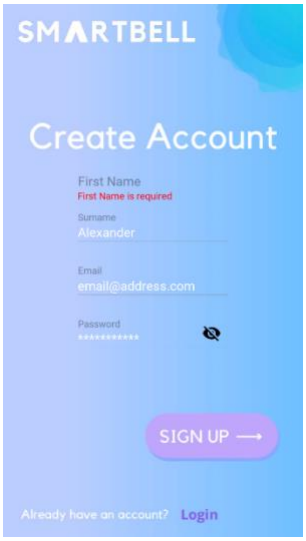
```
def createAccount(self):
    # method called when user taps the Sign Up button to check validity of details entered by user
    self.firstName_valid = False  # variable which indicates that a valid value for 'firstName' has been inputted by the user
    self.surnameValid = False  # variable which indicates that a valid value for 'surname' has been inputted by the user
    self.emailValid = False  # variable which indicates that a valid value for 'email' has been inputted by the user
    self.passwordValid = False  # variable which indicates that a valid value for 'password' has been inputted by the user
    if self.ids.firstName.text == "":  # if no data is inputted...
        self.ids.firstName_error.opacity = 1  # ...then an error message is displayed
    else:
        self.firstName = self.ids.firstName.text  # inputted first name assigned to a variable
        self.ids.firstName_error.opacity = 0  # error message removed
        self.firstName_valid = True  # variable which indicates that a valid value has been inputted by the user
    if self.ids.surname.text == "":  # if no data is inputted...
        self.ids.surname_error.opacity = 1  # ...then an error message is displayed
    else:
        self.surname = self.ids.surname.text  # inputted surname assigned to a variable
        self.ids.surname_error.opacity = 0  # error message removed
        self.surnameValid = True  # variable which indicates that a valid value has been inputted by the user
    if self.ids.email.text == "":  # if no data is inputted...
        self.ids.email_error_blank.opacity = 1  # ...then an error message is displayed
        self.ids.email_error_invalid.opacity = 0  # invalid email error message is removed
    else:
        email = self.ids.email.text
        if re.search(".+@.+\..+", email) != None:  # regular expression used to check that inputted email is in a valid email format
            self.email = self.ids.email.text  # inputted email assigned to a variable
            self.ids.email_error_blank.opacity = 0  # error message removed
            self.ids.email_error_invalid.opacity = 0  # invalid email error message removed
            self.emailValid = True  # variable which indicates that a valid value has been inputted by the user
        else:
            self.ids.email_error_blank.opacity = 0  # blank data error message removed
            self.ids.email_error_invalid.opacity = 1  # invalid email error message displayed
    if self.ids.password.text == "":  # if no data is inputted...
        self.ids.password_error_blank.opacity = 1  # ...then an error message is displayed
```

```
54          self.ids.password_error_invalid.opacity = 0  # invalid password error
55  message is removed
56      else:
57          password = self.ids.password.text
58          if re.search("(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[a-zA-
59  Z\d@$!%*?&]{8,}", password) != None:
60              # Checks that inputted password valid
61              self.password = self.ids.password.text  # inputted password
62  assigned to a variable
63              self.hashedPassword = (hashlib.new("sha3_256",
64  self.password.encode())).hexdigest()
65              self.password = None
66              # Creates a hash of the user's password so that it is stored
67  securely on the database, as it is sensitive data.
68              self.ids.password_error_blank.opacity = 0  # error message removed
69              self.ids.password_error_invalid.opacity = 0  # invalid password
70  error message is removed
71              self.passwordValid = True  # variable which indicates that a valid
72  value has been inputted by the user
73              if self.firstName_valid and self.surnameValid and self.emailValid
74  and self.passwordValid:  # checks if all the data values have been inputted
75  and are valid
76                  self.createAccountID()  # method called to create a unique
77  accountID for the user
78          else:
79              self.ids.password_error_blank.opacity = 0  # blank data error
80  message removed
81              self.ids.password_error_invalid.opacity = 1  # invalid password
82  error message displayed
```

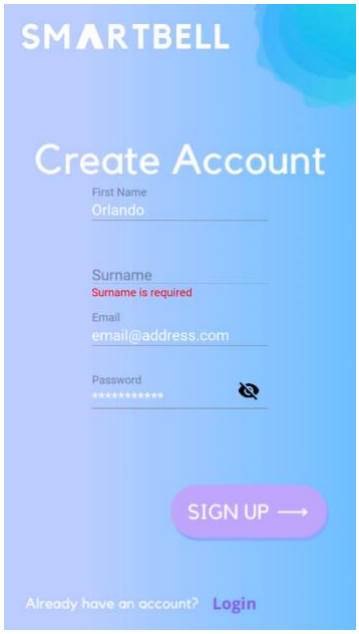
Explanation

Each time the createAccount method is called, the attributes firstName_valid, surnameValid, emailValid and passwordValid are set to *False* (**Lines 4-10**). Following this, the validity of the text inputted for each of the 4 required data items is checked by accessing the text property of each text field (**Line 12, Line 21, Line 30, Line 51**). If the text inputted is valid, the respective validity attribute is set to *True*. Each data item is validated as follows:

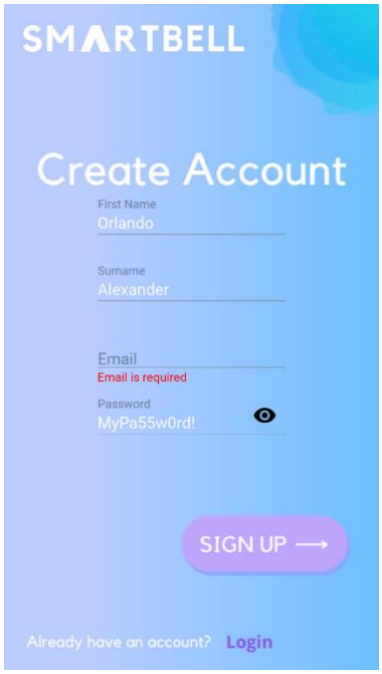
1. firstName
    - Only requirement for the text input is that the input string is not empty (**Line 12**)
    - Error messages if the input string is empty (**Line 13**):



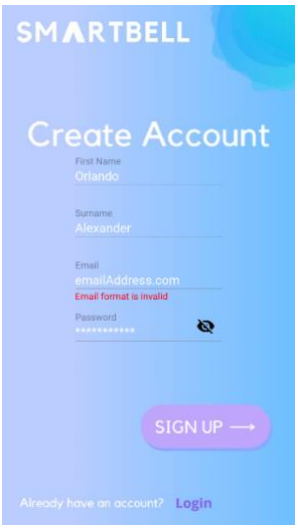
2. surname
    - Only requirement for the text input is that the input string is not empty (**Line 21**)
    - Error message if the input string is empty (**Line 22**):

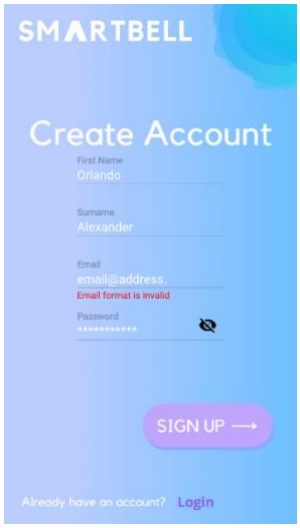


3. email
    - Two requirements for text input:
        1. Input string is not empty (**Line 30**)
        2. Input string is a valid email address format (**Line 37**):
            - **Regular expression** used to verify whether input string is a valid email address format (see below)
    - Error message if the input string is empty (**Line 31**):

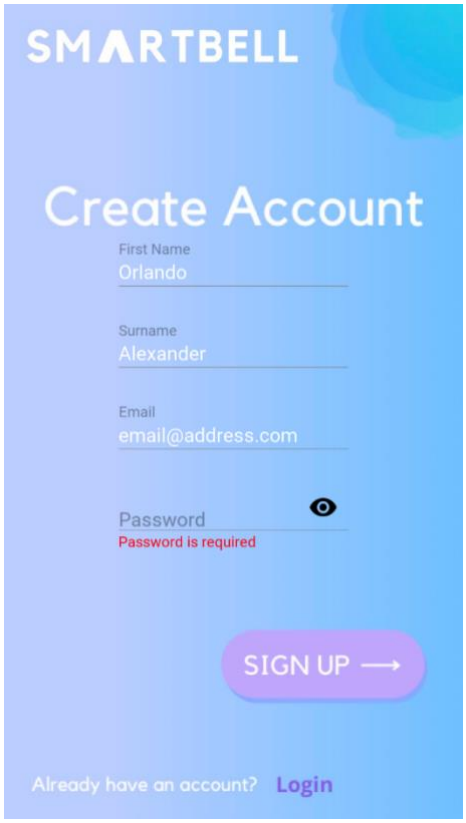


- Error message if the input string is not a valid email address format (**Line 49**):

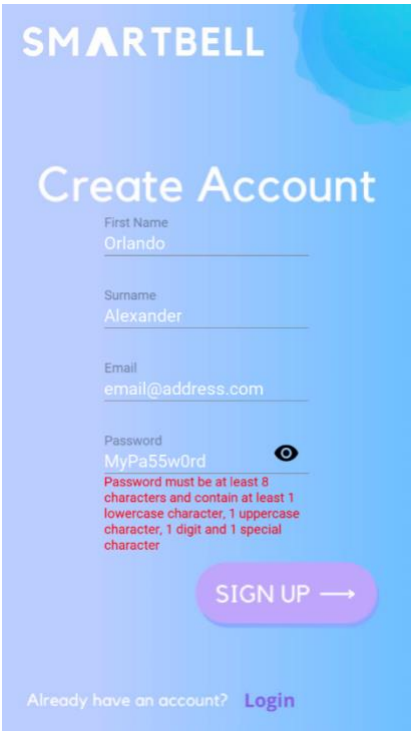


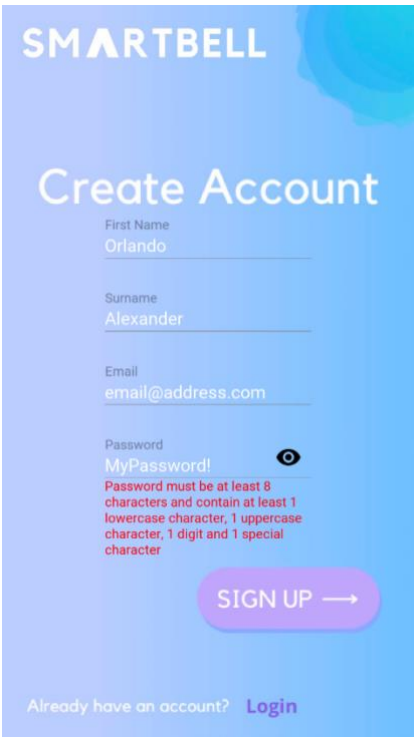


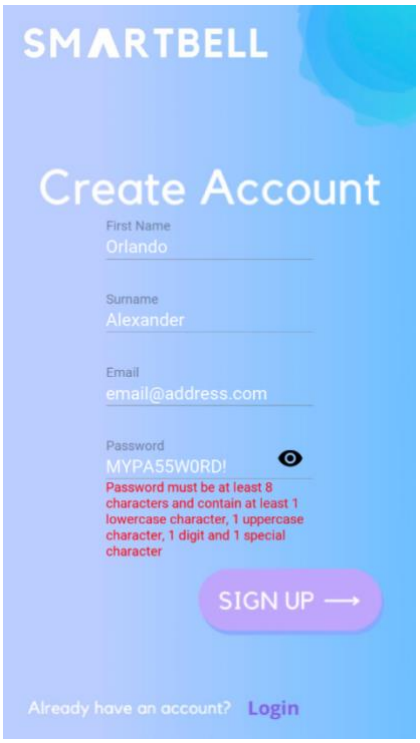
4. password
    - Two requirements for text input:
        1. Input string is not empty (**Line 51**)
        2. Input string is a valid password format (**Line 58**):
            - **Regular expression** used to verify whether input string is a valid password format (see below)
    - If the text input is valid, the password plaintext is **hashed** (see below)
    - Error message if the input string is empty (**Line 52**):



- Error message if the input string is not a valid email address format (**Line 81**):

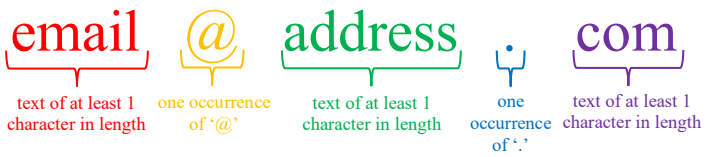






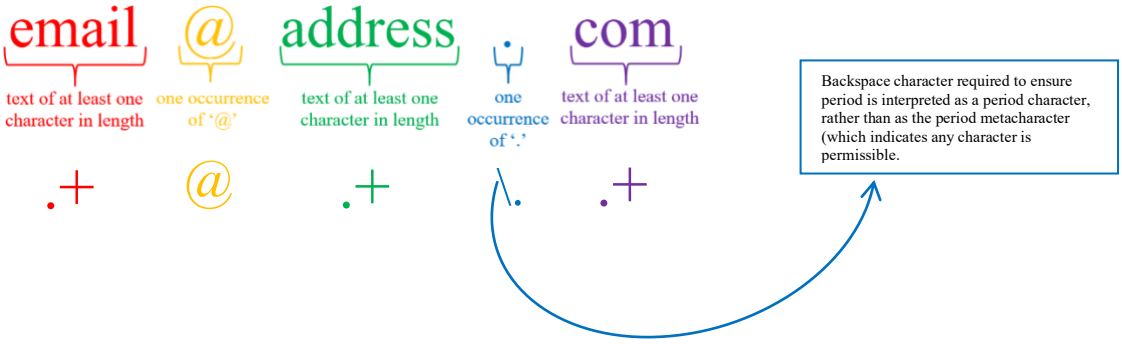
Regular expression – email address

To create the **regular expression** to check whether the email address inputted by the user was valid, I began by using **abstraction** to determine the format of a **valid email address**:



Using the abstraction above and the following *regex metacharacters,* I was able to create the required regular expression:

Regex Metacharacter	Description
.	Any character (except newline character)
+	At least 1 occurrence of the previous character/metacharacter
\	Used to express a metacharacter as a normal character



Therefore, the required **regular expression** to check the validity of the email address input is:

```
.+@.+\..+
```

To apply this regular expression to the email address inputted by the user (email), the search function of the *Python* module re is used (**Line 37**). If the regular expression is not satisfied, the search function returns None, hence the criteria for the selection statement on **Line 37**.

Regular expression – password

To create the **regular expression** to check whether the password inputted by the user was valid, I began by using **abstraction** to determine the 5 conditions for a **valid password**:

1. Must contain at least 1 lowercase letter
2. Must contain at least 1 uppercase letter
3. Must contain at least 1 digit
4. Must contain at least 1 special character
5. Must be at least 8 characters in length

Using the abstraction above and the following *regex metacharacters,* I was able to create the required regular expression:

Regex Metacharacter	Description
?=	**Lookahead** – asserts what characters must follow from the point in the string that the lookahead sits. Not that the lookahead does not 'consume' any characters in the string – after the closing parenthesis of the lookahead statement, the regex machine is left in the same place as it started (www.rexegg.com, n.d.).
*	Zero or more occurrences of the previous characters/metacharacter
[]	A set of characters
\ d	A digit
{*digit*,}	At least *digit* number of occurrences

To check that the password string satisfies each of the conditions 1-4 listed above, I used the **lookahead** regex syntax:

1. **(?=.*[a-z])** → Starting from the beginning of the string, the rest of the string must contain zero or more occurrences of any character, followed by 1 occurrence of a lowercase letter.

2. **(?=.*[A-Z])** → Starting from the beginning of the string, the rest of the string must contain zero or more occurrences of any character, followed by 1 occurrence of an uppercase letter.

3. **(?=.*\d)** → Starting from the beginning of the string, the rest of the string must contain zero or more occurrences of any character, followed by 1 occurrence of a digit.

4. **(?=.*[@$!%*?&])** → Starting from the beginning of the string, the rest of the string must contain zero or more occurrences of any character, followed by 1 occurrence of a special character.

As each **lookahead** statement does not move the regex engine along the string, if expressions 1-4 are chained together, the regex engine will still remain at the start of the string for each expression. As such, the regex expression is able to check whether there is at least 1 occurrence

of a lowercase letter, an uppercase letter, a digit and a special character at any point in the password string.

To check that the password string satisfies condition 5 (as listed above), I used the following regex expression:

5. **[a-zA-Z\d@$!%*?&]{8,}** → At least 8 characters from the set [a-zA-Z/d@$!%*?&], which includes all lowercase characters, all uppercase character, all digits and all special characters. In other words, the string is at least 8 characters in length.

Therefore, the required **regular expression** to check the validity of the password input is:

```
(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[a-zA-Z\d@$!%*?&]{8,}
```

To apply this regular expression to the password inputted by the user (password), the search function of the *Python* module re is used (**Line 58**). If the regular expression is not satisfied, the search function returns None, hence the criteria for the selection statement on **Line 59**.

Password hashing

The personal data associated with a user's account (e.g. their audio response messages or doorbell pairing) can all be accessed through the mobile app using their account ID. When a user is signed into the mobile app, their account ID is stored locally on the mobile phone. However, when a user signs into their account for the first time, their account ID must be retrieved from the MySQL *users* table. This process is executed in the verifyUser method of the SignIn class by passing the user's inputted password to the MySQL database and retrieving the associated account ID (if they have inputted the correct password). Therefore, the user's password is the gateway to full access of the personal data associated with their account.

To ensure the user's account cannot be compromised, the password stored in the MySQL *users* table is hashed using the SHA3-256 hashing algorithm. The hashed form of the *plaintext password* is called the *message digest*, and, since SHA3-256 is a one-way hashing algorithm, the plaintext value of the password cannot be re-obtained from its hashed form in a computational time period. Therefore, the only means of accessing the user's account ID (and therefore personal data) through the mobile app is through knowledge of the plaintext value of the user's password, which is discarded after it is inputted by the user (**Line 65**). This protects against the risks posed by a hacker gaining illegitimate access to the MySQL, as it is impossible to obtain the plaintext value of the user's password even through access to the MySQL *user* table.

The user's plaintext password (password) is encrypted on **Line 63** by applying the hashlib module to the bytes encoded format of the user's plaintext password. This process returns an object, to which the hexdigest method is applied to return the *message digest* of the user's plaintext password in hex format. This hashed password value, along with the non-sensitive data inputted by the user, is stored in the MySQL *users* table by sending a *post* request to the AWS *Elastic Beanstalk environment* in the updateUsers method.

Mobile App – Record and store audio input and output audio files

When designing and testing the mobile app on my PC, I used the pyaudio module to record and play the user's personalised audio responses without issue. However, when I deployed the mobile app to my iPhone, I received the following error:

```
ModuleNotFoundError: No module named 'pyaudio'
```

From my research, I discovered that pyaudio was not compatible with iOS. Following this, I spent many days researching a range of iOS-compatible audio modules for *Python* – it was incredibly difficult to find a module which would enable audio input on my mobile phone.

Eventually, I discovered the audiostream module, which I was able to successfully import the module onto my mobile phone. As the documentation for the module was very limited, I conducted many tests to understand the functionalities of the module and work out the properties required to accurately record audio. For example, I compared the audio bytes of a single *sample* recorded using audiostream with the audio bytes of a single *sample* recorded using pyaudio and tweaked the properties of the audiostream module until the audio bytes matched.

```
1  class RecordAudio():
2      # 'recordAudio' class allows user to record personalised audio messages
3      def __init__(self, **kw):
4          # initialises constant properties for the class
5          super().__init__(**kw)
6          self.bufferRate = 60  # variables which stores the number of audio
7  buffers recorded per second
8          self.audioData = []  # creates list to store the audio bytes recorded
9          self.mic = get_input(rate=8000, buffersize=2048,
10 callback=self.micCallback, source='default')  # initialises the class
11 'get_input' from the module 'audiostream' with the properties required to
12 ensure the audio is recorded correctly
13
14     def micCallback(self, buffer):
15         # method which is called by the class 'get_input' to store recorded
16 audio data (each buffer of audio samples)
17         self.audioData.append(buffer)  # appends each buffer (chunk of audio
18 data) to variable 'self.audioData'
19
20     def start(self):
21         # method which begins the process of recording the audio data
22         self.mic.start()  # starts the method 'self.mic' recording audio data
23         Clock.schedule_interval(self.readChunk, 1 / self.bufferRate)  # calls
24 the method 'self.readChunk' to read and store each audio buffer (2048 samples)
25 60 times per second
26
27     def readChunk(self, bufferRate):
28         # method which coordinates the storing of the bytes from each buffer
29         self.mic.poll()  # calls 'get_input(callback=self.mic_callback,
30 source='mic', buffersize=2048)' to read the byte content. This byte content is
31 then dispatched to the callback method 'self.micCallback'
32
33     def stop(self):
34         # method which terminates and saves the audio recording
35         Clock.unschedule(self.readChunk)  # un-schedules the calling of
36 'self.readChunk'
37         self.mic.stop()  # stops recording audio
38         return self.audioData
```

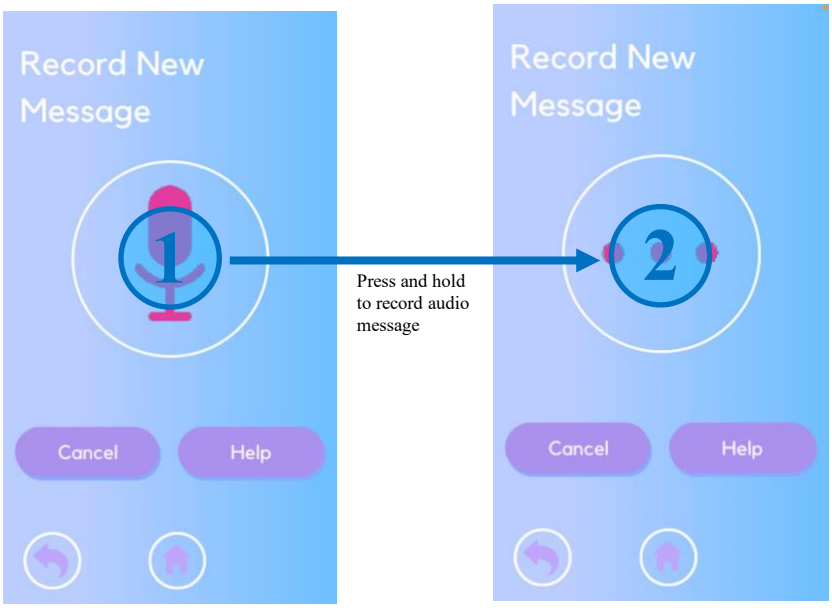
Explanation

When the recordAudio class is initialised (**Line 4**), an object of the get_input class called mic is created (**Line 11**). To successfully record the audio, the following *keyword arguments* are passed to get_input:

- rate
  - Set to *8000*
  - This means that the audio will be sampled 8000 times per second by get_input
- buffersize
  - Set to *2048*
  - This means that the audio samples are read and stored in chunks of 2048 samples
- *Callback function* for the method micCallback
  - Called by the poll method of get_input to store each audio buffer
- source
  - Set to *'default'*
  - Microphone source on mobile phone

Record audio

To initiate the recording, the start method of the recordAudio class (**Line 22**) is called as a *thread* when the user presses and holds the button (*Kivy widget* with ID button_recordAudio) which overlays the microphone icon (*Kivy widget* with ID recordAudio – labelled **1**) in the mobile app:



The start method must be called within a new *thread* (recordAudio_thread) so that the gif (labelled **2**) can continue looping in the main thread. The gif labelled **2** is an important **usability** feature, as it indicates to the user that they are currently recording audio. This ensures the user speaks at the correct time to successfully create their personalised audio message.

Requiring the user to continue holding while they record their audio message allows the user to customise the length of their audio message.

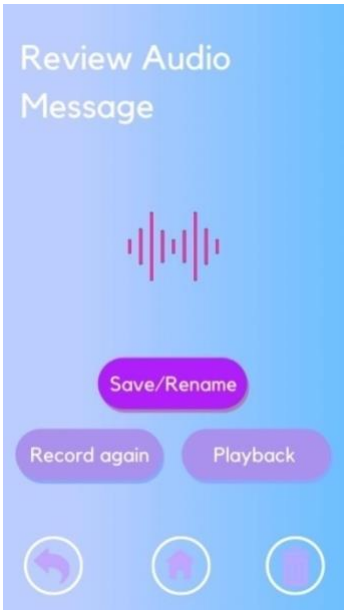
Within the custom start method, the start method of the get_input object mic is called (**Line 24**) to begin the audio sampling process. From my testing, the ideal audio recording was achieved when the latest audio buffer was read and stored 60 times per second. Therefore, in the start method of the object mic, the schedule_interval method of the Clock class is used to call the method readChunk 60 times per second. **(Line 25)**.

On **Line 33** in readChunk, the poll method of the object mic is called, which triggers the *callback function,* passing the bytes in the latest buffer (buffer) as an argument to micCallback. The method poll is named as such because this process is effectively 'polling' the object mic for the latest audio data. In the callback function micCallback, the bytes in buffer are appended to the array audioData (**Line 19**), which is an attribute of the class recordAudio.

When the user releases the button button_recordAudio, the stop method (**Line 33)** of the recordAudio class is called. To terminate the reading and storing of audio buffers, the regular calling of the method readChunk is un-scheduled (**Line 35**). To terminate the audio recording, stop method of the object mic is called (**Line 37**). Finally, the audio bytes recorded and stored in the array audioData is returned, *pickled* and stored locally on the mobile app:

```
with open(messagePath,"wb") as file:  # create .pkl file with name equal to
the messageID in write bytes mode
    pickle.dump(audioData, file)  # 'pickle' module serializes the data stored
in Python list object 'audioData' into a byte stream stored in .pkl file
    file.close()  # closes the file
```

Playback audio recording



The *audio playback* screen is shown when the user releases the button button_recordAudio, terminating the recording of their audio message:

```
on_release:
    recordAudio.source = root.recordAudio_end #changes
the image source for the ID 'recordAudio'
    root.stopRecording() #calls the method
'stopRecording' from the parent class in the Python
application
```

To listen back to an audio message they have recorded, the user must press the '*Playback'* button (*Kivy widget* with ID button_playbackAudio). Depending on the status, the selected audio message will be stored in 1 of 3 locations/formats:

1. *.pkl* file stored remotely in the S3 storage bucket *nea-audio-messages*
    - Occurs if user has previously recorded and saved the audio message, but never played it back through the mobile app
2. *.pkl* file stored locally on the mobile app
    - Occurs if user has just recorded the audio message
3. *.wav* file store locally on the mobile app:
    - Occurs if user has previously recorded, saved and played back the audio message through the mobile app

State 1

If the audio message is in **state 1**, the audio bytes must first be downloaded from the AWS S3 storage *nea-audio-messages*. All interactions with S3 storage are processed by the REST API server, so a *POST* request is sent to the route *'/downloadS3*, along with the required data downloadData, which includes the messageID of the audio message to be downloaded:

```
downloadData = {"bucketName": "nea-audio-messages",
                "s3File": self.messageID}  # creates the dictionary which
stores the metadata required to download the .pkl file of the personalised
audio message from AWS S3
response = requests.post(serverBaseURL + "/downloadS3", downloadData)
audioData = pickle.loads(response.content)  # unpickles the byte string
```

Note that the pickle module must be used to de-serialize the byte stream stored in the downloaded byte string (response.content) into a *Python* list object.

State 2

If the audio message is in **state 2**, the audio bytes must first be loaded from local storage on the mobile app:

```
self.messagePath = join(self.filepath, "audioMessage_tmp.pkl") # filepath to
store .pkl file of audio bytes for audio message which is not yet saved by
user
with open(self.messagePath, "rb") as file:
    audioData = pickle.load(file)
    file.close()  # closes the file
```

Note that the path for all audio messages in **state 2** (messagePath) is the same as this is a temporary file store before the message is saved by the user, given a unique messageID and uploaded to AWS S3.

State 1 and 2

Now that the *Python* list object audioData storing the audio bytes for the desired audio message is loaded, these audio bytes must be written to a *.wav* file so that the audio message can be outputted by the mobile phone:

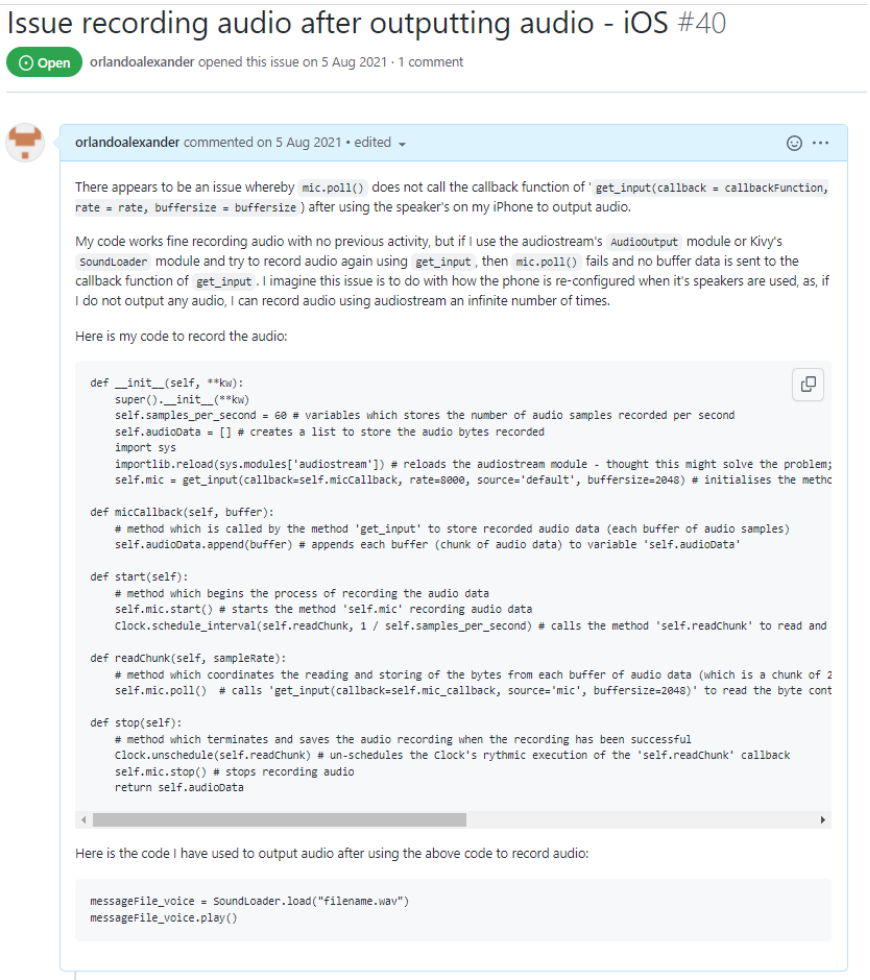
```
messageFile = wave.open(fileName + ".wav", "wb") # load .wav file in write
bytes mode
messageFile.setnchannels(1)  # audiostream module records in single audio
channel
messageFile.setsampwidth(2) # bytes per audio sample
messageFile.setframerate(8000) # samples recorded per second
messageFile.writeframes(b''.join(audioData)) # join together each element in
the bytes list 'audioData' (each element is an audio buffer)
messageFile.close()
```

State 1, 2 and 3

To playback the audio message, the *Kivy* module SoundLoader is used:

```
messageFile_voice = SoundLoader.load(fileName + ".wav")
messageFile_voice.play()
```

During *Development*, I encountered an issue with the audio playback; the audio input failed when it was used after outputting an audio message. After conducting in-depth troubleshooting, I discovered this to be a bug with the audiostream module, whereby the microphone configuration was adversely affected by audio output. To find a solution, I posted an *Issue Log* on the GitHub repository:



Issue recording audio after outputting audio - iOS #40

Open orlandoalexander opened this issue on 5 Aug 2021 · 1 comment

orlandoalexander commented on 5 Aug 2021 • edited

There appears to be an issue whereby `mic.poll()` does not call the callback function of `get_input(callback = callbackFunction, rate = rate, buffersize = buffersize)` after using the speaker's on my iPhone to output audio.

My code works fine recording audio with no previous activity, but if I use the audiostream's `AudioOutput` module or Kivy's `SoundLoader` module and try to record audio again using `get_input`, then `mic.poll()` fails and no buffer data is sent to the callback function of `get_input`. I imagine this issue is to do with how the phone is re-configured when it's speakers are used, as, if I do not output any audio, I can record audio using audiostream an infinite number of times.

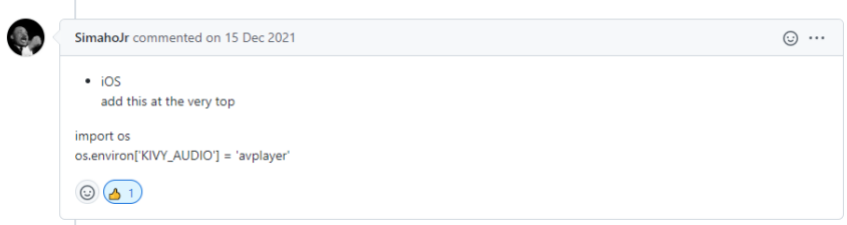
Here is my code to record the audio:

```
def __init__(self, **kw):
    super().__init__(**kw)
    self.samples_per_second = 60 # variables which stores the number of audio samples recorded per second
    self.audioData = [] # creates a list to store the audio bytes recorded
    import sys
    importlib.reload(sys.modules['audiostream']) # reloads the audiostream module - thought this might solve the problem;
    self.mic = get_input(callback=self.micCallback, rate=8000, source='default', buffersize=2048) # initialises the metho

def micCallback(self, buffer):
    # method which is called by the method 'get_input' to store recorded audio data (each buffer of audio samples)
    self.audioData.append(buffer) # appends each buffer (chunk of audio data) to variable 'self.audioData'

def start(self):
    # method which begins the process of recording the audio data
    self.mic.start() # starts the method 'self.mic' recording audio data
    Clock.schedule_interval(self.readChunk, 1 / self.samples_per_second) # calls the method 'self.readChunk' to read and

def readChunk(self, sampleRate):
    # method which coordinates the reading and storing of the bytes from each buffer of audio data (which is a chunk of 1
    self.mic.poll()  # calls 'get_input(callback=self.mic_callback, source='mic', buffersize=2048)' to read the byte cont

def stop(self):
    # method which terminates and saves the audio recording when the recording has been successful
    Clock.unschedule(self.readChunk) # un-schedules the Clock's rythmic execution of the 'self.readChunk' callback
    self.mic.stop() # stops recording audio
    return self.audioData
```

Here is the code I have used to output audio after using the above code to record audio:

```
messageFile_voice = SoundLoader.load("filename.wav")
messageFile_voice.play()
```

I received the following workaround solution:



SimahoJr commented on 15 Dec 2021

- iOS
  add this at the very top

import os
os.environ['KIVY_AUDIO'] = 'avplayer'

1

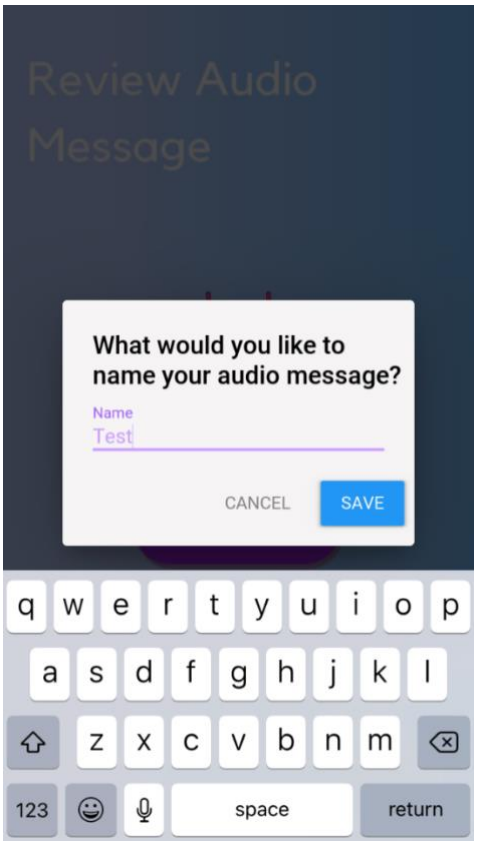
I added the following in the first line of the *Python* file for the mobile app:

```
import os
os.environ['KIVY_AUDIO'] = 'avplayer'  # control the kivy environment to
ensure audio input can be accepted following audio output
```

While this resolved the audio input issue, the audio output became somewhat inconsistent, as I will address in the *Evaluation* section.

Save audio recording

To save their personalised audio response which they have recorded, the user must press the '*Save/Rename'* button (*Kivy widget* with ID button_saveAudio):



To save the message, it must have a unique *message ID* which identifies it:

```
chars = string.ascii_uppercase +
string.ascii_lowercase + string.digits

self.messageID = ''.join(random.choice(chars) for i
in range(16))  # the 'random' module randomly
selects 16 characters from the string 'chars' to
form the unique messageID
```

Using the string module, a variable chars is created which stores a concatenated string of the uppercase and lowercase alphabetic characters and all the digits (0-9).
Using the random module, a random selection of 16 characters from the chars string are joined together to create a unique messageID for the user's audio message.

To store the details for this new audio message in the MySQL database, a *POST* request is made to the *'/update_audioMessages'* route of the REST API server, along with a *json* object storing the details about the audio message:

```
dbData_update = {'messageID': self.messageID, 'messageName': self.messageName,
'messageText': self.messageText, 'accountID': self.accountID,
'initialCreation': str(self.initialRecording)}  # json object which stores the
metadata required for the AWS server to update the MySQL database
requests.post(serverBaseURL + "/update_audioMessages",dbData_update)  # sends
post request to 'update_audioMessages' route on AWS server to insert the data
about the audio message
```

This will create a *record* in the *audioMessages* table of the MySQL database as follows:

messageID	messageName	messageText	accountID
GvJajwNaR9VPUEzY	Test	Null	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...

Note that the field *messageText* has the value '*Null*' because the audio response created by the user is in an audio format, rather than a text format (see below).

Finally, the audio message must be uploaded to the AS S3 storage bucket *nea-audio-messages* for 2 reasons:

- So that the Raspberry Pi doorbell can downloaded the audio messages and output it through its speaker
- So that the user can access the audio message through the mobile app on any device where they are logged into their account

Again, this process is done using the REST API sever as the intermediary body: the mobile app sends a *POST* request to the REST API route *'/uploadS3'*, along with a *json* object storing the required metadata about the audio message *and* the file storing the pickled audio bytes:

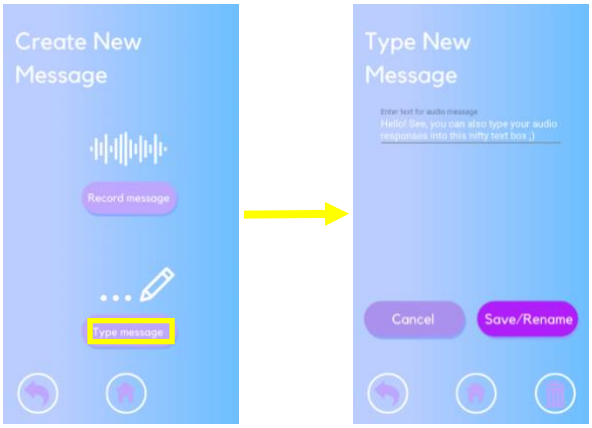
```
uploadData = {"bucketName": "nea-audio-messages",
                    "s3File": self.messageID}  # json object storing required
metadata to upload the audio message
file = {"file": open(self.messagePath, "rb")}  # opens the file to be sent
requests.post(serverBaseURL + "/uploadS3", files=file, data=uploadData)
# sends post request to 'uploadS3' route on AWS server to upload the pkl file
storing the data about the audio message to AWS S3
```

This will create a file in the AWS S3 storage bucket *nea-audio-messages* as follows:

Name	Type	Last modified	Size	Storage class
GvJajwNaR9VPUEzY	-	February 17, 2022, 16:43:24 (UTC+00:00)	51.3 KB	Standard

Create text-based audio responses:

To ensure the mobile app is **accessible** for users with speech disorders, personalised audio responses can also be stored in a text format, with the doorbell using text-to-speech software to output the messages in an audio format.

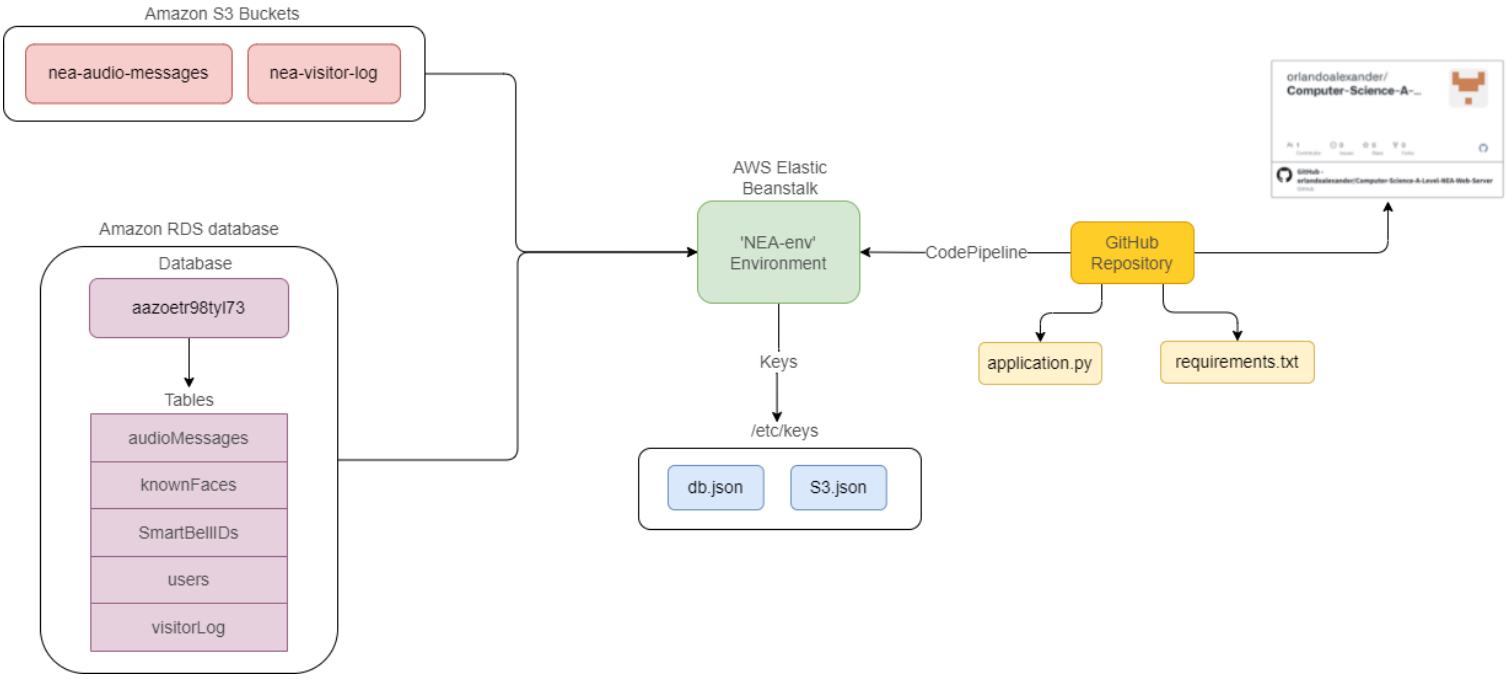


The process of saving audio responses created in a text formatted is much the same as the process for messages in an audio format, with the following exceptions:

- The field *messageText* in the *audioMessages* table will store the text string for the message response, rather than *'Null'*
- No files will be uploaded to AWS S3 storage buckets

REST API Server

REST API Server – File Structure

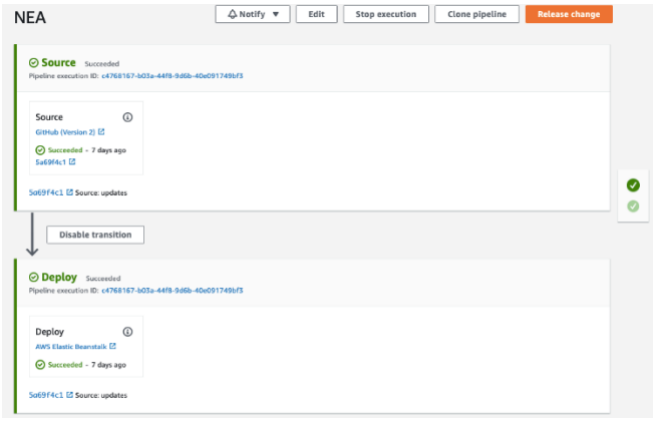


- The GitHub Repository https://github.com/orlandoalexander/Computer-Science-A-Level-NEA-Web-Server.git is used to store the *Python* file application.py which is hosted by the **AWS Elastic Beanstalk (EB) environment** 'NEA-env'. In addition to the application.py file, the GitHub repository also stores a text file (requirements.txt), which specifies which packages must be imported by the AWS *EB environment* to run the file application.py. By using **AWS CodePipeline** to create a live *pipeline* between the *public* GitHub repository and my *EB environment*, I was able to ensure that any updates that I *pushed* to the files in my GitHub repository would be automatically pushed to the *EB environment*. This setup was ideal for **iterative development**, as any *commits* that I *pushed* to the remote GitHub repository from my local IDE would be *deployed* automatically by the *pipeline*, enabling me to test and debug them quickly and easily. Where errors did arise, I would check the **logs** for my *EB environment* to resolve them:

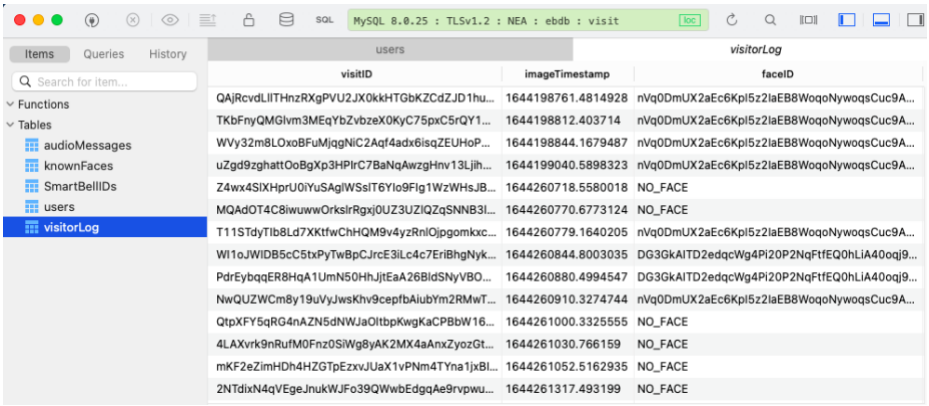
```
Feb  7 00:38:54 ip-172-31-18-149 web: [2022-02-07 00:38:54,346] ERROR in app: Exception on /verifyPairing [POST]
Feb  7 00:38:54 ip-172-31-18-149 web: Traceback (most recent call last):
Feb  7 00:38:54 ip-172-31-18-149 web: File "/var/app/venv/staging-LQM1lest/lib/python3.8/site-packages/flask/app.py", line 2447, in wsgi_app
Feb  7 00:38:54 ip-172-31-18-149 web: response = self.full_dispatch_request()
Feb  7 00:38:54 ip-172-31-18-149 web: File "/var/app/venv/staging-LQM1lest/lib/python3.8/site-packages/flask/app.py", line 1952, in full_dispatch_request
Feb  7 00:38:54 ip-172-31-18-149 web: rv = self.handle_user_exception(e)
Feb  7 00:38:54 ip-172-31-18-149 web: File "/var/app/venv/staging-LQM1lest/lib/python3.8/site-packages/flask_restful/__init__.py", line 271, in error_router
Feb  7 00:38:54 ip-172-31-18-149 web: return original_handler(e)
Feb  7 00:38:54 ip-172-31-18-149 web: File "/var/app/venv/staging-LQM1lest/lib/python3.8/site-packages/flask/app.py", line 1821, in handle_user_exception
Feb  7 00:38:54 ip-172-31-18-149 web: reraise(exc_type, exc_value, tb)
Feb  7 00:38:54 ip-172-31-18-149 web: File "/var/app/venv/staging-LQM1lest/lib/python3.8/site-packages/flask/_compat.py", line 39, in reraise
Feb  7 00:38:54 ip-172-31-18-149 web: raise value
Feb  7 00:38:54 ip-172-31-18-149 web: File "/var/app/venv/staging-LQM1lest/lib/python3.8/site-packages/flask/app.py", line 1950, in full_dispatch_request
Feb  7 00:38:54 ip-172-31-18-149 web: rv = self.dispatch_request()
Feb  7 00:38:54 ip-172-31-18-149 web: File "/var/app/venv/staging-LQM1lest/lib/python3.8/site-packages/flask/app.py", line 1936, in dispatch_request
Feb  7 00:38:54 ip-172-31-18-149 web: return self.view_functions[rule.endpoint](**req.view_args)
Feb  7 00:38:54 ip-172-31-18-149 web: File "/var/app/current/application.py", line 386, in verifyPairing
Feb  7 00:38:54 ip-172-31-18-149 web: return {'result':result[0]}
Feb  7 00:38:54 ip-172-31-18-149 web: TypeError: 'NoneType' object is not subscriptable
```

- 'NEA-env' is an **Elastic Beanstalk environment** which I created in the AWS console. The purpose of an *EB environment* is to collate several AWS resources in a single area. For my purposes, the *EB environment* 'NEA-env' is associated with the following **resources**:

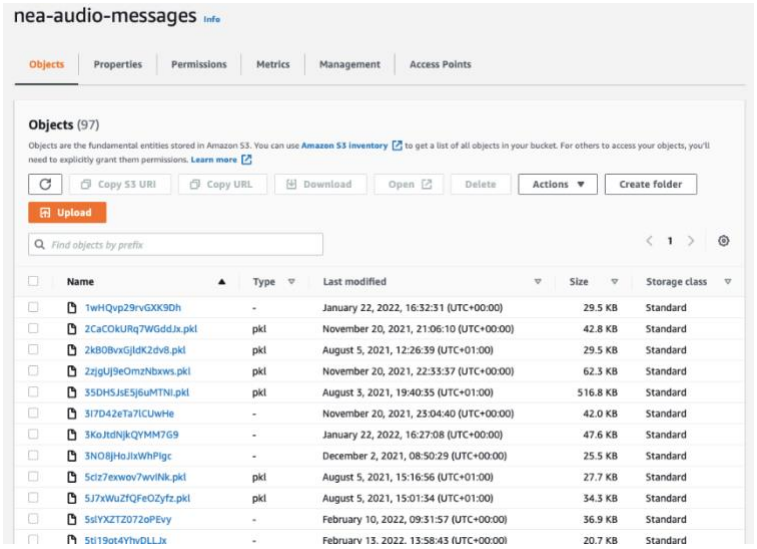
    1. **CodePipeline** – live pipeline to the GitHub repository storing the application program to be hosted by the *EB environment*:



    2. **RDS database** – MySQL database containing the 5 tables used by the system can be accessed directly by the application program hosted by the *EB environment* – accessed through database viewer **TablePlus**:

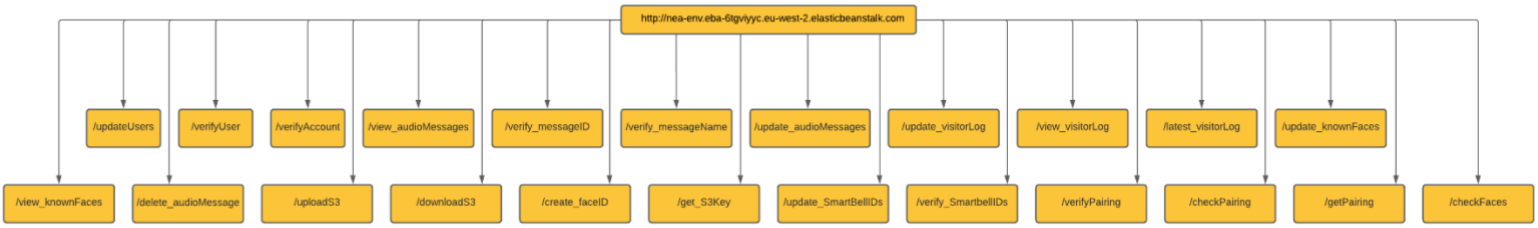


3. **S3 bucket** – NoSQL database containing 2 directories/buckets which store audio message files and visitor images:



- To access the associated RDS database and S3 buckets from the application program hosted by the *EB environment*, a **password** and **pair of keys** were required respectively. As explained below, I created a directory called 'keys' inside the 'etc' directory of the *EB environment* in which I stored 2 *json* files with the credentials for the RDS database and S3 buckets. As these *json* files were part of the *EB environment*, they could be accessed directly from the application program when it was running in the *EB environment*.

REST API Server – System Hierarchy Diagram



Each branch in the above diagram is a separate **path** in the *EB environment*, and can be accessed with the following URL structure:

http://nea-env.eba-6tgviyyc.eu-west-2.elasticbeanstalk.com/ path name

Elastic Beanstalk environment base URL

As each path of the application hosted by the *EB environment* is restricted to *POST* requests, the application behaves as a **REST API**: each path receives and returns data when it is called, rather than displaying a HTML-based webpage.

The primary purpose of the REST API is to perform processing which cannot be done directly by the mobile app. Therefore, the 2 main tasks of the REST API are:

1. Querying data in the **MySQL database**
2. Downloading/uploading data from/to the **S3 buckets**

Therefore, all the changes made to the **System Hierarchy structure** of the REST API during *Development* fell under 1 of 2 categories:

1. An additional interaction with the MySQL database or S3 buckets which I discovered to be necessary.
    - For example, the path verify_messageName was added to prevent the user from creating multiple message responses with the same name, by checking whether the message name inputted by the user already existed in the *audioMessages* table. If the message name already existed, the *Kivy* Animation class was used to shake the *'Save'* button, indicating to the user that the inputted message name was invalid.
2. It made sense programmatically to split 1 process into multiple processes, requiring additional paths.
    - For example, in the *Design* section the only path associated with adding a new user account to the *users* table was updateUsers. However, during the *Development* phase, I realised that it was necessary to check whether an account already existed with the email address inputted by the user, before the user's details were added to the *users* table. This additional step is executed biny the verifyUser path.

REST API Server – Modules

```
from flask import Flask, request, jsonify, send_file
from mysql import connector
import boto3
import pickle
import json
import random
import string
from cryptography.fernet import Fernet
```

The purpose of and general application of each imported Class/module/library is as follows:

- ***flask*** *(Module)*
  - Enables the *Python* application hosted by the *EB environment* to be turned into a web application by providing the required tools (pythonbasics.org, n.d.)
    - ***Flask*** *(Class imported from flask module)*
      - Wraps the *Python* file as the central object for the web application, so separate paths can be created for different functions of the web API
    - ***request*** *(Class imported from flask module)*
      - Enables the *Python* application to access data associated with a request made to a path on the web application
    - ***jsonify*** *(Class imported from flask module)*
      - Format data in a *json* format so it can be returned appropriately by the REST API – generally used to format the results of queries made to the RDS database
    - ***send_file*** *(Class imported form flask module)*
      - Return a file downloaded from S3 by the REST API to the client
- ***connector*** *(Class imported from mysql module)*
  - Enables *Python* application to establish a connection with the RDS database, so it can make queries to the database
- ***boto3*** *(Module)*
  - Used to interact with (upload/download) files stored in AWS S3 buckets
- ***json*** *(Module)*
  - Load the *json* files storing the RDS database and S3 bucket keys
- ***random*** *(Module)*
  - Generate random characters to create unique IDs (e.g. account ID or visit ID)
- ***string*** *(Module)*
  - Provides functionalities related to strings, such as creating a string of all the letters in the alphabet
- ***Fernet*** *(Class imported from cryptography.fernet module)*
  - Encrypt AWS S3 storage keys using symmetric encryption before they are returned to client by REST API

As the *EB environment* hosting the *Python* application must install any packages which are not preinstalled, the GitHub repository also contains a **requirements.txt** file which lists the packages to be imported, alongside the required version:

```
Flask==1.1.2
boto3==1.18.6
cryptography==35.0.0
```

REST API Server – Setup

As explained previously, the REST API is essentially a webpage whose paths can only be accessed via a *POST* request (when a view-only HTML-based webpage is loaded, a *GET* request is sent to the server). Therefore, the standard procedure to wrap a *Python* application as a webpage is used:

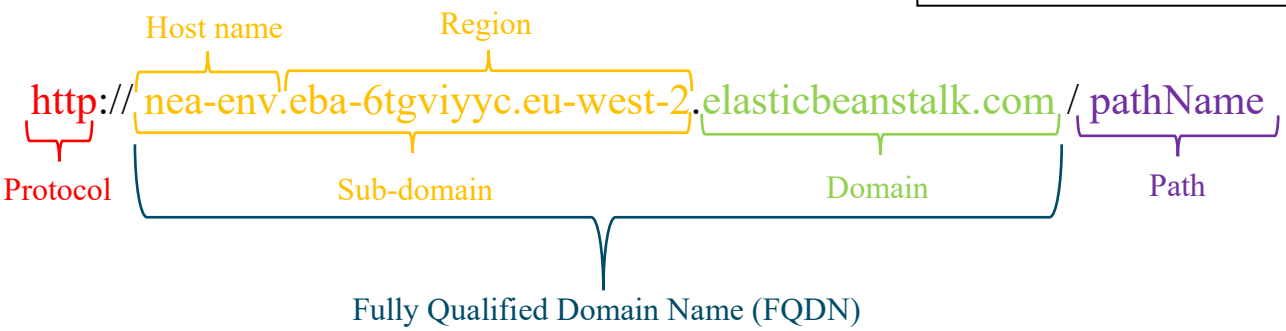
```
application = Flask(__name__) # wraps file using the Flask constructor and
stores it as the central object called 'application'
```

The constructor method of the `Flask` class takes the argument `__name__` (which is a special *Python* variable and corresponds to the name of the current module) and creates an object `application` which is used to create individual **paths/routes** on the webpage:

```
@application.route("/pathName", methods = ["POST"])
def myFunction():
    # do something
    return "Json string"
```

This creates a new path called '*pathName*'. As the *Python* file in which the `Flask` object `application` exists is hosted by an *EB environment* with the **endpoint/URL** '*http://nea-env.eba-6tgviyyc.eu-west-2.elasticbeanstalk.com*', the complete **URL** of the path '*pathName*' is as follows:

An endpoint refers to a URL which a *POST* request is made to



- **HTTP** protocol used as all requests use the *POST* request method, which is supported by HTTP
- **Sub-domain** is a domain within the larger *'elasticbeanstalk.com'* domain and it specifies the actual server on which the *EB environment* is running. The sub-domain has 2 parts:
  - **Host name** is the name of the environment itself, which I was able to choose
  - **Region** specifies where in the world the server hosting my *EB environment* is located
- **Domain** is the overarching name of the collection of servers hosting *EB environments* for AWS users globally
- **Path** is the name of the specific path within the *EB environment* to be accessed

Initially, I used the following approach to add a new path called *'pathName'* to the webpage:

```
def myFunction():
    return "Json string"

app.add_url_rule("/", "myPath", myFunction)
```

While this was a viable solution, it was a verbose method of creating a new path.

As demonstrated above, I decided to use the application.route **decorator** to simplify the process of adding a path. A decorator (indicated by the @ prefix) is a function which takes another function as an argument – in my case, the decorator application.route takes myFunction as an input, registering myFunction to the path *'myPath'*. As a result, when the path *'myPath'* is requested, the myFunction is called and its result is returned back to the client (Kim, 2020).

To understand the syntax of using the @ prefix, see the following 2 code snippets. Each achieves the same outcome, but *snippet 1* uses a decorator with @ prefix and *snippet 2* does not:

1. **With @ prefix**

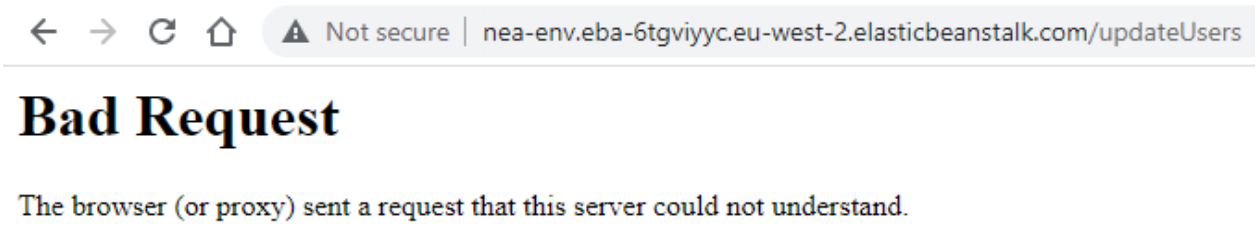
```
@application.route("/pathName", methods = ["POST"])
def myFunction():
    # do something
    return "Json string"
```

The function myFunction is an argument to the decorator application.route, which binds myFunction to the path *'pathName'*

2. **Without @ prefix**

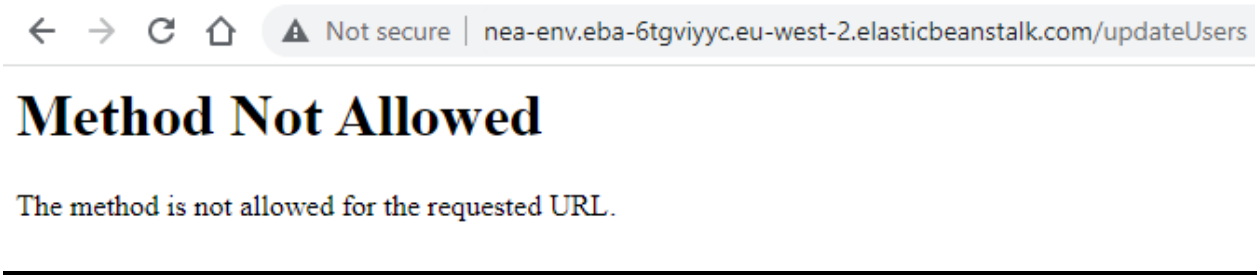
```
def myFunction():
    # do something
    return "Json string"

application.route("/pathName", methods = ["POST"])(myFunction)
```

Further to the *positional argument* which defines the path name (in the example above, this is "/pathName"), there is a second *keywords argument* passed to the application.route decorator: methods. methods is an array which defines which *request types* which are permitted to access the respective path. At first, I did not include the methods keyword argument, allowing all request types to communicate with the path. However, this led to the following error when the path was accessed through my browser:



This occurred because the function associated with the path *'updateUsers'* requires access to *json* data sent with the request. However, when a webpage path is retrieved directly through the browser as above, no data is sent with the request. In other words, the request type shown above is a **GET** request, whereas the function associated with the path requires a **POST** request type. In a **POST** request, data must be sent to the server in the request body. Therefore, to avoid an error message being shown when any of the REST API paths are accessed through a browser, I defined the methods keyword argument as follows: methods = ["POST"].

Doing so prevented the REST API path from being accessed through a browser – an example of **defensive programming**. Therefore, if a user did mistakenly attempt to access a REST API path through a browser, the following warning message would be shown in place of a generic error message:



REST API Server – Keys

As explained previously, the 2 main tasks of the REST API are:

1. Querying data in the **MySQL database**
2. Downloading/uploading data from/to the **S3 buckets**

As **keys** are required to interact with the MySQL RDS database and the AWS S3 storage server, my *Python* application code needed to utilise these keys during runtime. Initially, I naively stored these keys directly in the server code as constants on my public GitHub repository *Computer-Science-A-Level-NEA-Web-Server* which is pipelined to the AWS *EV environment*. However, AWS contacted me to inform me of this security breach:

Amazon Web Services has opened case 8653244881 on your behalf.

The details of the case are as follows:

Case ID: 8653244881
Subject: Action Required: Your AWS account 188213566607 is compromised
Severity: Urgent
Correspondence: Dear AWS customer,

We have become aware that the AWS Access Key AKIASXUTHDSHWWJCOXW6, belonging to User "", along with the corresponding Secret Key is publicly available online at https://github.com/orlandoalexander/Computer-Science-A-Level-NEA-Web-Server/blob/708299226c5d5285e936054edd236d8ac26abb07/application.py.

This poses a security risk to your account (including other account users), could lead to excessive charges from unauthorized activity, and violates the AWS Customer Agreement or other agreement with us governing your use of our service. To protect your account from excessive charges, we have temporarily limited your ability to use some AWS services. To remove the limits, please follow the instructions below.

To protect your account from excessive charges, we may terminate any suspected unauthorized resources on your account.

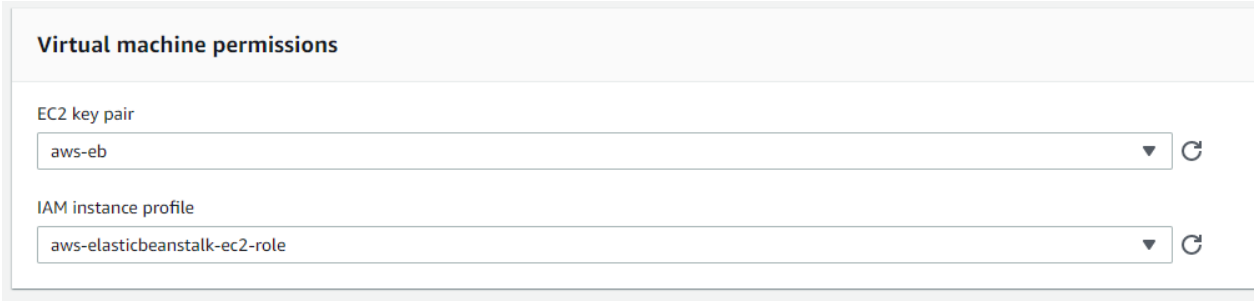
To resolve this, I needed to **store the keys directly in the *EB environment***, so that they could be accessed through the *Python* application stored on my GitHub repository without being stored directly in the *Python* application. To achieve this, I connected to the *EB environment* over SSH:

Connect to *EB environment* over SSH

Initially, I installed the EB CLI (Command Line Interface) by cloning the git repository *aws-elastic-beanstalk-cli-setup* and following the installation instructions:

```
git clone https://github.com/aws/aws-elastic-beanstalk-cli-setup.git
```

To configure the EB CLI, I used the **AWS console** first created an **EC2 key pair** with **public key** *'aws-eb'*. I also downloaded the **private key** as it must be inputted when using the EB CLI to establish an initial connection with my *EB environment* over SSH. To allow me to use the EC" key pair '*aws-eb*' to connect with my *EB environment*, I also configured the environment to use the same key pair:



I then ran the command `eb init` to establish a connection with the *EB environment* from the EB CLI, and followed the setup instructions:

1. **Select default region:**

```
Select a default region
1) us-east-1 : US East (N. Virginia)
2) us-west-1 : US West (N. California)
3) us-west-2 : US West (Oregon)
4) eu-west-1 : EU (Ireland)
5) eu-central-1 : EU (Frankfurt)
6) ap-south-1 : Asia Pacific (Mumbai)
7) ap-southeast-1 : Asia Pacific (Singapore)
8) ap-southeast-2 : Asia Pacific (Sydney)
9) ap-northeast-1 : Asia Pacific (Tokyo)
10) ap-northeast-2 : Asia Pacific (Seoul)
11) sa-east-1 : South America (Sao Paulo)
12) cn-north-1 : China (Beijing)
13) cn-northwest-1 : China (Ningxia)
14) us-east-2 : US East (Ohio)
15) ca-central-1 : Canada (Central)
16) eu-west-2 : EU (London)
17) eu-west-3 : EU (Paris)
18) eu-north-1 : EU (Stockholm)
19) eu-south-1 : EU (Milano)
20) ap-east-1 : Asia Pacific (Hong Kong)
21) me-south-1 : Middle East (Bahrain)
22) af-south-1 : Africa (Cape Town)
(default is 3): 16
```

2. **Select *EB* application to establish connection with:**

```
Select an application to use
1) NEA
2) EPQ-MQTT-Server
3) NEA-Server
4) NEA Web Server
5) Recycleapp1
6) RecycleApp
7) Recycle_App
8) Recycle App
9) Recycle
10) [ Create new Application ]
(default is 10): 1
```

3. **Select application platform:**

```
Select a platform.
1) .NET Core on Linux
2) .NET on Windows Server
3) Docker
4) GlassFish
5) Go
6) Java
7) Node.js
8) PHP
9) Packer
10) Python
11) Ruby
12) Tomcat
(make a selection): 10
```

4. **Select EC2 key pair to be used:**

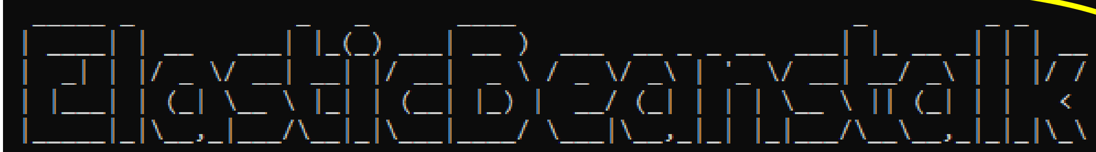
```
Select a keypair.
1) aws-eb
2) [ Create new KeyPair ]
(default is 1): 1
```

I

To verify my authenticity, I had to input the **private key** associated with the **public key** *'aws-eb'*. Once this **private key** was verified and saved locally, I was given the option to create a custom **passphrase** which could use in the future to access this private key.

To connect to my *EB environment* over SSH, I then ran the command eb ssh:

```
(.ebcli-virtual-env) C:\Users\aorla>eb ssh
INFO: Running ssh -i C:\Users\aorla\.ssh\aws-eb ec2-user@3.11.169.124
Enter passphrase for key 'C:\Users\aorla\.ssh\aws-eb':
```

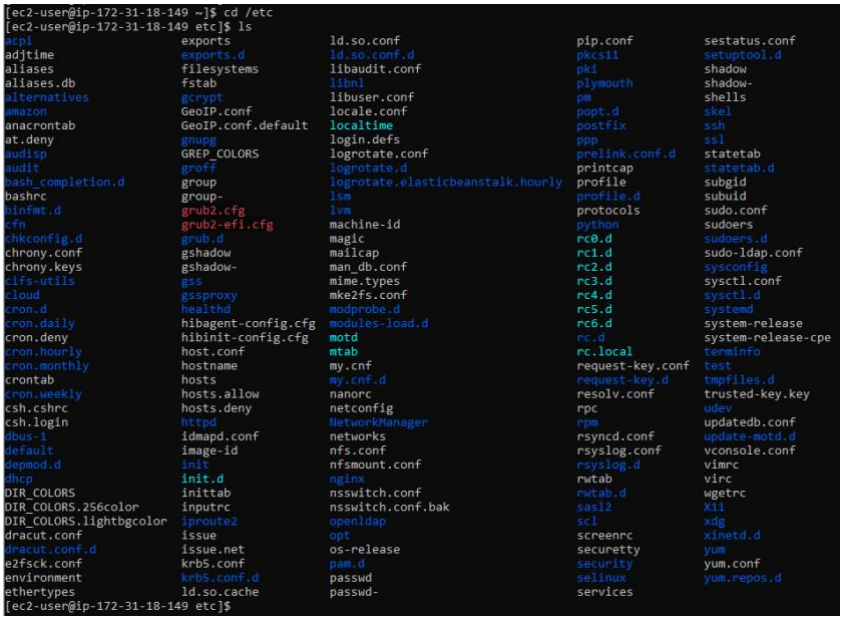


```
 Amazon Linux 2 AMI

 This EC2 instance is managed by AWS Elastic Beanstalk. Changes made via SSH
 WILL BE LOST if the instance is replaced by auto-scaling. For more information
 on customizing your Elastic Beanstalk environment, see our documentation here:
 http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/customize-containers-ec2.html

[ec2-user@ip-172-31-18-149 ~]$
```

Passphrase for '*aws-eb*' EC2 key pair inputted

As the *Elastic Beanstalk* operating system is **Linux** based, I used Linux command line syntax to store the **keys** in the *EB environment*.
The steps I used were as follows:

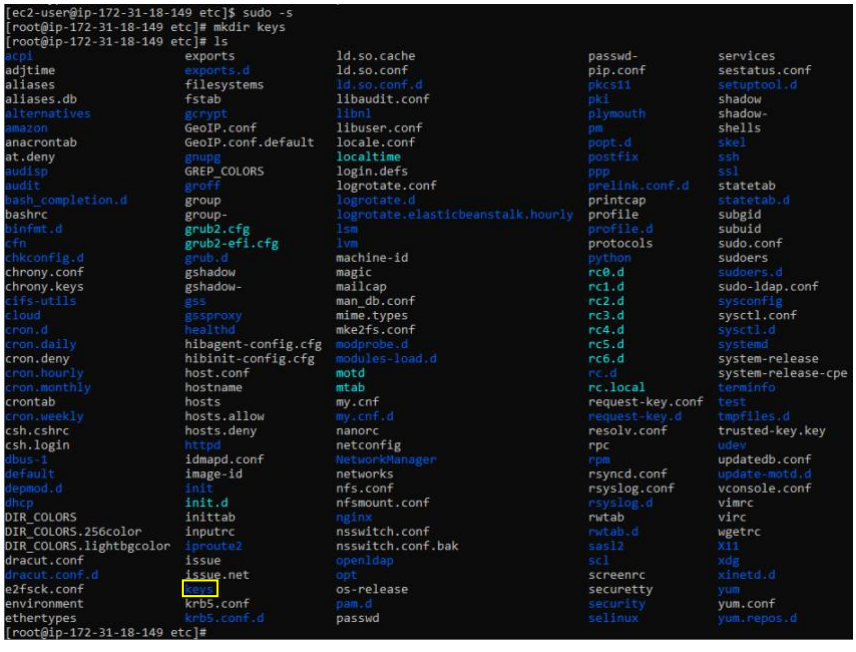
1. Navigate to the *'/etc'* directory
    - I used `cd /etc` to achieve this:



2. Create a directory *'keys'* inside the *'/etc'* directory to store the **key** files
    - I tried using `mkdir keys` to achieve this. However, this command through the following error:

```
[ec2-user@ip-172-31-18-149 etc]$ mkdir keys
mkdir: cannot create directory ‘keys’: Permission denied
[ec2-user@ip-172-31-18-149 etc]$
```

    - This error occurred because root privileges are required to create new directories. To resolve this, I used `sudo -s` to enable root user privileges. I then repeated the `mkdir keys` command successfully:



3. Create *json 2* files to store MySQL RDS database and AWS S3 keys respectively called: *'db.json'* and *'S3.json'*
    - I used `echo '{"key": "value"}' > db.json` and `echo '{"key": "value"}' > S3.json` to create the 2 *json* files with the required keys.
    - In the *json* file *db.json*, I stored the values for the *password*, *username* and *endpoint URL* respectively to access the MySQL RDS database:

```
[root@ip-172-31-18-149 keys]# echo '{"passwd": "[redacted]", "user": "orlandoalexander", "host": "aazoetr98tyl73.cnem9ngq
o5zs.eu-west-2.rds.amazonaws.com"}' > db.json
```

    - In the *json* file *S3.json*, I stored the pair of credentials required to upload and download files from buckets on AWS S3 storage:

```
[root@ip-172-31-18-149 keys]# echo '{"accessKey": "[redacted]", "secretKey": "[redacted]
[redacted]"}' > S3.json
```

REST API Server – Paths

The 23 individual paths on the REST API server each have specific purposes and perform specific operations. However, as explained previously, the operations of many of these paths are based on similar principles. Therefore, while the code for all the paths can be found in the *Appendix,* I am going to explain the design of the following 3 REST API paths:

1. /update_audioMessages
    - Insert the details for a new audio message created by the user through the mobile app into the MySQL table *audioMessages* OR update the details in the MySQL table *audioMessages* for an existing audio message
    - If the audio message has been recorded by the user through the mobile app (rather than typed), the audio bytes will be stored in the AWS S3 storage bucket *nea-audio-messages* by the REST API path */uploadS3*
2. /downloadS3
    - Download the image of a visitor captured by the *SmartBell* from the AWS S3 storage bucket *nea-visitor-log* and return this image to the client (i.e. the mobile app) OR download the audio bytes of an audio message recorded through the mobile app from the AWS S3 storage bucket *nea-audio-messages* and return these audio bytes to the client (i.e. the mobile app or the *SmartBell*)
3. /get_S3Key
    - Securely transmit the pair of keys required to interact with the AWS S3 storage buckets to the client (i.e. the *SmartBell*), allowing the client to upload the latest visitor image captured to the AWS S3 storage bucket *nea-visitor-log*.

/update_audioMessages

```
@application.route("/update_audioMessages", methods = ["POST"])
# route to add data about a new audio message to the 'audioMessages' table
def update_audioMessages():
    try:
        with open("/etc/keys/db.json", "r") as file: # load file storing
credentials to access RDS database
            keys = json.load(file) # convert the json file into a json object
        data = request.form # assigns data sent to API to variable ('data')
        mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),database="ebdb")  # initialise connection to database
        myCursor = mydb.cursor()  # initialises a cursor which allows
communication with 'mydb' (MySQL database session)
        if data['initialCreation'] == "False": # if audio message already
exists in database
            query = "UPDATE audioMessages SET messageName = '%s', messageText
= '%s' WHERE messageID = '%s'" % (data['messageName'], data['messageText'],
data['messageID']) # update message name and message text (if appropriate)
        else: # if audio message is to be added to database for the first time
            query = "INSERT INTO audioMessages (messageID, messageName,
messageText, accountID) VALUES ('%s', '%s', '%s', '%s')" % (data['messageID'],
data['messageName'], data['messageText'], data['accountID']) # data for new
audio message is inserted into the table 'audioMessages'
        myCursor.execute(query) # the query is executed in the MySQL database
        mydb.commit() # commits the changes to the MySQL database made by the
executed query
        return "success" # 'success' returned if changes are made successfully
    except:
        return "error" # 'error' returned if there is an error in the process
```

Explanation

The code for the function update_audioMessages (which is executed when the path *'update_audioMessages'* is requested) is contained within a *Try and Except* block, as shown on **Line 4** and **Line 27**. Where an error arising in the operation of the function, the string '*error'* is returned (**Line 28**). This is an example of **defensive programming**, as the client program which is sending a request to the REST API path will only ever receive 1 of 2 responses: *'success'* or *'error'*. Therefore, it is easy to program the client to react appropriately when the server throws an error – the client will never receive an unexcepted result. Moreover, by returning the string *'success'* upon successful completion of the function's operation, the client program is able to verify that requested process has been carried out successfully.

The credentials required to connect with the AWS-hosted RDS MySQL database are stored in the file *'db.json'* within *'/etc/keys'* directory of the *EB environment*. Therefore, the built-in *Python* function open is used to load this file (**Line 5**), and the load method of the json class is used to convert this *json* file into a *json* object which can be accessed much like a dictionary (**Line 7**). The *json* object contains the following 3 keys:

1. host
2. user
3. passwd

The values of these 3 keys must be passed as *keyword arguments* to the connector class method connect (**Line 9**) to create a session with the server for the database *'ebdb'*.

To communicate with tables in the MySQL database, *SQL queries* must be made. As the REST API is designed to dynamically perform database processing for multiple different user accounts with different values of input data, the specific SQL queries will vary each time a REST API path is called. However, as each REST API path is designed to perform a certain type of process, the structure of each query will remain constant. Therefore, I have used the *Python string operator* %s to dynamically alter the SQL query depending on the data sent with the *POST* request by the client. This data, which is sent in a *dictionary* format, is accessed on **Line 8** by accessing the form attribute of the request class. By replacing the appropriate *substrings* of the SQL query with '%s' (e.g. **Line 15**) and following the query string with % and a tuple containing the variable substring values to be substituted into the query, I was able to create dynamic SQL queries in response to the specific REST API path request.

The path *'update_audioMessages'* can execute 1 of 2 possible SQL queries, depending on the instruction sent by the client in the request data (i.e. the value of the initialCreation key):

1. Update the name and text (if appropriate) of an audio message already stored in the *audioMessages* table (**Line 15**)
    - Key names in the dictionary data for each variable substring value in SQL query:
        - messageName
        - messageText
        - messageID
    - The SQL *UPDATE* statement performs the following operation:

- Navigate to table *audioMessages*
- Consider record with field *messageID* = messageID
- Update the value of the fields as follows:
    1. *messageName* → messageName
    2. *messageText* → messageText

2. Insert all the details for a new audio message into the *audioMessages* table (**Line 19**)
    - Key names in the dictionary data for each variable substring value in SQL query:
        - messageID
        - messageName
        - messageText
        - accountID
    - The SQL *INSERT* statement performs the following operation:
        - Navigate to table *audioMessages*
        - Create a new record with field values as follows:
            1. *messageID* → messageID
            2. *messageName* → messageName
            3. *messageText* → messageText
            4. *accountID* → accountID

Once the required SQL query has been created, it can be executed using the myCursor object (**Line 23**) and the changes committed to the database (**Line 24**).

/downloadS3

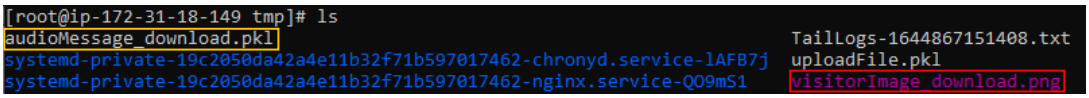
```
1  @application.route("/downloadS3", methods=["POST"])
2  # route to download pickled byte data of the user's personalised audio
3  messages or image file of visitor from AWS S3 storage
4  def downloadS3():
5      try:
6          with open("/etc/keys/S3.json", "r") as file:  # load file storing pair
7  of keys required to establish connection with S3 storage server
8              keys = json.load(file)  # convert the json file into a json object
9          data = request.form  # assigns the metadata sent to the API to a
10 variable ('data')
11         if data["bucketName"] == "nea-audio-messages":
12             fileName = "/tmp/audioMessage_download.pkl"  # file location in eb
13 environment
14         elif data["bucketName"] == "nea-visitor-log":
15             fileName = "/tmp/visitorImage_download.png"  # file location in eb
16 environment
17         s3 = boto3.client("s3", aws_access_key_id=keys["accessKey"],
18 aws_secret_access_key=keys["secretKey"])
19         # initialises a connection to the S3 client on AWS using the
20 'accessKey' and 'secretKey'
21         s3.download_file(Filename=fileName, Bucket=data["bucketName"],
22 Key=data["s3File"])
23         # downloads file name equal 's3file' from the S3 bucket with name
24 'bucketName' and stores this file in the '/tmp' directory of the eb
25 environment
26         return send_file(fileName, as_attachment=True) # returns the
27 downloaded file to the client
28     except:
29         return "error" # 'error' returned if there is an error in the proces
```

Explanation

To download a file from AWS S3 storage and return it to the client, 3 steps are required:

1. Initialise a connection with the AWS S3 client to access S3 storage
    - Connection established on **Line 17** using the boto3 module
    - The pair of **keys** required to establish this connection are stored locally in the *json* file *'S3.json'* in the *EB environment* directory *'/etc/keys'*
2. Download the required file from the appropriate S3 bucket and temporarily store it in the *EB environment*
    - The download_file method of the client class is used to download the required file from the AWS S3 storage server (**Line 21**)
    - The name of the file to be downloaded (assigned to the *keyword argument* Key in the download_file method) and the *S3 bucket* (assigned to the *keyword argument* Bucket in the download_file method) in which this file is stored are specified in the dictionary data sent with the *POST* request by the client:
        - The name of the file to downloaded is assigned to the key s3File in the dictionary data – this name corresponds to the ID value of the file to be downloaded as stored in the MySQL database (i.e. message ID or visit ID)
        - The name of the bucket from which the file is to be downloaded from is assigned to the key bucketName in the dictionary data

- The path at which the downloaded file is temporarily stored on the *EB environment* is specified by filename (**Line 12** and **Line 15** – value depends on whether file to be downloaded is an audio message or visitor image). The image below shows the contents of the *'/tmp'* folder on the *EB environment*, where the latest audio message file (*'audioMessage_download.pkl'*) and visitor image file (*'visitorImage_download.png'*) to be downloaded are stored:



3. Return the downloaded file to the client which made the request to the REST API path
   - The send_file class is used to return the downloaded file stored at filename to the client (**Line 26**). The client can access the file's byte content in the response returned by the REST API path.

/get_S3Key

```
1  @application.route("/get_S3Key", methods=["POST"])
2  # route to retrieve the pair of keys required to interact with AWS S3 storage
3  server
4  try:
5      with open("/etc/keys/db.json", "r") as file:  # load file storing
6  credentials to access RDS database
7          keys = json.load(file)  # convert the json file into a json object
8      data = request.form  # assigns the data sent to the API to a variable
9  ('data')
10     mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
11 passwd=(keys["passwd"]), database="ebdb")  # initialise connection to database
12     myCursor = mydb.cursor()  # initialises a cursor which allows
13 communication with 'mydb' (MySQL database session)
14     with open("/etc/keys/S3.json", "r") as file:  # load file storing pair of
15 keys required to establish connection with S3 storage server
16         keys_S3 = json.load(file)  # convert the json file into a json object
17     query = "SELECT * FROM users WHERE accountID = '%s'" % (data["accountID"])
18     myCursor.execute(query)
19     result = myCursor.fetchone()# retrieve first matching record from database
20     if result != 0: # verifies if an account exists with the specified
21 accountID
22         key = data["accountID"].encode()  # key must be encoded as bytes
23         fernet = Fernet(key)  # instantiates Fernet encryption object using
24 user's accountID as the encryption key
25         accessKey_encrypted = fernet.encrypt(keys_S3["accessKey"].encode())
26 # use Fernet class instance to encrypt the string - string must be encoded to
27 byte string before it is encrypted
28         secretKey_encrypted= fernet.encrypt(keys_S3["secretKey"].encode())
29 # use Fernet class instance to encrypt the string - string must be encoded to
30 byte string before it is encrypted
31         encryptedKeys_dict = {'accessKey_encrypted':
32 accessKey_encrypted.decode(),
33                               'secretKey_encrypted':
34 secretKey_encrypted.decode()}  # keys must be decoded to be jsonified and
35 returned by API
36         return encryptedKeys_dict
37     else:
38         return "error"
39 except:
40     return "error"
```

Explanation

By uploading the latest visitor image captured directly from the *SmartBell* Raspberry Pi device to the AWS S3 storage bucket *nea-visitor-log* (rather than through the *'upload_S3'* REST API path), the latency between the visitor ringing the doorbell and the user receiving the image of the visitor through the mobile app is reduced. However, to upload an image file to the AWS S3 storage bucket, the Raspberry Pi requires the pair of keys, as explained previously. As these keys can also be used to download personal data from either of the S3 storage buckets, it is important that these keys are kept secure. Therefore, it is not viable to store these keys locally on the Raspberry Pi.

To solve this, the keys are retrieved from the secure and remote *EB environment* each time they are required by the Raspberry Pi using the REST API path *'get_S3key'*, as shown on **Line 14**. To ensure the transmission of the keys to the client (i.e. the Raspberry Pi) is secure, and to avoid a *man-in-the-middle* attack, the keys are *symmetrically encrypted* using the Fernet class. As shown on **Line 24**, the byte-encoded format of the user's accountID is used as the *encryption key*, allowing the Raspberry Pi requiring use of the keys to easily decrypt them. As explain in the *Design* section, the unique accountID for each user is 43 characters in length, as this is the required length for the Fernet encryption key.

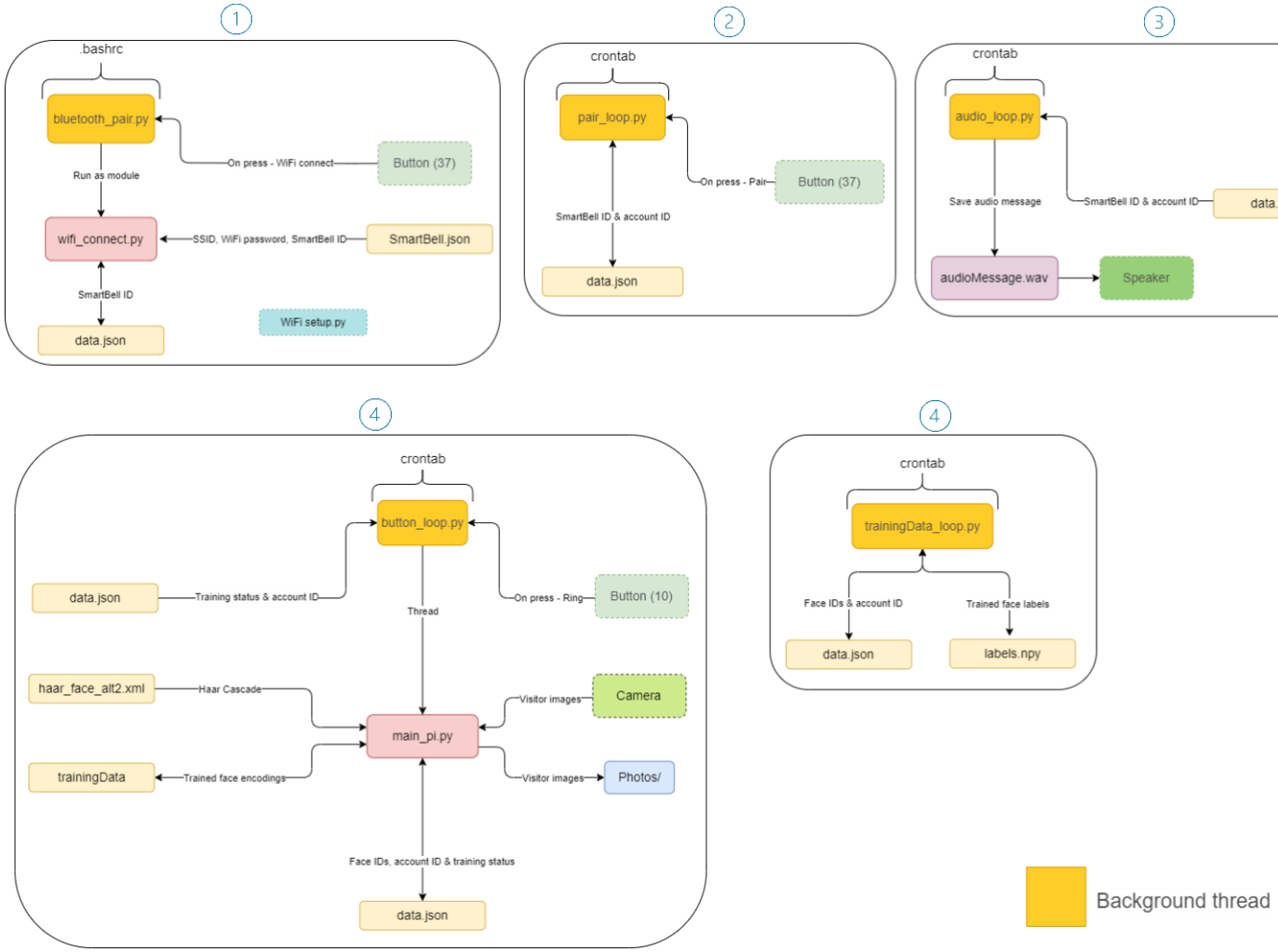
The table below shows the stages of **encryption of the access key:**

Variable	Variable value
accountID	**'MRynYQhtvlRuTdt9KY9etQwtdmR4TVMx2592BR1htVH='**
key (bytes encoded)	b'MRynYQhtvlRuTdt9KY9etQwtdmR4TVMx2592BR1htVH='
accessKey	**'AKI [redacted] 3EM'**
accessKey_encoded (bytes encoded)	b'AKI [redacted] 3EM'
accessKey_encrypted	**'gAAAAABiDOnBH_eAfXgb2kMKCsz5iLxStO0kOAdszPMagczKq6BM YleXnNgqeyrS3FxwLPsXtWr0ycqMmHsOCdWERu9dqtzQ4fHPYtDqK PSSA3GFr9R_twQ='**

The encrypted access key and secret key are returned to the client in a dictionary on **Line 36** for the client to decrypt and use.

Raspberry Pi

Raspberry Pi – File Structure

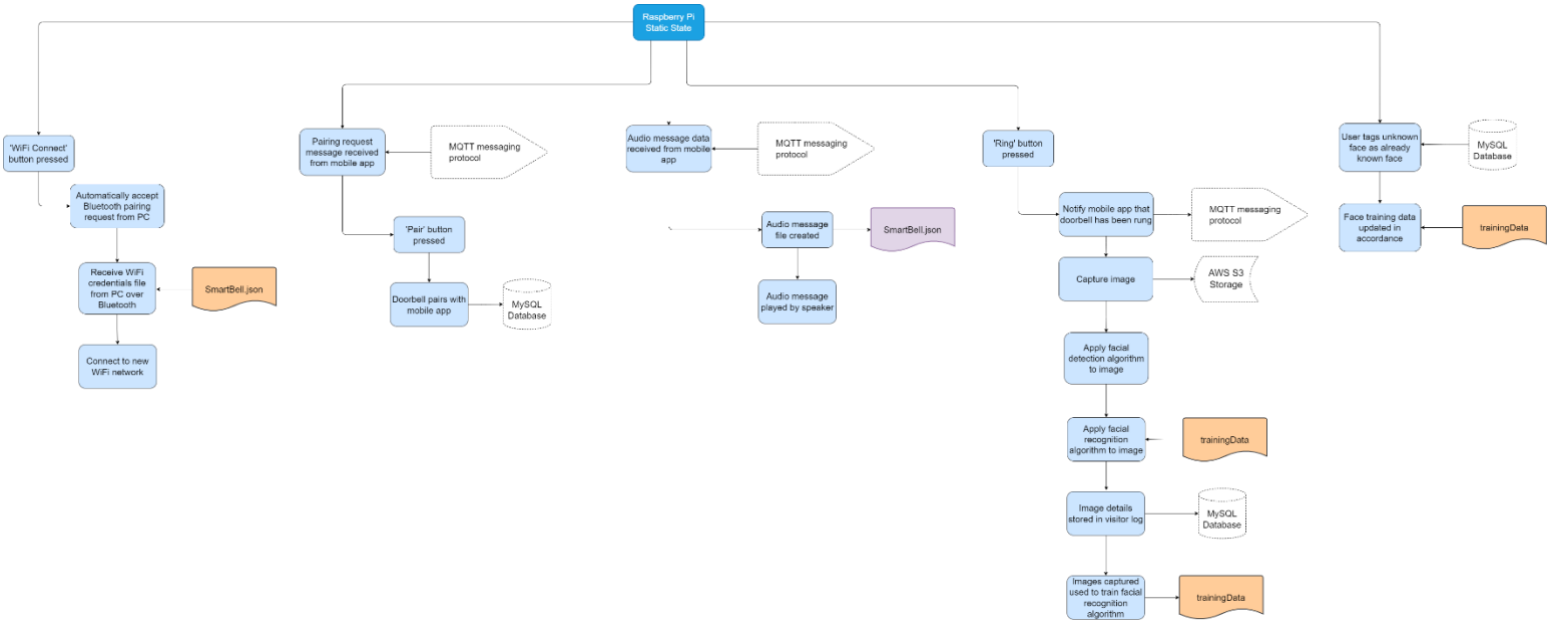


The Raspberry Pi and connected *peripherals* (the *camera* and 2 *buttons*) make up the physical **doorbell** element of the *SmartBell* system. As indicated by the numbered file groupings in the diagram above, the Raspberry Pi's complex file structure can be simplified into 4 categories which encapsulate all the functions of the Raspberry Pi doorbell:

1. *Connect the Raspberry Pi doorbell to WiFi using any PC*
2. *Pair the Raspberry Pi doorbell with user's account through the mobile app*
3. *Output user's audio messages they have created in the mobile app*
4. *Respond when the Raspberry Pi 'Ring' button is pressed by a visitor*

Raspberry Pi – System Hierarchy

The diagram below shows the 4 key functions of the Raspberry Pi, the hierarchy of operations within each function and the relationships with external services. As each of these 4 key behaviours can be executed at any time independently of the other behaviours, there is no hierarchical relationship between them.



Please note that references to **MySQL Database** and **AWS S3 Storage** should generally be read to include the **REST API** as an intermediary, since most communications with the MySQL Database and S3 Storage are executed by the REST API on behalf of the Raspberry Pi.

Raspberry Pi – Connect to WiFi using PC

The *Python* files bluetooth_pair.py and wifi_connect.py, along with input from the button connected to **GPIO pin** 37, together enable the user to set up a WiFi connection for their Raspberry Pi doorbell. This is essential: to communicate with the REST API server and mobile app, the Raspberry Pi doorbell must be connected to the internet, and WiFi must be used as requiring a cabled Ethernet connection would be inconvenient and would make the doorbell inaccessible for users with little technical knowledge.

1. User runs a command line program WiFi setup.py on any PC. This program walks the user through the steps required to initially set up their Raspberry Pi doorbell and asks the user to input a unique ID for the doorbell and the network SSID and password for their WiFi connection:

```
Welcome to your SmartBell!
This guide will walk you through the process of connecting your SmartBell to the internet.

Is this the first time setting up your SmartBell? (y/n) y

Please enter a unique name for you SmartBell: Test

Please enter the network SSID you would like to connect to: myWiFi
Please enter the passkey for your network with SSID 'myWiFi': myPassword

Please now select 'SmartBell' from the list of available bluetooth devices on your laptop and then and then
press the 'WiFi connect' button on your doorbell.
Once you have successfully paired with your SmartBell, please transfer the file called 'SmartBell.json' to the
SmartBell doorbell.
```

2. The command line program stores the WiFi credentials and doorbell ID in a *json* file SmartBell.json on the user's PC:

```
{"ssid": "myWiFi", "psswd": "myPassword", "id": "Test"}
```

3. The user pairs their PC with the Raspberry Pi doorbell over Bluetooth by pressing the doorbell's *'WiFi connect'* button (i.e. the button connected to GPIO pin 37) and then selecting *SmartBell* from the available devices:
   i. As the Raspberry Pi is **headless** (meaning there is no monitor, keyboard or mouse attached), the primary challenge was enabling the user to easily pair their PC with the Raspberry Pi, as the standard Bluetooth pairing procedure requires several verification steps to be completed through the Raspberry Pi's GUI
   ii. The Linux command line tool bluetoothctl is run on launch of bluetooth_pair.py automatically make the Raspberry Pi discoverable and ready to pair on boot:

bluetooth_pair.py

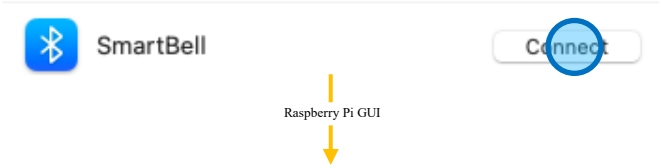
```
# make Raspberry Pi discoverable on boot.
os.system("""sudo bluetoothctl <<EOF
power on
discoverable on
pairable on
EOF
""")
```

Command Line Output

```
Agent registered
[bluetooth]# power on
[bluetooth]# discoverable on
[bluetooth]# pairable on
Changing power on succeeded
Changing discoverable on succeeded
```

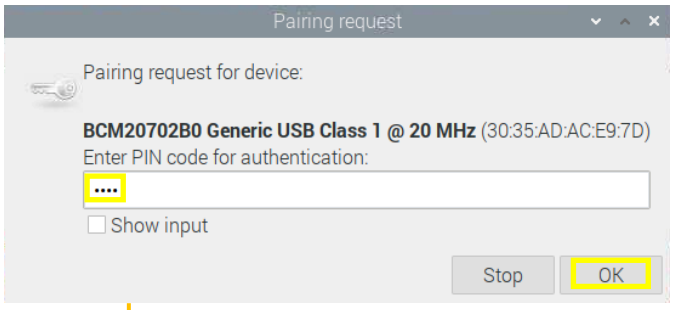
The system class of the os module is used to execute statements directly in the command line, as if they were being inputted via the terminal.
Using the Bluetooth configuration tool bluetoothctl, the command power on is used to turn on the Raspberry Pi's Bluetooth adapter.
The commands discoverable on and pairable on ensure the PC can view the Raspberry Pi in the list of available devices and then initiate the pairing process.

iii. After the user selects to pair with Raspberry Pi (called *'SmartBell'*), they must press the *'WiFi connect'* button on their doorbell to authenticate the Bluetooth pairing process and ensure only the owner of the doorbell can create a new Wifi connection:



User selects to pair with Raspberry Pi (i.e. *'SmartBell'*) from their PC.

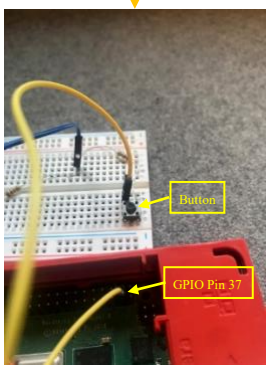
Raspberry Pi GUI



To accept the Bluetooth pairing request on the Raspberry Pi, the PIN code *'0000'* must be inputted into the text box and then *OK* button must be selected.

To interact with the authentication dialogue box, it *must* be visible. Therefore, the `pyautogui` module is also used in *'bluetooth_pair.py'* to hide any windows that may be opened on boot, the command `@lxterminal` is run on boot to upon a terminal window.

Press 'WiFi connect' button



To connect a button to the Raspberry Pi, the `RPi.GPIO` module is used to interface with the GPIO pins:

```
import RPi.GPIO as GPIO # Raspberry Pi GPIO library
GPIO.setmode(GPIO.BOARD) # Use physical pin numbering
GPIO.setup(37, GPIO.IN, pull up down=GPIO.PUD DOWN)
# Set pin 37 to be input pin and set initial value to be pulled low (off)
```

To detect the button press, an *infinite while loop* is used:

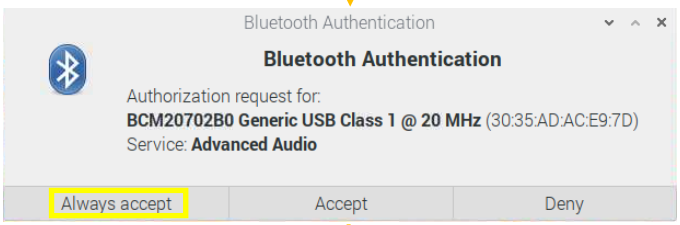
```
while True:
    if GPIO.input(37) == GPIO.HIGH and threading.active_count() == 1:
# when button pressed and pairing process thread not already initiated
        thread_run = threading.Thread(target=pair) # create pairing thread
        thread_run.start()
```

`pair` thread

```
import pyautogui # enable automation of key presses

def pair():
    # key entries to accept pairing request:
    pyautogui.write("0000")
    pyautogui.press("enter")
```

When the user presses the *'WiFi connect'* button, a function `pair` is called in a *thread*. In the `pair` function, the `write` class of the `pyautogui` module is used to automatically input the required PIN code and the `press` class is used to automatically select the *'OK'* box.

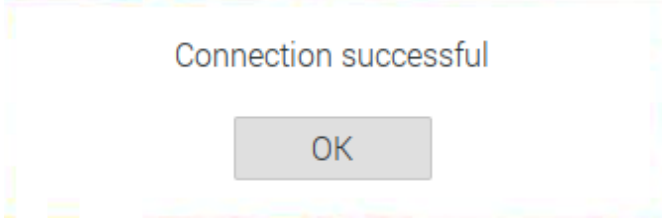
Raspberry Pi GUI



After the PIN code is inputted, a second dialogue box is shown. Here, the *'Always accept'* button must be selected automatically using the `press` method of the `pyautogui` method:

```
pyautogui.press("enter")
```

Raspberry Pi GUI



4. Using their PC, the user transmits the *json* file SmartBell.json to the Raspberry Pi:
   i. To automatically receive *json* files transmitted over Bluetooth, I set up an **OBEX Push Serve** on the Raspberry Pi (Douglas6, 2013):

- Install the required software using the command sudo apt-get install obexpushd
- Navigate to *system* file dbus-org.bluez.service and added the compatibility flag -C to the Bluetooth daemon:

```
GNU nano 5.4                          /etc/systemd/system/dbus-org.bluez.service
[Unit]
Description=Bluetooth service
Documentation=man:bluetoothd(8)
ConditionPathIsDirectory=/sys/class/bluetooth

[Service]
Type=dbus
BusName=org.bluez
ExecStart=/usr/libexec/bluetooth/bluetoothd -C
NotifyAccess=main
#WatchdogSec=10
#Restart=on-failure
CapabilityBoundingSet=CAP_NET_ADMIN CAP_NET_BIND_SERVICE
LimitNPROC=1
ProtectHome=true
ProtectSystem=full

[Install]
WantedBy=bluetooth.target
Alias=dbus-org.bluez.service
```

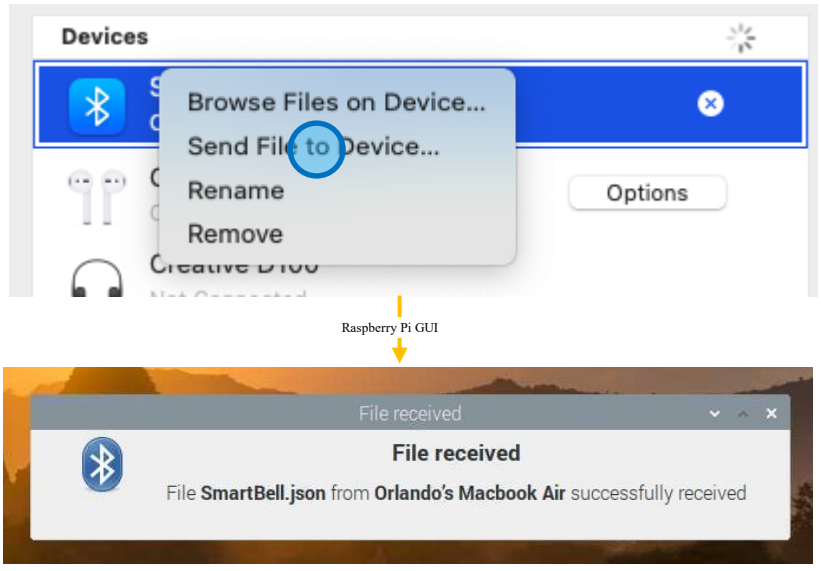
- Create a *system* file to automate the obexpush process:

```
GNU nano 5.4                          /etc/systemd/system/obexpush.service
[Unit]
Description=OBEX Push service
After=bluetooth.service
Requires=bluetooth.service

[Service]
ExecStart=/usr/bin/obexpushd -B23 -o /home/pi/Desktop/NEA/ComputerScience-NEA-RPi/bluetooth -n
Restart=always

[Install]
WantedBy=multi-user.target
```

Launch the obexpushd as a background process (indicated by flag -B) which can transmit files over Bluetooth on channel 23 and save them to */home/pi/Desktop/NEA/ComputerScience-NEA-RPi/bluetooth*

- Set the above service to run on boot using the command sudo systemctl enable obexpush. This is necessary as obexpush must be running when the PC pairs with the Raspberry Pi to ensure it recognises the obexpush service.

   ii. When the user selects to transmit SmartBell.json to the Raspberry Pi, the file is automatically received and saved:



The received file SmartBell.json is saved to a directory */home/pi/Desktop/NEA/ComputerScience-NEA-RPi/bluetooth*, which is inside the directory containing all the files required by the Raspberry Pi.

pair thread

```
import wifi_connect # import wifi_connect.py file as a module
path = '/home/pi/Desktop/NEA/ComputerScience-NEA-RPi/bluetooth/'
if len(os.listdir(path)) != 0: # if file has been received
    wifi_connect.run() # run module to connect Raspberry Pi to WiFi
```

When the pair thread verifies that the SmartBell.json file has been received, the main function run of *'wifi_connect.py'*, which is imported as module, is called.

5. The Raspberry Pi uses the network SSID and password stored in the received *json* file SmartBell.json to automatically establish a connection to the user's WiFi:

wifi_connect.py

```
1  if mySSID != '': # if user has sent a new SmartBell ID from their PC
2      try: # try except statement required as only needs to kill the
3  wpa_supplicant process if there is one running
4          os.system('sudo killall wpa_supplicant') # kills the wpa_supplicant
5  process
6          time.sleep(5)
7      except:
8          pass
9      command = (('wpa_passphrase "{}" "{}" | sudo tee -a
10 /etc/wpa_supplicant/wpa_supplicant.conf').format(mySSID, passkey))
11     # appends the correctly formatted network data to the WiFi configuration
12 file 'wpa_supplicant'
13     os.system(command) # execute command through terminal
14     time.sleep(5)
15     os.system('sudo wpa_supplicant -B -c
16 /etc/wpa_supplicant/wpa_supplicant.conf -i wlan0')
17     # wpa_supplicant automatically selects best network from
18 'wpa_supplicant.conf' to connect with and runs the WiFi connection process
```

Explanation

The variable mySSID is loaded from the received *json* file SmartBell.json; if it is not an empty string, this indicates that the user wishes to connect the Raspberry Pi to a new WiFi with SSID mySSID (**Line 1**). The command line software wpa_supplicant is used to control the wireless connection process on the Raspberry Pi. Therefore, to set up a new wireless connection, 3 steps must be carried out (Guoan, 2017):

i. Any current wpa_supplicant process must be terminated – this will only be possible if the Raspberry Pi is already connected to a WiFi network
   1) On **Line 4**, the system class of the os module is used to run sudo killall wpa_supplicant.

ii. The wireless configuration file wpa_supplicant.config must be updated to include the credentials for the new WiFi network that the user wants to connect the Raspberry Pi to
   1) The utility wpa_passphrase is used on **Line 9** is used to correctly format the SSID (mySSID) and password (passkey) stored in SmartBell.json for the WiFi configuration file wpa_supplicant.config:

```
pi@SmartBell:~ $ wpa_passphrase "{myWiFi}" "{myPassword}"

network={
        ssid="{myWiFi}"
        #psk="{myPassword}"
        psk=41c7a2ff336749dc43c48d47dba1ee7ed7fa98cfe28cd126af49c4ed88056634
}
```

2) The *pipe* key (|) is used in conjunction with the tee command to store the output of the wpa_passphrase utility in the WiFi configuration file wpa_supplicant.config (**Line 9**). Including -a after the tee command ensures that the new WiFi credentials are *appended* to the wpa_supplicant.config file, rather than overwriting the existing WiFi credentials, which would cause inconvenience for the user. The wpa_supplicant.config file is shown below:

```
  GNU nano 3.2        /etc/wpa_supplicant/wpa_supplicant.conf

ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=GB

network={
        ssid="BT-CHAK2C"
        psk="kaletsky12"
        key_mgmt=WPA-PSK
        disabled=1
}

network={
        ssid="iPhone"
        psk="orlando:)"
        key_mgmt=WPA-PSK
        disabled=1
}

network={
```

iii. A new wpa_supplicant connection process must be started to connect to the new WiFi network:

1) By running wpa_supplicant on the updated WiFi configuration file wpa_supplicant.config, the Raspberry Pi's *wireless network card* wlan0 is connected to the best available network (**Line 16**)
2) To ensure the wireless connection process is run in the background, the flag -B is added after wpa_supplicant (**Line 15**)

6. The Raspberry Pi stores the unique ID in the local *json* file data.json so the doorbell can be paired with the user's mobile app:

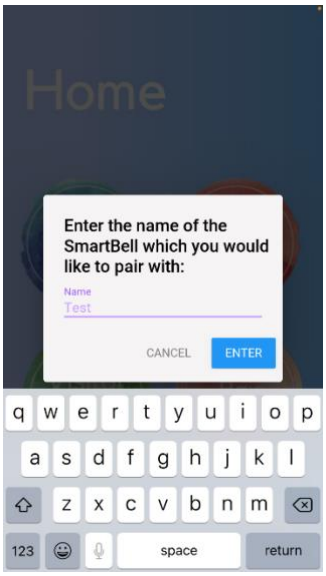
```
with open(join(path,'data.json'), 'r') as jsonFile:
    data = json.load(jsonFile)
data['id'] = SmartBellID # assign updated SmartBell ID
with open(join(path,'data.json'), 'w') as jsonFile:
    json.dump(data, jsonFile)
```

If the user adds a new value for SmartBellID in the SmartBell.json, the id field of data.json will be updated using the json module

Raspberry Pi – Pair with Mobile App

The *Python* file pair_loop.py, along with input from the button connected to **GPIO pin** 37, allow the user to pair their mobile app with their Raspberry Pi doorbell. This ensures that the user receives a notification and image of the visitor through the mobile app when the doorbell is rung, as well allowing the user to play audio messages through the doorbell's speaker from the mobile app.

1. Using the mobile app, the user enters the unique ID of the doorbell they wish to pair with:



The mobile app uses *Objective-C* to communicate with the Raspberry Pi over MQTT. Once the user has inputted a valid SmartBellID (as verified by the REST API path *'/checkPairing'*), the user's accountID is published as a message (publishData) to the MQTT topic *"id/*SmartBellID*"*:

```
MQTTPython = autoclass('MQTT')
mqtt = MQTTPython.alloc().init()
mqtt.publishData = publishData
mqtt.publishTopic = f"id/{self.piID}"
mqtt.publish()
```

2. When the Raspberry Pi *Python* file pair_loop.py receives a pairing request from the mobile app, the user must press the doorbell's *'Pair'* button (i.e. the button connected to GPIO pin 37) to confirm the pairing connection with their doorbell. Requiring the user to physically press a button to complete the pairing process ensures that the user can only connect to their own doorbell, ensuring only the intended user can view images of visitors at their doorstep.

```
def on_connect(client, userdata, flags, rc):
    # called when connection to MQTT broker established
    if rc == 0: # if connection is successful
        with open(join(path,'data.json')) as jsonFile:
            data = json.load(jsonFile)
        SmartBellID = str(data['id']) # retrieve SmartBell ID
        client.subscribe(f"id/{SmartBellID}") # mobile app
publishes to topic when it wishes to pair a user's account
with the doorbell
        client.message_callback_add(f"id/{SmartBellID}",
on_message) # add callback when message received to the topic
to indicate pairing request
```

Once the Raspberry Pi has established a connection with the MQTT broker **CloudMQTT**, on_connect is called to retrieve the doorbell's unique SmartBellID assigned by the user during the WiFi setup process. The Raspberry Pi then subscribes to the MQTT topic *"id/*SmartBellID*"*. By using the message_callback_add method of the MQTT client class paho.mqtt.client, a callback is created so that the on_message function is called when a message is published to the topic *"id/*SmartBellID*"* (indicating that the user has initiated a pairing request via the mobile app).

Pairing message received

```
def on_message(client, userData, msg):
    # callback function called when pairing request sent via
MQTT by mobile app
    time_start = time.time()
    while True:
        if GPIO.input(37) == GPIO.HIGH: # if user presses
'Pair' button on doorbell
            connectDoorbell(msg) # pair the doorbell with the
user's account
            break
        elif time.time() - time_start > 60: # pairing request
expires 60 seconds after it is initiated
            break
```

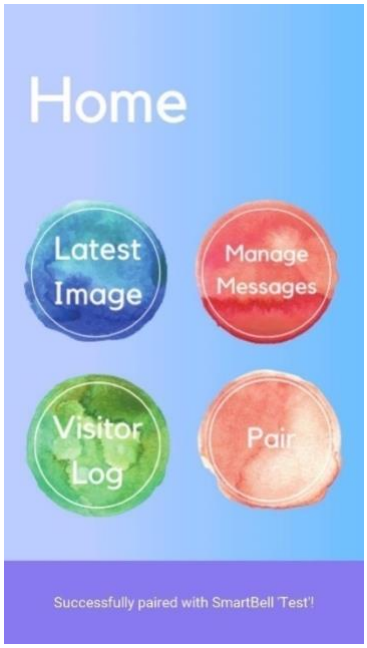
The on_message callback receives the MQTT message published to *"id/*SmartBellID*"* in bytes format as msg (i.e. the user's account ID). Using an *infinite while loop*, the Raspberry Pi continually checks the input status of the **GPIO Pin** 37. If the pin's state is HIGH, this indicates current is flowing between GPIO Pin 37 and the GND (ground) pin, and therefore the *'Pair'* button must have been pressed down to complete the circuit. As this physical input verifies that the user who initiated the pairing request owns the doorbell, the function connectDoorbell is called to complete the pairing process.

3. Using pair_loop.py, the doorbell stores the account ID of the user it is paired with in the *json* file data.json, allowing it to receive instructions to play audio messages sent via **MQTT** from the user's mobile app:

```
def connectDoorbell(msg):
    # pair doorbell with user's account
    with open(join(path,'data.json')) as jsonFile:
        data = json.load(jsonFile)
    SmartBellID = str(data['id'])
    accountID = msg.payload.decode() # decode payload sent by
user from mobile app (i.e. users' account ID)
    data['accountID'] = accountID # store the user's account ID
in the 'data.json' file as now paired with that account
    with open(join(path,'data.json'), 'w') as jsonFile:
        json.dump(data, jsonFile)
    data_accountID = {"accountID": accountID, 'id': SmartBellID}
    paired = requests.post(serverBaseURL +
"/update_SmartBellIDs", data_accountID).text # store pairing
details in MySQL table
```

To access the accountID of the user wishing to pair with the Raspberry Pi doorbell, the msg data sent via MQTT must be decoded into a UTF-8 string using the inbuilt decode function. Using the json module, this value of accountID can be stored locally in the data.json file. Finally, to notify the user that the pairing has been successful, the paired accountID and SmartBellID are stored in the MySQL table *SmartBellIDs* using the REST API path *'/update_SmartBellIDs'*.Finally, to

4. When the MySQL table *SmartBellIDs* is updated to register the pairing between the user's account and the doorbell, the mobile app briefly displays a *snackbar* to confirm that the pairing has been successful:



Raspberry Pi – Output Audio Messages

The *Python* file audio_loop.py plays the user's audio messages through the speaker connected to the Raspberry Pi doorbell when the user instructs them to be played through the mobile app.

1. The file audio_loop.py reads the account ID that the doorbell is paired with from the *json* file data.json and subscribes to the MQTT audio message topic for that account ID:

```
client = mqtt.Client()
client.username_pw_set(username="myUsername", password =
"myPassword")
client.on_connect = on_connect # creates callback for successful
connection with broker
client.connect("hairdresser.cloudmqtt.com", 18973)
# parameters for broker web address and port number

client.loop_forever() # indefinitely checks for messages on topics
that client is subscribed to
```

To receive the audio messages sent over MQTT, audio_loop.py must first establish a connection with the MQTT broker **CloudMQTT**. This is achieved by creating an object client of paho.mqtt module's Client class. By passing the required credentials to the methods username_pw_set and connect, a connection is established with the server. The method loop_forever forces the program to indefinitely poll the audio topics the client is subscribed to for messages.

Client connects to MQTT broker

```
def on_connect(client, userdata, -flags, rc):
    with open(join(path, 'data.json'), 'r') as jsonFile:
        data = json.load(jsonFile)
        accountID = data['accountID'] # load account ID

        client.subscribe(f"message/audio/{accountID}")
        client.message_callback_add(f"message/audio/{accountID}",
playAudio)
        client.subscribe(f"message/text/{accountID}")
        client.message_callback_add(f"message/text/{accountID}",
playText)
checkThread = threading.Thread(target=checkAccountID, args =
(accountID,)) # thread created to check whether any changes to the
account ID doorbell is paired with
checkThread.start()
```

The client object must subscribe to the same topic which the mobile app (via the REST API server) publishes the audio message details to. Moreover, it is important that *only* the client object and mobile app are communicating along this topic, to ensure the doorbell only outputs the desired audio message. Therefore, the user's accountID to specify the topic name as it is unique and common to both devices. As the processing required to output audio responses depends on whether the message is text based or bytes based, the client object subscribes to separate topics and creates separate callback functions which receive the messages published to their respective topics.

Run thread checkThread

```
def checkAccountID(currentID, ):
    while True:
        with open(join(path,'data.json')) as jsonFile:
            time.sleep(0.5)
            data = json.load(jsonFile)
        newID = str(data['accountID'])
        if newID != currentID: # if account ID has been changed
            # reconfigure topics that the Raspberry Pi is
subscribed as the doorbell's ID has been updated:
            client.unsubscribe(f"message/audio/{currentID}")
            client.unsubscribe(f"message/text/{currentID}")
            client.subscribe(f"message/audio/{newID}")
            client.message_callback_add(f"message/audio/{newID}",
playAudio)
            client.subscribe(f"message/text/{newID}")
            client.message_callback_add(f"message/text/{newID}",
playText)
            currentID = newID # set new values for accountID and
currentID for future comparisons
```

The threaded function checkAccountID ensures audio messages transmitted over MQTT are still received and outputted when the doorbell is paired with a new user account. By continually comparing the value of accountID stored locally in data.json (newID) with a variable currentID which stores the account ID referenced in the MQTT subscription topics, checkAccountID detects when the doorbell is paired with a new account. Using the unsubscribe method, the client object unsubscribes from the existing MQTT subscription topics and subscribes to the MQTT topics for audio message communication with the user newID.

2. When the user uses the mobile app to select an audio message to be played through the doorbell that they are paired with, the details of this audio message are published as a message to the MQTT audio message topic for their account ID:

```
MQTTPython = autoclass('MQTT')
mqtt = MQTTPython.alloc().init()
mqtt.publishData = str(self.messageText)
mqtt.publishTopic = f"message/text/{self.accountID}"
mqtt.publish()
```

Topic	Message
message/text/eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCrozmKR=	This is my test audio response!

If the user selects an audio response which is in a text format, the mobile app will use the *Objective-C* class mqtt to publish the actual message string (messageText) to the MQTT topic: *message/text/eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCrozmKR=*

```
MQTTPython = autoclass('MQTT')
mqtt = MQTTPython.alloc().init()
mqtt.publishData = str(self.messageID)
mqtt.publishTopic = f"message/audio/{self.accountID}"
mqtt.publish()
```

Topic	Message
message/text/eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCrozmKR=	This is my test audio response!

If the user selects an audio response which is in an audio format, the mobile app will use the *Objective-C* class mqtt to publish the messageID of the audio message (so that it can be downloaded from AWS S3 storage) to the MQTT topic: *message/audio/eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCrozmKR=*

3. The Raspberry Pi doorbell paired with the user's account ID receives the audio message details and saves the audio message to a local file audioMessages.wav:

message/audio/accountID

```
def playAudio(client, userData, msg):
    # output recorded audio message through doorbell's speaker
    messageID = msg.payload.decode() # decode payload sent via MQTT from
mobile app (i.e. messageID)
    downloadData = {"bucketName": "nea-audio-messages","s3File":
messageID}
# creates the dictionary which stores the metadata required to download
the pkl file of the personalised audio message from AWS S3
    response = requests.post(serverBaseURL + "/downloadS3", downloadData)
# send request to REST API path to download pickled audio message bytes
from AWS S3
    audioData = pickle.loads(response.content) # unpickles the bytes
string
messageFile = wave.open(join(path,"audioMessage.wav"), "wb")
messageFile.setnchannels(1) # audio stream module is single channel
messageFile.setsampwidth(2) # 2 bytes per audio sample (sample width)
messageFile.setframerate(8000) # 8000 samples per second
messageFile.writeframes(b''.join(audioData)) # write audio bytes to .wav
file
```

When the Raspberry Pi receives a message on the MQTT topic for audio-based responses, the callback function playAudio is invoked. The topic message is passed to playAudio as an object msg; to retrieve the messageID of the audio message to be played, the payload attribute of msg must be decoded. The pickled audio bytes of messageID can be downloaded sending a *POST* request to the REST API path *'/downloadS3'*, along with a *json* object downloadData storing the messageID of the audio message to be downloaded. Using wave, the audio bytes list audioData (storing each audio buffer) is written in the correct format to a local file audioMessages.wav.

message/text/accountID

```
def playText(client, userData, msg):
    # output typed (text-based) audio message through doorbell's speaker
    messageText = msg.payload.decode()
    TtS(messageText) # convert message text into audio file
```

When the Raspberry Pi receives a message on the MQTT topic for text-based responses, the callback function playText is invoked. As the message stores the message response string messageText, a function TtS is called to convert the text into an audio file.

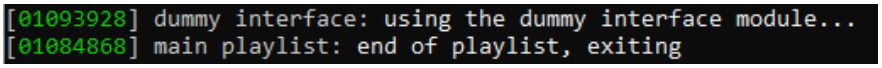
Call function TtS

```
def TtS(text):
    language = "en"
    TtS_obj = gTTS(text=text, lang=language, slow=False) # create text-to-
speech object
    TtS_obj.save(join(path,"audioMessage.wav")) # save text to speech
object as .wav file
```

The module gTTS, which is based on Google Translates text-to-speech API, is called to create an audio object which can be saved to a local file audioMessages.wav. This text-to-speech feature ensures that the doorbell is accessible to all users, as it allows those with speech disorders to create personalised audio responses.

4. The audio file audioMessages.wav is played through the speaker connected to the Raspberry Pi:

```
os.system("cvlc --play-and-exit {}".format(join(path,'audioMessage.wav')))
# play audio message directly through system using command line tool
'cvlc'
```



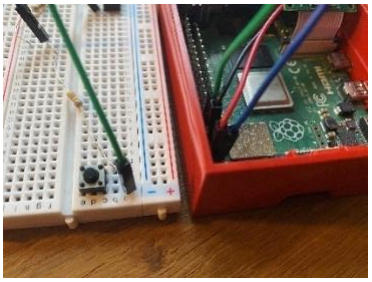
```
[01093928] dummy interface: using the dummy interface module...
[01084868] main playlist: end of playlist, exiting
```

Having encountered several issues using *Python* libraries to output the audio response audioMessages.wav, I was able to successfully use the command line tool cvlc to output audioMessages.wav through the speaker. To allow the user to play unlimited audio messages through the doorbell, I added --play-and-exit to ensure that cvlc automatically quit after each audio message was outputted.

Raspberry Pi – Respond when Doorbell is Rung

Python files button_loop.py and main_pi.py, along with input from the button connected to **GPIO pin** 10 and the camera connected to the Raspberry Pi, together capture the image of the visitor when the doorbell is rung and upload it to AWS S3, allowing the user to view the image in the mobile app. A facial recognition algorithm is also applied to the captured image, allowing the name of the visitor to be displayed in the mobile app if they are identified, and the new images of the visitor are used to train the data set for the facial recognition algorithm.

1. The visitor presses the doorbell's *'ring'* button (i.e. the button connected to GPIO pin 10):



To connect a button to the Pi, the RPi.GPIO module is used to interface with the Raspberry GPIO pins:

```
import RPi.GPIO as GPIO # Raspberry Pi GPIO library
GPIO.setmode(GPIO.BOARD) # Use physical pin numbering
GPIO.setup(10, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
# Set pin 10 to be input pin and set initial value to be pulled low (off)
```

Note that the parameter pull_up_down of the setup class is set to GPIO.PUD_DOWN. This indicates that the pin should be read as low when the button is not pushed and high when the button is pushed.

2. The *Python* file button_loop.py calls the *Python* file main_pi.py in a *thread* to capture the image of the visitor and carry out the associated image processing:

The function detect_buttonPressed calls the run function main_pi.py inside a *thread* thread_run, enabling multiple simultaneous threads of main_pi.py to run, reducing the length of the doorbell's inactivity between rings:

```
def detect_buttonPressed():
    while True: # Run forever
        if GPIO.input(10) == GPIO.HIGH and os.path.isfile(join(path, 'data.json')) == True:
            with open(join(path,'data.json')) as jsonFile:
                data = json.load(jsonFile)
                training = data['training'] # load training state
            if (threading.active_count() == 1 or training == 'True') and 'accountID' in data:
                thread_run = threading.Thread(target =runThread)
                # calls 'run' function of main_pi.py in a thread
                thread_run.start()
```

When the 'Ring' button (i.e. the button connected to GPIO pin 10) is pressed, it is set to state HIGH, triggering the initial selection statement in the function detect_buttonPressed (assuming the user has already set up their doorbell and so the file data.json exists). The second selection statement determines whether it is safe to create a new thread of main_pi.py – further to requiring that the doorbell is paired with a user's accountID, it is imperative to ensure that only one process is ever attempting to access the Raspberry Pi camera at a time. If a second process attempts to create a camera instance, the following error occurs:

```
mmal: mmal_vc_port_enable: failed to enable port vc.null_sink:in:0(OPQV): ENOSPC
mmal: mmal_port_enable: failed to enable connected port (vc.null_sink:in:0(OPQV))0x1cb3f60 (ENOSPC)
mmal: mmal_connection_enable: output port couldn't be enabled
```

As each thread of main_pi.py requires access to the camera, one of the following two conditions must be met to create a new thread of main_pi.py:

1. There are currently no active threads of main_pi.py (i.e. the main thread of button_loop.py is the only 1 thread running)
2. All the active threads of main_pi.py are in the training state (determined by the status of training), and so no longer require access to the camera

During *Development*, I noticed that the main_pi.py file would sometimes crash when it was called from the function detect_buttonPresssed in button_loop.py. Analysing the logs, I noticed this occurred due to issues establishing a connection with the MQTT to broker. To resolve this, I added the following code to ensure detect_buttonPresssed, and therefore main_pi.py, was only called once an internet connection was established:

```
while True:
    try:
        url.urlopen('http://google.com') # attempts to open 'google.com'
        detect_buttonPressed()
        break
    except: # if no internet connection is established yet, then wait 5 secs
        time.sleep(5)
```

3. main_pi.py uses the doorbell's camera to capture multiple images of the visitor and identify the least blurry image in which a face is detected

When the function run in main_pi.py is called, the method captureImage inside the class buttonPressed is called:

```
buttonPressed().captureImage()
```

Each time captureImage is called, up to two attempts are made to capture an image of the visitor to ensure maximum chances of capturing a high quality image which can be processed by the facial recognition algorithm without causing excessive latency between the doorbell being rung and the user receiving the image of the visitor through the mobile app. To capture an image, the Raspberry Pi must interface with the PiCamera module attached to Raspberry Pi:

```
self.camera = PiCamera() # create instance of Raspberry Pi camera
self.rawCapture = PiRGBArray(self.camera) # using PiRGBArray increases efficiency when
accessing camera stream
time.sleep(0.155) # delay to allow camera to warm up

while attempts < 2: # make up to two attempts to capture high quality image of visitor
    self.rawCapture.truncate(0) # clear any data from the camera stream
    self.camera.capture(self.rawCapture, format="bgr") # captures camera stream in 'bgr' format
    self.faceBGR = cv.flip(self.rawCapture.array,0) # flip image in vertical plane
```

The object camera enables the *Python* code to interact with the PiCamera module. Using PiRGBArray to wrap the camera object ensures the Raspberry Pi has direct access to the camera module byte stream, avoiding delays due to compression. Before any images are captured, I discovered that a brief delay (~0.155s) must be included to give the camera sensor time to initialise.

Using the capture method of the camera object and passing the object rawCapture (which enables direct to the camera's bytes stream), the byte stream at that instance is captured and stored in a 3D array. This 3D array is stored in the array attribute of the rawCapture object with *BGR* format as OpenCV requires image arrays with *BGR* pixel structure at their core for its image processing. The matrix stored has the following structure, with the three *BGR* pixel values (0-255) at each *x-y* coordinate on the image stored in the innermost array (i.e. the ratio of blue-to-red-to-green). The index of each innermost *BGR* array within the outer 2D array represents the *x-y* coordinate where that ***BGR* pixels** exists:

```
[[[141 123  97]
  [142 124  98]
  [141 124  99]
  ...
  [118  96  69]
  [116  91  62]
  [116  91  62]]

 [[123  71  50]
  [124  72  51]
  [123  69  54]
  ...
  [136 126 110]
  [136 127 108]
  [133 124 105]]
```

While the above *BGR* array is suitable for display to the user, before the facial data processing algorithms can be implemented, the image arrays must be converted into two separate formats:

1. Grayscale – **OpenCV** module
2. RGB – **facial-recognition** module

To **convert to grayscale**, I used OpenCV's cvtColor class, specifying the COLOR_BGR2GRAY conversion method to be used:

```
self.faceGray = cv.cvtColor(self.faceBGR, cv.COLOR_BGR2GRAY) # change to grayscale for OpenCV
```

As each pixel in a grayscale image is a single number which represents the brightness of the pixel on a scale 0-255, where 0 is black and 255 is white, the array for the grayscale image is 2D. The index of each element within its array alongside the index of its array within the outermost array provide the *x-y* coordinate for each **grayscale pixel**, describing the brightness of the image at each point of the image:

```
[[116 117 118 ...  82  83  83]
 [113 116 116 ...  81  84  83]
 [117 118 117 ...  79  82  83]
 ...
 [ 75  76  88 ... 115 115 116]
 [ 77  80  99 ... 116 117 115]
 [ 73  84 102 ... 115 116 115]]
```

To **convert to RGB**, I used OpenCV's cvtColor class, specifying the COLOR_BGRRGB conversion method to be used:

```
self.faceRGB = cv.cvtColor(self.faceBGR, cv.COLOR_BGR2RGB) # convert to RGB format
```

As *RGB* pixel structure is essentially the same as *BGR* pixel structure, except the values of the blue and red pixels are swapped:

- $B_{BGR} = R_{RGB}$
- $R_{BGR} = B_{RGB}$

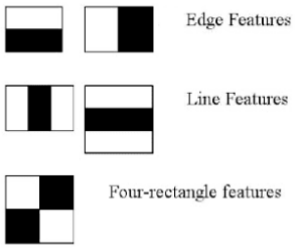
Therefore, the 3D array/matrix above with *BGR* format would like this:

```
[[[97 123  141]
  [98 124  142]
  [99 124  141]
  ...
  [69  96  118]
  [62  91  116]
  [62  91  116]

 [[50  71  123]
  [51  72  124]
  [54  69  123]
  ...
  [110 126 136]
  [108 127 136]
  [105 124 133]]
```

To determine whether a second image of the visitor should be captured by the camera before **facial recognition** is attempted, I decided to apply a **facial detection** algorithm to the initial captured image to verify whether a face can be located, rather than directly applying the facial recognition algorithm to the initial captured image. If a face can be detected, the facial recognition algorithm is applied directly. If a face cannot be detected, a second image of the visitor is captured. As the facial detection algorithm is significantly quicker to apply than the facial recognition algorithm, this ensures the greatest possibility that the Raspberry Pi applies the facial recognition algorithm to a high-quality image of the visitor's face, whilst minimising the time for the facial recognition process. To implement the facial detection algorithm, I used a **Haar Cascades Classifier**:

```
haarCascade = cv.CascadeClassifier(join(path,"haar_face_alt2.xml")) # reads in Haar cascade file
```

Haar Cascades work by training a classifier on positive and negative data points (i.e. grayscale photos with and without faces present) to detect the pattern of pixel brightness gradients which are associated with a face. This process is executed on each training image using a sliding window which moves across each *x-y* coordinate of the image and calculates the five following rectangular properties which describe the **pixel brightness gradients**:



Using this data, the distribution of pixel brightness values can be mapped for a typical face. This data is stored in an *xml* file.

During my in-development testing, I discovered that the most suitable Haar Cascade Classifier was *haar_face_alt2.xml* as the training data set aligned most closely with the typical visitor image captured by the Rasberry Pi camera module. The general structure of the *haar_face_alt2.xml* file is as follows:

```
<_>
    <maxWeakCount>16</maxWeakCount>
    <stageThreshold>-4.9842400550842285e+00</stageThreshold>
    <weakClassifiers>
      <_>
        <internalNodes>
          0 -1 9 -2.1110000088810921e-02</internalNodes>
        <leafValues>
          1.2435649633407593e+00 -1.5713009834289551e+00</leafValues></_>
      <_>
        <internalNodes>
          0 -1 10 2.0355999469757080e-02</internalNodes>
        <leafValues>
          -1.6204780340194702e+00 1.1817760467529297e+00</leafValues></_>
```

Having created the Haar Cascade Classifier object haarCascade by reading in the *xml* storing the Haar-like features, I used the detectMultiScale method to identify the location of the faces within the grayscale version of the image captured by the Raspberry Pi camera faceGray.

```
faceDetected = haarCascade.detectMultiScale(self.faceGray, scaleFactor=1.01, minNeighbors=6)    # returns
rectangular coordinates of a
face
```

Two arguments are passed to detectMultiScale to specify the properties of the facial detection process:

1. scaleFactor specifies the percentage by which the image is resized on each iteration of the facial detection algorithm, as the face size used in the imported Haar cascade is constant, but face sizes in the test image faceGray may vary depending on the visitor's positioning. As I have set scaleFactor to *'1.01'*, the image is reduced by 1% on each iteration. As this is a small step for each resizing, there is a high chance of correctly detecting the faces in the test image, although the algorithm processing does take slightly longer than if a larger value for scaleFactor was used.
2. minNeighbours specifies the minimum number of positive rectangles (storing the pixel brightness gradient) that must be adjacent to each candidate rectangle for the candidate rectangle to be actually positive. A higher value of minNeighbours will result in less face detections, but those faces that are detected are more likely to be correct.

As faceDetected is a 2D array which stores lists with the coordinates of the four points that enclose each detected face, the number of faces detected by in the test image can be calculated using the inbuilt len function:

```
num_faceDetected = len(faceDetected) # returns number of faces detected in image
```

Further to ensuring that at least one face can be identified in the test image, it is important also to verify whether the quality of the test image is sufficient for the facial recognition algorithm to attempt to determine the name associated with the visitor's face. To achieve this, the **Laplacian operator** is applied to the test image faceGray using the OpenCV module Laplacian:

```
blurFactor = cv.Laplacian(self.faceGray, cv.CV_64F).var() # returns blurriness of image
```

The **Laplacian operator** identifies areas in image with steep changes in gradient value (brightness) of pixels in a grayscale image, so it is often used for edge detection (i.e. where there is a steep gradient of pixel value, there is likely to be an edge present). However, in my application, I am interested in whether or not there are lots edges in the image, as blurry images have few edges and vice versa. Therefore, I used the var operation to find the **variance** of possible edge and non-edge areas of the image. If the variance is low, this indicates that there is a low spread of pixel brightness values, indicative of blurry image with few clear edges. Moreover, if the variance is high, this indicates there is a large spread in pixel brightness, indicative of a non-blurry image with clear images.

If the following two conditions are satisfied, the image is determined to be suitable for further processing:

1. `num_faceDetected >= 1`
2. `blurFactor >= 25`

4. A thread within main_pi.py is created to format and upload the captured image

To reduce latency between the doorbell being rung and the user receiving the image of visitor *and* their name (if they can be identified), the image of the visitor is formatted and uploaded in a thread which runs parallel to the execution of the facial recognition algorithm in the main thread:

```
self.uploadImage = threading.Thread(target=self.formatImage, args=(self.faceBGR,))
self.uploadImage.start()  # starts the thread which will run in pseudo-parallel to the rest of the program
```

To display the captured image of the visitor in the mobile app without distorting the image, the image visitorImage must be formatted before it is transmitted to the mobile app. The target pixel width and height of the visitor image are determined by applying a scaling factor to the pixle width and height of the mobile phone screen:

```
visitorImage_cropped_w = round(int(windowSize_mobile[0]) * 0.93) # target width of visitor image
visitorImage_cropped_h = round(int(windowSize_mobile[1]) * 0.54) # target height of visitor image
```

The pixel dimensions of the mobile app screen are stored in an array windowSize_mobile. Note that the round function must be applied to the calculated target pixel dimensions to ensure that visitorImage_cropped_w and visitorImage_cropped_h are both integers as they both represent a pixel value, which must be a whole number.

As the image must be resized by an equal amount in the vertical and horizontal plane to avoid distorting the image, it is only necessary to calculate the required resize scaling factor in one plane:

```
scaleFactor = visitorImage_cropped_h / visitorImage.shape[0] # factor by which height of image must be
scale down to fit screen
```

Note that the shape property of the OpenCV image visitorImage is a tuple which stores the number of *rows*, *columns* and *colour channels*. Therefore, the pixel height of the image can be found by retrieving the number of rows in the image (stored at index [0] in shape).

Using the constant resize factor scaleFactor, the width and height of the visitor image can be adjusted equally using OpenCV's resize module:

```
visitorImage = cv.resize(visitorImage, int(visitorImage.shape[1] * scaleFactor),
                int(visitorImage.shape[0] * scaleFactor)), interpolation=cv.INTER_AREA) # scales down width
and height of image to match required image height
```

The new visitorImage array will have the *correct height* (i.e. visitorImage_cropped_h). However, its width will not be equal to visitorImage_cropped_w as the resize factor used referenced the height of the image only. To resolve this, the image must be cropped in the horizontal plane:

```
visitorImage_centre_x = visitorImage.shape[1]//2 # x-coordinate of horizontal middle of image
visitorImage_x = visitorImage_centre_x - visitorImage_cropped_w // 2 # start x-coordinate of visitor image
```

The *x*-coordinate value of the horizontal centre of the image (visitorImage_centre_x) can be determined by using *floored division* to divide the width of the image by two. Note that *floored division* returns only the integer part of any calculation, which is necessary here as pixels must be an integer value. To crop the image to the desired width about the centre of the image, the start *x*-coordinate (visitorImage_x) is set to half the desired image width less than the *x*-coordinate value of the horizontal image centre.

Using the start *x*-coordinate and the target image width, the visitor image can be cropped horizontally:

```
visitorImage_cropped = visitorImage[0:visitorImage.shape[0],
                       visitorImage_x:visitorImage_x + visitorImage_cropped_w] # crops image width
```

visitorImage_cropped is a 2D array, with the outermost array representing each row which contains an inner-array storing the value the pixel for each column in the row:

Columns

Rows	0	1	2	3	4	5	6	7
0	104	105	105	106	106	107	107	106
1	105	105	105	106	106	107	107	106
2	105	105	105	107	106	107	106	106
3	105	106	107	106	106	106	105	106
4	106	105	105	106	105	105	105	107
5	106	106	105	106	105	105	106	106
6	106	106	105	106	105	106	106	106
7	105	106	104	105	106	107	106	106

As the outermost array (i.e. the image rows) is unchanged, the height of the image is unchanged. However, a sub-array is extracted from the inner-array of each row (as represented by yellow rectangle above) which corresponds to the pixel values stored in the columns of the image which are to be displayed.

Once the image is correctly formatted, it can be uploaded to **AWS S3 storage** using *Python's* boto3 module, so that it can be downloaded by the mobile app and displayed to the user. To create a connection with the AWS S3 storage client, an *access key* and *secret key* must be provided. To avoid any security breaches, these keys are only stored on the remote AWS S3 server. To access these keys, a request is made to the REST API server path *'get_*S3Key', which returns the keys encoded with the user's account ID. Using the user's account ID, these keys are decoded (accessKey and secretKey) and used to create a connection with the AWS S3 storage client in the method uploadAWS_image:

```
s3 = boto3.client("s3", aws_access_key_id=accessKey, aws_secret_access_key=secretKey) # initialises a
connection to the S3 client on AWS
s3.upload_file(Filename=self.path_visitorImage, Bucket=kwargs["Bucket"], Key=kwargs["Key"])  # uploads the
image file to the S3 bucket called 'nea-visitor-log'
```

The method upload_file of the AWS S3 client connection object s3 is called to upload the formatted visitor image to AWS S3 storage. The three arguments passed to upload_file are as follows:

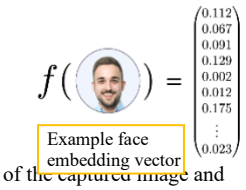
1. *Filename* – path to formatted visitor image on Raspberry Pi (path_visitorImage)
2. *Bucket* – name of bucket on AWS S3 storage where formatted visitor image is to be uploaded to ('*nea-visitor-log'*)
3. *Key* – name to be assigned to formatted visitor image in AWS S3 storage bucket (visitID)

Values passed as arguments to the uploadAWS_image function and stored in the dictionary kwargs. Using the key names (*Bucket* and *Key*), the required values can be accessed.

5. The data set of image encodings and associated labels stored in trainingData are read in by main_pi.py, enabling the facial recognition algorithm to be applied to the captured visitor image(s). If the visitor's face can be identified, the *json* file data.json is used to determine the face ID of the visitor.

Initially, I used OpenCV's in-built facial recognition algorithm. However, despite refining the algorithm threshold levels and optimising the preparation of the captured images before they were fed into the algorithm, my in-development testing found the accuracy of the facial recognition algorithm to be around 20%. To resolve this, I conducted further research into *Python* facial recognition algorithms. I decided to use to use the following libraries instead:

1. **dlib** – facial recognition network which enables implementation of 'deep metric learning' to create the *face encodings* (i.e. a vector storing 128 values which quantify a face) of the captured visitor images. These *face encodings* are saved to a file when the facial recognition algorithm is trained, and then compared with the images passed to the facial recognition algorithm to identify the visitor's name.



Example face embedding vector

2. **facial-recognition** – performs the facial recognition process by analysing the *face encodings* of the captured image and comparing them with the tagged *face encodings* stored in the trained data set.

The *face encodings* vector data is stored locally on the Raspberry Pi in trainingData. Therefore, firstly the facial recognition algorithm attempts to load trainingData:

```
fileName = join(path,"trainingData") # load training data (i.e. face encodings)
if not os.path.isfile(fileName): # if training data file doesn't exist (algorithm run for first time)
    return 'Unknown', False
data = pickle.loads(open(fileName, "rb").read()) # unpickle the face encodings
```

To avoid an error arising when the facial recognition algorithm is run for the first time, the isfile method of os class path is used to check whether the trainingData file exists. If it does not, the facial recognition algorithm is terminated by returning *'Unknown'*. If trainingData can be opened, the character stream stored in the file is *unpickled* and reconstructed into the dictionary object (data) module which stores the *face encodings* in the required format for the facial_recognition module. The dictionary data has two keys:

1. ***Encodings*** – 2D array where each element in the inner array stores the vector of numerical properties which describe each face in the training data:

```
{'encodings': [array([-8.81491527e-02,  6.61701038e-02,  5.76362200e-02, -3.60169671e-02,
       -8.19923058e-02, -6.39431085e-03, -9.72492620e-02,  2.05755495e-02,
        1.57165989e-01, -6.41372651e-02,  2.22493127e-01,  2.39041541e-02,
       -2.17836648e-01,  4.79959883e-03, -2.88938545e-02,  1.36327744e-01,
       -1.69954479e-01,  2.80711707e-03, -8.51075500e-02, -1.02522932e-01,
        5.76867163e-02,  7.19637722e-02,  8.46662670e-02,  6.77256361e-02,....]
```

2. ***Labels*** – an array which stores the label value associated with each vector of face encodings:

```
'labels': [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 3, 2, 2, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]}
```

To compare the *face encodings* stored in the *Encodings* key of data with the captured image, a vector of *face encodings* for the captured image must be generated using the face_encodings module of face_recognition:

```
encodings = face_recognition.face_encodings(faceRGB)
```

As encodings is an array which stores the vector *face encodings* for each face in the visitor image faceRGB, the algorithm iterates through each separate face encoding encoding in the test image:

```
for encoding in encodings: # iterate through face encodings in the visitor image
    matches = face_recognition.compare_faces(data["encodings"], encoding) # compare encoding of face in
visitor image with encodings in trained data set
    # matches contain array with boolean values True and False for each face encoding in 'data'
    if True in matches:
        matchedIndexes = [index for (index, match) in enumerate(matches) if match] # store indexes of
training set face encodings which match a face encoding in visitor image
```

Using the module compare_faces of the face_recognition library to compare the vector face encodings of the images in the training data set (stored in the dictionary data) with the vector face encoding of the test image (encoding), an array matches is created:

```
[True, True, True, True, True, True, False, False, True, True, True, True, True, True, True, True, True, True, True, False, False, False, False,
True, False, False, False, False, False, False, True, True, True, True, True, True, True]
```

Each element in matches stores a Boolean value which indicates whether the face encodings of the training image at the same index in the dictionary data match with the face encoding of test image. Using *list comprehension,* the enumerate function is used to iterate through the matches array and store the index of each value (i.e match) in matches which is *True* in the array matchedIndexes. As such, matchedIndexes stores a list of all the indexes of the training set face encodings which match a face encoding in the visitor image.

To identify which face label in the training data set has the strongest match with the visitor image, it is necessary to find the face label which has the most associations with indexes in matchedIndexes:

```
for index in matchedIndexes: # loop over the matched indexes and store a count for each recognized  face
    label = data["labels"][index] # get the label associated with the face encoding at index 'index'
    #increase count for the name we got
    labelCount[label] = labelCount.get(label, 0) + 1 # increment counter (value) for label (key) of face
encoding stored at 'index'
label = max(labelCount, key=labelCount.get) # return key with greatest value (i.e. label with greatest number of
matches)
```

As the face encodings and their associated labels are stored at the same index in their respective arrays, by iterating through matchedIndexes and obtaining the index of each face encoding which had a match with the visitor image, it is possible to find the label of each match. The dictionary labelCount stores the label of each match as a separate key, and increments its value by one each time a matched face encoding corresponds to the same key. This is possible as dictionary keys must be unique, so if the label corresponding to the index in matchedIndexes is already a key in labelCount its value will be altered, rather than a new key created. To increment the value by one, the current value of the key is found using the dictionary tool get and one is added to this value.

Having iterated through each index at which a matched face encoding is found, the *Python* in-built function max is used to find the key (i.e. label) in labelCount which has the greatest value. While it would be reasonable to assume that this key/label (stored as label) should be considered by default to be the correct label for the visitor image, during my in-development testing, I discovered an issue: where a test image matched with only one/two out of many training images for a particular label, the algorithm still considered this a successful match. Therefore, to refine the algorithm so that it did not produce any *false positives*, I implemented a feature which would enable it to ignore 'weak' matches:

```
matchCount = labelCount[label] # number of matches for label assigned to visitor image
actualCount = 0 # stores total number of face encodings with same label as label assigned to visitor image
for i in data['labels']: # iterate through each label in trained data set
    if i == label:
        actualCount +=1
if matchCount/actualCount > 0.5: # if visitor image matches more than 50% of training data set images with same
label
    return label, True
else:
    return 'Unknown', False
```

By accessing the value stored at the key label in labelCount, the number of *matches* between the visitor image and the training data set images with the same label is assigned to matchCount. Using a *for* loop, the *total* number of labels in the training data set with the same label as assigned to the visitor image is found and assigned to actualCount. To determine whether there was a sufficiently 'strong' match between the visitor image and the images in the training data set, I decided to set the threshold level at 50%; if over half the images in the training data set with a certain label match the visitor image, that particular label is considered to be the correct identifier for the visitor image. If not, the name of the visitor image is considered '*Unknown*'.

If *True* is the second value returned by the facial recognition algorithm, then it is necessary to find the face ID associated with the label assigned to the visitor's image. Each face ID associated with the user's account is stored locally on the Raspberry Pi in a *json file*:

```
with open(join(path,'data.json')) as jsonFile:
    self.data = json.load(jsonFile)
    self.faceIDs = [faceID for faceID in self.data[self.accountID]["faceIDs"]]
```

Having used the class json.load to convert the *json file* into a *Python* dictionary object data, *list comprehension* is used to extract each element stored in the array at the nested key *'faceIDs'* (which is particular for the user's account ID) and store them in the list faceIDs. The face ID stored in faceIDs at the index of the label returned by the facial recognition algorithm is the visitor's face ID:

```
self.faceID = self.faceIDs[self.label]
```

The visitor's face ID (faceID) is stored in the MySQL table *visitorLog* so that it can be retrieved by the mobile app and used to download the image from AWS S3 storage.

6. If the visitor is successfully identified, the encodings of the new images are stored in trainingData and are associated with the existing label for the identified visitor. As this increases the size of the training data set, this improves the facial recognition algorithm:

Further to the images captured by the Raspberry Pi when the doorbell is rung, the training algorithm begins by attempting to capture at least 2 new images of the visitor, using the same criteria regarding the number of faces identified and the blurriness of the image. This is done to maximise the size of the training data set, which will increase the accuracy of the facial recognition algorithm as it will be trained on a greater range of variations of each face.

Before the *face encodings* for any face are calculated and added to the training data set, it must be confirmed whether each training image contains the face of the same visitor:

```
for faceRGB in self.trainingImages:
    if self.faceRecognised == True: # if face identified in original visitor images
        label, faceRecognised = self.recognise(faceRGB) # get label for captured training image
```

Iterating through each *RGB* image in the list trainingImages which stores the array for each training image, the facial recognition algorithm recognise is called on each image (if faceRecognised is *True*). To verify whether the face in each image in the data set of training images matches the face associated with the face ID of the visitor, the variable label (the label assigned to the face in the training image) and the attribute label (the label of the visitor) are compared. If they are equivilant, it is assumed that that particular training image is valid and it is added to the training data set:

```
boxes = face_recognition.face_locations(faceRGB,model='hog') # bounding box around face location in image
encodings = face_recognition.face_encodings(faceRGB, boxes) # compute the facial encodings for the face
for encoding in encodings: # loop through each face encoding in image
    self.encodings.append(encoding)
    self.labels.append(label)
```

To create the *face encodings* for each training image, it is necessary to determine the coordinate of the bounding boxe(s) boxes for the face(s) in the visitor image using the class face_recognition.face_locations. The coordinates of the bounding box are stored in a tuple in boxes:

[(290, 468, 558, 200)]

Using the class face_recognition.face_encodings, vector *face encodings* for the training image are stored. These encodings are appended to the attribute array encodings while the associated label label is appended to the attribute array labels. This ensures that the index of each *face encoding* and the associated label are stored at the same index in their respective arrays.

To complete the training algorithm, the *face encodings* and associated labels must be stored in a dictionary (as this is the required format for the facial recognition algorithm):

```
trainingData = pickle.loads(open(join(path,"trainingData"), "rb").read())
trainingData['encodings'].extend(self.encodings) # add latest visitor image encoding data to list
trainingData['labels'].extend(self.labels) # add latest visitor image label to list

f = open(join(path,"trainingData"), "wb")
f.write(pickle.dumps(trainingData)) # to open file in write mode
```

As the data stored in trainingData is a character stream which represents a dictionary trainingData storing the *face encodings* and associated labels, pickle.loads is used to reconstruct this dictionary. Once the new *face encodings* and label arrays are added to the dictionary trainingData, pickle.dumps is used to convert the updated training image data set dictionary into a character stream which can be stored in trainingData.

7. If the visitor cannot be identified, the encodings of the new images are stored alongside a new label for the visitor in trainingData, and a new face ID is stored in data.json. If, through the mobile app, the user subsequently identifies the visitor to be a known face, the *Python* file trainingData_loop.py updates data.json and trainingData to store the correct face IDs and labels for the encoded data of the visitor's face.

The *Python* file trainingData_loop.py runs as a background thread on the Raspberry Pi, which continually calls the REST API server path *'checkFaces'*. The server path returns the face IDs which all correspond to the same face name for a particular user's account.

```
query = "SELECT faceName FROM knownFaces WHERE accountID = '%s' GROUP BY faceName HAVING COUNT(faceName)
> 1" % (data['accountID']) # select all face names which appear in more than one record in 'knownFaces'
for the same user account
myCursor.execute(query)  # the query is executed in the MySQL database
```

The SQL query selects any duplicate face names associated with the same user's account using the following SQL conditions:

- *GROUP BY*
  - Returns values (i.e. *faceName*) based on the field *faceName*. As such, each unique value of *faceName* associated with a user's account is returned, rather than all the values of *faceName* associated with a user's account
- *HAVING*
  - Much like the *WHERE* clause, the *HAVING* clause applies a selection criteria to the *GROUP BY* clause, restricting the rows affected by the *GROUP BY* clause
- *COUNT*
  - This *aggregate SQL function* returns the total number of records with the same value of *faceName* for a particular value of *faceName* associated with the user's account

To retrieve the face IDs for each of the duplicate face names, adjust the SQL database to remove the unique records in *knownFaces* storing the face details for faces which have been determined to be the same and update the *visitorLog* to reflect these changes, the following algorithm is used:

```
for faceName in result: # iterate through duplicate face names
    query = "SELECT faceID FROM knownFaces WHERE faceName = '%s' AND accountID = '%s' " % (
    faceName[0], data['accountID']) # get the face ID for each duplicate face name
    myCursor.execute(
        query)  # query is executed in the MySQL database which the variable 'myCursor' is connected to
    result = myCursor.fetchall()
    faceIDs.append(result)
    faceIDs_delete = result[1:] # get face IDs of duplicate names which are to be deleted from database
    faceID_keep = result[0][0] # get base face ID which is to remain stored in database
    for faceID in faceIDs_delete:
        query = "DELETE FROM knownFaces WHERE faceID = '%s' AND accountID = '%s'" % (faceID[0],
data['accountID']) # delete face IDs of duplicate names
        myCursor.execute(query)  # the query is executed in the MySQL database which the variable
'myCursor' is connected to
        query = "UPDATE visitorLog SET faceID = '%s' WHERE faceID = '%s' AND accountID = '%s'" % (
        faceID_keep, faceID[0], data['accountID']) # update visitor log to store same face ID for all
visits where visitor has same name
        myCursor.execute(query)  # the query is executed in the MySQL database which the variable
'myCursor' is connected to
mydb.commit()  # commits the changes to the MySQL database made by the executed query
response = jsonify(faceIDs)
return response
```

Once the Raspberry Pi function receives the response from the REST API server path *'checkFaces'* storing face IDs of duplicate face names, the Raspberry Pi updates the face IDs stored in the local file data.json and the *face encodings* stored in trainingData using the following algorithm:

```
trainingData = pickle.loads(open(join(path,"trainingData"), "rb").read()) # load known face encodings
newFaceID_update = faceIDs_update.pop(0)[0] # access and remove first face ID of the face IDs assigned to
the duplicate name - all face IDs associated with the duplicate name will be assigned to this new face ID
newLabel = faceIDs.index(newFaceID_update) # value of label of new face ID
oldFaceID_update = faceIDs_update.pop(0)[0] # old face ID which is to be assigned to new face ID
oldLabel = faceIDs.index(oldFaceID_update) # label of old face ID
faceIDs[oldLabel] = newFaceID_update # replace old face ID with new face ID
for (index, label) in enumerate(trainingData['labels']): # iterate through labels stored in
'trainingData' and replace the old labels with the new label, so the new label/face ID will be tagged to
the face encodings of the duplicate face names
    if label == oldLabel: # if label needs to be updated
        trainingData['labels'][index] = newLabel # change label value of old label to new label
data[accountID]["faceIDs"] = faceIDs # save updated face IDs
with open(join(path,'data.json'),'w') as jsonFile:
    json.dump(data, jsonFile)
with open(join(path,'trainingData'),'wb') as file:
    file.write(pickle.dumps(trainingData)) # store updated face encodings and associated labels
```

Raspberry Pi – Bootup Background Threads

To ensure the doorbell is accessible for all users, regardless of technical ability, it is important that it can be set up and used 'out-of-the-box'. This means that all the functionalities explained previously must be available for use as soon as the doorbell is powered on. Moreover, given the nature of a doorbell, these functionalities must remain available while the doorbell is powered on. Therefore, to achieve this, I decided to run all the following *Python* files **on bootup** as **background threads**:

1. pair_loop.py
2. trainingData_loop.py
3. button_loop.py
4. audio_loop.py
5. bluetooth_pair.py

Following my research into running files from boot on a Raspberry Pi, I discovered 2 possible methods (Dexter Industries, 2015):

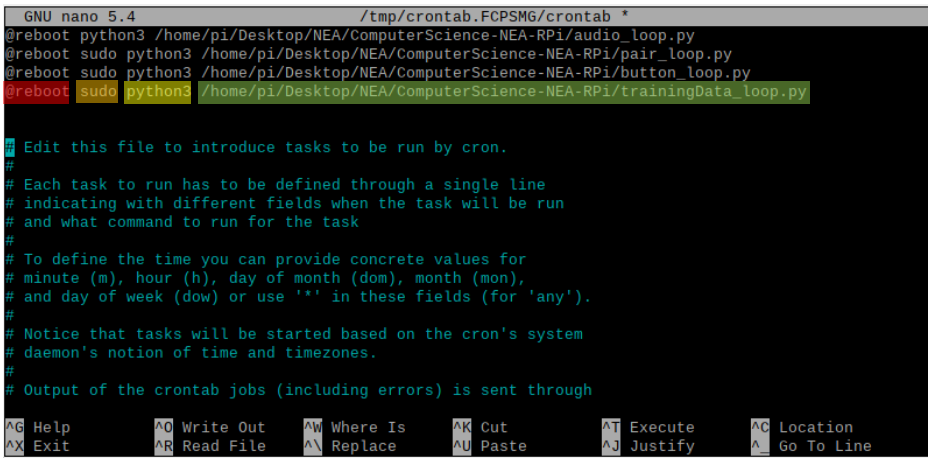
1. Create a **crontab job** to run the required files on reboot of the Raspberry Pi as a background thread
2. Modify the **.bashrc** file to add a statement which runs the required files on reboot of the Raspberry Pi, when a new terminal window is opened and when a new SSH connection is made with the Raspberry Pi

While I initially opted for the second method (i.e. modifying the '.bashrc' file). However, when I connected to the Raspberry Pi over **SSH** or using **VNC viewer** during *Development*, as new threads of the *Python* files which I had set up to run as a background thread at boot were initiated. This caused several issues; for example, when the Rasperry Pi doorbell 'ring' button was pressed, the 2 threads of the main_pi.py file (created by the 2 background threads of the button_loop.py file) made 2 simultaneous attempts to connect with the Raspberry Pi camera, causing the following error:

```
mmal: mmal_vc_port_enable: failed to enable port vc.null_sink:in:0(OPQV): ENOSPC
mmal: mmal_port_enable: failed to enable connected port (vc.null_sink:in:0(OPQV))0x1cb3f60 (ENOSPC)
mmal: mmal_connection_enable: output port couldn't be enabled
```

As a result, the background thread of the button_loop.py file crashed, meaning I could not test whether it was behaving as intended.

To resolve these issues, I used the first method and created **crontab jobs** to run the *Python* files numbered 1-4 above as background threads by using the terminal command sudo crontab -e:



There are 4 elements to the structure of each crontab job:

1. **Crontab job type** → @reboot specifies that the following command to run the *Python* file is to be run when the Raspberry Pi boots up
2. **Privileges** → sudo must be used for the last 3 *Python* files as they must write to files stored on the Raspberry Pi, so they require root privileges to run
3. **Python version** → python3 indicates that the *Python* files should be executed using version 3 of *Python*
4. **File path** → crontab jobs run from root, so the complete file path to the *Python* file must be used

Although the crontab jobs worked effectively in running the *Python* files numbered 1-4 above from boot, running *Python* file number 5 (bluetooth_pair.py) from boot using a crontab job caused it to crash. To discover where the issue was arising, I boot the Raspberry Pi to the *Command Line Interface* (rather than the *Graphical Interface*) as this would allow me to see the logs of the crontab jobs. The issue was importing the pyautogui module (which is used to enable the automated key presses required to accept Bluetooth pairing requests), since pyautogui is not designed to run on a headless device.

To workaround this issue, I took the following steps:

1. Through independent investigation, I discovered that the pyautogui module was imported without error when the *Python* file bluetooth_pair.py was called through a terminal window.

2. As statements in the **.bashrc** file are run when a new terminal window is opened (rather than just at boot, as this is the case for a crontab job), I added a statement at the end of the .bashrc file to run the *Python* file bluetooth_pair.py:

```
 GNU nano 3.2                    /home/pi/.bashrc                    Modified

  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi



sudo python3 /home/pi/Desktop/NEA/ComputerScience-NEA-RPi/bluetooth_pair.py &

# & ensures program runs in the background
```

Note the use of & at the end of the statement; this ensures the *Python* file runs as a background thread.

3. To open a new terminal on boot (which in turn causes the statement in the *'.bashrc'* file to be executed, as explained in step no. 2), I navigated to the autostart file using the command sudo nano /etc/xdg/lxsession/LXDE-pi/autostart. With this file open, I added @lxterminal at the end of the file to open the terminal on boot:

```
 GNU nano 3.2          /etc/xdg/lxsession/LXDE-pi/autostart

@lxpanel --profile LXDE-pi
@pcmanfm --desktop --profile LXDE-pi
@xscreensaver -no-splash
@lxterminal
```

Testing

Please refer to the testing video: https://youtu.be/hvcWZEq3k6o

Test number	Test purpose	Test data	Test action	Expected outcome	Actual outcome (video timestamp)
1	Automatically generate *json* file storing unique doorbell ID and WiFi credentials	SmartBellID = *'Test'* ssid = *'VM870701'* psswd = *'REDACTED'*	*Python* file *WiFi setup.py* is executed	*Python* programme prompts user to input unique ID for doorbell and WiFi credentials *SmartBell*.json file created and stored on PC	00:18
2	Connect to doorbell over Bluetooth		Navigate to list of available Bluetooth devices on PC	Raspberry Pi is discoverable from user's PC as a Bluetooth device called 'SmartBell'	00:38
3	Connect to doorbell over Bluetooth		Raspberry Pi *'WiFi connect'* GPIO button pressed by user	Raspberry Pi automatically enters verification code to confirm the Bluetooth pairing with the PC	00:53
4	Connect doorbell to WiFi and assign doorbell its unique ID	*SmartBell.json* file	Bluetooth file transmission utility used on PC to send *SmartBell.json* file	*SmartBell.json* file is transmitted over Bluetooth to Raspberry Pi *SmartBell.json* is received and saved automatically by Raspberry Pi	01:15

5	Connect doorbell to WiFi	*SmartBell.json* file		WiFi manager (*wpa_supplicant*) is restarted Raspberry Pi transitions from no WiFi connection → connecting to WiFi → connected to WiFi	01:44
6	Create new user account during initial launch of mobile app	firstname = *'Orlando'* surname = *'Alexander'* email = *'email@address.com'* password = *'MyPa55w0rd!'*	User taps *'Sign Up'* button	User account created User signed into new account Mobile app *Messages* screen displayed	02:11
7	Hide/show password		User taps *'closed eye'* icon next to password	Password characters are revealed	02:27
8	Valid password format	password = *'MyPa55w0rd'*	User taps *'Sign Up'* button	Error message displayed when user attempts to sign up due to invalid password	02:36
9	Initial mobile app launch walkthrough			Pulsing plus sign shown on *Messages* screen	02:43
10	Initial mobile app launch walkthrough		User attempts to create new message	Target view explaining the personalised audio messages is opened	02:47
11	User records audio response	User voice input	User presses and holds microphone icon	Audio recorded Rippling dots gif looped to indicate that audio is being recorded	02:58, 03:40 *and* 04:10

12	User plays back audio response	*.wav* file storing audio bytes recorded by user	User taps *'Playback'* button	Audio recording is outputted Rippling lines gif looped for duration of audio recording	03:03 *and* 04:01
13	User saves audio response		User taps *'SAVE/RENAME'* button after recording audio message	Dialog box prompting user to enter a name for the audio message is shown	03:05
14	User saves audio response	messageName = *'Test audio'*	User taps *'SAVE'* button in dialog box	Audio response stored on AWS S3 storage Details about audio response stored in *audioMessages* table of MySQL database Name of audio response shown in *Messages* screen	03:09
15	User types audio response	messageText = *'This is my test audio response!'*	User types message in multiline text box	User's message is shown on screen as they type	03:24
16	User re-records audio message	User voice input	User taps *'Record again'* button in *Review Audio Message* screen	Previously recorded audio message is discarded Screen to begin recording audio message is shown	03:44
17	User deletes audio message		User opens audio message and taps the bin icon	Audio response recorded removed from *audioMessages* table of MySQL database Audio response name no longer shown in *Messages* screen	04:06 *and* 04:31

18	User renames audio message		User taps *'SAVE/RENAME'* button on audio message that they have already saved	Dialog box prompting user to enter new name for audio message is shown	04:21
19	User renames audio message	messageName = *'Name 2'*	User taps *'SAVE'* button in dialog box	Updated name of audio response stored in *audioMessages* table of MySQL database Updated name of audio response shown in *Messages* screen	04:27
20	Help to record audio message displayed to user		User taps *'Help'* button on *Record New Message* screen	Snackbar briefly appears instructing user to *'Press and hold the microphone to speak'*	04:38
21	Pair user's account with doorbell		User taps *'Pair'* button on *Homepage* screen	Dialog box prompting user to enter name of doorbell they would like to pair with is shown	04:52
22	Pair user's account with doorbell	SmartBellID = *'Test'*	User taps *'ENTER'* button in dialog box	MQTT message transmitted announcing the pairing request with the concerned doorbell	04:54
23	Pair user's account with doorbell		Raspberry Pi *'Pair'* GPIO button pressed by user	Pairing between doorbell and mobile is confirmed Pairing status stored in *SmartBellIDs* table of MySQL database Snackbar briefly appears in mobile app confirming the pairing success	05:00

24	Check images stored on new account		User taps *'Latest Image'* button on the *Homepage* screen	Snackbar briefly appears in mobile app informing user that there are no images yet associated with their account	05:07
25	Doorbell rung by visitor for first time	Visitor's face	Raspberry Pi *'Ring'* GPIO button pressed by user	User notified that doorbell has been rung through mobile app Image of visitor captured Image of visitor displayed in mobile app Facial recognition algorithm applied to visitor's face *'Face unknown'* displayed in mobile app as face detected, but this is the first time the visitor has rung the doorbell so their name cannot be identified Visitor's face encodings used to train facial recognition data set	05:20 *and* 06:19
26	User responds to visitor with audio message		User taps *'Respond'* button in *Visitor Image* screen	User's personalised audio responses are shown	05:32
27	User responds to visitor with audio message		User taps *'YES!'* in dialog box asking them if they wish to play their audio message through the doorbell	Audio message is outputted through doorbell's speaker	05:35, 05:44 *and* 06:12

28	User identifies name of visitor	faceName = *'Orlando'*	User taps *'SAVE'* in dialog box prompting them to enter the visitor's name	Visitor name associated with visitor's face ID to improve facial recognition algorithm next time the visitor rings the doorbell	05:40
29	Doorbell rung by visitor (not first time)	Visitor's face	Raspberry Pi *'Ring'* GPIO button pressed by user	User notified that doorbell has been rung through mobile app Image of visitor captured Image of visitor displayed in mobile app Facial recognition algorithm applied to visitor's face Visitor name correctly displayed in mobile app Visitor's face encodings used to train facial recognition data set	05:52 – correct facial recognition 06:50 – correct facial recognition
30	User views visitor log sorted by date		User taps *'Visitor Log'* button on *homepage* screen User selects *'Date'* sorting option	Details of each visit displayed in scrolling list (visitor name, visit time and image of visitor) Visitor log list sorted by date, from most recent to oldest visit Average number of visits per day displayed	07:19

				Average time of visit displayed	
31	User views visitor log sorted by name		User taps '*Visitor Log*' button on *homepage* screen User selects '*Name*' sorting option	Details of each visit displayed in scrolling list (visitor name, visit time and image of visitor) Visitor log list sorted by name of visitor in alphabetical order Average number of visits per day displayed Average time of visit displayed	07:27
32	User signs out of account		User taps profile icon on *Homepage* screen	Dialog box shown asking user if they want to sign out and informing them of the consequences of doing so.	07:33
33	User signs out of account		Users taps '*YES*' on dialog box asking if they would like to sign out	User is signed out of the app *Login* screen is shown	07:35

Evaluation

Having spent many, many enjoyable hours designing and developing this project, I am incredibly proud of the outcome! The *SmartBell* doorbell and mobile app work seamlessly together to make the lives of those who work from home that little bit easier. With my solution, gone are the days of distracting doorbell jingles which so often interrupt important online meetings for home-workers. Instead, the user simply receives a discrete notification through the accessible mobile app which enables them to respond quickly and effectively to the visitor at their doorstep, without even leaving their desk!

Evaluation of Objectives

Objective	Evaluation	Associated test
Users must be able to **create individual accounts** on the mobile app which will allow them to access and play their personalised audio messages through personalised shortcut buttons.	I met this objective in its **entirety**. Using the mobile app, the user can create an individual account which is associated with their personalised audio messages. The user's account details are stored in the *users* table of the MySQL database, and the *accountID* field is used to link the user's account with the details of their audio message stored in the *audioMessages* table. *Kivy* is also used to create an accessible app GUI to access and play the user's personalised audio messages.	6, 7, 8, 32, 33
User must be able to **pair their doorbell with their account** via a simple GUI on the mobile app.	I met this objective in its **entirety**. The user can assign a unique ID to their doorbell during setup. Through the mobile app, the user can pair with the doorbell by inputting its ID and transmitting a pairing request using MQTT. To verify the pairing process, the user must press the *'Pair'* button on the doorbell. These pairing details are stored in the MySQL database table *SmartBellIDs*.	1, 2, 3, 4, 5, 21, 22, 23
The user must be able to **record or type**	I met this objective in its **entirety**.	11, 12, 13, 14, 15, 16, 17, 18, 19, 20

personalised audio message responses (to be played through the doorbell) and create shortcuts to access them quickly through the mobile app.	Using *KivyMD* to create text fields, the user can input the text for audio messages to be played by the doorbell. Enabling the user to record audio messages was a significantly greater challenge due to the restrictions imposed by Apple; however, using the *audiostream* module and many days of troubleshooting, I embedded the audio message recording feature into the mobile app. The details for all the audio messages are stored in the MySQL table *audioMessages*, and the bytes for the recorded audio messages are stored on AWS S3 storage. Each time the user accesses their audio messages through the mobile app, the name and content of each message is downloaded from the server and formatted so the user can easily access it. It should be noted that, due to a bug with the *audiostream* module, the mobile app sometimes failed to playback the audio through the mobile app. However, I have considered this issue to be beyond my control and I await a bug fix from the developers of the *audiostream* module	
The doorbell will attempt to **identify every individual** who presses the button.	I met this objective in its **entirety**. Throughout the *Development* process I continually improved the facial recognition algorithm. Initially, I relied entirely on *Open-CV*'s facial recognition capabilities. However, despite my persistent attempts to improve the accuracy of the algorithm, it seemed impossible to obtain consistently accurate results regardless of how well I prepared the training images. To resolve this, I used the *facial-*	25, 29

	recognition module instead (which is built using deep learning module *dlib*). This produced drastically more accurate facial recognition results, including recognising individuals based on images of them as a child.	
When the doorbell is pressed, the user will receive a **notification through their phone**, along with the name of the visitor (if identified). When this notification is opened, the user will be shown an image of the visitor in the app.	I **partially** met this objective. Using MQTT, the user receives a notification in a timely manner displaying an image of the visitor (downloaded from AWS S3 storage) and their name (if identified) through the mobile app when the doorbell they are paired with is rung. However, the user must have the mobile app already open for the notification be received, as pushing standard banner notifications to the user's mobile requires an **Apple Developer** subscription which costs $100. For this reason, I decided not to use push notifications.	25, 29
When visitor image is shown to user in the mobile app, the GUI will also have several large buttons to enable user to **select the desired audio message to be played through the doorbell.**	The audio messages created by the user are also available for selection when the notification is received. When an audio message is selected, its details are transmitted via MQTT to the doorbell which the user is paired with and outputted by the doorbell's speaker.	26, 27
When the doorbell is pressed, details about the visit will be added to the **visitor log**, which will show the name of the visitor (if identifiable), the time of the visitor, the image of the visitor and an option to download the image to their camera role. This visitor log will be accessible via the mobile app, with the option to sort/filter which visits are displayed. Various	I **partially** met this objective. The details about each visit are stored in the MySQL table *visitorLog* and, using a merge sort algorithm, are displayed to the user through the mobile app in a variety of ways depending on the user's preference, alongside the associated visitor image. Moreover, several aggregate SQL functions are used successfully to display summary statistics about the visits. However, in line with my focus on data protection in	30, 31

summary statistics about the visits will also be displayed.	the *Analysis* section, I decided to remove the feature which allowed the user to download the image of the visitor, as this would mean the visitor's image could be shared without their consent.	
Following each doorbell ring where the visitor was not identifiable, the user will be given the option to **enter the name of the visitor via the mobile app**. The user will be required to ask the visitor for consent to store their tagged image. This will ensure that the correct name is assigned to the visitor's face the next time that it is detected.	I **partially** met this objective. If the visitor cannot be identified, the Raspberry Pi will generate a new face ID which is associated with the visitors' face encodings. The user will also have the option to input the visitor's name through the mobile app, which will be stored in the MySQL table *knownFaces* alongside the visitor's face ID. If the same visitor rings the doorbell again, the Raspberry Pi identify their face ID and the mobile app will retrieve their name from *knownFaces*. If the user inputs a name for the visitor which already exists associated with their account, the Raspberry Pi will update its face training model to combine all the face encodings of visitors with the same name into a single face ID. As the user cannot download the images of visitors from the mobile app, the visitor images cannot be shared and therefore the visitor does not need to provide consent for their tagged image to be stored.	28

Further to the objectives outline above, I also designed and created a feature which enabled the user to **connect their doorbell to a WiFi network using Bluetooth and a PC**. This is particularly useful as it ensures the doorbell can be used out-of-the-box and without requiring the user to connect it to a monitor and keyboard to run the setup process.

Improvements to the Solution

To **improve** the *SmartBell* product, I would consider the following:

1. **Create weatherproof case for doorbell**
    - This would improve the usability and appearance of the doorbell, as the Raspberry Pi and its peripherals (i.e. the buttons, speaker and camera) would be contained in a single unit which could easily be attached to the user's door in all weathers. Moreover, the CAD case would be designed to ensure that the camera was placed at the correct height to capture the optimal image of the visitor's face, improving the accuracy of the facial recognition algorithm.
    - I would take measurements of the Raspberry Pi and attached peripherals to create a 2D technical drawing of the case.
    - Use CAD software (e.g. Fusion 360), I would convert this 2D drawing into a 3D design which could be sliced using Ultimaker Cura and printed using a 3D printer.
    - Once the Raspberry Pi and attached peripherals are enclosed within the 3D case, I would also use a waterproof sealant to ensure the Raspberry Pi and the peripherals were not damaged by weather conditions that the doorbell may be exposed to outdoors.

2. **Enable user to receive push notifications when doorbell is rung**
    - This would improve the convenience of the mobile app, as the user could receive push notifications through their mobile phone when the doorbell is rung regardless of whether the mobile app is open.
    - Implementing this solution would be possible if I had extra funds, as it would require me to subscribe to the **Apple Developer Programme,** which costs $100 per annum.

3. **Introduce option for user to connect doorbell to Bluetooth-enabled speaker**
    - This would expand the possible functionality of the *SmartBell* doorbell, as it would enable the doorbell to also behave as a standard doorbell which simply plays a jingle when it is rung.
    - To implement this extended functionality, I would use a similar approach to the one used to create a Bluetooth pairing between the user's PC and the Raspberry Pi.
    - I would also integrate a new feature into the mobile which would allow the user to choose whether the doorbell plays a jingle through the Bluetooth speaker when it is rung or whether it notifies the user through the mobile app, or both.

Feedback from Primary Client

Upon completion of my project, I arranged an interview with the Primary Client, Maria Kramer, to present the solution. For logistical purposes, this interview was conducted remotely using corporate video software so that Maria could easily see the mobile app and doorbell as I walked her through the functionalities outlined by the objectives I constructed for the project based on my initial interview with Maria during the *Analysis* phase.

Having seen all the functionalities of the *SmartBell*, including recording audio messages in the mobile app, receiving notifications when the doorbell is rung through the mobile app, identifying the names of visitors and outputting audio responses through the doorbell, could you give me your overall reflections on the product?

Maria: You know, it's very impressive. I can't believe you managed to do it – both physically and the software. The hardware is works well with the software, and the software is slick. I would say, overall, it's brilliant!

Could you explain any specific features of the solution that were unique and that you particularly liked?

Maria: Firstly, I think it's amazing that you can create these audio messages to a particular person and ring the bell and that's great. I must say that the image recognition was quite impressive considering that we're doing this also through a phone screen, and it still managed to correctly identify my face having only taken one picture of my face. So that's brilliant. I also think, is that the software I think is very clear and simple. I love the four buttons; it makes it very easy quickly respond to a visitor at the door without being distracted. I also think it's highly interactive and intuitive; it has really good two-way feedback.

Were there any improvements that you think could be made to the solution?

Maria: I think, as a next step, I suppose you would develop the hardware so it was contained in an enclosure or box. You could argue that you're more the software developer and you would potentially give that job to someone else who specialises in hardware design.

Having taken you through each of the objectives for the solution, to what extent do you think I successfully met each objective?

Maria: I think you explained all the functionalities very well and I can see clearly see how they match against each of your objective points. So, overall, it seems to me that you have successfully met each of your objectives.

Response to Feedback from Primary Client

I was delighted with the feedback from my Primary Client, Maria Kramer, as she was incredibly positive about the product. It was clear from her feedback that the *SmartBell* fulfilled the criteria she was looking for in the solution, as we had discussed initially during the *Analysis* phase.

In her feedback, Maria highlighted three key **features of the solution that she found to be unique and particularly appealing**:

1. **Facial recognition**
    a. Accuracy of face identification
    b. Ability to train algorithm successfully on a very small training data set
2. **Accessible mobile app UI**
    a. Mobile app home screen is very clear and simple
    b. Mobile app is intuitive to use
3. **Ease of playing audio messages through doorbell**
    a. Communicate with visitors at the door without being distracted

Further to being central to my objectives, the above three features mentioned by Maria were all general weaknesses of the existing solutions I identified in the *Analysis* phase, and are most applicable to my target market: people who work from home. Therefore, it is rewarding to discover that my project has both fulfilled the objectives I laid out during the *Analysis* phase *and* it has met the needs of the customer base who would be using it.

In terms of **improvements to the solution**, Maria suggested that I might consider further development of the *SmartBell* hardware, primarily a case for the doorbell. Interestingly, this aligned with one of the possible improvements I had previously identified. Therefore, if I were to develop this project further, I would focus on designing a case for the doorbell which made the *SmartBell* totally ready for market. However, as the key possible improvement identified by my primary client was related to hardware, it can be concluded that the software itself is ready for market. This was rewarding to hear, as my focus for the project was on the software development.

Bibliography

BBC News (2004). Global warming “biggest threat.” *news.bbc.co.uk*. [online] 9 Jan. Available at: http://news.bbc.co.uk/1/hi/sci/tech/3381425.stm.

British Geological Society (n.d.). *The greenhouse effect*. [online] British Geological Survey. Available at: https://www.bgs.ac.uk/discovering-geology/climate-change/how-does-the-greenhouse-effect-work/ [Accessed 29 Mar. 2021].

Call, M. (2020). *Why Is Behavior Change so Hard?* [online] accelerate.uofuhealth.utah.edu. Available at: https://accelerate.uofuhealth.utah.edu/explore/why-is-behavior-change-so-hard [Accessed 31 Mar. 2021].

CIA World Factbook (2021). *United Kingdom - The World Factbook*. [online] www.cia.gov. Available at: https://www.cia.gov/the-world-factbook/countries/united-kingdom/ [Accessed 31 Mar. 2021].

Committee on Climate Change (2019). *Net Zero The UK’s contribution to stopping global warming*. [online] *Climate Change Committee*, p.137. Available at: https://www.theccc.org.uk/publication/net-zero-the-uks-contribution-to-stopping-global-warming/ [Accessed 29 Mar. 2021].

Department for Transport (2019). *Transport Statistics Great Britain 2019*. [online] Available at: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/870647/tsgb-2019.pdf [Accessed 31 Mar. 2021].

Easton, M. (2017). How much of your area is built on? *BBC News*. [online] 9 Nov. Available at: https://www.bbc.co.uk/news/uk-41901294 [Accessed 31 Mar. 2021].

European Environment Agency (2020). *Transport: increasing oil consumption and greenhouse gas emissions hamper EU progress towards environment and climate objectives — European Environment Agency*. [online] www.eea.europa.eu. Available at: https://www.eea.europa.eu/publications/transport-increasing-oil-consumption-and/increasing-oil-consumption-and-ghg [Accessed 31 Mar. 2021].

Fullwood, C. (2019). *Alternative heating sources - what you need to know*. [online] onehome.org.uk. Available at: https://onehome.org.uk/your-home/199-alternative-heating-sources-what-you-need-to-know [Accessed 31 Mar. 2021].

hazardex (2008). *No single solution to the looming energy crisis*. [online] www.hazardexonthenet.net. Available at: https://www.hazardexonthenet.net/article/18504/No-single-solution-to-the-looming-energy-crisis.aspx [Accessed 29 Mar. 2021].

Jacobson, M. (2019). The 7 reasons why nuclear energy is not the answer to solve climate change. *DiCaprio Foundation*. [online] 24 Jun. Available at: https://www.leonardodicaprio.org/the-7-reasons-why-nuclear-energy-is-not-the-answer-to-solve-climate-change/ [Accessed 30 Mar. 2021].

Jevtic, A. (2016). *10 Countries that Are Running Out of Oil*. [online] Insider Monkey. Available at: https://www.insidermonkey.com/blog/10-countries-that-are-running-out-of-oil-489357/ [Accessed 29 Mar. 2021].

Jos Lelieveld et al. (2019). *Proceedings of the National Academy of Sciences*.

MacKay, D. (2008). *Sustainable Energy - Without The Hot Air*. UIT Cambridge. Available free online from www.withouthotair.com.

Nations Encyclopaedia (2014). *United Kingdom*. [online] Nationsencyclopedia.com. Available at: https://www.nationsencyclopedia.com/economies/Europe/United-Kingdom.html [Accessed 31 Mar. 2021].

Nations Online (2010). *Map of European Russia - Nations Online Project*. [online] Nations Online Project. Available at: https://www.nationsonline.org/oneworld/map/European-Russia-map.htm [Accessed 29 Mar. 2021].

Norwegian Petroleum (2015). *Exports of Norwegian oil and gas - Norwegianpetroleum.no*. [online] Norwegianpetroleum.no. Available at: https://www.norskpetroleum.no/en/production-and-exports/exports-of-oil-and-gas/ [Accessed 29 Mar. 2021].

Office for National Statistics (2016). *UK energy: how much, what type and where from? - Office for National Statistics*. [online] Ons.gov.uk. Available at: https://www.ons.gov.uk/economy/environmentalaccounts/articles/ukenergyhowmuchwhattypeandwherefrom/2016-08-15 [Accessed 29 Mar. 2021].

Office of Nuclear Energy (2018). *INFOGRAPHIC: How Much Power Does A Nuclear Reactor Produce?* [online] Energy.gov. Available at: https://www.energy.gov/ne/articles/infographic-how-much-power-does-nuclear-reactor-produce [Accessed 30 Mar. 2021].

Opus Energy (2020). *Biomass: The renewable energy source supporting the zero-carbon transition*. [online] www.opusenergy.com. Available at: https://www.opusenergy.com/blog/biomass-supports-zero-carbon-transition/ [Accessed 31 Mar. 2021].

Power Stations of the UK. (n.d.). *Power Stations of the UK*. [online] Available at: https://www.powerstations.uk [Accessed 26 Mar. 2021].

Rhodes, A., Gazis, E. and Gross, R. (2017). *Is the UK facing an electricity security crisis?* [online] Imperial College London - Energy Futures Lab, pp.4–5. Available at: https://imperialcollegelondon.app.box.com/s/gjldk43cwcoc0qw4upayu6kyrbsh9ylx [Accessed 26 Mar. 2021].

Roberts, T. and Clark, H. (2018). *Nuclear Electricity in the UK*. *https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/789655/Nuclear_electricity_in_the_UK.pdf*, p.64.

Schipani, V. (2018). *Wind Energy's Carbon Footprint*. [online] FactCheck.org. Available at: https://www.factcheck.org/2018/03/wind-energys-carbon-footprint/ [Accessed 30 Mar. 2021].

Shellenberger, M. (2018). *If Nuclear Power Is So Safe, Why Are We So Afraid Of It?* [online] Forbes. Available at: https://www.forbes.com/sites/michaelshellenberger/2018/06/11/if-nuclear-power-is-so-safe-why-are-we-so-afraid-of-it/?sh=29c6c8706385 [Accessed 30 Mar. 2021].

Stringer, B. (2014). *How much of England's countryside is protected?* [online] Barney's Blog. Available at: https://barneystringer.wordpress.com/2014/04/03/how-much-of-englands-countryside-is-protected/ [Accessed 31 Mar. 2021].

Taft, N. (2016). *The dangers of being dependent on foreign oil*. [online] Fuel Freedom Foundation. Available at: https://www.fuelfreedom.org/dangers-dependent-foreign-oil/ [Accessed 29 Mar. 2021].

Trane (2021). *Heat Pump Or Furnace? What's The Best Way To Heat Your Home? | Trane Topics*. [online] IR Trane Content Hub. Available at:

https://www.trane.com/residential/en/resources/heat-pump-vs-furnace-what-heating-system-is-right-for-you/ [Accessed 31 Mar. 2021].

U.S. Energy Information Administration (2020). *Nuclear power and the environment - U.S. Energy Information Administration (EIA)*. [online] Eia.gov. Available at: https://www.eia.gov/energyexplained/nuclear/nuclear-power-and-the-environment.php [Accessed 30 Mar. 2021].

Union of Concerned Scientists (2014). *Environmental Impacts of Natural Gas*. [online] Union of Concerned Scientists. Available at: https://www.ucsusa.org/resources/environmental-impacts-natural-gas [Accessed 30 Mar. 2021].

United Nations Development Programme (2007). *Human Development Report 2007/2008*. *http://hdr.undp.org/sites/default/files/reports/268/hdr_20072008_en_complete.pdf*.

Vekony, A. (2021). *Air Source Heat Pump 2021 (Cost, Types, Savings)*. [online] www.greenmatch.co.uk. Available at: https://www.greenmatch.co.uk/heat-pump/air-source-heat-pump#how-it-works [Accessed 31 Mar. 2021].

Walker, D. (2019). *Electric planes are here – but they won't solve flying's CO_2 problem*. [online] The Conversation. Available at: https://theconversation.com/electric-planes-are-here-but-they-wont-solve-flyings-co-problem-125900 [Accessed 31 Mar. 2021].

Wikipedia. (2021). *Trolleybus*. [online] Available at: https://en.wikipedia.org/wiki/Trolleybus [Accessed 31 Mar. 2021].

World Nuclear Association (2021). *Nuclear Power in the United Kingdom |UK Nuclear Energy - World Nuclear Association*. [online] World-nuclear.org. Available at: https://www.world-nuclear.org/information-library/country-profiles/countries-t-z/united-kingdom.aspx [Accessed 30 Mar. 2021].

Appendix

Mobile app

https://github.com/orlandoalexander/ComputerScience-NEA-App

main.py

```
import os
os.environ[
    'KIVY_AUDIO'] = 'avplayer' # control the kivy environment to ensure audio
input can be accepted following audio output
from kivymd.app import MDApp
from kivymd.uix.textfield import MDTextField
from kivymd.uix.dialog import MDDialog
from kivymd.uix.taptargetview import MDTapTargetView
from kivymd.uix.button import MDFlatButton, MDRaisedButton
from kivymd.uix.list import TwoLineAvatarListItem
from kivymd.uix.list import ImageLeftWidget
from kivy.uix.image import AsyncImage
from kivy.uix.boxlayout import BoxLayout
from kivy.core.audio import SoundLoader
from kivy.uix.screenmanager import ScreenManager, Screen, SlideTransition,
NoTransition
from kivy.lang import Builder
from kivy.clock import Clock
from kivy.animation import Animation
from kivy.storage.jsonstore import JsonStore
from os.path import join
import re
import random
import string
import requests
import hashlib
from threading import Thread
import time
import math
import wave
from pyobjus import autoclass
from audiostream import get_input
import pickle


serverBaseURL = "http://nea-env.eba-6tgviyyc.eu-west-2.elasticbeanstalk.com/"
# base URL to access AWS elastic beanstalk environment


class WindowManager(ScreenManager):
    # 'WindowManager' class used for transitions between GUI windows
    pass


class Launch(Screen, MDApp):
    # coordinate the correct launch screen for the mobile app depending on the
current status of the app
```

```
    def __init__(self, **kw):
        super().__init__(**kw)
        self.statusUpdate()
        Clock.schedule_once(self.finishInitialising)  # Kivy rules are not
applied until the original Widget (Launch) has finished instantiating, so must
delay initialisation

    def finishInitialising(self, dt):
        # applies launch processes which require access to Kivy ids
        self.manager.transition = NoTransition() # set transition type
        if self.initialUse == True: # initial launch of mobile app
            self.manager.current = "SignUp"
        elif self.loggedIn == True: # user already logged in
            createThread_ring(self.accountID, self.filepath) # connect to MQTT
broker to receive messages when visitor presses doorbell
            self.manager.current = "Homepage"  # if the user is already logged
in, screen 'Homepage' is called to allow the user to navigate the app
        else: # not initial launch of app and user not logged in
            self.manager.current = "SignIn"

    def statusUpdate(self):
        # update the attributes assigned to the locally cached data about the
user details and the app status
        self.filepath = MDApp.get_running_app().user_data_dir # path to
readable/writeable directory to store local data
        jsonFilename = join(self.filepath,"jsonStore.json")  # if file name
already exists, it is assigned to 'self.filename'. If filename doesn't already
exist, file is created locally on the mobile phone
        self.jsonStore = JsonStore(jsonFilename)  # wraps the json file as a
json object
        if not self.jsonStore.exists("localData"): # if the mobile app is
running for the first time, the key 'localData' will not exist
            self.jsonStore.put("localData", initialUse=True, loggedIn=False,
accountID="", paired=False) # sets launch properties
        self.initialUse = self.jsonStore.get("localData")["initialUse"]
        self.loggedIn = self.jsonStore.get("localData")["loggedIn"]
        self.paired = self.jsonStore.get("localData")["paired"]
        self.accountID = self.jsonStore.get("localData")["accountID"]

    def dismissDialog(self, instance):
        # called when 'Cancel' is tapped on the dialog box
        self.dialog.dismiss()  # closes the dialog box

    def openSnackbar(self):
        # controls the opening animation of the snackbar
        animation = Animation(pos_hint={"center_x": 0.5, "top":
self.topHeight}, d=0.03)  # end properties of the snackbar animation's opening
motion
        animation.start(self.ids.snackbar)  # executes the opening animation

    def dismissSnackbar(self):
        # controls the closing animation of the snackbar
        time.sleep(self.sleepTime)  # delay before snackbar is closed
        animation = Animation(pos_hint={"center_x": 0.5, "top": 0}, d=0.03)  #
end properties of the snackbar animation's closing motion
        animation.start(self.ids.snackbar)  # executes the closing animation


class Homepage(Launch):
    # home screen allows user to navigate to each screen in mobile app

    def pairSelect(self, **kw):
```

```
        # called when user selects 'Pair' button on home screen
        super().__init__(**kw)
        self.statusUpdate()
        if self.paired == False: # user account not paired with doorbell
            title = f"Enter the name of the SmartBell which you would like to
pair with:"
            self.pairDialog(title) # open dialog to allow user to pair with
doorbell
        else:
            self.piID=self.paired
            self.alreadyPaired_dialog()

    def account(self):
        # called when user taps 'Account' icon on home screen
        self.statusUpdate()
        self.signOut_dialog() # open dialog which gives user the option to
sign out of their account

    def signOut(self, instance):
        # signs user out of their account
        self.dialog.dismiss()
        self.jsonStore.put("localData", initialUse=self.initialUse,
loggedIn=False, accountID='', paired=False)
        self.statusUpdate()

MDApp.get_running_app().manager.get_screen('SignUp').ids.firstName.text = ''
        MDApp.get_running_app().manager.get_screen('SignUp').ids.surname.text
= ''
        MDApp.get_running_app().manager.get_screen('SignUp').ids.email.text =
''
        MDApp.get_running_app().manager.get_screen('SignUp').ids.password.text
= ''
        MDApp.get_running_app().manager.get_screen('SignIn').ids.email.text =
''
        MDApp.get_running_app().manager.get_screen('SignIn').ids.password.text
= ''
        self.manager.current = "SignIn"

    def signOut_dialog(self):
        # creates dialog box to ask user to confirm whether they would like to
sign out of their account
        self.dialog = MDDialog(
            title='Sign out?',
            text = 'If you sign out, you will be unpaired from any existing
connection with a SmartBell.',
            auto_dismiss=False,
            type="custom",
            buttons=[MDFlatButton(text="NO", text_color=((128 / 255), (128 /
255), (128 / 255), 1),
                                  on_press=self.dismissDialog),
                     MDRaisedButton(text='YES', md_bg_color=(136 / 255, 122 /
255, 239 / 255, 1),
                                    on_press=self.signOut)])  # creates the
dialog box with the required properties for the user to input the name of the
audio message recorded/typed
        self.dialog.open()  # opens the dialog box

    def pairDialog(self, title):
        # creates dialog to enable user to enter ID of SmartBell to pair with
        self.dialog = MDDialog(
            title=title,
            auto_dismiss=False,
```

```
            type="custom",
            content_cls=DialogContent(),  # content class
            buttons=[MDFlatButton(text="CANCEL", text_color=((128 / 255), (128
/ 255), (128 / 255), 1),
                                  on_press=self.dismissDialog),
                     MDRaisedButton(text='ENTER', md_bg_color=(136 / 255, 122
/ 255, 239 / 255, 1),
                                    on_press=self.pair)])  # creates the
dialog box with the required properties for the user to input the name of the
audio message recorded/typed
        self.dialog.open()  # opens the dialog box

    def alreadyPaired_dialog(self):
        # creates dialog asking user to confirm whether they would like to
alter their current SmartBell pairing
        self.dialog = MDDialog(
            title=f"You are currently paired with SmartBell '{self.piID}'.\nDo
you want to unpair / pair with a new SmartBell?",
            auto_dismiss=False,
            type="custom",
            buttons=[MDFlatButton(text="CANCEL", text_color=((128 / 255), (128
/ 255), (128 / 255), 1),
                                  on_press=self.dismissDialog),
                     MDRaisedButton(text='UNPAIR / PAIR', md_bg_color=(136 /
255, 122 / 255, 239 / 255, 1),

on_press=self.dismissDialog_alreadyPaired)])
        self.dialog.open()  # opens the dialog box

    def dismissDialog_alreadyPaired(self, instance):
        # creates dialog to enable to unpair from current SmartBell or pair
with new SmartBell
        self.dialog.dismiss()  # closes the dialog box
        title = f"Enter the name of the SmartBell which you would like to pair
with. Alternatively, enter 'unpair' to unpair from SmartBell '{self.piID}':"
        self.pairDialog(title)

    def pair(self, instance):
        # executes SmartBell pairing process
        self.dialog.dismiss()
        for object in self.dialog.content_cls.children:  # iterates through
the objects of the dialog box where the user inputted the name of the audio
message they recorded/typed
            if isinstance(object, MDTextField):  # if the object is an
MDTextField
                newID = object.text
        if newID.lower() == 'unpair':
            self.jsonStore.put("localData", initialUse=self.initialUse,
loggedIn=self.loggedIn, accountID=self.accountID,
                               paired=False) # write updated data to the json file
            self.statusUpdate()  # update launch variables
            publishData = ""
            id = self.piID
            pairing = False
            pair_thread = Thread(target=pairThread, args=(self.accountID, id,
pairing, self.jsonStore))
            pair_thread.start()
        else:
            self.piID= newID
            publishData = str(self.accountID)
            dbData_id = {'id': self.piID}
            response = (requests.post(serverBaseURL + "/checkPairing",
```

```
dbData_id)).text
            if response == 'notExists': # if SmartBell with ID entered by user
doesn't exist
                self.ids.snackbar.text = f"No SmartBell with the name
'{self.piID}' exists!"
                self.topHeight = 0.13
                self.sleepTime = 5
                self.openSnackbar() # calls the method which creates the
snackbar animation
                thread_dismissSnackbar = Thread(target=self.dismissSnackbar,
args=(),
                                                daemon=False)  #
initialises an instance of the 'threading.Thread()' method
                thread_dismissSnackbar.start()  # starts the thread which will
run in pseudo-parallel to the rest of the program
            elif response == 'exists':
                pairing = True
                pair_thread = Thread(target=pairThread, args=(self.accountID,
self.piID, pairing, self.jsonStore))
                pair_thread.start()
        MQTTPython = autoclass('MQTT') # interface between Python code and
objective C class 'MQTT'
        mqtt = MQTTPython.alloc().init()
        mqtt.publishData = publishData
        mqtt.publishTopic = f"id/{self.piID}"
        mqtt.publish() # publish message to MQTT topic 'id/piID' via objective
C class 'MQTT'


class SignUp(Launch):
    # 'SignUp' class allows user to create an account

    def createAccount(self):
        # called when user taps the Sign Up button to check validity of
details entered by user
        self.firstName_valid = False  # variable which indicates that a valid
value for 'firstName' has been inputted by the user
        self.surnameValid = False  # variable which indicates that a valid
value for 'surname' has been inputted by the user
        self.emailValid = False  # variable which indicates that a valid value
for 'email' has been inputted by the user
        self.passwordValid = False  # variable which indicates that a valid
value for 'password' has been inputted by the user
        if self.ids.firstName.text == "":  # if no data is inputted...
            self.ids.firstName_error.opacity = 1  # ...then an error message
is displayed
        else:
            self.firstName = self.ids.firstName.text  # inputted first name
assigned to a variable
            self.ids.firstName_error.opacity = 0  # error message removed
            self.firstName_valid = True  # variable which indicates that a
valid value has been inputted by the user
        if self.ids.surname.text == "":  # if no data is inputted...
            self.ids.surname_error.opacity = 1  # ...then an error message is
displayed
        else:
            self.surname = self.ids.surname.text  # inputted surname assigned
to a variable
            self.ids.surname_error.opacity = 0  # error message removed
            self.surnameValid = True  # variable which indicates that a valid
value has been inputted by the user
        if self.ids.email.text == "":  # if no data is inputted...
```

```
            self.ids.email_error_blank.opacity = 1  # ...then an error message
is displayed
            self.ids.email_error_invalid.opacity = 0  # invalid email error
message is removed
        else:
            email = self.ids.email.text
            if re.search(".+@.+\..+", email) != None:  # regular expression
used to check that inputted email is in a valid email format
                self.email = self.ids.email.text  # inputted email assigned to
a variable
                self.ids.email_error_blank.opacity = 0  # error message
removed
                self.ids.email_error_invalid.opacity = 0  # invalid email
error message removed
                self.emailValid = True  # variable which indicates that a
valid value has been inputted by the user
            else:
                self.ids.email_error_blank.opacity = 0  # blank data error
message removed
                self.ids.email_error_invalid.opacity = 1  # invalid email
error message displayed
        if self.ids.password.text == "":  # if no data is inputted...
            self.ids.password_error_blank.opacity = 1  # ...then an error
message is displayed
            self.ids.password_error_invalid.opacity = 0  # invalid password
error message is removed
        else:
            password = self.ids.password.text
            if re.search("(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[a-zA-
Z\d@$!%*?&]{8,}", password) != None:
                # Checks that inputted password is at least 8 characters
                # Regular expression used to check that password contains at
least 1 lowercase character,
                # 1 uppercase character, 1 digit and 1 special character
                self.password = self.ids.password.text  # inputted password
assigned to a variable
                self.hashedPassword = (hashlib.new("sha3_256",
self.password.encode())).hexdigest()
                # Creates a hash of the user's password so that it is stored
secureley on the database, as it is sensitive data

                self.ids.password_error_blank.opacity = 0  # error message
removed
                self.ids.password_error_invalid.opacity = 0  # invalid
password error message is removed
                self.passwordValid = True  # variable which indicates that a
valid value has been inputted by the user
                if self.firstName_valid and self.surnameValid and
self.emailValid and self.passwordValid:  # checks if all the data values have
been inputted and are valid
                    self.createAccountID()  # method called to create a unique
accountID for the user
            else:
                self.ids.password_error_blank.opacity = 0  # blank data error
message removed
                self.ids.password_error_invalid.opacity = 1  # invalid
password error message displayed

    def createAccountID(self):
        # creates a unique accountID for the user
        data_accountID = {"field": "accountID"}
        self.accountID = requests.post(serverBaseURL + "/create_ID",
```

```
data_accountID).text
        self.updateUsers()

    def updateUsers(self):
        # add user's details to 'users' table in AWS RDS database
        dbData_update = {}  # dictionary which stores the metadata required
for the AWS server to make the required query to the MySQL database
        dbData_update["accountID"] = self.accountID  # adds the variable
'accountID' to the dictionary 'dbData'
        dbData_update["firstName"] = self.firstName  # adds the variable
'firstName' to the dictionary 'dbData'
        dbData_update["surname"] = self.surname  # adds the variable 'surname'
to the dictionary 'dbData'
        dbData_update["email"] = self.email  # adds the variable 'email' to
the dictionary 'dbData'
        dbData_update[
            "password"] = self.hashedPassword  # adds the variable
'hashPassword' to the dictionary 'dbData'
        response = requests.post(serverBaseURL + "/verifyAccount",
                                 dbData_update)  # sends post request to
'verifyAccount' route on AWS server to check whether the email address
inputted is already associated with an account
        self.topHeight = 0.13 # snackbar properties
        self.sleepTime = 5
        if response.text == "exists":  # if the inputted email address is
already associated with an account
            self.ids.snackbar.text = "Account with this email address already
exists. Login instead"  # creates specific text for the generic Label which is
used as a snackbar in a varity of scenarios in the app
            self.ids.snackbar.font_size = 24
            self.openSnackbar() # calls the method which creates the snackbar
animation
            thread_dismissSnackbar = Thread(target=self.dismissSnackbar,
args=(),
                                            daemon=False)  # initialises
an instance of the 'threading.Thread()' method
            thread_dismissSnackbar.start()  # starts the thread which will run
in pseudo-parallel to the rest of the program
        else:
            response = requests.post(serverBaseURL + "/updateUsers",
                                     dbData_update)  # sends post request to
'updateUsers' route on AWS server with user's inputted data to be stored in
the database
            if response.text == "error":  # if an error occurs when adding the
user's data into the 'users' table
                self.ids.snackbar.text = "Error creating account. Please try
again later"  # creates specific text for the generic Label which is used as a
snackbar in a varity of scenarios in the app
                self.ids.snackbar.font_size = 30
                self.openSnackbar() # calls the method which creates the
snackbar animation
                thread_dismissSnackbar = Thread(target=self.dismissSnackbar,
args=(),
                                                daemon=False)  #
initialises an instance of the 'threading.Thread()' method
                thread_dismissSnackbar.start()  # starts the thread which will
run in pseudo-parallel to the rest of the program
            else:
                self.statusUpdate() # get latest value of 'paired'
                self.jsonStore.put("localData", initialUse=self.initialUse,
                                   loggedIn=True, accountID=self.accountID,
                                   paired=self.paired)  # updates json object to
```

```
reflect that user has successfully created an account
                self.statusUpdate()  # update launch variables


                createThread_ring(self.accountID, self.filepath) # connect to
MQTT broker to receive messages when visitor presses doorbell as now logged in
and have unique accountID

                self.manager.transition = NoTransition()  # creates a cut
transition type

                if self.initialUse == True:
                    self.manager.current = "MessageResponses_add"  # switches
to 'MessageResponses_add' GUI
                    self.manager.current.__init__()
                else:
                    self.manager.current = "Homepage"  # switches to
'Homepage' GUI
                    self.manager.current.__init__()



class SignIn(Launch):
    # 'SignIn' class allows users to log into their account

    def signIn(self):
        # called when user taps the Sign In button
        self.emailValid = False  # variable which indicates that a valid value
has been inputted by the user
        self.passwordValid = False  # variable which indicates that a valid
value has been inputted by the user
        if self.ids.email.text == "":  # if no data is inputted...
            self.ids.email_error_blank.opacity = 1  # ...then an error message
is displayed
            self.ids.email_error_invalid.opacity = 0  # invalid email error
message is removed
        else:
            self.email = self.ids.email.text  # inputted email assigned to a
variable
            self.ids.email_error_blank.opacity = 0  # error message removed
            self.emailValid = True
        if self.ids.password.text == "":  # if no data is inputted...
            self.ids.password_error_blank.opacity = 1  # ...then an error
message is displayed
            self.ids.password_error_invalid.opacity = 0  # invalid password
error message is removed
        else:
            self.password = self.ids.password.text  # inputted password
assigned to a variable
            self.hashedPassword =
(hashlib.new("sha3_256",self.password.encode())).hexdigest()
            # Creates a hash of the user's password so that it can be compared
to the hashed version stored on the database
            self.ids.password_error_invalid.opacity = 0  # error message
removed
            self.passwordValid = True  # variable which indicates that a valid
value has been inputted by the user
            if self.emailValid and self.passwordValid:  # checks if all the
data values have been inputted and are valid
                self.verifyUser()

    def verifyUser(self):
```

```
        # verifies the email and password entered by the user
        dbData_verify = {}  # dictionary which stores the metadata required
for the AWS server to make the required query to the MySQL database
        dbData_verify["email"] = self.email  # adds the variable 'email' to
the dictionary 'dbData'
        dbData_verify["password"] = self.hashedPassword  # adds the variable
'password' to the dictionary 'dbData'
        response = (requests.post(serverBaseURL + "/verifyUser",
dbData_verify)).json()[
            'result']  # sends a post request to the 'verifyUser' route of the
AWS server to validate the details (email and password) entered by the user
        if response == "none":  # if the details inputted by the user don't
match an existing account
            self.topHeight =0.1
            self.sleepTime = 3.5
            self.openSnackbar() # calls the method which creates the snackbar
animation
            thread_dismissSnackbar = Thread(target=self.dismissSnackbar,
args=(),
                                            daemon=False)  # initialises
an instance of the 'threading.Thread()' method
            thread_dismissSnackbar.start()  # starts the thread which will run
in pseudo-parallel to the rest of the program
        else:
            self.accountID = response  # if the user inputs details which
match an account stored in the MySQL database, their unique accountID is
returned
            dbData_accountID = {'accountID': self.accountID}
            response = (requests.post(serverBaseURL + "/getPairing",
dbData_accountID).json())[
                'result']  # sends post request to 'verifyAccount' route on
AWS server to check whether the email address inputted is already associated
with an account
            if response == 'none':  # if there is no doorbell pairing for this
account
                self.paired = self.jsonStore.get("localData")["paired"]
                self.jsonStore.put("localData", initialUse=self.initialUse,
loggedIn=True, accountID=self.accountID,
                                   paired=self.paired)  # updates json object to
reflect that user has successfully signed in
            else:
                doorbellID = response
                self.jsonStore.put("localData", initialUse=self.initialUse,
loggedIn=True, accountID=self.accountID,
                                   paired=doorbellID)  # updates json object to
reflect that user has successfully signed in
            self.statusUpdate()  # update launch variables

            createThread_ring(self.accountID, self.filepath) # connect to MQTT
broker to receive messages when visitor presses doorbell as now logged in

            self.manager.transition = NoTransition()  # creates a cut
transition type

            self.manager.current = "Homepage"  # switches to 'Homepage' GUI
            self.manager.current.__init__()


class MessageResponses_add(Launch):
    # 'MessageResponses_add' class allows user to add audio response messages
to be played through the doorbell.
```

```
    def __init__(self, **kw):
        # retrieves the data for existing audio messages
        super().__init__(**kw)
        dbData_view = {}  # dictionary which stores the metadata required for
the AWS server to make the required query to the MySQL database
        dbData_view["accountID"] = self.accountID  # adds the variable
'accountID' to the dictionary 'dbData'
        response = (requests.post(serverBaseURL + "/view_audioMessages",
dbData_view)).json()[
            'result']  # sends a post request to the 'view_audioMessages'
route of the AWS server to fetch all the data about all the audio messages
associated with that user
        if response == 'none':
            self.numMessages = 0
            self.messageData = []
        else:
            self.messageData = response
            self.numMessages = self.messageData["length"]
        self.numPages = int(math.ceil(self.numMessages / 3))
        self.currentPage = 0
        self.currentMessage = -3
        self.previewMessages = False
        try:
            self.targetView.stop()  # close the target view
        except:
            pass
        Clock.schedule_once(
            self.finishInitialising)  # Kivy rules are not applied until the
original Widget (Launch) has finished instantiating, so must delay
initialisation

    def finishInitialising(self, dt):
        # applies initialisation processes which require access to Kivy ids
        if self.initialUse == True and self.numMessages == 0: # if the mobile
app is running for the first time and the user has zero audio messages
            # create animation:
            animation = Animation(color=[1, 1, 1, 0], duration=0.1)  # become
invisible
            animation += Animation(color=[1, 1, 1, 0], duration=1)  #
invisible
            animation += Animation(color=[1, 1, 1, 1], duration=0.1)  # become
visible
            animation += Animation(color=[1, 1, 1, 1], duration=1)  # visible
            animation.repeat = True  # animation loops forever
            animation.start(self.ids.plusIcon) # apply animation to image with
id 'plusIcon'
        self.audioMessage_create(1, 3) # calls method to display user's
current audio messages


    def audioMessage_create(self, currentPage, currentMessage):
        # displays the user's current audio messages and allows users to
create new personalised audio messages which can be played through the
doorbell
        if self.numMessages == 0:  # if the user has not yet recorded any
audio messages
            self.addMessage_target()  # calls the method which opens the
target view widget which explains to the user what a personalised audio
message is
            self.ids.plusIcon.pos_hint = {"x": 0.17, "y": 0.5}  # sets the
position for the plus icon
            self.ids.button_audioMessage_1.disabled = False  # activates the
```

```
button to open the target view widget
        else:  # if the user has already added a personalised audio message
            self.jsonStore.put("localData", initialUse=False,
loggedIn=self.loggedIn, accountID=self.accountID,
                               paired=self.paired)
            self.statusUpdate()
            if ((self.currentPage + int(currentPage)) > 0) and
(self.currentPage + int(currentPage)) <= (
            self.numPages):  # if the new page number that is being
transitioned to is positive and less than or equal to the total number of
pages required to display all the user's audio messages
                self.currentPage += int(
                    currentPage)  # updates the value of the variable
'self.currentPage' depending on whether the user has moved forwards or
backwards a page
                if self.currentMessage + int(currentMessage) >= 0:
                    self.currentMessage += int(
                        currentMessage)  # updates the value of the variable
'self.currentMessage' depending on whether the user has moved forwards or
backwards a page
                if self.currentPage < self.numPages:  # if the current page is
not the final page required to display all the user's audio messages, then all
three icons must be filled with the name of an audio message
                    self.ids.audioMessage_name1.text =
self.messageData[str(self.currentMessage)][
                        1]  # name of the audio message in the top left icon
is the first item in the tuple
                    self.ids.audioMessage_name2.text =
self.messageData[str(self.currentMessage + 1)][
                        1]  # name of the audio message in the top right icon
is the first item in the tuple
                    self.ids.audioMessage_name3.text =
self.messageData[str(self.currentMessage + 2)][
                        1]  # name of the audio message in the bottom left
icon is the first item in the tuple
                    self.ids.plusIcon.pos_hint = {"x": 0.66,
                                                  "y": 0.24}  # position of
plus icon image to create a new audio message
                    self.ids.button_audioMessage_1.disabled = False
                    self.ids.button_audioMessage_2.disabled = False
                    self.ids.button_audioMessage_3.disabled = False
                    self.ids.button_plusIcon.disabled = False
                elif self.currentPage == self.numPages:  # if the current page
is the final page required to display all the user's audio messages
                    self.ids.audioMessage_name1.text = ""  # resets the text
value for the top left icon
                    self.ids.audioMessage_name2.text = ""  # resets the text
value for the top right icon
                    self.ids.audioMessage_name3.text = ""  # resets the text
value for the bottom left icon
                    if self.numMessages % 3 == 1:  # modulus used to determine
if there is one audio message on the final page
                        # code below sets up the icons and buttons for the GUI
when the user has already added one audio message on the page:
                        self.ids.audioMessage_name1.text =
self.messageData[str(self.currentMessage)][
                            1]  # name of the audio message in the top left
icon is the first item in the tuple
                        self.ids.plusIcon.pos_hint = {"x": 0.65,
                                                      "y": 0.5}  # position of
plus icon image to create a new audio message
                        self.ids.button_audioMessage_1.disabled = False
```

```
                        self.ids.button_audioMessage_2.disabled = False
                        self.ids.button_audioMessage_3.disabled = True
                        self.ids.button_plusIcon.disabled = True
                    elif self.numMessages % 3 == 2:  # modulus used to
determine if there are two audio messages on the final page
                        # code below sets up the icons and buttons for the GUI
when the user has already added two audio messages on the page:
                        self.ids.audioMessage_name1.text =
self.messageData[str(self.currentMessage)][
                            1]  # name of the audio message in the top left
icon is the first item in the tuple
                        self.ids.audioMessage_name2.text =
self.messageData[str(self.currentMessage + 1)][
                            1]  # name of the audio message in the top right
icon is the first item in the tuple
                        self.ids.plusIcon.pos_hint = {"x": 0.17,
                                                      "y": 0.24}  # position
of plus icon image to create a new audio message
                        self.ids.button_audioMessage_1.disabled = False
                        self.ids.button_audioMessage_2.disabled = False
                        self.ids.button_audioMessage_3.disabled = False
                        self.ids.button_plusIcon.disabled = True
                    elif self.numMessages % 3 == 0:  # modulus used to
determine if there are three audio messages on the final page
                        # code below sets up the icons and buttons for the GUI
when the user has already added three audio messages on the page:
                        self.ids.audioMessage_name1.text =
self.messageData[str(self.currentMessage)][
                            1]  # name of the audio message in the top left
icon is the first item in the tuple
                        self.ids.audioMessage_name2.text =
self.messageData[str(self.currentMessage + 1)][
                            1]  # name of the audio message in the top right
icon is the first item in the tuple
                        self.ids.audioMessage_name3.text =
self.messageData[str(self.currentMessage + 2)][
                            1]  # name of the audio message in the bottom lft
icon is the first item in the tuple
                        self.ids.plusIcon.pos_hint = {"x": 0.66,
                                                      "y": 0.24}  # position
of plus icon image to create a new audio message
                        self.ids.button_audioMessage_1.disabled = False
                        self.ids.button_audioMessage_2.disabled = False
                        self.ids.button_audioMessage_3.disabled = False
                        self.ids.button_plusIcon.disabled = False
        self.ids.plusIcon.opacity = 1  # sets the opacity of the plus icon (to
add more audio messages) to 1 after placing it/them in the correct position on
the screen depending on how many audio messages the user has already added


    def addMessage_target(self):
        # instantiates a target view widget which explains how to utilise
audio messages
        titleSpaces = '             ' # spaces required to align title text
correctly
        descriptionSpaces = '                ' # spaces required to align
description text correctly
        # create target view widget with required properties:
        self.targetView = MDTapTargetView(
            widget=self.ids.button_audioMessage_1,
            title_text="{}Add an audio message".format(titleSpaces),
            description_text="{0}You can create personalised\n{0}audio
```

```
responses which can\n{0}be easily selected in the\n{0}SmartBell app and played by\n{0}"
                                "your SmartBell when a visitor\n{0}comes to the door.".format(descriptionSpaces),
            widget_position="left_top",
            outer_circle_color=(49 / 255, 155 / 255, 254 / 255),
            target_circle_color=(145 / 255, 205 / 255, 241 / 255),
            outer_radius=370,
            title_text_size=33,
            description_text_size=27,
            cancelable=False)

    def openTarget(self):
        # controls the opening of the target view
        if self.targetView.state == "close":  # if the target view is currently closed
            self.targetView.start()  # opens the target view
            self.ids.button_continueIcon.disabled = False  # activates the continue icon button
            animation = Animation(opacity=1, d=0.1)  # automatic animation which gradually increases the opacity of the continue icon image from 0 to 1
            animation.start(self.continueIcon)  # starts the animation of the continue icon image
        else:  # if the target view is currently open
            self.targetView.stop()  # close the target view
            self.ids.button_continueIcon.disabled = True  # deactivates the continue icon button
            animation = Animation(opacity=0, d=0.1)  # automatic animation which gradually decreases the opacity of the continue icon image from 1 to 0
            animation.start(self.continueIcon)  # starts the animation of the continue icon image

    def openMessage(self, buttonNum):
        # opens correct audio message data when user taps on name of an audio message
        if buttonNum == 1:
            self.messageDetails = self.messageData[str(self.currentMessage)]
        elif buttonNum == 2:
            self.messageDetails = self.messageData[str(self.currentMessage + 1)]
        elif buttonNum == 3:
            self.messageDetails = self.messageData[str(self.currentMessage + 2)]
        if self.messageDetails[2] == "Null":
            self.manager.current = "MessageResponses_viewAudio"  # switches to 'MessageResponses_createAudio' GUI
            self.manager.current_screen.__init__()  # initialises the running instance of the 'MessageResponses_createAudio' class
            self.manager.current_screen.messageDetails_init(
                self.messageDetails)  # calls the 'messageDetails_init' method of the running instance of the 'MessageResponses_createAudio' class
        else:
            self.manager.current = "MessageResponses_createText"  # switches to 'MessageResponses_createAudio' GUI
            self.manager.current_screen.__init__()  # initialises the running instance of the 'MessageResponses_createAudio' class
            self.manager.current_screen.messageDetails_init(
                self.messageDetails)  # calls the 'messageDetails_init' method of the running instance of the 'MessageResponses_createAudio' class

    def respondAudio_select(self):
        # called when user selects Respond button when visitor image displayed
```

```
in mobile app
        self.previewMessages = True
        self.ids.previewMessages.opacity = 1  # changes background image to
instruct user to select images

    def respondAudio_preview(self, currentMessage):
        # formats audio message preview displayed when user selects an audio
message to be played through doorbell
        messageNum = (self.currentPage - 1) * 3 + currentMessage - 1  #
calculates message number so that messageID can be retrieved from json object
'self.messageData'
        self.messageID = self.messageData[str(messageNum)][0]
        self.messageName = self.messageData[str(messageNum)][1]
        self.messageText = self.messageData[str(messageNum)][2]
        if len(self.messageText) < 25:
            self.maxLength = len(self.messageText)
        else:
            self.maxLength = 25
        self.previewMessage_dialog()

    def respondAudio_new(self):
        pass


    def cancelRespond_dialog(self):
        # creates dialog box which asks user to confirm whether they would
like to cancel their audio message response after the doorbell has been rung
        self.dialog = MDDialog(
            title="Are you sure you want to cancel your response?",
            auto_dismiss=False,
            type="custom",
            buttons=[MDFlatButton(text="NO", text_color=((128 / 255), (128 /
255), (128 / 255), 1),
                                  on_press=self.dismissDialog),
                     MDRaisedButton(text="YES", md_bg_color=(136 / 255, 122 /
255, 239 / 255, 1),
                                    on_press=self.cancelRespond)])  # creates
the dialog box with the required properties for the user to input the name of
the audio message recorded/typed
        self.dialog.open()  # opens the dialog box


    def cancelRespond(self, instance):
        # called if user indicates that they would like to cancel their audio
message response
        self.dialog.dismiss()
        if self.manager.get_screen('VisitorImage').ids.faceName.text == 'Face
unknown':
            self.updateFaces_dialog()
        self.manager.current = "Homepage"

    def previewMessage_dialog(self):
        # creates dialog box which displays preview of audio message response
before user selects to play message through mobile app
        if self.messageText == "Null":
            text = "[b][i]This is an audio message[/i][/b]" # markup used to
increase accessibility and usability
        else:
            text = "[b]Message preview [/b]\n\n[i]" +
self.messageText[:self.maxLength] + "[/i]"
        self.dialog = MDDialog(
            title="Play message '{}' through your
```

```
SmartBell?".format(self.messageName),
            text=text,
            auto_dismiss=False,
            type="custom",
            buttons=[MDFlatButton(text="NO", text_color=((128 / 255), (128 /
255), (128 / 255), 1),
                                  on_press=self.dismissDialog),
                     MDRaisedButton(text="YES!",
                                    on_press=self.transmitMessage)])  #
creates the dialog box with the required properties for the user to input the
name of the audio message recorded/typed
        self.dialog.open()  # opens the dialog box


    def transmitMessage(self, instance):
        # transmits details about audio message response to Raspberry Pi over
MQTT
        self.dialog.dismiss()
        MQTTPython = autoclass('MQTT') # invoke Objective C class class to
transmit audio message
        mqtt = MQTTPython.alloc().init()
        if self.messageText != "Null":
            mqtt.publishData = str(self.messageText)
            mqtt.publishTopic = f"message/text/{self.accountID}"
        else:
            mqtt.publishData = str(self.messageID)
            mqtt.publishTopic = f"message/audio/{self.accountID}"
        mqtt.publish()
        if self.manager.get_screen('VisitorImage').ids.faceName.text == 'Face
unknown':
            self.updateFaces_dialog()

    def updateFaces_dialog(self):
        # create dialog box to enable user to input name of visitor if they
couldn't be recognised to train facial recognition algorithm
        self.dialog = MDDialog(
            title="Enter visitor's name so SmartBell can identify them next
time:",
            auto_dismiss=False,
            type="custom",
            content_cls=DialogContent(),
            buttons=[MDFlatButton(text="CANCEL", text_color=((128 / 255), (128
/ 255), (128 / 255), 1),
                                  on_press=self.dismissDialog),
                     MDRaisedButton(text="SAVE",
                                    on_press=self.knownFaces_update)])  #
creates the dialog box with the required properties for the user to input the
name of the audio message recorded/typed
        self.dialog.open()  # opens the dialog box


    def knownFaces_update(self, instance):
        # update details of identified faces in SQL database
        global faceID
        self.dialog.dismiss()
        for object in self.dialog.content_cls.children:  # iterates through
the objects of the dialog box where the user inputted the name of the visitor
            if isinstance(object, MDTextField):
                faceName = object.text
        data_knownFaces = {"faceName": faceName, "faceID": faceID}
        requests.post(serverBaseURL + "/update_knownFaces", data_knownFaces)
        self.manager.get_screen('VisitorImage').ids.faceName.text = faceName
```

```
class MessageResponses_create(Launch):
    # 'MessageResponses_create' class allows the user to select whether they
would like to record their audio message using their voice as the input or
type their audio message using the on-screen keyboard as their input. The GUI
is created in the kv file.
    pass


class MessageResponses_createAudio(Launch):
    # 'MessageResponses_createAudio' class allows users to record their
personalised audio message which can be played through their SmartBell
doorbell

    def __init__(self, **kw):
        # loads the images/gifs required to create the GUI for the user to
record their audio message
        super().__init__(**kw)
        self.initialRecording = True
        self.recordAudio_static = "SmartBell_audioRecord_static.png"  # loads
the static mic image file
        self.recordAudio_listening = "SmartBell_audioRecord_listening.zip"  #
loads the zip file used to create the second part of the gif displayed when
the user is recording their audio message
        self.recordAudio_loading = "SmartBell_audioRecord_loading.zip"  #
loads the zip file used to create the third part of the gif displayed when the
user is recording their audio message
        self.recordAudio_end = "SmartBell_audioRecord_end.zip"  # loads the
zip file used to create the final part of the gif displayed when the user is
recording their audio message
        Clock.schedule_once(
            self.finishInitialising) # Kivy rules are not applied until the
original Widget (Launch) has finished instantiating, so must delay
initialisation

    def finishInitialising(self, dt):
        # applies launch processes which require access to Kivy ids
        self.ids.recordAudio.source = self.recordAudio_static  # sets the
source of the image with id 'recordAudio' to the static microphone image
        self.ids.button_recordAudio.disabled = False  # activates the button
to record an audio message

    def rerecordAudio(self, messageDetails):
        # called when user selects Record after previewing their audio message
recording
        self.messageDetails = messageDetails
        self.initialRecording = False

    def startRecording(self):
        # begins the process of recording the user's audio message
        self.jsonStore.put("localData", initialUse=False,
loggedIn=self.loggedIn, accountID=self.accountID,
                           paired=self.paired)
        self.statusUpdate()
        self.recordAudio = RecordAudio()  # instantiates the method which is
used to control the recording of the user's audio message
        recordAudio_thread = Thread(target=self.recordAudio.start, args=(),
                                    daemon=False)  # initialises the
instance of thread which is used to record the user's audio input
        self.ids.recordAudio.anim_loop = 0  # sets the number of loops of the
```

```
gif to be played (which uses the zio file
'SmartBell_audioRecord_listening.zip') to infinite as the length of the user's
audio message is indeterminate
        self.ids.recordAudio.size_hint = 4, 4
        self.ids.recordAudio.source = self.recordAudio_listening  # changes
the source of the image with the id 'recordAudio'
        self.startTime = time.time()  # sets up timer to record how long
button to record audio is held for
        recordAudio_thread.start()  # starts the thread instance called
'self.recordAudio_thread'

    def stopRecording(self):
        # terminates the recording of the user's audio message
        self.ids.recordAudio.size_hint = 2, 2
        self.ids.recordAudio.source = self.recordAudio_static  # changes the
source of the image with the id 'recordAudio'
        endTime = time.time()  # stops the timer which records how long button
to record audio is held for
        if (endTime - self.startTime) <= 1:  # if this audio recording is too
short, an error message will be returned
            self.recordAudio.falseStop()  # calls the method which clears the
data stored from the recording as the recording was too short and therefore is
invalid
            self.ids.snackbar.font_size = 30
            self.ids.snackbar.text = "Press and hold the microphone to speak"
# creates specific text for the generic Label which is used as a snackbar in a
varity of scenarios in the app
            self.topHeight =0.13
            self.sleepTime = 3.5
            self.openSnackbar() # calls the method which opens the snackbar to
indicate that the audio message recorded was too short
            dismissSnackbar_thread = Thread(target=self.dismissSnackbar,
args=(),
                                            daemon=False)  # initialises
an instance of the thread which closes the snackbar after 3.5 seconds.
            dismissSnackbar_thread.start()  # starts the thread
'self.dismissSnackbar_thread'
        else:  # if the button to record audio is held for more than one
second
            self.ids.button_recordAudio.disabled = True  # disables the button
to record an audio message
            audioData = self.recordAudio.stop()  # calls the method which
terminates the recording of the user's voice input and saves the audio data
            messagePath = join(self.filepath, "audioMessage_tmp.pkl")
            with open(messagePath,
                      "wb") as file:  # create pkl file with name equal to the
messageID in write bytes mode
                pickle.dump(audioData,
                            file)  # 'pickle' module serializes the data
stored in the Python list object 'audioData' into a byte stream which is
stored in pkl file
                file.close()  # closes the file
            self.manager.current = "MessageResponses_viewAudio"  # switches to
'MessageResponses_viewAudio' GUI
            self.manager.current_screen.__init__()  # initialises the running
instance of the 'MessageResponses_createAudio' class
            if self.initialRecording == False:
                self.manager.current_screen.messageDetails_init(
                    self.messageDetails)  # calls the 'messageDetails_init'
method of the running instance of the 'MessageResponses_createAudio' class

    def helpAudio(self):
```

```
        # instructs the user how to record an audio message if they press the
'Help' button
        self.ids.snackbar.font_size = 30
        self.ids.snackbar.text = "Press and hold the microphone to speak"  #
creates specific text for the generic Label which is used as a snackbar in a
varity of scenarios in the app
        self.topHeight =0.13
        self.sleepTime = 3.5
        self.openSnackbar() # calls the method which opens the snackbar to
indicate that the audio message recorded was too short
        dismissSnackbar_thread = Thread(target=self.dismissSnackbar, args=(),
                                        daemon=False)  # initialises an
instance of the thread which closes the snackbar after 3.5 seconds.
        dismissSnackbar_thread.start()  # starts the thread
'self.dismissSnackbar_thread'


class RecordAudio():
    # 'RecordAudio' class allows user to create personalised audio messages
using voice input
    def __init__(self, **kw):
        # initialises constant properties for the class
        super().__init__(**kw)
        self.bufferRate = 60  # variables which stores the number of audio
buffers recorded per second
        self.audioData = []  # creates a list to store the audio bytes
recorded
        self.mic = get_input(callback=self.micCallback, rate=8000,
source='default',
                             buffersize=2048)  # initialises the class
'get_input' from the module 'audiostream' with the properties required to
ensure the audio is recorded correctly

    def micCallback(self, buffer):
        # called by the class 'get_input' to store recorded audio data (each
buffer of audio samples)
        self.audioData.append(buffer)  # appends each buffer (chunk of audio
data) to variable 'self.audioData'

    def start(self):
        # begins the process of recording the audio data
        self.mic.start()  # starts the method 'self.mic' recording audio data
        Clock.schedule_interval(self.readChunk,
                                1 / self.bufferRate)  # calls the method
'self.readChunk' to read and store each audio buffer (2048 samples) 60 times
per second

    def readChunk(self, bufferRate):
        # coordinates the reading and storing of the bytes from each buffer of
audio data (which is a chunk of 2048 samples)
        self.mic.poll()  # calls 'get_input(callback=self.mic_callback,
source='mic', buffersize=2048)' to read the byte content. This byte content is
then dispatched to the callback method 'self.micCallback'


    def falseStop(self):
        # terminates the audio recording when the duration of audio recording
is less than 1 second
        self.audioData = []  # clears the list storing the audio data
        Clock.unschedule(self.readChunk)  # un-schedules the Clock's execution
of 'self.readChunk'
        self.mic.stop()  # stops recording audio
```

```
    def stop(self):
        # terminates and saves the audio recording when the recording has been
successful
        Clock.unschedule(self.readChunk)  # un-schedules the calling
'self.readChunk'
        self.mic.stop()  # stops recording audio
        return self.audioData


class MessageResponses_view(Launch):
    # 'MessageResponses_view' Class displays the user's names of the user's
recorded audio messages

    def __init__(self, **kw):
        self.statusUpdate()
        Screen.__init__(self, **kw) # only initialise the screen as no need to
initialise Launch again as this takes user to homepage
        self.audioRename = False

    def messageDetails_init(self, messageDetails):
        # called when changing the content of an audio message (re-recording
or re-typing)
        self.messageDetails = messageDetails
        self.messageID = self.messageDetails[0]
        self.messageName = self.messageDetails[1]
        self.messageText = self.messageDetails[2]
        self.initialRecording = False
        self.initialTyping = False
        # if the message is a text message
        if self.messageType == "Text":
            self.ids.messageText.text = self.messageText  # set the text in
the text box to the old value of the text
        elif os.path.isfile(join(self.filepath, (
                self.messageID + ".wav"))):  # added to remove old version of
audio recording with same messageID (audio file name) stored on app, when user
re-records the audio message
            os.remove(join(self.filepath, (self.messageID + ".wav")))

    def nameMessage_dialog(self):
        # allows the user to input the name of the audio message which they
recorded/typed
        if (self.initialRecording == False and self.messageType == "Voice") or
(
                self.initialTyping == False and self.messageType == "Text"):
            title = "Enter a new name for this audio message or press 'cancel'
to keep it named '{}':".format(
                self.messageName)
        else:
            title = "What would you like to name your audio message?"
        self.dialog = MDDialog(
            title=title,
            auto_dismiss=False,
            type="custom",
            content_cls=DialogContent(),  # content class
            buttons=[MDFlatButton(text="CANCEL", text_color=((128 / 255), (128
/ 255), (128 / 255), 1),
                                  on_press=self.dismissDialog),
                     MDRaisedButton(text="SAVE", md_bg_color=(136 / 255, 122 /
255, 239 / 255, 1),
                                    on_press=self.nameMessage)])  # creates
the dialog box with the required properties for the user to input the name of
```

```
the audio message recorded/typed
        self.dialog.open()  # opens the dialog box

    def dismissDialog(self, instance):
        # alters dismissDialog method in inherited Class - polymorphism
        super().dismissDialog(instance)
        if (self.initialRecording == False and self.messageType == "Voice") or
(
                self.initialTyping == False and self.messageType == "Text"):
            self.audioMessages_update()

    def nameMessage(self, instance):
        # called when the button 'Save' is tapped on the dialog box which
allows the user to input the name of the audio message they recorded/typed
        for object in self.dialog.content_cls.children:  # iterates through
the objects of the dialog box where the user inputted the name of the audio
message they recorded/typed
            if isinstance(object, MDTextField):  # if the object is an
MDTextField
                dbData_name = {}  # dictionary which stores the metadata
required for the AWS server to make the required query to the MySQL database
                self.messageName = object.text
                dbData_name[
                    "messageName"] = self.messageName  # adds the variable
'object.text' to the dictionary 'dbData_create'
                dbData_name[
                    "accountID"] = self.accountID  # adds the variable
'accountID' to the dictionary 'dbData_create'
                response = requests.post(serverBaseURL +
"/verify_messageName",
                                         dbData_name)  # sends post request to
'verify_messageName' route on AWS server to check whether the message name
that the user has inputted has already been assigned to one of their audio
messages
                if (object.text == "" or object.text == "Null" or
response.text == "exists" or len(
                        object.text) > 13):  # if the user has not inputted
any text or if the user's input is 'Null' (input used to indicate an audio
message' or if the message name is already in use by that user or if the
length of the message name is greater than 13 characters
                    xCood = int(
                        instance.pos[0])  # assigns current x coordinate of
the 'Save' button to the variable 'xCood'
                    yCood = int(
                        instance.pos[1])  # assigns current y coordinate of
the 'Save' button to the variable 'yCood'
                    animation = Animation(pos=((xCood + 7), yCood),
t="out_elastic",
                                          d=0.02)  # creates an animation
object which moves the 'Save' button to the right
                    animation += Animation(pos=((xCood - 7), yCood),
t="out_elastic",
                                           d=0.02)  # adds a sequential step
to the animation object which moves the 'Save' button to the left
                    animation += Animation(pos=(xCood, yCood),
t="out_elastic",
                                           d=0.02)  # adds a sequential step
to the animation object which moves the 'Save' button to its original position
                    animation.start(instance)  # starts the animation instance
                else:  # if the user has inputted a name which is not already
in use by that user and is 13 or less characters in length
                    if (self.initialRecording and self.messageType == "Voice")
```

```
or (
                                self.initialTyping == True and self.messageType ==
"Text"):  # if this audio message has just been recorded
                            self.messageID = self.createMessageID()  # calls the
method which creates a unique messageID for the audio message which the user
has created
                        if self.audioRename:  # if the user has played the audio
message, then it will have been converted to a wav file already, so this wav
file should be renamed with the newly created messageID
                            oldName = join(self.filepath, "audioMessage_tmp.wav")
                            newName = join(self.filepath, (self.messageID +
".wav"))
                            os.rename(oldName, newName)
                        self.dialog.dismiss()  # closes the dialog box
                        self.messageName = object.text  # assigns the text of the
MDTextField to the variable 'self.messageName' (the text is the name of the
audio message as inputted by the user)
                        self.audioMessages_update()  # calls the method to update
the MySQL table 'audioMessages'

    def createMessageID(self):
        # creates a unique messageID for the audio message created by the user
        while True:  # creates an infinite loop
            chars = string.ascii_uppercase + string.ascii_lowercase +
string.digits  # creates a concatenated string of all the uppercase and
lowercase alphabetic characters and all the digits (0-9)
            messageID = ''.join(random.choice(chars) for i in range(16))  #
the 'random' module randomly selects 16 characters from the string 'chars' to
form the unique messageID
            dbData_create = {"messageID": messageID}  # adds the variable
'messageID' to the dictionary 'dbData_create'
            response = requests.post(serverBaseURL + "/verify_messageID",
                                     dbData_create)  # sends post request to
'verify_messageID' route on AWS server to check whether the messageID that has
been generated for an audio message does not already exist
            if response.text == "notExists":  # if the messageID does not yet
exist
                break  # breaks the infinite loop
            else:
                pass
        return messageID  # returns the unique message ID generated for this
audio message

    def deleteMessage(self):
        # deletes user's audio message
        dbData_update = {}  # dictionary which stores the metadata required
for the AWS server to make the required query to the MySQL database
        dbData_update[
            "messageID"] = self.messageID  # adds the variable 'messageID' to
the dictionary 'dbData_update'
        response = requests.post(serverBaseURL + "/delete_audioMessages",
                                 dbData_update)  # sends post request to
'update_audioMessages' route on AWS server to insert the data about the audio
message which the user has created into the MySQL table 'audioMessages'


class MessageResponses_viewAudio(MessageResponses_view):
    # 'MessageResponses_viewAudio' Class displays an audio message which the
user has already recorded
    def __init__(self, **kw):
        super().__init__(**kw)
        self.messagePath = join(self.filepath, "audioMessage_tmp.pkl") #
```

```
filepath to store .pkl file of audio bytes for audio message which is not yet
saved by user
        self.playbackAudio_gif = "SmartBell_playbackAudio.zip"  # loads the
zip file used to create the audio playback gif
        self.playbackAudio_static = "SmartBell_playbackAudio.png"  # loads the
image file of the audio playback static image
        self.initialRecording = True
        self.initialTyping = None
        self.messageID = ""
        self.messageType = "Voice"

    def audioMessages_update(self):
        # updates the MySQL table to store the data about the audio message
created by the user
        dbData_update = {'messageID': self.messageID, 'messageName':
self.messageName, 'messageText': self.messageText, 'accountID':
self.accountID, 'initialCreation': str(self.initialRecording)}  # json object
which stores the metadata required for the AWS server to update the MySQL
database
        response = requests.post(serverBaseURL + "/update_audioMessages",
                                 dbData_update)  # sends post request to
'update_audioMessages' route on AWS server to insert the data about the audio
message which the user has created into the MySQL table 'audioMessages'
        try:  # wav file to be uploaded to AWS only exists if a new audio
message has been recorded - else if the audio message name has just been
changed, the statement will break
            self.uploadAWS()  # calls the method to upload the audio message
data to AWS S3
        except:
            pass
        self.manager.transition = NoTransition()  # creates a cut transition
type
        self.manager.current = "MessageResponses_add"  # switches to
'MessageResponses_add' GUI
        self.manager.current_screen.__init__()  # creates a new instance of
the 'MessageResponses_add' class

    def uploadAWS(self):
        # sends the data for the audio message recorded by the user as a pkl
file to the AWS elastic beanstalk environment, where it is uploaded to AWS s3
using 'boto3' module
        uploadData = {"bucketName": "nea-audio-messages",
                      "s3File": self.messageID}  # creates the dictionary
which stores the metadata required to upload the personalised audio message to
AWS S3 using the 'boto3' module on the AWS elastic beanstalk environment
        # if an audio message with the same messageID already exists, it will
be overwritten
        file = {"file": open(self.messagePath,
                             "rb")}  # opens the file to be sent using Flask's
'request' method (which contains the byte stream of audio data) and stores the
file in a dictionary
        requests.post(serverBaseURL + "/uploadS3", files=file,
                                 data=uploadData)  # sends post request to
'uploadS3' route on AWS server to upload the pkl file storing the data about
the audio message to AWS s3 using 'boto3'
        # This is done because the 'boto3' module cannot be installed on
mobile phones so the process of uploading the pkl file to AWS s3 using boto3
must be done remotely on the AWS elastic beanstalk environment
        os.remove(self.messagePath)

    def audioMessage_play(self):
        # allows user to playback the audio message which they have recorded
```

```
        if os.path.isfile(join(self.filepath, (self.messageID + ".wav"))): #
wav on app
            fileName = join(self.filepath, self.messageID)
        else:
            if os.path.isfile(self.messagePath): # pkl on app
                with open(self.messagePath, "rb") as file:
                    audioData = pickle.load(file)
                    file.close()  # closes the file
                    self.audioRename = True
                    fileName = join(self.filepath, "audioMessage_tmp")
            else: # pkl on AWS
                fileName = join(self.filepath, self.messageID)
                downloadData = {"bucketName": "nea-audio-messages",
                                "s3File": self.messageID}  # creates the
dictionary which stores the metadata required to download the pkl file of the
personalised audio message from AWS S3 using the 'boto3' module on the AWS
elastic beanstalk environment
                response = requests.post(serverBaseURL + "/downloadS3",
downloadData)
                audioData = pickle.loads(response.content)  # unpickles the
byte string
            messageFile = wave.open(fileName + ".wav", "wb") # load .wav file
in write bytes mode
            messageFile.setnchannels(1)  # audiostream module records in
single audio channel
            messageFile.setsampwidth(2) # bytes per audio sample
            messageFile.setframerate(8000) # samples recorded per second
            messageFile.writeframes(b''.join(audioData)) # join together each
element in the bytes list 'audioData' (each element is an audio buffer)
            messageFile.close()
        messageFile_voice = SoundLoader.load(fileName + ".wav")
        messageFile_voice.play()
        self.ids.playbackAudio.source = self.playbackAudio_gif
        self.audioLength = messageFile_voice.length
        thread_stopGif = Thread(target=self.stopGif, args=(),
                                daemon=False)  # initialises an instance
of the 'threading.Thread()' method
        thread_stopGif.start()  # starts the thread which will run in pseudo-
parallel to the rest of the program

    def stopGif(self):
        # ends the looping of the playback gif
        time.sleep(self.audioLength)
        self.ids.playbackAudio.source = self.playbackAudio_static

    def tmpAudio_delete(self):
        # deletes the temporary audio file
        if os.path.isfile(self.messagePath):
            os.remove(self.messagePath)


class MessageResponses_createText(MessageResponses_view):
    # 'MessageResponses_createText' Class displays a typed audio message which
the user has created
    def __init__(self, **kw):
        super().__init__(**kw)
        self.initialTyping = True
        self.initialRecording = None
        self.messageType = "Text"

    def saveMessage(self):
        # creates dialog box to specify name for typed audio message
```

```
        if (len(list((self.ids.messageText.text).strip())) > 80 or
len(list((self.ids.messageText.text).strip())) == 0):
            self.ids.snackbar.font_size = 30
            self.ids.snackbar.text = "Sorry, the text you have entered is
invalid!\nPlease make sure your message is between\n1 and 80 characters."  #
creates specific text for the generic Label which is used as a snackbar in a
varity of scenarios in the app
            self.topHeight =0.13
            self.sleepTime = 3.5
            self.openSnackbar() # calls the method which opens the snackbar to
indicate that the audio message recorded was too short
            dismissSnackbar_thread = Thread(target=self.dismissSnackbar,
args=(),
                                            daemon=False)  # initialises
an instance of the thread which closes the snackbar after 3.5 seconds.
            dismissSnackbar_thread.start()  # starts the thread
'self.dismissSnackbar_thread'
        else:
            self.nameMessage_dialog()

    def audioMessages_update(self):
        # updates the MySQL table to store the data about the audio message
created by the user
        messageText = self.ids.messageText.text
        #self.statusUpdate() # get latest value for accountID
        dbData_update = {}  # dictionary which stores the metadata required
for the AWS server to make the required query to the MySQL database
        dbData_update[
            "messageID"] = self.messageID  # adds the variable 'messageID' to
the dictionary 'dbData_update'
        dbData_update[
            "messageName"] = self.messageName  # adds the variable
'messageName' to the dictionary 'dbData_update'
        dbData_update[
            "messageText"] = messageText  # adds the 'null' variable
'fileText' to the dictionary 'dbData_update'
        dbData_update[
            "accountID"] = self.accountID  # adds the variable 'accountID' to
the dictionary 'dbData_update'
        dbData_update["initialCreation"] = str(
            self.initialTyping)  # adds the variable 'initialRecording' to the
dictionary 'dbData_update'
        response = requests.post(serverBaseURL + "/update_audioMessages",
                                 dbData_update)  # sends post request to
'update_audioMessages' route on AWS server to insert the data about the audio
message which the user has created into the MySQL table 'audioMessages'
        self.jsonStore.put("localData", initialUse=False,
loggedIn=self.loggedIn, accountID=self.accountID,
                           paired=self.paired)
        self.statusUpdate()  # update launch variables
        self.manager.transition = NoTransition()  # creates a cut transition
type
        self.manager.current = "MessageResponses_add"  # switches to
'MessageResponses_add' GUI
        self.manager.current_screen.__init__()  # creates a new instance of
the 'MessageResponses_add' class


class VisitorLog(Launch):
    # 'VisitorLog' class displays the details of the visits to the user's
doorbell
```

```
    def __init__(self, **kw):
        super().__init__(**kw)
        thread = Thread(target=self.schedule)
        thread.start()

    def schedule(self):
        Clock.schedule_once(self.visitorLog)

    def visitorLog(self, dt):
        # creates visitor log
        self.statusUpdate()
        self.get_averageRate()
        self.get_averageTime()
        dbData_accountID = {"accountID": self.accountID}
        self.visitors = requests.post(serverBaseURL + "/get_visitorLog",
dbData_accountID).json() # retrieve visitor log details associated with user's
account
        for index, visitDetails in enumerate(self.visitors): # iterate through
visit details in visitor log
            self.epoch = float(visitDetails[0][6:])
            self.date = time.strftime('%d-%m-%Y, %H:%M:%S',
time.gmtime(self.epoch)) # convert epoch time in seconds into actual time/date
            self.faceID = visitDetails[1]
            self.visitID = visitDetails[2]
            self.visitorImage_path = join(self.filepath, self.visitID +
'.png')
            self.get_visitorImage() # download visitor image for visit
'visitID'
            if self.faceID != 'NO_FACE': # if face identified for visit
                dbData_faceID = {"faceID": self.faceID}
                result = requests.post(serverBaseURL + "/get_faceName",
dbData_faceID).json()
                faceName = result[0]
                self.visitors[index][1] = faceName # replace face ID with face
name
            else: # if no face identified
                self.visitors[index][1] = 'No face identified'
            self.visitors[index][2] = self.visitorImage_path # store path to
visitor image associated with visit ID
            self.visitors[index] += (self.date,) # append the date to the
tuple 'visitors'
        self.visitorsFormatted = list(map(lambda a:(a[0][6:], a[1], a[2],
a[3]), self.visitors)) # create array storing required values
        self.displayLog('date')

    def displayLog(self, dateORname):
        # create scrolling list to display visitor log
        self.visitorsSorted = self.mergeSort(self.visitorsFormatted,
dateORname) # sort list storing visit details by visitor name or by date of
visit
        self.ids.container.clear_widgets() # clear existing visitor log scroll
list
        for visit in self.visitorsSorted:
            rowWidget = TwoLineAvatarListItem(text=f"Name: {visit[1]}",
secondary_text = f"Date: {visit[3]}") # create row widget with visitor name
and date
            rowWidget.add_widget(ImageLeftWidget(source= visit[2])) # add
associated visit image to row widget
            self.ids.container.add_widget(rowWidget) # add row widget to
scroll list

    def get_visitorImage(self):
```

```
        # download visitor images from AWS to display in visitor log
        downloadData = {"bucketName": "nea-visitor-log",
                        "s3File": self.visitID}  # creates the dictionary
which stores the metadata required to download the png file of the visitor
image from AWS S3 (via the server REST API)
        response = requests.post(serverBaseURL + "/downloadS3",
                                 downloadData)  # request sent to custom REST
API, which uses 'boto3' module to attempt to download the visitor image with
name 'visitID' from AWS S3
        responseMessage = response.content  # bytes content of message
returned by REST API
        time.sleep(0.5)  # time delay to reduce number of requests to AWS API,
reducing running costs
        visitorImage_data = responseMessage  # stores visitor image bytes data
        f = open(self.visitorImage_path,'wb')  # opens file to store image
bytes (opens in 'wb' format to enable bytes to be written to this file)
        f.write(visitorImage_data)  # writes visitor image bytes data to the
file
        f.close()
        time.sleep(0.5)


    def mergeSort(self, array, dateORname):
        # sorts visitor log by criteria specified by user
        if dateORname == 'date':
            index = 1 # specifies index of value in 'array' to be used to sort
array
            self.date = True
        else:
            index = 0 # specifies index of value in 'array' to be used to sort
array
            self.date = False
        if len(array) > 1: # if array has length greater than 1, continue
splitting it
            mid = len(array) // 2 # find middle index of array
            left = array[:mid] # store left half of array elements
            right = array[mid:] # store right half of array elements

            self.mergeSort(left, dateORname) # recursively call the mergesort
function on the left of array elements to sort left half
            self.mergeSort(right, dateORname) # recursively call the mergesort
function on the right of array elements to sort right half

            # merge and sort left and right array
            i = 0 # left array index
            j = 0 # right array index
            k = 0 # main array index
            while i < len(left) and j < len(right):
                # sort array by comparing and swapping values in tuple at
index 'index'
                if left[i][index] < right[j][index]: # value in left array
less than value in right array
                    array[k] = left[i] # save value in left index into merged
array
                    i += 1 # move to next index in left array
                else: # value in left array greater than or equal to value in
right array
                    array[k] = right[j]
                    j += 1 # move to next index in right array
                k += 1 # move to next index in merged array

            # move all remaining values in left array into merged array, as no
```

```
        # more values in right array to compare with
            while i < len(left):
                array[k] = left[i]
                i += 1
                k += 1

            # move all remaining values in right array into merged array, as
no more values in left array to compare with
            while j < len(right):
                array[k] = right[j]
                j += 1
                k += 1
        return array


    def get_averageRate(self):
        # get average number of visits per day
        dbData_accountID = {"accountID": self.accountID}
        response = requests.post(serverBaseURL + "/get_averageRate",
dbData_accountID)
        self.averageRate = str(round(response.json()['result'],1))
        text = f"Average number of visits per day: {str(self.averageRate)}
visits"
        self.ids.averageRate.text = text

    def get_averageTime(self):
        # get average time when doorbell is rung
        dbData_accountID = {"accountID": self.accountID}
        response = requests.post(serverBaseURL + "/get_averageTime",
dbData_accountID)
        self.averageTime = round(response.json()['result'],2)
        hours = int(self.averageTime) # total number of complete hours
        minutes = (self.averageTime * 60) % 60 # total number of minutes
remaining from complete hours
        self.averageTime = str("%02d:%02d" % (hours, minutes)) # zero padding
and two decimal places for each number
        text = f"Average time of visit (24hr): {self.averageTime}"
        self.ids.averageTime.text = text


class RingAlert(Launch):
    # 'RingAlert' Class displays a screen with a pulsating doorbell image when
the doorbell is rung
    pass


class VisitorImage(Launch):
    # 'VisitorImage' Class displays the image of the visitor when the doorbell
is rung

    def viewImage(self):
        # image of visitor and associated name (if identified) displayed in
mobile app
        global faceID
        self.statusUpdate()
        dbData_accountID = {"accountID": self.accountID}
        response = requests.post(serverBaseURL + "/latest_visitorLog",
dbData_accountID).json()['result']
        if response == 'none':
            self.manager.get_screen(
                'Homepage').ids.snackbar.text = 'No images captured by
SmartBell on your account'
```

```
            self.manager.current = 'Homepage'
            MDApp.get_running_app().manager.get_screen('Homepage').topHeight =
0.13
            MDApp.get_running_app().manager.get_screen('Homepage').sleepTime =
3
            self.manager.get_screen('Homepage').openSnackbar()  # calls the
method which creates the snackbar animation
            thread_dismissSnackbar =
Thread(target=self.manager.get_screen('Homepage').dismissSnackbar, args=(),
                                               daemon=False)  # initialises an
instance of the 'threading.Thread()' method
            thread_dismissSnackbar.start()  # starts the thread which will run
in pseudo-parallel to the rest of the program
        else:
            visitID = response[0]
            visitorImage_path = join(self.filepath, 'visitorImage.png')
            thread_visitorImage = Thread(target=visitorImage_thread, args=(
            visitID,visitorImage_path))  # thread created so image is
downloaded in the background, and so does not delay loading of screen
            thread_visitorImage.setDaemon(True)
            thread_visitorImage.start()
            data_visitID = {"visitID": visitID}
            response = requests.post(serverBaseURL + "/view_visitorLog",
data_visitID).json()
            faceID = response[1]
            if faceID != "NO_FACE":  # faceID is set to 'NO_FACE' when a face
cannot be detected in the image taken by the doorbell
                data_faceID = {"faceID": str(faceID)}
                faceName = None
                while faceName == None:  # loop until faceID record has been
added to db by Raspberry Pi (ensures no error arises in case of latency
between RPi inserting vistID data to db and mobile app retrieving this data
here)
                    faceName = requests.post(serverBaseURL + "/get_faceName",
data_faceID).json()[0]
                if faceName == "":
                    faceName = "Face unknown"
            else:
                faceName = "No face identified"
            self.ids.faceName.text = faceName

    def cancelRespond_dialog(self):
        # creates dialog box which asks user to confirm whether they want to
cancel their audio message response after the doorbell has been rung
        self.dialog = MDDialog(
            title="Are you sure you want to cancel your response?",
            auto_dismiss=False,
            type="custom",
            buttons=[MDFlatButton(text="NO", text_color=((128 / 255), (128 /
255), (128 / 255), 1),
                                  on_press=self.dismissDialog),
                     MDRaisedButton(text="YES", md_bg_color=(136 / 255, 122 /
255, 239 / 255, 1),
                                    on_press=self.cancelRespond)])  # creates
the dialog box with the required properties for the user to input the name of
the audio message recorded/typed
        self.dialog.open()  # opens the dialog box

    def cancelRespond(self, instance):
        # called if user indicates that they would like to cancel their audio
message response
        self.dialog.dismiss()
```

```
        if self.ids.faceName.text == 'Face unknown':
            self.updateFaces_dialog()
        self.manager.current = "Homepage"

    def updateFaces_dialog(self):
        # creates dialog box which gives user option to enter name of visitor
if they weren't identified by the facial recognition algorithm
        self.dialog = MDDialog(
            title="Enter visitor's name so SmartBell can identify them next
time:",
            auto_dismiss=False,
            type="custom",
            content_cls=DialogContent(),
            buttons=[MDFlatButton(text="CANCEL", text_color=((128 / 255), (128
/ 255), (128 / 255), 1),
                                  on_press=self.dismissDialog),
                     MDRaisedButton(text="SAVE",
                                    on_press=self.knownFaces_update)])  #
creates the dialog box with the required properties for the user to input the
name of the audio message recorded/typed
        self.dialog.open()  # opens the dialog box

    def knownFaces_update(self, instance):
        # store details about visitor's face in SQL database
        global faceID
        self.dialog.dismiss()
        for object in self.dialog.content_cls.children:  # iterates through
the objects of the dialog box where the user inputted the name of the visitor
            if isinstance(object, MDTextField):
                faceName = object.text
        data_knownFaces = {"faceName": faceName, "faceID": faceID}
        requests.post(serverBaseURL + "/update_knownFaces", data_knownFaces)
        self.ids.faceName.text = faceName


class DialogContent(BoxLayout):
    # 'DialogContent' Class contains the standard structure for a Kivy Dialog
box
    pass


class MyApp(MDApp):
    # 'MyApp' class is used to create app GUI by building 'layout.kv' file
    def build(self, **kw):
        layout = Builder.load_file("layout.kv")  # loads the 'layout.kv' file
        return layout


def visitorImage_thread(visitID, visitorImage_path):
    # downloads the image of the visitor when the doorbell is rung and
displays it in the mobile app
    try:
        for visitorImage in
MDApp.get_running_app().manager.get_screen('VisitorImage').ids.visitorImage.ch
ildren:
            visitorImage.opacity = 0
    except:
        pass
    MDApp.get_running_app().manager.get_screen(
        'VisitorImage').ids.loading.opacity = 1  # reset opacity of image
loading gif
    MDApp.get_running_app().manager.get_screen(
```

```
        'VisitorImage').ids.faceName.text = "Loading..."  # reset text of
visitor image name label

    downloadData = {"bucketName": "nea-visitor-log",
                    "s3File": visitID}  # creates the dictionary which stores
the metadata required to download the png file of the visitor image from AWS
S3 (via the server REST API)
    responseMessage = b'error'  # message returned by REST API if the  visitor
image uploaded by the Raspberry Pi is not yet available on AWS S3
    while responseMessage == b'error':  # while loop continue looping if
message returned by REST API is b'error', as visitor image uploaded by the
Raspberry Pi is not yet available on AWS S3
        response = requests.post(serverBaseURL + "/downloadS3",
                                 downloadData)  # request sent to custom REST
API, which uses 'boto3' module to attempt to download the visitor image with
name 'visitID' from AWS S3
        responseMessage = response.content  # bytes content of message
returned by REST API
        time.sleep(0.5)  # time delay to reduce number of requests to AWS API,
reducing running costs
    visitorImage_data = responseMessage  # stores visitor image bytes data
    f = open(visitorImage_path,
             'wb')  # opens file to store image bytes (opens in 'wb' format to
enable bytes to be written to this file)
    f.write(visitorImage_data)  # writes visitor image bytes data to the file
    f.close()
    time.sleep(0.5)
    visitorImage = AsyncImage(source=visitorImage_path,
                              pos_hint={"center_x": 0.5,
                                        "center_y": 0.53})  # AsyncImage loads
image as background thread, so doesn't hold up running of program if there is
a delay in loading the image
    visitorImage.reload()

MDApp.get_running_app().manager.get_screen('VisitorImage').ids.visitorImage.ad
d_widget(
        visitorImage)  # accesses screen ids of 'VisitorImage' screen and adds
the visitor image as a widget to a nested Kivy float layout
MDApp.get_running_app().manager.get_screen('VisitorImage').ids.loading.opacity
= 0 # set opacity of image loading gif to zero as image is loaded and
displayed

MDApp.get_running_app().manager.get_screen('VisitorImage').ids.loading.opacity
= 0 # set opacity of image loading gif to zero as image is loaded and
displayed


def createThread_ring(accountID, filepath):
    # interfaces with Objective C class 'MQTT' to create client which will
receive message when doorbell rung
    MQTTPython = autoclass(
        'MQTT') # autoclass used to load Objective-C class 'MQTT' and create a
Python wrapper around it
    mqtt = MQTTPython.alloc().init() # instance of the Objective-C 'MQTT'
class created
    mqtt.ringTopic = f"ring/{accountID}" # the Objective-C property
'ringTopic' is assigned
    mqtt.connect()  # call Objective-C method which connects to the MQTT
broker
    visitorImage_path = join(filepath, 'visitorImage.png') # path to store
visitor image on mobile app
    thread_ring = Thread(target=ringThread,args=(mqtt, visitorImage_path)) #
```

```
create thread which checks status of Objective-C property
'messageReceived_ring'
    thread_ring.start() # start the thread
    return


def createThread_visit(visitID):
    # creates thread which downloads visitor image from AWS S3 storage
    thread_visit = Thread(target=visitThread, args=(visitID,))
    thread_visit.start()


def ringThread(mqtt, visitorImage_path):
    # checks status of Objective C class attributes to verify whether MQTT
message indicating that doorbell has been rung is received
    while True:
        if mqtt.messageReceived_ring == 1: # if message received on topic
'ring/accountID' by Objective-C MQTT session instance (i.e. SmartBell doorbell
rung)
            try:  # runs successfully if visitor image already exists on app
                for visitorImage in
MDApp.get_running_app().manager.get_screen(
                        'VisitorImage').ids.visitorImage.children:
                    # 'visitorImage' is a nested Kivy float layout, whose
children are images of the visitor
                    visitorImage.opacity = 0  # set the opacity of each
existing vistor image to 0, so that the previous visitor image is not shown
            except:  # if visitor image doesn't already exist on app
                pass
            mqtt.messageReceived_ring = 0  # value of 'messageReceived_ring'
must be reset to 0 so that new messages can be detected in Python code
            mqtt.vibratePhone()  # calls Objective-C method to vibrate mobile
spacespacespMDApp.get_running_app().manager.get_screen('VisitorImage').ids.loa
ding.opacity = 1 # reset opacity of image loading gif
spacespacespMDApp.get_running_app().manager.get_screen('VisitorImage').ids.fac
eName.text = "Loading..." # reset text of visitor image name label
            MDApp.get_running_app().manager.get_screen(
                'VisitorImage').ids.loading.opacity = 1  # reset opacity of
image loading gif
            MDApp.get_running_app().manager.get_screen(
                'VisitorImage').ids.faceName.text = "Loading..."  # reset text
of visitor image name label
            MDApp.get_running_app().manager.current = "RingAlert"  # open Kivy
screen to notify user that the doorbell has been rung
            visitID = str(
                mqtt.messageData.UTF8String())  # decode message published to
topic 'ring/accountID' by Raspberry Pi doorbell
            createThread_visit(visitID) # visit thread only called once
doorbell is rung to reduce energy usage.
            downloadData = {"bucketName": "nea-visitor-log",
                            "s3File": visitID}  # creates the dictionary which
stores the metadata required to download the png file of the visitor image
from AWS S3 (via the server REST API)
            responseMessage = b'error'  # message returned by REST API if the
visitor image uploaded by the Raspberry Pi is not yet available on AWS S3
            while responseMessage == b'error':  # while loop continue looping
if message returned by REST API is b'error', as visitor image uploaded by the
Raspberry Pi is not yet available on AWS S3
                response = requests.post(serverBaseURL + "/downloadS3",
                                         downloadData)  # request sent to
custom REST API, which uses 'boto3' module to attempt to download the visitor
image with name 'visitID' from AWS S3
```

```
                responseMessage = response.content  # bytes content of message
returned by REST API
                time.sleep(0.5)  # time delay to reduce number of requests to
AWS API, reducing running costs
            visitorImage_data = responseMessage  # stores visitor image bytes
data
            f = open(visitorImage_path,
                     'wb')  # opens file to store image bytes (opens in 'wb'
format to enable bytes to be written to this file)
            f.write(visitorImage_data)  # writes visitor image bytes data to
the file
            f.close()
            time.sleep(0.5)
            MDApp.get_running_app().manager.current = "VisitorImage"
            visitorImage = AsyncImage(source=visitorImage_path,
                                      pos_hint={"center_x": 0.5,
                                                "center_y": 0.53}) #
AsyncImage loads image as background thread, so doesn't hold up running of
program if there is a delay in loading the image
            visitorImage.reload()
            mqtt.notifyPhone()  # calls Objective-C method to play
notification sound through mobile phone

MDApp.get_running_app().manager.get_screen('VisitorImage').ids.visitorImage.ad
d_widget(
                visitorImage)  # accesses screen ids of 'VisitorImage' screen
and adds the visitor image as a widget to a nested Kivy float layout
MDApp.get_running_app().manager.get_screen('VisitorImage').ids.loading.opacity
= 0 # set opacity of image loading gif to zero as image is loaded and
displayed
            MDApp.get_running_app().manager.get_screen(
                'VisitorImage').ids.loading.opacity = 0  # set opacity of
image loading gif to zero as image is loaded and displayed
        else:
            time.sleep(3)  # reduce energy usage of mobile as checking status
of 'messageReceived_ring' less frequently


def visitThread(visitID):
    # download visitor image from latest ring of doorbell and display in
mobile app
    global faceID
    while True:
        data_visitID = {"visitID": visitID}
        response = None
        while response == None:  # loop until visitID record has been added to
db by Raspberry Pi (ensures no error arises in case of latency between RPi
inserting vistID data to db and mobile app retrieving this data here)
            response = requests.post(serverBaseURL + "/view_visitorLog",
data_visitID).json()
            time.sleep(1)
        faceID = response[1]
        if faceID != "NO_FACE":  # faceID is set to 'NO_FACE' when a face
cannot be detected in the image taken by the doorbell
            data_faceID = {"faceID": str(faceID)}
            faceName = None
            while faceName == None:  # loop until faceID record has been added
to db by Raspberry Pi (ensures no error arises in case of latency between RPi
inserting vistID data to db and mobile app retrieving this data here)
                faceName = requests.post(serverBaseURL + "/get_faceName",
data_faceID).json()[0]
            if faceName == "":
```

```
                faceName = "Face unknown"
        else:
            faceName = "No face identified"

MDApp.get_running_app().manager.get_screen('VisitorImage').ids.faceName.text =
faceName
        break


def pairThread(accountID, id, pairing, jsonStore):
    # checks pairing status between mobile app and doorbell
    start_time = time.time()
    dbData_id = {'id': id}
    MDApp.get_running_app().manager.get_screen('Homepage').topHeight = 0.13
    MDApp.get_running_app().manager.get_screen('Homepage').sleepTime = 5
    while True:
        response = (requests.post(serverBaseURL + "/verifyPairing",
dbData_id).json())[
            'result']  # sends post request to 'verifyAccount' route on AWS
server to check whether the email address inputted is already associated with
an account
        if response == accountID and pairing == True:
            MDApp.get_running_app().manager.get_screen(
                'Homepage').ids.snackbar.text = f"Successfully paired with
SmartBell '{id}'!"

MDApp.get_running_app().manager.get_screen('Homepage').openSnackbar()
            loggedIn = jsonStore.get("localData")["loggedIn"]
            accountID = jsonStore.get("localData")["accountID"]
            jsonStore.put("localData", initialUse=False, loggedIn=loggedIn,
accountID=accountID, paired=id)
            break
        elif response == '' and pairing == False:
            MDApp.get_running_app().manager.get_screen(
                'Homepage').ids.snackbar.text = f"Successfully unpaired from
SmartBell '{id}'"

MDApp.get_running_app().manager.get_screen('Homepage').openSnackbar()
            break
        elif response != accountID and response != None and response != '' and
pairing == True:
            MDApp.get_running_app().manager.get_screen(
                'Homepage').ids.snackbar.text = 'Error pairing SmartBell.
Please ensure you\ninput the correct name for your SmartBell'

MDApp.get_running_app().manager.get_screen('Homepage').openSnackbar()
            break
        elif time.time() - start_time > 60:
            MDApp.get_running_app().manager.get_screen(
                'Homepage').ids.snackbar.text = 'Error pairing SmartBell.
Please ensure you\ninput the correct name for your SmartBell'

MDApp.get_running_app().manager.get_screen('Homepage').openSnackbar()
            break
        time.sleep(1)
    thread_dismissSnackbar =
Thread(target=MDApp.get_running_app().manager.get_screen('Homepage').dismissSn
ackbar, args=(),
                                   daemon=False)  # initialises an instance
of the 'threading.Thread()' method
    thread_dismissSnackbar.start()  # starts the thread which will run in
pseudo-parallel to the rest of the program
```

```
if __name__ == "__main__":  # when the program is launched, if the name of the file is the main program (i.e. it is not a module being imported by another file) then this selection statement is True
    MyApp().run()  # the run method is inherited from the 'MDApp' class which is inherited by the class 'MyApp'
```

mqtt.m

```
#import "mqtt.h"
#import <AudioToolbox/AudioToolbox.h>

/** defines the methods for the 'MQTT' class
 */
@implementation MQTT
- (id) init {

    return self;
}

- (void) connect {

        MQTTCFSocketTransport*transport = [[MQTTCFSocketTransport alloc] init];
        transport.host = @"hairdresser.cloudmqtt.com";
        transport.port = 18973;

        MQTTSession *session = [[MQTTSession alloc] init];
        session.delegate = self;
        session.transport = transport;
        session.userName = @"yrczhohs";
        session.password = @"qPSwbxPDQHEI";
        session.protocolLevel = MQTTProtocolVersion31;
        self.messageReceived_ring = 0;

        [session connectAndWaitTimeout:30];

        [session subscribeToTopic:self.ringTopic atLevel:2];
}

- (void)newMessage:(MQTTSession *)session data:(NSData *)data onTopic:(NSString
*)topic qos:(MQTTQosLevel)qos retained:(BOOL)retained mid:(unsigned int)mid {

        NSString* dataStr = [[NSString alloc] initWithData:data
                                        encoding:NSUTF8StringEncoding];

    if (![dataStr isEqualToString:@""] && [topic isEqualToString:self.ringTopic] ){
            self.messageData = dataStr;
            self.messageReceived_ring = 1;
    }
}
```

```
- (void) publish {

    MQTTCFSocketTransport*transport = [[MQTTCFSocketTransport alloc] init];
    transport.host = @"hairdresser.cloudmqtt.com";
    transport.port = 18973;

    MQTTSession *session = [[MQTTSession alloc] init];
    session.transport = transport;
    session.userName = @"yrczhohs";
    session.password = @"qPSwbxPDQHEI";
    session.protocolLevel = MQTTProtocolVersion31;
    [session setDelegate:self];
    [session connectAndWaitTimeout:30];

    NSData* data = [self.publishData dataUsingEncoding:NSUTF8StringEncoding];
    [session publishData:data
                 onTopic:self.publishTopic
                  retain:NO
                     qos:1];
    [session disconnect ];
}

- (void)vibratePhone; {

    AudioServicesPlayAlertSound(4095);
}

- (void)notifyPhone; {

     AudioServicesPlayAlertSound(1022);

}

@end
```

mqtt.h

```
/** MQTTClient header file must be imported to access associated protocols required to implement MQTT
 */
#import "MQTTClient.h"


/** 'MQTT' is name of custom class which conforms to protocol 'MQTTSessionDelegate' (a protocol is used to declare methods and properties which are independent of a class). Therefore, the 'MQTT' class implements the methods associated with the 'MQTTSessionDelegate' protocol.
 */
@interface MQTT: NSObject <MQTTSessionDelegate>


/** The following properties are properties of the 'MQTT' class
 */
@property (nonatomic) int messageReceived_ring;
@property (nonatomic, assign) NSString *messageData;
@property (nonatomic, assign) NSString *ringTopic;
@property (nonatomic, assign) NSString *publishData;
@property (nonatomic, assign) NSString *publishTopic;

@end
```

layout.kv

```
#: import NoTransition kivy.uix.screenmanager.NoTransition
WindowManager: #initiates and orders all screens/classes
    Launch:
    SignUp:
    SignIn:
    Homepage:
    MessageResponses_add:
    MessageResponses_create:
    MessageResponses_createAudio:
    MessageResponses_viewAudio:
    MessageResponses_createText:
    RingAlert:
    VisitorImage:
    VisitorLog:



<MDTextField>
#creates default settings for MDTextField objects
    mode: "line" #default mode for text field is a rectangle
    size_hint_x: 0.5 #default width of text field is half the width of the
screen
    color_mode: "custom"
    use_bubble: False
    line_color_focus: 1,1,1,1 #rgba color of rectangle outline when the user
begins to input text into the text field (white)
    line_color_normal: 128/255,128/255,128/255,1

<Button>
#creates default settings for buttons
    background_color: 1,1,1,0 #sets all buttons default background colour to
transparent


<SignUp>:
    name: "SignUp" #defines the screen name used to switch between the screens

    FloatLayout: #class used in Kivy to organise objects in the GUI without
any constraints

        Image:
            id: signUp #unique identifier for this specific image
            source: "SmartBell_signUp.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        MDTextField:
            id: firstName #unique identifier for this specific textfield
            hint_text: "First Name" #hint text indicates what data the user
should input into this text field
            pos_hint: {"center_x": 0.5, "center_y": 0.67} #position of text
field relative to screen
        MDLabel:
            id: firstName_error #unique identifier for this specific label
            text: "First Name is required   " #notifies user that input is
required if they have not inputted any data
```

```
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.75, "center_y": 0.64} #position of label
relative to screen
            opacity: 0 #by default the opacity value is 0 so that the label
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1

        MDTextField:
            id: surname #unique identifier for this specific textfield
            hint_text: "Surname" #hint text indicates what data the user
should input into this text field
            pos_hint: {"center_x": 0.5, "center_y": 0.57} #position of text
field relative to screen
        MDLabel:
            id: surname_error #unique identifier for this specific label
            text: "Surname is required" #notifies user that input is required
if they have not inputted any data
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.75, "center_y": 0.54} #position of label
relative to screen
            opacity: 0 #by default the opacity value is 0 so that the label
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1

        MDTextField:
            id: email #unique identifier for this specific textfield
            hint_text: "Email" #hint text indicates what data the user should
input into this text field
            pos_hint: {"center_x": 0.5, "center_y": 0.47} #position of text
field relative to screen
        MDLabel:
            id: email_error_blank #unique identifier for this specific label
            text: "Email is required" #notifies user that input is required if
they have not inputted any data
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.75, "center_y": 0.44} #position of label
relative to screen
            opacity: 0 #by default the opacity value is 0 so that the label
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1
        MDLabel:
            id: email_error_invalid #unique identifier for this specific label
            text: "Email format is invalid" #notifies user that inputted email
format is invalid
```

```
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.75, "center_y": 0.44} #position of label
relative to screen
            opacity: 0 #by default the opacity value is 0 so that the label
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1

        MDTextField:
            id: password #unique identifier for this specific textfield
            hint_text: "Password" #hint text indicates what data the user
should input into this text field
            pos_hint: {"center_x": 0.5, "center_y": 0.37} #position of text
field relative to screen
            password: True #inputted text is replaced by bullet points to
conceal the password as it is sensitive information
            icon_right: "eye-off" #by default, the inputted text is replaced
by bullet points so the eye icon status default to 'eye-off'

        MDLabel:
            id: password_error_blank #unique identifier for this specific
label
            text: "Password is required" #notifies user that input is required
if they have not inputted any data
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.75, "center_y": 0.34} #position of label
relative to screen
            opacity: 0 #by default the opacity value is 0 so that the label
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1
        MDLabel:
            id: password_error_invalid #unique identifier for this specific
label
            text: "Password must be at least 8 characters and contain at least
1 lowercase character, 1 uppercase character, 1 digit and 1 special character"
#notifies user that input is required if they have not inputted any data
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.53, "center_y": 0.3} #position of label
relative to screen
            size_hint_x: 0.56 #ensures the text in the label does not run
beyond the corresponding text field
            opacity: 0 #by default the opacity value is 0 so that the label
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1
```

```
        Label:
            id: snackbar
            canvas.before:
                Color:
                    rgba: (136/255,122/255,239/255,1)
                Rectangle:
                    pos: self.pos
                    size: self.size
            text_color: (1,1,1,1)
            valign: "middle"
            halign: 'center'
            size_hint: 1, 0.1
            pos_hint: {"center_x": 0.5, "top":0}


        Button:
            id: passwordIcon_button #unique identifier for this specific
button
            size_hint: 0.09, 0.04
            pos_hint: {"center_x": 0.7, "center_y": 0.375}
            on_press:
                password.icon_right = "eye" if password.icon_right == "eye-
off" else "eye-off" #if the icon was previously "eye-off", then it is changed
to "eye". If the icon was previously "eye", then it is changed to "eye-off"
                password.password = False if password.password is True else
True #the property 'password' of the text field 'password' is inverted when
the icon is tapped by the user

        Button:
            id: signUp_button #unique identifier for this specific button
            size_hint: 0.5, 0.11 #size of button relative to screen
            pos_hint: {"center_x": 0.7, "center_y": 0.19} #position of button
relative to screen
            on_press: root.createAccount() #calls Python method
'createAccount'

        Button:
            id: signIn_button #unique identifier for this specific button
            size_hint: 0.2, 0.05 #size of button relative to screen
            pos_hint: {"center_x": 0.67, "center_y": 0.04} #position of button
relative to screen
            on_press: root.manager.current = "SignIn"  # switches to 'SignIn'
screen


<SignIn>:

    name: "SignIn" #defines the screen name used to switch between the screens

    FloatLayout: #class used in Kivy to organise objects in the GUI without
any constraints

        Image:
            id: signIn #unique identifier for this specific image
            source: "SmartBell_signIn.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        MDTextField:
```

```
            id: email #unique identifier for this specific textfield
            hint_text: "Email" #hint text indicates what data the user should
input into this text field
            pos_hint: {"center_x": 0.5, "center_y": 0.47} #position of text
field relative to screen
        MDLabel:
            id: email_error_blank #unique identifier for this specific label
            text: "Email is required" #notifies user that input is required if
they have not inputted any data
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.75, "center_y": 0.44} #position of label
relative to screen
            opacity: 0 #by default the opacity value is 0 so that the label
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1
        MDLabel:
            id: email_error_invalid #unique identifier for this specific label
            text: "Email format is invalid" #notifies user that inputted email
format is invalid
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.75, "center_y": 0.44} #position of label
relative to screen
            opacity: 0 #by default the opacity value is 0 so that the label
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1

        MDTextField:
            id: password #unique identifier for this specific textfield
            hint_text: "Password" #hint text indicates what data the user
should input into this text field
            pos_hint: {"center_x": 0.5, "center_y": 0.37} #position of text
field relative to screen
            password: True #inputted text is replaced by a bullet point icon
to conceal the password as it is sensitive information
            icon_right: "eye-off"
        MDLabel:
            id: password_error_blank #unique identifier for this specific
label
            text: "Password is required" #notifies user that input is required
if they have not inputted any data
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.75, "center_y": 0.34} #position of label
relative to screen
            opacity: 0 #by default the opacity value is 0 so that the label
```

```
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1
        MDLabel:
            id: password_error_invalid #unique identifier for this specific
label
            text: "Password must be at least 8 characters and contain at least
1 lowercase character, 1 uppercase character, 1 digit and 1 special character"
#notifies user that input is required if they have not inputted any data
            theme_text_color: "Custom"
            font_style: "Caption" #font style for error message
            text_color: 1,0,0,1 #red in rgba color values, which is
appropriate for an error message
            halign: "left" #horizontally align the text on the left of the
label so that the start of the text can easily be aligned with the edge of the
corresponding text field
            pos_hint: {"center_x": 0.53, "center_y": 0.3} #position of label
relative to screen
            size_hint_x: 0.56 #ensures the text in the label does not run
beyond the corresponding text field
            opacity: 0 #by default the opacity value is 0 so that the label
and error message cannot be seen, but when an error is made by the user the
opacity value is changed to 1

        Label:
            id: snackbar
            canvas.before:
                Color:
                    rgba: (136/255,122/255,239/255,1)
                Rectangle:
                    pos: self.pos
                    size: self.size
            text: "Incorrect username or password"
            text_color: (1,1,1,1)
            font_size: 36
            valign: "middle"
            halign: 'center'
            size_hint: 1, 0.1
            pos_hint: {"center_x": 0.5, "top":0}

        Button:
            id: passwordIcon_button #unique identifier for this specific
button
            size_hint: 0.09, 0.04
            pos_hint: {"center_x": 0.7, "center_y": 0.375}
            on_press:
                password.icon_right = "eye" if password.icon_right == "eye-
off" else "eye-off" #if the icon was previously "eye-off", then it is changed
to "eye". If the icon was previously "eye", then it is changed to "eye-off"
                password.password = False if password.password is True else
True #the property 'password' of the text field 'password' is inverted when
the icon is tapped by the user

        Button:
            id: signIn_button #unique identifier for this specific button
            size_hint: 0.5, 0.11 #size of button relative to screen
            pos_hint: {"center_x": 0.7, "center_y": 0.19} #position of button
relative to screen
            on_press: root.signIn() #calls Python method 'signIn'

        Button:
            id: signUp_button #unique identifier for this specific button
            size_hint: 0.2, 0.05 #size of button relative to screen
```

```
            pos_hint: {"center_x": 0.67, "center_y": 0.04} #position of button
relative to screen
            on_press: root.manager.current = "SignUp" # switches to 'SignUp'
screen

<Homepage>:

    name: "Homepage"

    FloatLayout:

        Image:
            id: Homepage #unique identifier for this specific image
            source: "SmartBell_homepage.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen


        Button:
            id: button_addMessage
            size_hint: 0.23,0.12
            pos_hint: {"x": 0.62, "y": 0.52}
            on_press:
                root.manager.current = "MessageResponses_add"
                root.manager.current_screen.__init__()

        Button:
            id: button_latestImage
            size_hint: 0.23,0.12
            pos_hint: {"x": 0.15, "y": 0.52}
            on_press:
                root.manager.current = "VisitorImage"
                root.manager.current_screen.viewImage()

        Button:
            id: button_visitorLog
            size_hint: 0.23,0.12
            pos_hint: {"x": 0.15, "y": 0.25}
            on_press:
                root.manager.current = "VisitorLog"
                root.manager.current_screen.visitorLog('')

        Button:
            id: button_pair
            size_hint: 0.23,0.12
            pos_hint: {"x": 0.62, "y": 0.25}
            on_press:
                root.manager.current_screen.pairSelect()

        Button:
            id: button_account #unique identifier for this specific button
            pos_hint: {"center_x": 0.85, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press: root.account() #calls Python method 'account'

        Label:
            id: snackbar #unique identifier for this specific label
            canvas.before:
            #draws GUI - the 'before' aspect of the instruction ensures that
```

```
these graphics are the first aspect of the label to be created and so are
behind the text of the label
                Color:
                    rgba: (136/255,122/255,239/255,1)
                Rectangle:
                    pos: self.pos
                    size: self.size
            text_color: (1,1,1,1)
            font_size: 30
            valign: "middle" #align the text vertically in the middle of the
label
            halign: "center" #align the text horizontally in the centre of the
label
            size_hint: 1, 0.13
            pos_hint: {"center_x": 0.5, "top":0}



<MessageResponses_add>:

    name: "MessageResponses_add" #defines the screen name used to switch
between the screens
    continueIcon: continueIcon #maps Python variable onto kivy ID

    FloatLayout:

        Image:
            id: addMessages #unique identifier for this specific image
            source: "SmartBell_addMessages.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        Image:
            id: previewMessages #unique identifier for this specific image
            source: "SmartBell_previewMessages.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen
            opacity: 0


        Button:
            id: button_continueIcon #unique identifier for this specific
button
            disabled: True #button is disabled so cannot be tapped
            size_hint: 0.23,0.1
            pos_hint: {"x":0.44,"y": 0.13}
            on_press: root.manager.current = "MessageResponses_create" #
switches to 'MessageResponses_create' screen

        Button:
            id: button_homepage #unique identifier for this specific button
            pos_hint: {"center_x": 0.52, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press:
                if root.previewMessages == False: root.manager.current =
"Homepage"
```

```
                elif root.previewMessages == True: root.cancelRespond_dialog()

        Button:
            id: button_forward #unique identifier for this specific button
            pos_hint: {"center_x": 0.85, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press: root.audioMessage_create(1, 3) # calls parent method
'audioMessage_create' with the arguments (1,3) to indicate that the forward
button has been pressed

        Button:
            id: button_backward #unique identifier for this specific button
            pos_hint: {"center_x": 0.15, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press: root.audioMessage_create(-1,-3) # calls parent method
'audioMessage_create' with the arguments (-1,-3) to indicate that the
backwards button has been pressed

        Button:
            id: button_audioMessage_1
            size_hint: 0.23,0.12
            pos_hint: {"x": 0.14, "y": 0.52}
            disabled: True
            on_press:
                if root.previewMessages == True and audioMessage_name1.text !=
"": root.respondAudio_preview(1)
                elif root.previewMessages == True and audioMessage_name1.text
== "":  root.respondAudio_new()
                elif root.previewMessages == False and audioMessage_name1.text
!= "": root.openMessage(1)
                elif root.previewMessages == False and audioMessage_name1.text
== "":  root.openTarget() #opens the target view

        Button:
            id: button_audioMessage_2
            size_hint: 0.23,0.12
            pos_hint: {"x": 0.62, "y": 0.52}
            disabled: True
            on_press:
                if root.previewMessages == True and audioMessage_name2.text !=
"": root.respondAudio_preview(2)
                elif root.previewMessages == True and audioMessage_name2.text
== "":  root.respondAudio_new()
                elif root.previewMessages == False and audioMessage_name2.text
!= "": root.openMessage(2)
                elif root.previewMessages == False and audioMessage_name2.text
== "":  root.manager.current = "MessageResponses_create"

        Button:
            id: button_audioMessage_3
            size_hint: 0.23,0.12
            pos_hint: {"x": 0.14, "y": 0.26}
            disabled: True
            on_press:
                if root.previewMessages == True and audioMessage_name3.text !=
"": root.respondAudio_preview(3)
                elif root.previewMessages == True and audioMessage_name3.text
== "":  root.respondAudio_new()
                elif root.previewMessages == False and audioMessage_name3.text
!= "": root.openMessage(3)
                elif root.previewMessages == False and audioMessage_name3.text
== "":  root.manager.current = "MessageResponses_create"
```

```
        Button:
            id: button_plusIcon #unique identifier for this specific button
            size_hint: 0.23,0.12
            pos_hint: {"x": 0.64, "y": 0.26}
            disabled: True #button is disabled so cannot be tapped
            on_press:
                if root.previewMessages == False: root.manager.current =
"MessageResponses_create" # switches to 'MessageResponses_create' screen
                if root.previewMessages == True: root.respondAudio_new()

        Image:
            id: plusIcon #unique identifier for this specific image
            source: "plusIcon.png"
            size_hint: 0.17,0.17
            opacity: 0

        Image:
            id: continueIcon #unique identifier for this specific image
            source: "continueIcon.png"
            size_hint: 0.19,0.19
            pos_hint: {"x":0.44,"y": 0.13}
            opacity: 0


        Label:
            id: audioMessage_name1 #unique identifier for this specific label
            size_hint: 0.2,0.1
            pos_hint: {"x": 0.15, "y": 0.542}
            halign: "center" #align the text horizontally in the centre of the
label
            font_size: 36
            bold: True

        Label:
            id: audioMessage_name2 #unique identifier for this specific label
            size_hint: 0.2,0.1
            pos_hint: {"x": 0.63, "y": 0.542}
            halign: "center" #align the text horizontally in the centre of the
label
            font_size: 36
            bold: True

        Label:
            id: audioMessage_name3 #unique identifier for this specific label
            size_hint: 0.2,0.1
            pos_hint: {"x": 0.15, "y": 0.27}
            halign: "center" #align the text horizontally in the centre of the
label
            font_size: 36
            bold: True



<MessageResponses_create>:

    name: "MessageResponses_create" #defines the screen name used to switch
between the screens

    FloatLayout:
```

```
        Image:
            id: createMessages #unique identifier for this specific image
            source: "SmartBell_createMessages.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        Button:
            id: button_recordMessage #unique identifier for this specific
button
            pos_hint: {"center_x": 0.54, "center_y": 0.535}
            size_hint: 0.43,0.09
            on_press:
                root.manager.current = "MessageResponses_createAudio"
                root.manager.current_screen.__init__()

        Button:
            id: button_recordMessage #unique identifier for this specific
button
            pos_hint: {"center_x": 0.54, "center_y": 0.22}
            size_hint: 0.43,0.09
            on_press:
                root.manager.current = "MessageResponses_createText"
                root.manager.current_screen.__init__()

        Button:
            id: button_homepage #unique identifier for this specific button
            pos_hint: {"center_x": 0.52, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press: root.manager.current = "Homepage"

        Button:
            id: button_backward #unique identifier for this specific button
            pos_hint: {"center_x": 0.15, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press:
                root.manager.current = root.manager.previous()
                root.manager.current_screen.__init__()



<MessageResponses_createAudio>:

    name: "MessageResponses_createAudio" #defines the screen name used to
switch between the screens

    FloatLayout:

        Image:
            id: createMessages_audio #unique identifier for this specific
image
            source: "SmartBell_createMessages_audio.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        Image:
            id: recordAudio #unique identifier for this specific image
```

```
            source: "SmartBell_audioRecord_static.png"
            anim_delay: 0 #sets the delay between each image in the zip file
to 0 so that the gif is played smoothly
            anim_loop: 1 #sets the limit for the number of loops of the gif to
1
            allow_stretch: True
            keep_ratio: True
            size_hint: 2,2
            pos_hint: {"center_x":0.5, "center_y": 0.6}

        Button:
            id: button_recordAudio #unique identifier for this specific image
            size_hint: 0.3,0.25
            pos_hint: {"center_x": 0.5, "center_y": 0.58}
            on_press:
                root.startRecording() #calls the method 'startRecording' from
the parent class in the Python application
            on_release:
                recordAudio.source = root.recordAudio_end #changes the image
source for the ID 'recordAudio'
                root.stopRecording() #calls the method 'stopRecording' from
the parent class in the Python application


        Button:
            id: button_helpAudio #unique identifier for this specific image
            size_hint: 0.45,0.1
            pos_hint: {"center_x": 0.74, "center_y": 0.25}
            on_press:
                root.helpAudio() #calls the method to open a dialog box which
allows the user to input the name of the audio message which they recorded

        Button:
            id: button_cancelAudio #unique identifier for this specific image
            size_hint: 0.45,0.1
            pos_hint: {"center_x": 0.25, "center_y": 0.25}
            on_press:
                root.manager.current = "MessageResponses_add" # switches to
'MessageResponses_add' screen
                root.manager.current_screen.__init__()

        Button:
            id: button_homepage #unique identifier for this specific button
            pos_hint: {"center_x": 0.52, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press: root.manager.current = "Homepage"

        Button:
            id: button_backward #unique identifier for this specific button
            pos_hint: {"center_x": 0.15, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press:
                root.manager.current = root.manager.previous()
                root.manager.current_screen.__init__()

        Label:
            id: snackbar #unique identifier for this specific label
            canvas.before:
            #draws GUI - the 'before' aspect of the instruction ensures that
these graphics are the first aspect of the label to be created and so are
behind the text of the label
                Color:
```

```
                    rgba: (136/255,122/255,239/255,1)
                Rectangle:
                    pos: self.pos
                    size: self.size
            text_color: (1,1,1,1)
            valign: "middle" #align the text vertically in the middle of the
label
            halign: "center" #align the text horizontally in the centre of the
label
            size_hint: 1, 0.13
            pos_hint: {"center_x": 0.5, "top":0}


<MessageResponses_viewAudio>:

    name: "MessageResponses_viewAudio"

    FloatLayout:

        Image:
            id: MessageResponses_viewAudio #unique identifier for this
specific image
            source: "SmartBell_playbackMessages_audio.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        Image:
            id: playbackAudio #unique identifier for this specific image
            source: "SmartBell_playbackAudio.png"
            anim_delay: 0 #sets the delay between each image in the zip file
to 0 so that the gif is played smoothly
            anim_loop: 0 #sets an infinite loop limit for the gif
            pos_hint: {"center_x":0.5, "center_y": 0.6}

        Button:
            id: button_playbackAudio #unique identifier for this specific
image
            size_hint: 0.45,0.1
            pos_hint: {"center_x": 0.74, "center_y": 0.25}
            on_press:
                root.audioMessage_play() #calls the method 'audioMessage_play'
from the parent class in the Python application

        Button:
            id: button_rerecordAudio #unique identifier for this specific
image
            size_hint: 0.45,0.1
            pos_hint: {"center_x": 0.25, "center_y": 0.25}
            on_press:
                root.manager.current = "MessageResponses_createAudio" #
switches to 'MessageResponses_createAudio' screen
                root.manager.current_screen.__init__()
                if root.initialRecording == False:
root.manager.current_screen.rerecordAudio(root.messageDetails)

        Button:
            id: button_saveAudio #unique identifier for this specific image
            size_hint: 0.45,0.1
            pos_hint: {"center_x": 0.5, "center_y": 0.375}
```

```
            on_press:
                root.nameMessage_dialog()  #calls the method
'nameMessage_dialog' from the parent class in the Python application

        Button:
            id: button_backward #unique identifier for this specific button
            pos_hint: {"center_x": 0.15, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press:
                root.tmpAudio_delete()
                if root.initialRecording == False: root.manager.current =
"MessageResponses_add"
                elif root.initialRecording == True: root.manager.current =
root.manager.previous()
                root.manager.current_screen.__init__()

        Button:
            id: button_homepage #unique identifier for this specific button
            pos_hint: {"center_x": 0.52, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press:
                root.tmpAudio_delete()
                root.manager.current = "Homepage"

        Button:
            id: button_deleteMessage #unique identifier for this specific
button
            pos_hint: {"center_x": 0.85, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press:
                root.deleteMessage() #calls the method 'deleteMessage' from
the parent class in the Python application
                root.manager.current = "MessageResponses_add"
                root.manager.current_screen.__init__()



<MessageResponses_createText>:

    name: "MessageResponses_createText"

    FloatLayout:

        Image:
            id: createMessages_text #unique identifier for this specific image
            source: "SmartBell_createMessages_text.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        MDTextField:
            id: messageText #unique identifier for this specific textfield
            hint_text: "Enter text for audio message" #hint text indicates
what data the user should input into this text field
            multiline: True
            size_hint_x: 0.765
            pos_hint: {"center_x": 0.5, "center_y": 0.7} #position of text
field relative to screen
```

```
        Button:
            id: button_saveAudio #unique identifier for this specific image
            size_hint: 0.43,0.1
            pos_hint: {"center_x": 0.74, "y": 0.2}
            on_press:
                root.saveMessage()  #calls the method 'saveMessage' from the
parent class in the Python application

        Button:
            id: button_cancelAudio #unique identifier for this specific image
            size_hint: 0.43,0.1
            pos_hint: {"center_x": 0.26, "y": 0.2}
            on_press:
                root.manager.current = "MessageResponses_add" # switches to
'MessageResponses_add' screen
                root.manager.current_screen.__init__()

        Button:
            id: button_backward #unique identifier for this specific button
            pos_hint: {"center_x": 0.15, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press:
                root.manager.current = "MessageResponses_add"
                root.manager.current_screen.__init__()

        Button:
            id: button_homepage #unique identifier for this specific button
            pos_hint: {"center_x": 0.52, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press: root.manager.current = "Homepage"

        Button:
            id: button_deleteMessage #unique identifier for this specific
button
            pos_hint: {"center_x": 0.85, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press:
                root.deleteMessage() #calls the method 'deleteMessage' from
the parent class in the Python application
                root.manager.current = "MessageResponses_add"
                root.manager.current_screen.__init__()

        Label:
            id: snackbar #unique identifier for this specific label
            canvas.before:
            #draws GUI - the 'before' aspect of the instruction ensures that
these graphics are the first aspect of the label to be created and so are
behind the text of the label
                Color:
                    rgba: (136/255,122/255,239/255,1)
                Rectangle:
                    pos: self.pos
                    size: self.size
            text_color: (1,1,1,1)
            valign: "middle" #align the text vertically in the middle of the
label
            halign: "center" #align the text horizontally in the centre of the
label
            size_hint: 1, 0.13
            pos_hint: {"center_x": 0.5, "top":0}
```

```
<RingAlert>:

    name: "RingAlert"

    FloatLayout:
        Image:
            id: ringAlert_background
            source: "SmartBell_ringAlert_background.zip"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen
            anim_delay: 0.3
            anim_loop: 0

        Image:
            id: ringAlert_background
            source: "SmartBell_ringAlert_button.zip"
            pos_hint: {"center_x": 0.502, "center_y": 0.202}
            size_hint: 0.77,0.77
            anim_delay: 0.04
            anim_loop: 0


<VisitorImage>:

    name: "VisitorImage"

    FloatLayout:
        Image:
            id: background #unique identifier for this specific image
            source: "SmartBell_visitorImage.png"
            size: self.size #sets image to same size as screen
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        FloatLayout:
            id: visitorImage

        Image:
            id: border
            source: "SmartBell_visitorImage-border.png"
            size: self.size
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        Image:
            id: label
            source: "SmartBell_visitorImage-label.png"
            size: self.size
            allow_stretch: True #allows image to be stretched so that it fills
the screen
            keep_ratio: False #allows image proportions to be altered so that
it fills the screen

        Image:
```

```
            id: loading
            source: "SmartBell_loading.zip"
            anim_delay: 0.075
            anim_loop: 0
            size_hint: 2, 2
            pos_hint: {"center_x":0.5,"center_y": 0.5} #position of image
relative to parent screen

        Label:
            id: faceName
            text: "Loading..."
            text_size: self.size # dimensions for text
            font_size: 28 # font size for text
            halign: 'center' # alignment of text
            valign: 'middle'
            pos_hint: {"center_x": 0.83, "center_y": 0.6}
            size_hint: 0.21,0.1
            canvas.before:
                PushMatrix
                Rotate:
                    angle: -70
                    origin: self.center
            canvas.after:
                PopMatrix

        Button:
            id: button_homepage #unique identifier for this specific button
            pos_hint: {"center_x": 0.52, "center_y": 0.07}
            size_hint: 0.2,0.1
            on_press: root.cancelRespond_dialog()

        Button:
            id: button_respond #unique identifier for this specific button
            pos_hint: {"center_x": 0.52, "center_y": 0.2}
            size_hint: 0.4,0.1
            on_press:
                root.manager.transition = NoTransition()
                root.manager.current = "MessageResponses_add"
                root.manager.current_screen.respondAudio_select()



<VisitorLog>:
    name: 'VisitorLog'

    Image:
        id: background
        source: "SmartBell_visitorLog.png"
        size: self.size #sets image to same size as screen
        allow_stretch: True #allows image to be stretched so that it fills the
screen
        keep_ratio: False #allows image proportions to be altered so that it
fills the screen


    Button:
        id: button_homepage #unique identifier for this specific button
        pos_hint: {"center_x": 0.52, "center_y": 0.07}
        size_hint: 0.2,0.1
        on_press:
            root.manager.current = "Homepage"
```

```
    FloatLayout:

        MDSwitch:
            pos_hint: {'center_x': .5, 'center_y': .3}
            width: dp(64)
            selected_color: 37/255, 41/255, 88/255, 1
            on_active:
                if root.date == True: root.displayLog('name')
                else: root.displayLog('date')

        MDLabel:
            id: dateORname #unique identifier for this specific label
            text: "     Date                          Name" #notifies user that
input is required if they have not inputted any data
            theme_text_color: "Custom"
            text_color: 1,1,1,1 #white
            halign: "center"
            pos_hint: {"center_x": 0.5, "center_y": 0.3} #position of label
relative to screen

        MDLabel:
            id: averageRate #unique identifier for this specific label
            theme_text_color: "Custom"
            text_color: 1,1,1,1 #white
            halign: "center"
            pos_hint: {"center_x": 0.5, "center_y": 0.22} #position of label
relative to screen

        MDLabel:
            id: averageTime #unique identifier for this specific label
            theme_text_color: "Custom"
            text_color: 1,1,1,1 #white
            halign: "center"
            pos_hint: {"center_x": 0.5, "center_y": 0.17} #position of label
relative to screen


    ScrollView:
        size_hint: 0.8,0.45
        pos_hint: {"center_x": 0.5, "top": 0.8}
        MDList:
            id: container



<DialogContent>:
    name: "DialogContent"
    orientation: "vertical"
    size_hint_y: None
    height: "40dp"

    MDTextField:
        id: messageName #unique identifier for this specific text field
        size_hint_x: 1
        hint_text: "Name"
        color_mode: 'custom'
        line_color_normal: [202/255,154/255,254/255,1] #color of the line
around the text field in its normal state
        line_color_focus: [202/255,154/255,254/255,1] #color of the line
```

```
around the text field when the user attempts to close the dialog box before
entering a name for the audio message
```

Rest API Server

https://github.com/orlandoalexander/Computer-Science-A-Level-NEA-Web-Server

application.py

```
from flask import Flask, request, jsonify, send_file
from mysql import connector
import boto3
import json
import random
import string
from cryptography.fernet import Fernet
import time

application = Flask(
    __name__)  # wraps file using the Flask constructor and stores it as the
central object called 'application'


@application.route("/")
# homepage route
def test():
    return "Working"  # if the pipeline and server is working, the text
'Working' is displayed when the homepage route is accessed


@application.route("/updateUsers", methods=["POST"])
# route to add a new user to the 'users' table
def updateUsers():
    try:
        with open("/etc/keys/db.json", "r") as file:
            keys = json.load(file)
        data = request.form  # assigns the data sent to the API to a variable
('data')
        mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                                 database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
        myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
        query = "INSERT INTO users(accountID, firstName, surname, email,
password) VALUES ('%s','%s','%s','%s','%s')" % (
        data['accountID'], data['firstName'], data['surname'], data['email'],
data[
            'password'])  # MySQL query to add the data sent with the API to
the appropriate columns in the 'users' table
        myCursor.execute(query)  # executes the query in the MySQL database
        mydb.commit()  # commits the changes to the MySQL database made by the
executed query
        return "success"  # confirms that MySQL database was successfully
updated
    except:
        return "error"  # signifies that an error ocurred when adding the
user's data in the 'users' table


@application.route("/verifyUser", methods=["POST"])
# route to verify a user's sign in details
```

```
def verifyUser():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    query = "SELECT EXISTS (SELECT * FROM users WHERE email = '%s' AND
password = '%s')" % (data['email'], data[
        'password'])  # 'query' variable stores string with MySQL command that
is to be executed. The '%s' operator is used to insert variable values into
the string.
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = (myCursor.fetchone())[
        0]  # returns the first result of the query result (accountID), if
there is a result to be returned
    if result == 1:
        query = "SELECT accountID FROM users WHERE email = '%s' AND password =
'%s'" % (data['email'], data[
            'password'])  # 'query' variable stores string with MySQL command
that is to be executed. The '%s' operator is used to insert variable values
into the string.
        myCursor.execute(
            query)  # the query is executed in the MySQL database which the
variable 'myCursor' is connected to
        result = (myCursor.fetchone())  # returns the first result of the
query result (accountID), if there is a result to be returned
        return {'result': result[0]}
    else:
        return {'result': 'none'}


@application.route("/verifyAccount", methods=["POST"])
# route to verify that a user's account doesn't already exist
def verifyAccount():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT EXISTS(SELECT * FROM users WHERE email = '%s')" % (data[
        'email'])  # 'query' variable stores string with MySQL command that is
to be executed. The '%s' operator is used to insert variable values into the
string.
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()[0]  # returns 1 if there is a result and 0 if
not
    if result == 1:
        return "exists"  # the string 'exists' is returned if the user's
```

```
inputted details match an account which already exists in the 'users' MySQL
table
    else:
        return "notExists"  # the string 'notExists' is returned if the user's
inputted details do not match an account which already exists in the 'users'
MySQL table


@application.route("/view_audioMessages", methods=["POST"])
# route to determine how many audio messages a particular user has and what
the display names are and file details are for these messages
def view_audioMessages():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT EXISTS(SELECT * FROM audioMessages WHERE accountID =
'%s')" % (data[
        'accountID'])  # 'query' variable stores string with MySQL command
that is to be executed. The '%s' operator is used to insert variable values
into the string.
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()[0] # returns '1' if records exist with user's
account ID (i.e. if the user has created audio messages) and '0' if not
    if result == 1:
        query = "SELECT messageID, messageName, messageText FROM audioMessages
WHERE accountID = '%s'" % (data['accountID'])
        myCursor.execute(
            query)  # the query is executed in the MySQL database which the
variable 'myCursor' is connected to
        result = myCursor.fetchall()  # returns all the results of the query
        result_dict = dict()  # creates a dictionary to store the results from
the executed query
        result_dict["length"] = len(
            result)  # add the key 'length' to the dictionary to store the
number of audio messages stored in the 'audioMessages' MySQL table for the
concerned user
        for i in result:
            result_dict[str(result.index(i))] = i  # adds the name of each
audio message and the respective data from the field 'fielText' to the
dictionary with keys of an incrementing numerical value
        return {'result': result_dict}  # returns a jsonfied object of the
results dictionary using the method 'jsonify'
    else:
        return {'result': 'none'}


@application.route("/verify_messageID", methods=["POST"])
# route to check whether the messageID that has been generated for an audio
message does not already exist
def verify_messageID():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
```

```
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT EXISTS(SELECT * FROM audioMessages WHERE messageID =
'%s')" % (data[
        'messageID'])  # 'query' variable stores string with MySQL command
that is to be executed. The '%s' operator is used to insert variable values
into the string.
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()[
        0]  # returns all the results of the query result (messageName and
messageText), if there is a result to be returned
    if result == 1:
        return "exists"  # the string 'exists' is returned if the messageID
generated is already used by another audio message in the 'audioMessages'
table
    else:
        return "notExists"  # the string 'notExists' is returned if the
messageID generated is not already used by another audio message in the
'audioMessages' table


@application.route("/verify_messageName", methods=["POST"])
# route to check whether the message name that the user has inputted has
already been assigned to one of their audio messages
def verify_messageName():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT EXISTS(SELECT * FROM audioMessages WHERE messageName =
'%s' AND accountID = '%s')" % (
    data['messageName'], data[
        'accountID'])  # 'query' variable stores string with MySQL command
that is to be executed. The '%s' operator is used to insert variable values
into the string.
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()[
        0]  # returns all the results of the query result (messageName and
messageText), if there is a result to be returned
    if result == 1:
        return "exists"  # the string 'exists' is returned if the message name
is already assigned one of the user's audio messages in the 'audioMessages'
table
    else:
        return "notExists"  # the string 'notExists' is returned if the
message name is not already assigned one of the user's audio messages in the
'audioMessages' table
```

```
@application.route("/update_audioMessages", methods=["POST"])
# route to add data about a new audio message to the 'audioMessages' table
def update_audioMessages():
    try:
        with open("/etc/keys/db.json", "r") as file:  # load file storing
credentials to access RDS database
            keys = json.load(file)  # convert the json file into a json object
        data = request.form  # assigns the data sent to the API to a variable
('data')
        mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                                 database="ebdb")  # initialise connection to
database
        myCursor = mydb.cursor()  # initialises a cursor which allows
communication with 'mydb' (MySQL database session)
        if data['initialCreation'] == "False":  # if audio message already
exists in database
            query = "UPDATE audioMessages SET messageName = '%s', messageText
= '%s' WHERE messageID = '%s'" % (
            data['messageName'], data['messageText'],
            data['messageID'])  # update message name and message text (if
appropriate)
        else:  # if audio message is to be added to database for the first
time
            query = "INSERT INTO audioMessages (messageID, messageName,
messageText, accountID) VALUES ('%s', '%s', '%s', '%s')" % (
            data['messageID'], data['messageName'], data['messageText'],
data['accountID'])
            # data for new audio message is inserted into the table
'audioMessages'
        myCursor.execute(query)  # the query is executed in the MySQL database
        mydb.commit()  # commits the changes to the MySQL database made by the
executed query
        return "success"  # 'success' returned if changes are made
successfully
    except:
        return "error"  # 'error' returned if there is an error in the process


@application.route("/update_visitorLog", methods=["POST"])
# route to add data about a new visit to the 'visitorLog' table
def update_visitorLog():
    try:
        with open("/etc/keys/db.json", "r") as file:
            keys = json.load(file)
        data = request.form  # assigns the data sent to the API to a variable
('data')
        mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                                 database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
        myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
        query = "INSERT INTO visitorLog (visitID, imageTimestamp, faceID,
accountID) VALUES ('%s', '%s', '%s', '%s')" % (
        data['visitID'], data['imageTimestamp'], data['faceID'], data[
            'accountID'])  # 'query' variable stores string with MySQL command
that is to be executed. The '%s' operator is used to insert variable values
into the string.
```

```
        myCursor.execute(
            query)  # the query is executed in the MySQL database which the
variable 'myCursor' is connected to
        mydb.commit()  # commits the changes to the MySQL database made by the
executed query
        return "success"
    except:
        return "error"


@application.route("/view_visitorLog", methods=["POST"])
# retrieve details for a particular visit from the table 'visitorLog'
def view_visitorLog():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT imageTimestamp, faceID FROM visitorLog WHERE visitID =
'%s'" % (data["visitID"])
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()
    return jsonify(result)


@application.route("/get_visitorLog", methods=["POST"])
# route to retrieve data about the visits associated with a user's account
from the 'visitorLog' table
def get_visitorLog():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT imageTimestamp, faceID, visitID FROM visitorLog WHERE
accountID = '%s'" % (data["accountID"])
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchall()
    return jsonify(result)


@application.route("/get_faceName", methods=["POST"])
# retrieve the name of a visitor given their face ID from the table
'knownFaces'
def get_faceName():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
```

```
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query =  "SELECT faceName FROM knownFaces WHERE faceID = '%s'" %
(data["faceID"])
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()
    return jsonify(result)


@application.route("/get_averageTime", methods=["POST"])
# find the average time of day when the user's doorbell is rung
def get_averageTime():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT AVG(SUBSTRING(imageTimestamp,1,5)) FROM visitorLog WHERE
accountID = '%s'" % (data["accountID"])
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()[0]
    return {'result': result}


@application.route("/get_averageRate", methods=["POST"])
# find the average number of visits per day to the user's doorbell
def get_averageRate():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT COUNT(*) FROM visitorLog WHERE accountID = '%s'" %
(data["accountID"]) # retrieve total number of visits for a user's account
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    count = myCursor.fetchone()[0] # total number of visits to user's
dooorbell
    query = "SELECT MIN(SUBSTRING(imageTimestamp, 7)) FROM visitorLog WHERE
accountID = '%s'" % (data["accountID"]) # retrieve time of first recorded
visit to user's doorbell
    myCursor.execute(query)
```

```
    initialTime = myCursor.fetchone()[0] # time when user's doorbell first
rung
    currentTime = time.time()
    totalDays = (currentTime-float(initialTime))/24/3600 # total days passed
since first visit to user's doorbell
    averageRate = count/totalDays # average number of visits to user's
doorbell
    return {'result': averageRate}


@application.route("/latest_visitorLog", methods=["POST"])
# retrieve the details about the latest visit the user's doorbell
def latest_visitorLog():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT EXISTS(SELECT visitID, faceID FROM visitorLog WHERE
accountID = '%s' ORDER BY imageTimestamp DESC)" % (
    data["accountID"])
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()[
        0]  # returns all the results of the query result (messageName and
messageText), if there is a result to be returned
    if result == 1:
        query = "SELECT visitID, faceID FROM visitorLog WHERE accountID = '%s'
ORDER BY imageTimestamp DESC" % (
        data["accountID"])
        myCursor.execute(
            query)  # the query is executed in the MySQL database which the
variable 'myCursor' is connected to
        result = myCursor.fetchone()
        return {'result': result}
    else:
        return {'result': 'none'}


@application.route("/update_knownFaces", methods=["POST"])
# route to add data about a new audio message to the 'knownFaces' table
def update_knownFaces():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    if data[
        'faceName'] == "":  # if this is the first time the record with this
faceID has been added to the database (from the raspberry pi)
```

```
        query = "INSERT INTO knownFaces (faceID, faceName, accountID) VALUES
('%s', '%s', '%s')" % (
        data['faceID'], data['faceName'], data[
            'accountID'])  # 'query' variable stores string with MySQL command
that is to be executed. The '%s' operator is used to insert variable values
into the string.
    else:  # called when the user has entered the name for the faceID value
and wants to store this value
        query = "UPDATE knownFaces SET faceName = '%s' WHERE faceID = '%s'" %
(data['faceName'], data['faceID'])
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    mydb.commit()  # commits the changes to the MySQL database made by the
executed query
    return "success"



@application.route("/delete_audioMessages", methods=["POST"])
# delete audio message record from 'audioMessage' table
def delete_audioMessages():
    try:
        with open("/etc/keys/db.json", "r") as file:
            keys = json.load(file)
        data = request.form  # assigns the data sent to the API to a variable
('data')
        mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                                 database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
        myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
        query = "DELETE FROM audioMessages WHERE messageID = '%s'" %
(data['messageID'])
        myCursor.execute(
            query)  # the query is executed in the MySQL database which the
variable 'myCursor' is connected to
        mydb.commit()  # commits the changes to the MySQL database made by the
executed query
        return "success"
    except:
        return "error"


@application.route("/uploadS3", methods=["POST"])
# route to upload byte data of the user's personalised audio messages
def uploadS3():
    try:
        with open("/etc/keys/S3.json", "r") as file:
            keys = json.load(file)
        data = request.form  # assigns the metadata sent to the API to a
variable ('data')
        file = request.files[
            "file"]  # assigns the txt file storing the bytes of the audio
message to the variable 'file'
        fileName = "/tmp/uploadFile.pkl"
        file.save(fileName)  # temporarily saves the txt file in the "/tmp"
folder on the AWS server
        s3 = boto3.client("s3", aws_access_key_id=keys["accessKey"],
aws_secret_access_key=keys[
            "secretKey"])  # initialises a connection to the S3 client on AWS
```

```
using the 'accessKey' and 'secretKey' sent to the API
        s3.upload_file(Filename=fileName, Bucket=data["bucketName"], Key=data[
            "s3File"])  # uploads the txt file to the S3 bucket called 'nea-
audio-messages'. The name of the txt file when it is stored on S3 is the
'messageID' of the audio message which is being stored as a txt file.
        return "success"
    except:
        return "error"


@application.route("/downloadS3", methods=["POST"])
# route to download pickled byte data of the user's personalised audio
messages or image file of visitor from AWS S3 storage
def downloadS3():
    try:
        with open("/etc/keys/S3.json", "r") as file:  # load file storing pair
of keys required to establish connection with S3 storage server
            keys = json.load(file)  # convert the json file into a json object
        data = request.form  # assigns the metadata sent to the API to a
variable ('data')
        if data["bucketName"] == "nea-audio-messages":
            fileName = "/tmp/audioMessage_download.pkl"  # file location in eb
environment
        elif data["bucketName"] == "nea-visitor-log":
            fileName = "/tmp/visitorImage_download.png"  # file location in eb
environment
        s3 = boto3.client("s3", aws_access_key_id=keys["accessKey"],
aws_secret_access_key=keys["secretKey"])
        # initialises a connection to the S3 client on AWS using the
'accessKey' and 'secretKey'
        s3.download_file(Filename=fileName, Bucket=data["bucketName"],
Key=data["s3File"])
        # downloads file name equal 's3file' from the S3 bucket with name
'bucketName' and stores this file in the '/tmp' directory of the eb
environment
        return send_file(fileName, as_attachment=True) # returns the
downloaded file to the client
    except:
        return "error" # 'error' returned if there is an error in the process


@application.route("/create_ID", methods=["POST"])
# route to create a unique ID (face ID, visit ID or account ID)
def create_ID():
    data = request.form
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    while True:  # creates an infinite loop
        chars = string.ascii_uppercase + string.ascii_lowercase +
string.digits  # creates a concatenated string of all the uppercase and
lowercase alphabetic characters and all the digits (0-9)
        ID = (''.join(random.choice(chars) for i in
                      range(43))) + '='  # unique message ID is compatible
format for fernet encryption
        if data["field"] == "visitID":
```

```
            query = "SELECT EXISTS(SELECT * FROM visitorLog WHERE visitID =
'%s')" % (
                ID)  # 'query' variable stores string with MySQL command that
is to be executed. The '%s' operator is used to insert variable values into
the string.
        elif data["field"] == "accountID":
            query = "SELECT EXISTS(SELECT * FROM users WHERE accountID =
'%s')" % (
                ID)  # 'query' variable stores string with MySQL command that
is to be executed. The '%s' operator is used to insert variable values into
the string.
        elif data["field"] == "faceID":
            query = "SELECT EXISTS(SELECT * FROM knownFaces WHERE faceID =
'%s')" % (
                ID)  # 'query' variable stores string with MySQL command that
is to be executed. The '%s' operator is used to insert variable values into
the string.
        myCursor.execute(
            query)  # the query is executed in the MySQL database which the
variable 'myCursor' is connected to
        result = myCursor.fetchone()[
            0]  # returns the first result of the query result (accountID), if
there is a result to be returned
        if result == 0:
            break
    return ID


@application.route("/get_S3Key", methods=["POST"])
# route to retrieve the pair of keys required to interact with AWS S3 storage
server
def get_S3Key():
    try:
        with open("/etc/keys/db.json", "r") as file:  # load file storing
credentials to access RDS database
            keys = json.load(file)  # convert the json file into a json object
        data = request.form  # assigns the data sent to the API to a variable
('data')
        mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]), database="ebdb")  # initialise connection to database
        myCursor = mydb.cursor()  # initialises a cursor which allows
communication with 'mydb' (MySQL database session)
        with open("/etc/keys/S3.json", "r") as file:  # load file storing pair
of keys required to establish connection with S3 storage server
            keys_S3 = json.load(file)  # convert the json file into a json
object
        query = "SELECT * FROM users WHERE accountID = '%s'" %
(data["accountID"])
        myCursor.execute(query)
        result = myCursor.fetchone() # retrieve first matching record from
MySQL database
        if result != 0: # verifies if an account exists with the specified
accountID
            key = data["accountID"].encode()  # key must be encoded as bytes
            fernet = Fernet(key)  # instantiates Fernet encryption object
using user's accountID as the encryption key
            accessKey_encrypted =
fernet.encrypt(keys_S3["accessKey"].encode())  # use Fernet class instance to
encrypt the string - string must be encoded to byte string before it is
encrypted
            secretKey_encrypted= fernet.encrypt(keys_S3["secretKey"].encode())
# use Fernet class instance to encrypt the string - string must be encoded to
```

```
byte string before it is encrypted
            encryptedKeys_dict = {'accessKey_encrypted':
accessKey_encrypted.decode(),
                                  'secretKey_encrypted':
secretKey_encrypted.decode()}  # keys must be decoded to be jsonified and
returned by API
            return encryptedKeys_dict
        else:
            return "error"
    except:
        return "error"

@application.route("/update_SmartBellIDs", methods=["POST"])
# route to udpate details about SmartBell pairings in table 'SmartBellIDs'
def update_SmartBellIDs():
    try:
        with open("/etc/keys/db.json", "r") as file:
            keys = json.load(file)
        data = request.form  # assigns the data sent to the API to a variable
('data')
        mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                                 database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
        myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
        if 'accountID' not in data:  # if request is to add a new id, not a
new accountID
            query = "INSERT INTO SmartBellIDs (id) VALUES ('%s')" % (data[
                'id'])  # 'query' variable stores string with MySQL command
that is to be executed. The '%s' operator is used to insert variable values
into the string.
        else:
            query = "UPDATE SmartBellIDs SET accountID = ('%s') WHERE id =
'%s'" % (data['accountID'], data[
                'id'])  # 'query' variable stores string with MySQL command
that is to be executed. The '%s' operator is used to insert variable values
into the string.
        myCursor.execute(
            query)  # the query is executed in the MySQL database which the
variable 'myCursor' is connected to
        mydb.commit()  # commits the changes to the MySQL database made by the
executed query
        return 'True'
    except:
        return 'False'


@application.route("/verifyPairing", methods=["POST"])
# route to check whether user has successfully paired with the a particular
SmartBell
def verifyPairing():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
```

```
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT accountID FROM SmartBellIDs WHERE id = ('%s')" %
(data['id'])
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()  # returns the first result of the query
result (accountID), if there is a result to be returned
    return {'result': result[0]}


@application.route("/checkPairing", methods=["POST"])
# route to check whether a particular SmartBell ID already exists
def checkPairing():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT EXISTS(SELECT * FROM SmartBellIDs WHERE id = ('%s'))" %
(data['id'])
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()[
        0]  # returns the first result of the query result (accountID), if
there is a result to be returned
    if result == 1:
        return 'exists'
    else:
        return 'notExists'


@application.route("/getPairing", methods=["POST"])
# route to retrieve the pairing status for a particular user account
def getPairing():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT EXISTS(SELECT * FROM SmartBellIDs WHERE accountID =
('%s'))" % (data['accountID'])
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchone()[
        0]  # returns the first result of the query result (accountID), if
there is a result to be returned
    if result == 1:
        query = "SELECT id FROM SmartBellIDs WHERE accountID = ('%s')" %
(data['accountID'])
```

```
        myCursor.execute(
            query)  # the query is executed in the MySQL database which the
variable 'myCursor' is connected to
        result = myCursor.fetchone()
        return {'result': result[0]}
    else:
        return {'result': 'none'}


@application.route("/checkFaces", methods=["POST"])
# route to check whether the duplicate face names exist for the same user's
account
def checkFaces():
    with open("/etc/keys/db.json", "r") as file:
        keys = json.load(file)
    data = request.form  # assigns the data sent to the API to a variable
('data')
    mydb = connector.connect(host=(keys["host"]), user=(keys["user"]),
passwd=(keys["passwd"]),
                             database="ebdb")  # initialises the database
using the details sent to API, which can be accessed with the 'request.form()'
method
    myCursor = mydb.cursor()  # initialises a cursor which allows
communication with mydb (MySQL database)
    query = "SELECT faceName FROM knownFaces WHERE accountID = '%s' GROUP BY
faceName HAVING COUNT(faceName) > 1" % (
    data['accountID']) # select all face names which appear in more than one
record in 'knownFaces' for the same user account
    myCursor.execute(query)  # the query is executed in the MySQL database
which the variable 'myCursor' is connected to
    result = myCursor.fetchall()
    faceIDs = []
    for faceName in result: # iterate through duplicate face names
        query = "SELECT faceID FROM knownFaces WHERE faceName = '%s' AND
accountID = '%s' " % (
        faceName[0], data['accountID']) # get the face ID for each duplicate
face name
        myCursor.execute(
            query)  # the query is executed in the MySQL database which the
variable 'myCursor' is connected to
        result = myCursor.fetchall()
        faceIDs.append(result)
        faceIDs_delete = result[1:] # get the face IDs of duplicate names
which are to be deleted from database
        faceID_keep = result[0][0] # get the base face ID which is to remain
stored in database
        for faceID in faceIDs_delete:
            query = "DELETE FROM knownFaces WHERE faceID = '%s' AND accountID
= '%s'" % (faceID[0], data['accountID']) # delete face IDs of duplicate names
            myCursor.execute(query)  # the query is executed in the MySQL
database which the variable 'myCursor' is connected to
            query = "UPDATE visitorLog SET faceID = '%s' WHERE faceID = '%s'
AND accountID = '%s'" % (
            faceID_keep, faceID[0], data['accountID']) # update visitor log to
store same face ID for all visits where visitor has same name
            myCursor.execute(query)  # the query is executed in the MySQL
database which the variable 'myCursor' is connected to
    mydb.commit()  # commits the changes to the MySQL database made by the
executed query
    response = jsonify(faceIDs)
    return response
```

```
if __name__ == "__main__":  # if the name of the file is the main program (not
a module imported from another file)...
    application.run(debug=True)  # ...then the API server begins running
```

Raspberry Pi

https://github.com/orlandoalexander/ComputerScience-A-Level-NEA-RPi

main_pi.py

```
#!/usr/bin/python3
import cv2 as cv
from picamera.array import PiRGBArray
from picamera import PiCamera
import time
import json
import boto3
from os.path import join
import os
import threading
import paho.mqtt.client as mqtt
import requests
from cryptography.fernet import Fernet
import face_recognition
import pickle

path = "/home/pi/Desktop/NEA/ComputerScience-NEA-RPi"

serverBaseURL = "http://nea-env.eba-6tgviyyc.eu-west-2.elasticbeanstalk.com/"
# base URL to access AWS elastic beanstalk environment

haarCascade = cv.CascadeClassifier(join(path,"haar_face_alt2.xml")) # reads in
the xml haar cascade file

windowSize_mobile = (640, 1136) # mobile phone screen size in pixels

class buttonPressed():
    def __init__(self):
        with open(join(path,'data.json'), 'r') as jsonFile: # ensures up-to-
date value for accountID is used
            time.sleep(0.5) # avoids concurrent access errors for 'data.json'
file
            self.data = json.load(jsonFile) # load json file as json object
            self.accountID = str(self.data['accountID']) # retrieve user's
account ID stored under key 'accountID' in json object 'data'
        self.visitID = self.create_visitID() # create unique visit ID for the
visit
        self.publish_message_ring() # transmit MQTT message to the mobile app
to notify of visit
        if self.accountID not in self.data: # if data for account not stored
(i.e. doorbell not paired with a user account)
            self.data.update({self.accountID:{"faceIDs":[]}}) # updates json
file to create empty parameter to store names of known visitors associated
with a specific accountID
            with open(join(path,'data.json'),'w') as jsonFile:
                json.dump(self.data, jsonFile) # write json object data to
json file 'data.json' so stored permanently on Raspberry Pi

    def captureImage(self):
        # called when doorbell 'ring' button pressed
        self.camera = PiCamera() # create instance of Raspberry Pi camera
        self.rawCapture = PiRGBArray(self.camera) # using PiRGBArray increases
efficiency when accessing camera stream
        self.trainingImages = []
```

```
        self.faceDetected = False
        time.sleep(0.155) # delay to allow camera to warm up
        attempts = 0
        while attempts < 2: # make up to two attempts to capture high quality
image of visitor
            faceRGBImages = []
            self.rawCapture.truncate(0) # clear any data from the camera
stream
            self.camera.capture(self.rawCapture, format="bgr") # captures
camera stream in 'bgr' format (opencv array requires this)
            self.faceBGR = cv.flip(self.rawCapture.array,0) # flip image in
vertical (0) axis, as camera module is upside down when attached to Raspberry
Pi
            self.faceGray = cv.cvtColor(self.faceBGR, cv.COLOR_BGR2GRAY) #
change visitor image to grayscale for OpenCV operations
            self.faceRGB = cv.cvtColor(self.faceBGR, cv.COLOR_BGR2RGB) #
change visitor image to rgb for face-recognition library operations
            if attempts == 0: # first image of visitor captured
                self.uploadImage = threading.Thread(target=self.formatImage,
args=(self.faceBGR,), daemon=False)
                self.uploadImage.start()  # starts the thread which will run
in pseudo-parallel to the rest of the program to format image for display in
mobile app GUI
            faceRGBImages.append(self.faceRGB)
            # check if face exists as much quicker than doing facial
recognition (so can check whether need to capture another image):
            faceDetected = haarCascade.detectMultiScale(self.faceGray,
scaleFactor=1.01, minNeighbors=6)  # returns rectangular coordinates of face
bounding box
            blurFactor = cv.Laplacian(self.faceGray, cv.CV_64F).var() #
calculate bluriness of visitor image
            num_faceDetected = len(faceDetected) # number of faces detected in
image
            if num_faceDetected >= 1 and blurFactor >= 25: # if at least 1
face has been detected and image isn't blurry, image considered suitable
                self.trainingImages.extend(faceRGBImages) # save visitor
images in RGB format to train the facial recognition algorithm
                self.faceDetected = True
                break # suitable image captured so do not attempt capturing
images again
            else:
                attempts +=1
        if self.faceDetected == True: # if face detected in visitor image
            self.facialRecognition() # run facial recognition algorithm
        else: # if no face detected in visitor image
            self.faceID = "NO_FACE"
            self.update_visitorLog() # update SQL database to store visit
details
            self.camera.close() # terminate camera instance
            quit()


    def recognise(self, faceRGB):
        # facial recognition algorithm
        fileName = join(path,"trainingData") # load the face encodings and
labels for trained image data set
        if not os.path.isfile(fileName): # training image data set doesn't
exist yet
            return 'Unknown', False # face not recognised
        data = pickle.loads(open(fileName, "rb").read()) # reconstruct
dictionary object 'data' from character stream stored in fileName file
        encodings = face_recognition.face_encodings(faceRGB) # create face
```

```
encodings for visitor image (test image)
        for encoding in encodings: # iterate through face encodings in the
visitor image as there may be multiple faces in the visitor image
            matches = face_recognition.compare_faces(data["encodings"],
encoding) # compare encoding of face in visitor image with encodings in
trained data set
            # matches contain array with boolean values True and False for
each face encoding in 'data'
            if True in matches: # if there is at least one match for the test
image in the training image data set
                matchedIndexes = [index for (index, match) in
enumerate(matches) if match] # store indexes of training set face encodings
which match a face encoding in visitor image
                labelCount = {}
                for index in matchedIndexes: # loop over the matched indexes
and store a count for each face in training data set which matches the test
image
                    label = data["labels"][index] # get the label associated
with the face encoding at index 'index'
                    labelCount[label] = labelCount.get(label,0) + 1 #
increment counter (value) for label (key) of face encoding stored at 'index',
indicating that test image has match with this label
                label = max(labelCount, key=labelCount.get) # return key with
greatest value (i.e. label of face in training image data set with greatest
number of matches)
                matchCount = labelCount[label] # number of matches for label
assigned to visitor image
                actualCount = 0 # stores total number of face encodings with
same label as label assigned to visitor image
                for i in data['labels']: # iterate through each label in
trained data set
                    if i == label: # label in trained data set is same as
label assigned to visitor image
                        actualCount +=1
                if matchCount/actualCount > 0.5: # if visitor image matches
more than 50% of training data set images with same label
                    return label, True
                else:
                    return 'Unknown', False
            else:
                return 'Unknown', False
        return 'Unknown', False


    def facialRecognition(self):
        # driver method for facial recognition algorithm
        self.faceIDs = []
        with open(join(path,'data.json')) as jsonFile:
            self.data = json.load(jsonFile) # load json object from
'data.json' file
            self.faceIDs = [faceID for faceID in
self.data[self.accountID]["faceIDs"]] # store all face IDs associated with
user's account in array

        self.label, self.faceRecognised = self.recognise(self.faceRGB) #
execute facial recognition algorithm to retrieve label for captured visitor
image

        if self.faceRecognised == True: # if face recognised in visitor image
            self.faceID = self.faceIDs[self.label] # retrieve face ID for face
label detected in visitor image
        else: # if no face recognised in visitor image
```

```
            self.faceID = self.create_faceID() # create new face ID for face
in visitor image

        self.update_visitorLog() # update SQL database to store visit details

        if self.faceID not in self.faceIDs: # if visitor image is new face
            self.faceIDs.append(self.faceID)
            self.label = self.faceIDs.index(self.faceID) # create new label
which corresponds to index of new face ID in face IDs array
            self.update_knownFaces() # store details for new face

        with open(join(path,'data.json'), 'w') as jsonFile:
            json.dump(self.data, jsonFile)
        self.thread_updateTraining =
threading.Thread(target=self.updateTraining, args=(), daemon=False) # create
thread to train facial recognition algorithm
        self.thread_updateTraining.start()  # starts the thread which will run
in pseudo-parallel to the rest of the program to train facial recognition
algorithm

    def train(self, faceRGB, label): # train facial recognition algorithm
        boxes = face_recognition.face_locations(faceRGB,model='hog') #
bounding box around face location in image
        encodings = face_recognition.face_encodings(faceRGB, boxes) # compute
the facial encodings for the face
        for encoding in encodings: # loop through each face encoding in image
            self.encodings.append(encoding)
            self.labels.append(label)
        return self.encodings, self.labels


    def updateTraining(self):
        # driver method for training algorithm
        self.encodings = [] # store face encodings for trained data set
        self.labels = [] # store corresponding face labels for trained data
set
        attempts = 0

        while attempts < 2: # make up to two attempts to capture high quality
image of visitor
            self.rawCapture.truncate(0) # clear any data from the camera
stream
            self.camera.capture(self.rawCapture, format="bgr") # captures
camera stream in 'bgr' format (opencv array requires this)
            self.faceBGR = cv.flip(self.rawCapture.array,0) # flip image in
vertical (0) axis, as camera module is upside down when attached to Raspberry
Pi
            self.faceGray = cv.cvtColor(self.faceBGR, cv.COLOR_BGR2GRAY) #
change visitor image to grayscale for OpenCV operations
            self.faceRGB = cv.cvtColor(self.faceBGR, cv.COLOR_BGR2RGB) #
change visitor image to rgb for face-recognition library operations
            faceDetected = haarCascade.detectMultiScale(self.faceGray,
scaleFactor=1.01, minNeighbors=6)  # returns rectangular coordinates of face
bounding box
            blurFactor = cv.Laplacian(self.faceGray, cv.CV_64F).var() #
calculate bluriness of visitor image
            num_faceDetected = len(faceDetected) # number of faces detected in
image
            if num_faceDetected >= 1 and blurFactor >= 25: # if at least 1
face has been detected and image isn't blurry, image considered suitable
                self.trainingImages.extend(self.faceRGB) # save visitor images
in RGB format to train the facial recognition algorithm
```

```
            attempts +=1

        self.camera.close()

        self.data[self.accountID]["faceIDs"] = self.faceIDs # update face IDs
in json object
        self.data['training'] = 'True' # status message to indicate that
training is complete and camera instance has been closed, so can now ring
doorbell again with new camera instance (avoids concurrent camera access
errors)

        with open(join(path,'data.json'), 'w') as jsonFile:
            json.dump(self.data, jsonFile)

        for faceRGB in self.trainingImages: # iterate through image arrays for
each training image
            if self.faceRecognised == True: # if face identified in test
visitor image
                label, faceRecognised = self.recognise(faceRGB) # get face
label for training image
                if label == self.label: # if training image  has same label as
label of test visitor image
                    self.encodings, self.labels = self.train(faceRGB, label) #
train facial recognition algorithm and update face encodings and labels
            elif self.faceRecognised == False: # if no face identified in test
visitor image
                self.encodings, self.labels = self.train(faceRGB, self.label)
# train facial recognition algorithm and update face encodings and labels for
new face label

        fileName = join(path, "trainingData")  # load the face encodings and
labels for trained image data set
        if os.path.isfile(fileName): # training image data set exists
            trainingData = pickle.loads(open(join(path,"trainingData"),
"rb").read()) # reconstruct dictionary object 'trainingData' from character
stream stored in 'trainingData' file
            trainingData['encodings'].extend(self.encodings) # append latest
version of visitor image face encodings to training data
            trainingData['labels'].extend(self.labels) # append latest version
of visitor image labels to training data
        else: # training image data set doesn't exist
            trainingData = {'encodings': self.encodings, 'labels':
self.labels} # create new dictionary storing face encodings and labels for
trained data set

        f = open(join(path,"trainingData"), "wb")
        f.write(pickle.dumps(trainingData)) # dump training data dictionary
object as character stream to 'trainingData'
        f.close()

        self.data['training'] = 'False'
        with open(join(path,'data.json'), 'w') as jsonFile:
            json.dump(self.data, jsonFile)
        client.disconnect() # disconnect client from MQTT broker
        quit()


    def create_visitID(self):
        # creates a unique visitID for each visit
        data_vistID = {"field": "visitID"} # dictionary passed to REST API
path to specify that required ID is visit ID
        visitID = requests.post(serverBaseURL + "/create_ID",
```

```
data_vistID).text # REST API path generate new unique visit ID
        return visitID


    def create_faceID(self):
        # creates a unique faceID for the face captured
        data_faceID = {"field": "faceID"} # dictionary passed to REST API path
to specify that required ID is face ID
        faceID = requests.post(serverBaseURL + "/create_ID", data_faceID).text
# REST API path generate new unique face ID
        return faceID


    def update_visitorLog(self):
        # store visit details in SQL table 'visitorLog'
        data_visitorLog = {"visitID": self.visitID, "imageTimestamp":
(str(time.strftime("%H.%M"))+','+str(time.time())), "faceID": self.faceID,
"accountID": self.accountID} # data to be stored in SQL table 'visitorLog'
        requests.post(serverBaseURL + "/update_visitorLog", data_visitorLog) #
REST API path store data in SQL databese
        return

    def update_knownFaces(self):
        # store details for new face in SQL table 'knownFaces'
        data_knownFaces = {"faceID": self.faceID, "faceName": "", "accountID":
self.accountID} # data to be stored in SQL table 'knownFaces'
        requests.post(serverBaseURL + "/update_knownFaces", data_knownFaces) #
REST API path store data in SQL database
        return

    def formatImage(self, visitorImage):
        # format visitor image so suitable shape to be displayed in mobile app
        visitorImage_cropped_w = round(int(windowSize_mobile[0]) * 0.93) #
target width of visitor image
        visitorImage_cropped_h = round(int(windowSize_mobile[1]) * 0.54) #
target height of visitor image
        scaleFactor = visitorImage_cropped_h / visitorImage.shape[0] # factor
by which height of image must be scale down to fit screen
        visitorImage = cv.resize(visitorImage,
                                 (int(visitorImage.shape[1] * scaleFactor),
int(visitorImage.shape[0] * scaleFactor)),
                                 interpolation=cv.INTER_AREA) # scales down
width and height of image to match required image height
        visitorImage_centre_x = visitorImage.shape[1]//2 # x-coordinate of
horizontal middle of image
        visitorImage_x = visitorImage_centre_x - visitorImage_cropped_w // 2
# start x-coordinate of visitor image
        if visitorImage_x < 0: # if desired start x-coordinate of image is
negative, set start x-coordinate to 0
            visitorImage_x = 0
        visitorImage_cropped = visitorImage[0:visitorImage.shape[0],
                               visitorImage_x:visitorImage_x +
visitorImage_cropped_w] # crops image width to fit screen
        self.path_visitorImage = join(path, 'Photos/visitorImage.png')
        cv.imwrite(self.path_visitorImage, visitorImage_cropped) # store
formatted visitor image locally on Raspnberry Pi
        self.uploadAWS_image(Bucket="nea-visitor-log", Key = self.visitID) #
upload formatted visitor image to AWS S3 storage

    def uploadAWS_image(self, **kwargs):
        # uploads image to AWS S3 storage
        fernet = Fernet(self.accountID.encode()) # instantiate Fernet class
```

```
# with users accountID as the key
        data_S3Key = {"accountID": self.accountID} # dictionary passed to REST
API path to encode AWS S3 keys
        encodedKeys = requests.post(serverBaseURL + "/get_S3Key",
data_S3Key).json() # REST API path returns json object with encoded keys
        accessKey =
fernet.decrypt(encodedKeys["accessKey_encrypted"].encode()).decode() # decode
access key using 'accountID' encryption key
        secretKey =
fernet.decrypt(encodedKeys["secretKey_encrypted"].encode()).decode() # decode
secret key using 'accountID' encryption key
        s3 = boto3.client("s3", aws_access_key_id=accessKey,
aws_secret_access_key=secretKey)  # initialises a connection to the S3 client
on AWS using the access key and secret key
        s3.upload_file(Filename=self.path_visitorImage,
Bucket=kwargs["Bucket"], Key=kwargs["Key"])  # uploads the image file to the
S3 bucket called 'nea-visitor-log'.
        return

    def publish_message_ring(self):
        # transmit MQTT message to mobile app to notify that doorbell has been
rung
        client.publish("ring/{}".format(self.accountID),
"{}".format(str(self.visitID))) # publish MQTT message to 'ring/accountID'
topic
        return


def on_connect(client, userdata, flags, rc):
    # callback function called if successful connection to MQTT broker
    if rc == 0: # if connection is successful
        client.publish('connected','')
    else:
        # attempts to reconnect
        client.on_connect = on_connect
        client.username_pw_set(username="yrczhohs", password = "qPSwbxPDQHEI")
        client.connect("hairdresser.cloudmqtt.com", 18973)

def run():
    # driver function when doorbell is rung
    client.username_pw_set(username="yrczhohs", password = "qPSwbxPDQHEI") #
specify MQTT broker connection details
    client.on_connect = on_connect # creates callback for successful
connection with broker
    client.connect("hairdresser.cloudmqtt.com", 18973) # connect to MQTT
broker
    buttonPressed().captureImage()

client = mqtt.Client()
```

button_loop.py

```
#!/usr/bin/python3
import RPi.GPIO as GPIO # Import Raspberry Pi GPIO library
import time
import main_pi
import threading
import paho.mqtt.client as mqtt
import urllib.request as url
import os
from os.path import join
import json

path = "/home/pi/Desktop/NEA/ComputerScience-NEA-RPi"

GPIO.setwarnings(False) # ignore warnings
GPIO.setmode(GPIO.BOARD) # use physical pin numbering
GPIO.setup(10, GPIO.IN, pull_up_down=GPIO.PUD_DOWN) # set pin 10 to be an
input pin and set initial value to be pulled low (off)

def runThread():
    main_pi.run()

def detect_buttonPressed():
    # detect when visitor presses doorbell 'ring' button
    while True:
        if GPIO.input(10) == GPIO.HIGH and os.path.isfile(join(path,
'data.json')) == True: # if doorbell 'ring' button is pressed and 'data.json'
file exists (indicates doorbell has been set up)
            with open(join(path,'data.json')) as jsonFile:
                data = json.load(jsonFile)
                training = data['training'] # store training status
            if (threading.active_count() == 1 or training == 'True') and
'accountID' in data: # to avoid concurrent camera access issues, doorbell can
be rung in new thread if there is currently no doorbell thread OR when the
doorbell thread is in training state, AND the doorbell is paired with a user's
account ID
                thread_run = threading.Thread(target=runThread)
                thread_run.start()

while True: # block program until doorbell is connected to internet
    try:
        url.urlopen('http://google.com') # attempts to open 'google.com'
        detect_buttonPressed()
        break
    except: # if no internet connection is established yet, then wait 5 secs
        time.sleep(5)
```

trainingData_loop.py

```
import cv2 as cv
import requests
import threading
import time
from os.path import join
import json
import numpy as np
from picamera import PiCamera
import pickle
from threading import Thread


path = "/home/pi/Desktop/NEA/ComputerScience-NEA-RPi"

serverBaseURL = "http://nea-env.eba-6tgviyyc.eu-west-2.elasticbeanstalk.com/"
# base URL to access AWS elastic beanstalk environment


def checkFaces():
    # check whether SQL table 'vsitorLog' contains any duplicate face names
associated with the same user account
    while True:
        with open(join(path,'data.json'), 'r') as jsonFile: # ensures up-to-
date value for accountID is used
            time.sleep(0.5) # avoids concurrent access errors for 'data.json'
file
            data = json.load(jsonFile)  # load json file as json object
            accountID = str(data['accountID'])  # retrieve user's account ID
stored under key 'accountID' in json object 'data'
        dbData_accountID = {'accountID': accountID}
        faceIDs_update = (requests.post(serverBaseURL + "/checkFaces",
dbData_accountID).json()) # request to REST API path to check whether there
are any face IDs with the same name for the same user account
        if len(faceIDs_update) != 0: # if face name duplicates exist
            faceIDs_update = faceIDs_update[0] # access required 2D array
which is first element stored inside 3D array
            updateLabels_thread = Thread(target = updateLabels, args =
(faceIDs_update, accountID)) # function called in thread so can update another
label simultaneously
            updateLabels_thread.start()
        time.sleep(5)

def updateLabels(faceIDs_update, accountID):
    # update labels for face encodings in trained data set where there are
duplicate face names for identified faces associated with the same user
account
    while True: # avoids running update to labels stored in 'trainingData'
while main program is running by checking if the camera is in use
        try:
            camera = PiCamera()
            break
        except: # if unable to instantiate instance of 'PiCamera', indicates
that camera is currently in use by 'main_pi.py'
            time.sleep(5)
    camera.close() # close camera instance
    time.sleep(30) # delay to allow time to complete image training and save
'trainingData'
    faceIDs = []
```

```
    with open(join(path,'data.json')) as jsonFile:
        data = json.load(jsonFile)
        for faceID in data[accountID]["faceIDs"]:
            faceIDs.append(faceID) # store all the faceIDs associated with
user's account
    trainingData = pickle.loads(open(join(path,"trainingData"), "rb").read())
# load known face encodings
    newFaceID_update = faceIDs_update.pop(0)[0] # access and remove first face
ID of the face IDs assigned to the duplicate name - all face IDs associated
with the duplicate name will be assigned to this new face ID
    newLabel = faceIDs.index(newFaceID_update) # label of new face ID
    oldFaceID_update = faceIDs_update.pop(0)[0] # old face ID which is to be
assigned to 'newFaceID_update'
    oldLabel = faceIDs.index(oldFaceID_update) # label of old face ID
    faceIDs[oldLabel] = newFaceID_update # replace old face ID with new face
ID
    for (index, label) in enumerate(trainingData['labels']): # iterate through
labels stored in 'trainingData' and replace the old labels with the new label,
so the face encodings of the duplicate face names will be tagged with the new
label
        if label == oldLabel: # if label needs to be updated
            trainingData['labels'][index] = newLabel # change label value of
old label to new label
    data[accountID]["faceIDs"] = faceIDs # save updated face IDs
    with open(join(path,'data.json'),'w') as jsonFile:
        json.dump(data, jsonFile)
    with open(join(path,'trainingData'),'wb') as file:
        file.write(pickle.dumps(trainingData)) # store updated face encodings
and associated labels

checkFaces_thread = threading.Thread(target = checkFaces, args =())
checkFaces_thread.start()
```

audio_loop.py

```
import paho.mqtt.client as mqtt
import wave
import pickle
from os.path import join
import os
import requests
from gtts import gTTS
import time
import urllib.request as url
import json
import threading

serverBaseURL = "http://nea-env.eba-6tgviyyc.eu-west-2.elasticbeanstalk.com/"

path = "/home/pi/Desktop/NEA/ComputerScience-NEA-RPi"

def playAudio(client, userData, msg):
    # output recorded audio message through doorbell's speaker
    messageID = msg.payload.decode() # decode payload sent via MQTT from
mobile app (i.e. message ID)
    downloadData = {"bucketName": "nea-audio-messages",
                                    "s3File": messageID}  # creates the
dictionary which stores the metadata required to download the pkl file of the
personalised audio message from AWS S3
    response = requests.post(serverBaseURL + "/downloadS3", downloadData) #
send request to REST API path to download pickled audio message bytes from AWS
S3 and return them
    audioData = pickle.loads(response.content) # unpickles the bytes string
    messageFile = wave.open(join(path,"audioMessage.wav"), "wb")
    messageFile.setnchannels(1) # audio stream module is single channel
    messageFile.setsampwidth(2) # 2 bytes per audio sample (sample width)
    messageFile.setframerate(8000) # 8000 samples per second
    messageFile.writeframes(b''.join(audioData)) # write audio bytes to .wav
file
    messageFile.close()
    os.system("cvlc --play-and-exit {}".format(join(path,'audioMessage.wav')))
# play audio message directly through system using command line tool 'cvlc' -
'play-and-exit' used to quit the player after audio played

def playText(client, userData, msg):
    # output typed (text-based) audio message through doorbell's speaker
    messageText = msg.payload.decode() # decode payload sent via MQTT from
mobile app (i.e. message text)
    TtS(messageText) # convert message text into audio file
    os.system("cvlc --play-and-exit {}".format(join(path,'audioMessage.wav')))
# play audio message directly through system using command line tool 'cvlc' -
'play-and-exit' used to quit the player after audio played

def TtS(text):
    # convert message text into audio file
    language = "en"
    TtS_obj = gTTS(text=text, lang=language, slow=False) # create text-to-
speech object
    TtS_obj.save(join(path,"audioMessage.wav")) # save text to speech object
as .wav file
    return

def checkAccountID(currentID, client):
    # check whether account ID associated with doorbell has been updated (i.e.
doorbell paired with new user)
```

```
    while True:
        with open(join(path,'data.json')) as jsonFile:
            time.sleep(0.5)
            data = json.load(jsonFile)
        latestID = str(data['accountID']) # load latest user account ID
doorbell paired with
        if latestID != currentID: # if account ID has been changed
            # reconfigure topics that the Raspberry Pi is subscribed to as the
doorbell's ID has been updated:
            client.unsubscribe(f"message/audio/{currentID}")
            client.unsubscribe(f"message/text/{currentID}")
            client.subscribe(f"message/audio/{latestID}")
            client.message_callback_add(f"message/audio/{latestID}",
playAudio)
            client.subscribe(f"message/text/{latestID}")
            client.message_callback_add(f"message/text/{latestID}", playText)
            currentID = latestID # set new values for accountID and currentID
for future comparisons
        time.sleep(5)

def on_connect(client, userdata, flags, rc):
    # callback function called when program connect to MQTT broker
    with open(join(path, 'data.json'), 'r') as jsonFile:
        time.sleep( 0.5)
        data = json.load(jsonFile)
        accountID = data['accountID']
    if rc == 0: # if connection is successful
        client.publish("audio", "ready")
        client.subscribe(f"message/audio/{accountID}") # subscribe to topic
'message/audio/accountID'
        client.message_callback_add(f"message/audio/{accountID}", playAudio) #
create callback for function 'playAuio' when message published to topic
'message/audio/accountID'
        client.subscribe(f"message/text/{accountID}")
        client.message_callback_add(f"message/text/{accountID}", playText)
        checkThread = threading.Thread(target=checkAccountID, args =
(accountID,client)) # create thread which checks whether account ID doorbell
paired with has been updated
        checkThread.start()
    else:
        # attempts to reconnect
        client.on_connect = on_connect
        client.username_pw_set(username="yrczhohs", password = "qPSwbxPDQHEI")
        client.connect("hairdresser.cloudmqtt.com", 18973)

while True: # block program until user set up doorbell is set up
    if os.path.isfile(join(path,'data.json')) == False: # cannot pair doorbell
with mobile app unless 'data.json' file has been created, as 'data.json'
stores doorbell's unique Smart Bell ID and user's account ID required for
mobile app to pair with doorbell
        time.sleep(5)
    else:
        with open(join(path,'data.json'), 'r') as jsonFile:
            time.sleep(0.5)
            data = json.load(jsonFile)
        if 'accountID' in data:
            break

while True: # block program until doorbell is connected to internet
    try:
        url.urlopen('http://google.com')
        break
```

```
    except:
        time.sleep(5)



client = mqtt.Client()
client.username_pw_set(username="yrczhohs", password = "qPSwbxPDQHEI")
client.on_connect = on_connect # creates callback for successful connection
with broker
client.connect("hairdresser.cloudmqtt.com", 18973) # parameters for broker web
address and port number
client.loop_forever()
```

pair_loop.py

```
import paho.mqtt.client as mqtt
import urllib.request as url
from os.path import join
import RPi.GPIO as GPIO # Import Raspberry Pi GPIO library
import os
import json
import time
import threading
import requests

path = "/home/pi/Desktop/NEA/ComputerScience-NEA-RPi"

serverBaseURL = "http://nea-env.eba-6tgviyyc.eu-west-2.elasticbeanstalk.com/"
# base URL to access AWS elastic beanstalk environment

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD) # use physical pin numbering
GPIO.setup(37, GPIO.IN, pull_up_down=GPIO.PUD_DOWN) # set pin 37 to be an
input pin and set initial value to be pulled low (off)

while True: # block program until user set up doorbell
    if os.path.isfile(join(path, 'data.json')) == False: # check if user has
already set up doorbell
        time.sleep(5)
    else:
        break

def on_message(client, userData, msg):
    # callback function called when pairing request received
    time_start = time.time()
    while True:
        if GPIO.input(37) == GPIO.HIGH: # check if user pressed 'pair' button
on doorbell
            connectDoorbell(msg) # pair doorbell with user account
            break
        elif time.time() - time_start > 60:
            break


def connectDoorbell(msg):
    # pair doorbell with user account
    with open(join(path,'data.json')) as jsonFile:
        data = json.load(jsonFile)
    SmartBellID = str(data['id']) # unique ID of doorbell
    accountID = msg.payload.decode() # user account ID that doorbell is to
pair with
    with open(join(path,'data.json'), 'r') as jsonFile:
        data = json.load(jsonFile)
    data['accountID'] = accountID
    with open(join(path,'data.json'), 'w') as jsonFile:
        json.dump(data, jsonFile)
    data_accountID = {"accountID": accountID, 'id': SmartBellID}
    paired = requests.post(serverBaseURL + "/update_SmartBellIDs",
data_accountID).text # REST API path to store new pairing connection in SQL
table 'SmartBellIDs'
    client.publish(f'pair/{accountID}', paired) # publish MQTT message to
topic 'pair/accountID' to notify mobile app that pairing has been successful
```

```
def checkID(currentID):
    # check whether doorbell ID has been updated
    while True:
        with open(join(path,'data.json')) as jsonFile:
                data = json.load(jsonFile)
        latestID = str(data['id']) # latest value of doorbell ID
        if latestID != currentID: # if doorbell ID has been changed
            SmartBellID = currentID = latestID
            # reconfigure topics that the Raspberry Pi is subscribed to as the
doorbell's ID has been updated:
            client.unsubscribe(f"id/{currentID}")
            client.subscribe(f"id/{SmartBellID}")
            client.message_callback_add(f"id/{SmartBellID}", on_message)
        time.sleep(5)


def on_connect(client, userdata, flags, rc):
    # callback function called when program connect to MQTT broker
    if rc == 0: # if connection is successful
        with open(join(path,'data.json')) as jsonFile:
            data = json.load(jsonFile)
        SmartBellID = str(data['id'])
        client.publish(f"connected/{SmartBellID}")
        client.subscribe(f"id/{SmartBellID}")
        client.message_callback_add(f"id/{SmartBellID}", on_message)
        checkThread = threading.Thread(target=checkID, args = (SmartBellID,))
# create thread which checks whether doorbell ID has been updated
        checkThread.start()

        checkThread.start()
    else:
        # attempts to reconnect
        client.on_connect = on_connect
        client.username_pw_set(username="yrczhohs", password = "qPSwbxPDQHEI")
        client.connect("hairdresser.cloudmqtt.com", 18973)


while True:
    try:
        url.urlopen('http://google.com')
        break
    except:
        time.sleep(5)


client = mqtt.Client()
client.username_pw_set(username="yrczhohs", password = "qPSwbxPDQHEI")
client.on_connect = on_connect # creates callback for successful connection
with broker
client.connect("hairdresser.cloudmqtt.com", 18973) # parameters for broker web
address and port number
client.loop_forever()
```

bluetooth_pair.py

```
#!/usr/bin/python3
import time
import RPi.GPIO as GPIO # Import Raspberry Pi GPIO library
import os
import wifi_connect
import pyautogui
import subprocess
import threading

GPIO.setwarnings(False)
GPIO.setmode(GPIO.BOARD) # use physical pin numbering
GPIO.setup(37, GPIO.IN, pull_up_down=GPIO.PUD_DOWN) # set pin 37 to be an
input pin and set initial value to be pulled low (off)


if os.path.isfile('/home/pi/Desktop/NEA/ComputerScience-NEA-
RPi/bluetooth/SmartBell.json') == True:
    os.remove('/home/pi/Desktop/NEA/ComputerScience-NEA-
RPi/bluetooth/SmartBell.json')

# make Raspberry Pi discoverable on boot.
os.system("""sudo bluetoothctl <<EOF
power on
discoverable on
pairable on
EOF
""")

devices = []
paired_devices = (subprocess.getoutput("""sudo bluetoothctl <<EOF
paired-devices
EOF""")).split('Device ')[1:] # list of paired devices

for paired_device in paired_devices: # iterate through paired devices
    devices.append(paired_device[0:17]) # create list with addresses of paired
devices

for device in devices: # iterate through paired devices and remove pairing
    command = ("""sudo bluetoothctl remove {}""").format(device) # remove
device pairing
    os.system(command)

def pair():
    # pair Raspberry Pi with PC over bluetooth
    pyautogui.write("0000") # pairing code
    pyautogui.press("enter") # press enter key
    time.sleep(2)
    pyautogui.press("enter")
    pyautogui.press("enter")
    start = time.time()
    while time.time() - start <120:
        path = '/home/pi/Desktop/NEA/ComputerScience-NEA-RPi/bluetooth/'
        if len(os.listdir(path)) != 0: # check if file with wifi details
received by Raspberry Pi
            wifi_connect.run() # connect to WiFi
            break


pyautogui.keyDown("ctrl")
```

```
pyautogui.keyDown("alt")
pyautogui.keyDown("d")

pyautogui.keyUp("ctrl")
pyautogui.keyUp("alt")
pyautogui.keyUp("d")

while True:
    if GPIO.input(37) == GPIO.HIGH and threading.active_count() == 1: # if
'pair' button pressed AND Raspberry Pi not currently pairing with a device
        thread_run = threading.Thread(target =pair)
        thread_run.start()
```

wifi_connect.py

```
import json
import os
import time
from os.path import join

path = "/home/pi/Desktop/NEA/ComputerScience-NEA-RPi"

def run():
    filePath = join(path, 'bluetooth') # filepath to open 'SmartBell.json'
file sent my PC over bluetooth storing wifi connection details
    file = join(filePath,str(os.listdir(filePath)[0]))
    with open(file, 'r') as f:
        data = json.load(f)
        mySSID = data['ssid'] # WiFi network SSID
        passkey = data['psswd'] # WiFi network password
        SmartBellID = data['id'] # unique SmartBell ID
    for file in os.listdir(filePath):
        os.remove(os.path.join(filePath, file))

    newData = {"id": SmartBellID, "training": False} # dictionary storing
SmartBell ID sent by user from PC and set 'training' to False (default)

    if os.path.isfile(join(path, 'data.json')) == False: # if the doorbell is
being set up for the first time
        with open(join(path,'data.json'), 'w') as jsonFile:
            json.dump(newData, jsonFile) # store SmartBell ID in 'data.json'

    elif SmartBellID != "": # if the user has sent a new SmartBell ID from
their PC
        with open(join(path,'data.json'), 'r') as jsonFile:
            data = json.load(jsonFile)
        data['id'] = SmartBellID # assign updated SmartBell ID
        with open(join(path,'data.json'), 'w') as jsonFile:
            json.dump(data, jsonFile)

    if mySSID != '': # if user has sent a new SmartBell ID from their PC
        try: # try except statement required as only needs to kill the
wpa_supplicant process if there is one running
            os.system('sudo killall wpa_supplicant') # kills the
wpa_supplicant process
            time.sleep(5)
        except:
            pass
        command = (('wpa_passphrase "{}" "{}" | sudo tee -a
/etc/wpa_supplicant/wpa_supplicant.conf').format(mySSID, passkey)) # appends
the correctly formatted network data to the WiFi configuration file
'wpa_supplicant'
        os.system(command) # execute command through terminal
        time.sleep(5)
        os.system('sudo wpa_supplicant -B -c
/etc/wpa_supplicant/wpa_supplicant.conf -i wlan0') # wpa_supplicant
automatically selects best network from 'wpa_supplicant.conf' to connect with
and runs the WiFi connection process
```

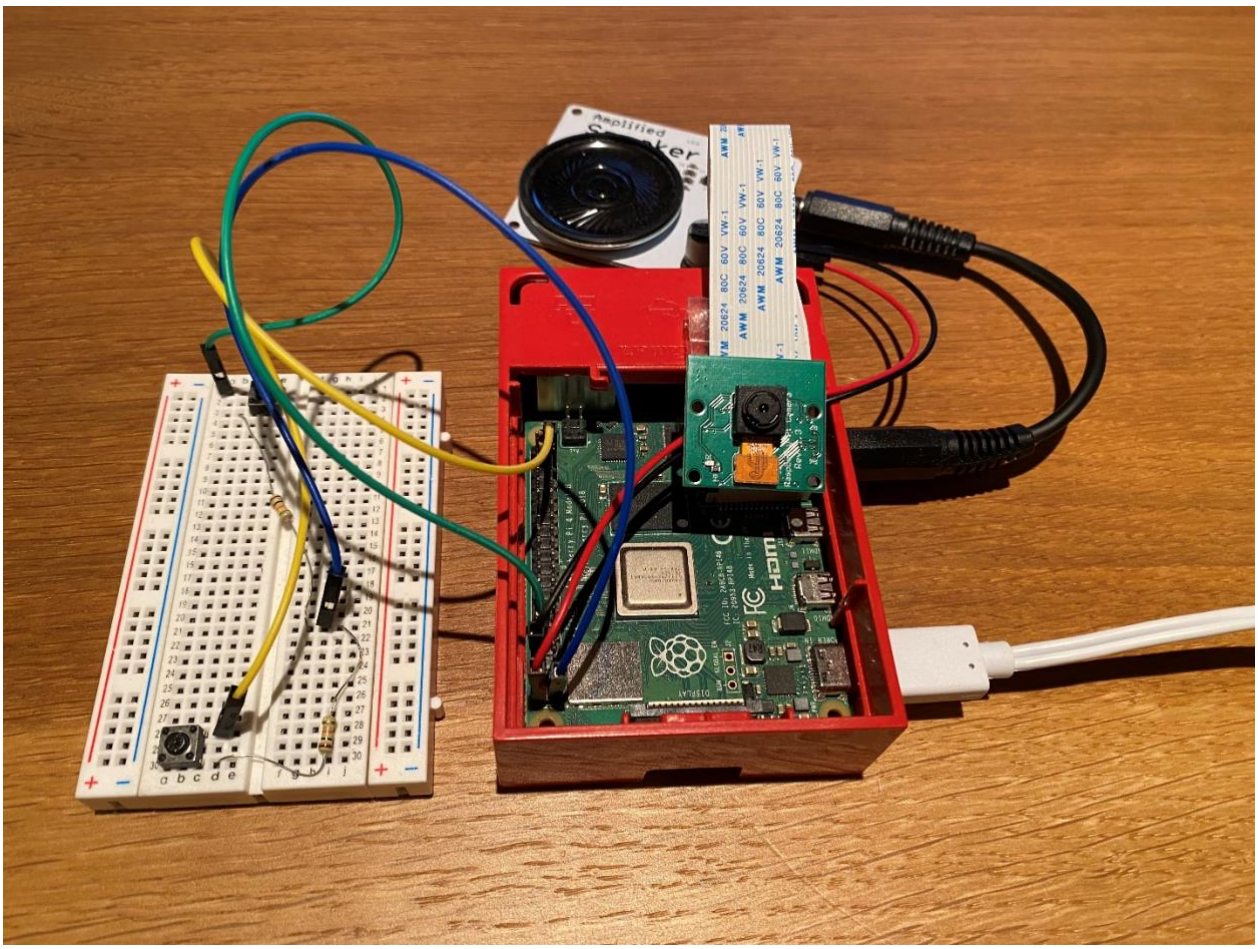
Personal Computer (WiFi Setup)

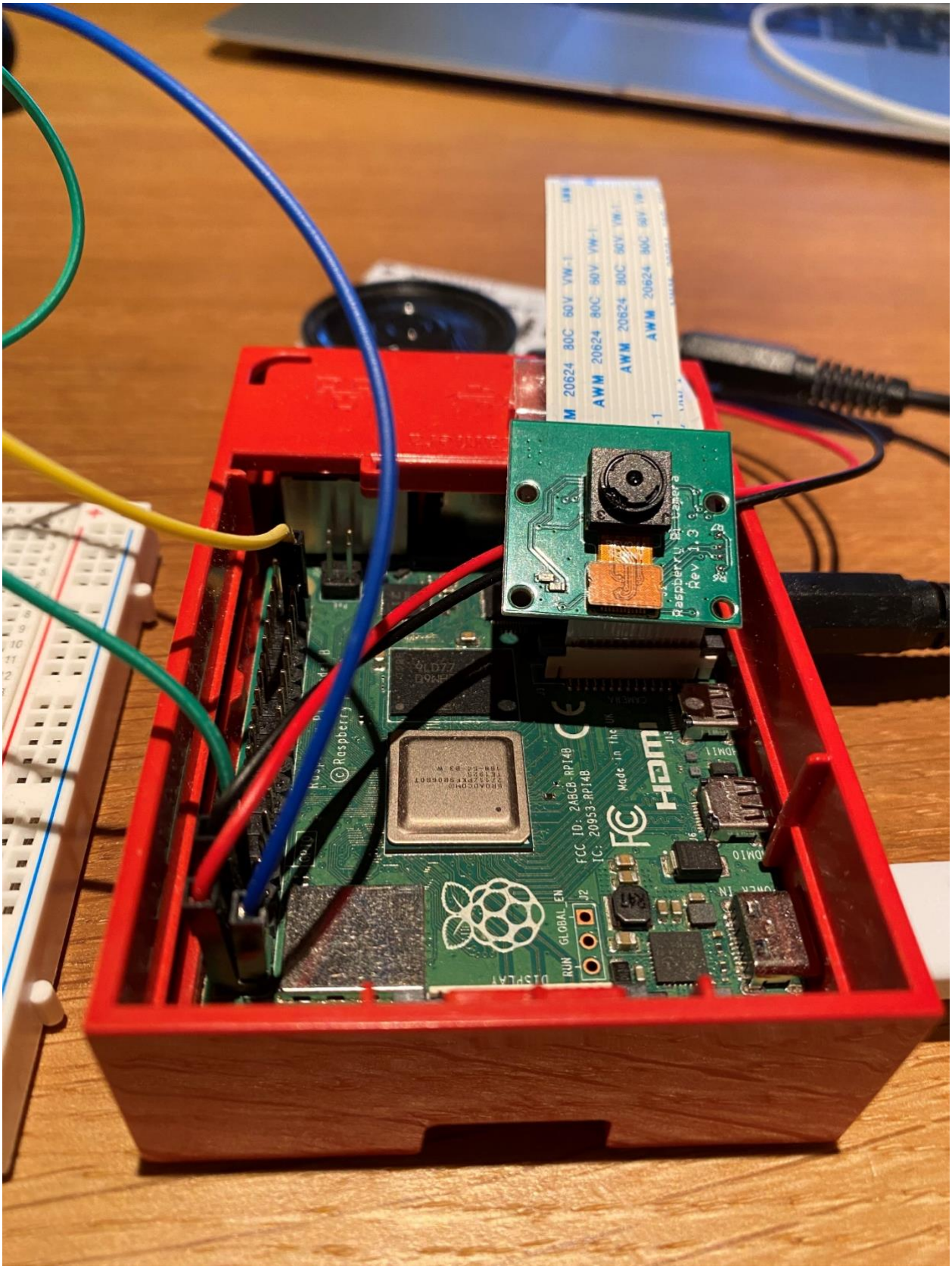
wifi setup.py

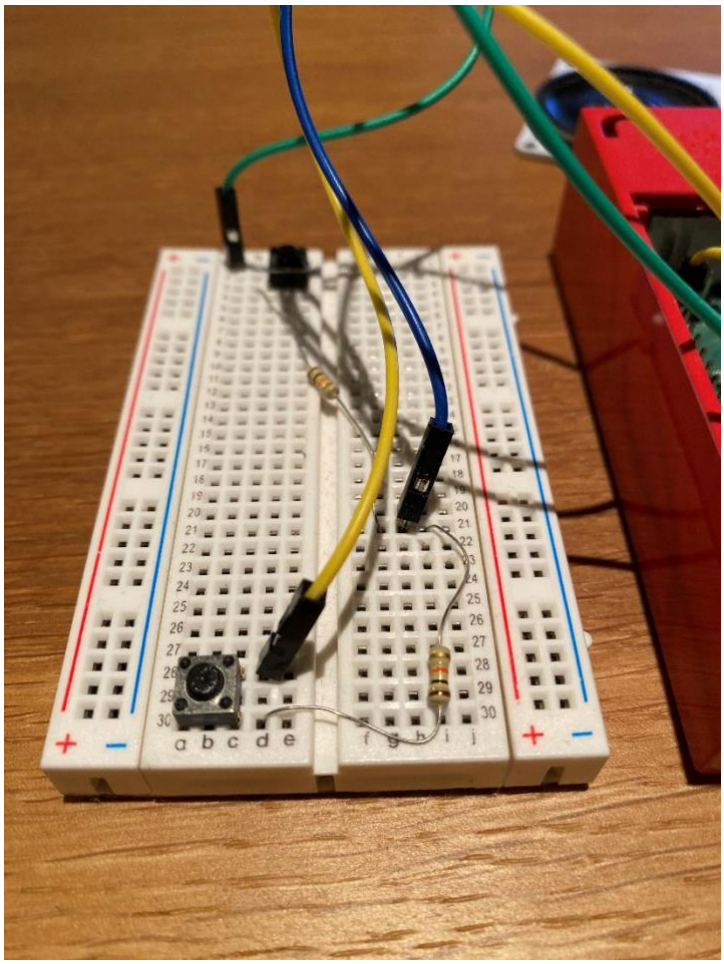
```
import json
import time
import paho.mqtt.client as mqtt
import threading
import requests

serverBaseURL = "http://nea-env.eba-6tgviyyc.eu-west-2.elasticbeanstalk.com/"
# base URL to access AWS elastic beanstalk environment
initialSetup = False
wifi = False
ssid = ''
psswd = ''


def wifi_connected(client, userData, msg):
    print('\nSmartBell has succesfully connected to the WiFi!')
    time.sleep(5)
    quit()


def wifi_notConnected():
    while True:
        time.sleep(30)
        print('\nSmartBell is not yet connected to the WiFi. Please try
again.')


def on_connect(client, userdata, flags, rc):
    global SmartBellID
    if rc == 0:
        client.subscribe(f'connected/{SmartBellID}')
        client.message_callback_add(f'connected/{SmartBellID}',
wifi_connected)
    else:
        client.username_pw_set(username="yrczhohs", password="qPSwbxPDQHEI")
        client.on_connect = on_connect  # creates callback for successful
connection with broker
        client.connect("hairdresser.cloudmqtt.com", 18973)  # parameters for
broker web address and port number


print(
    "Welcome to your SmartBell!\nThis guide will walk you through the process
of connecting your SmartBell to the internet.")

if input("\nIs this the first time setting up your SmartBell? (y/n) ").lower()
== "y":
    initialSetup = True
elif input("\nWould you like to set up a new WiFi connection for your
SmartBell? (y/n) ").lower() == "y":
    wifi = True

response = 'exists'
if initialSetup == True:
    SmartBellID = str(input("\nPlease enter a unique name for you SmartBell:
```

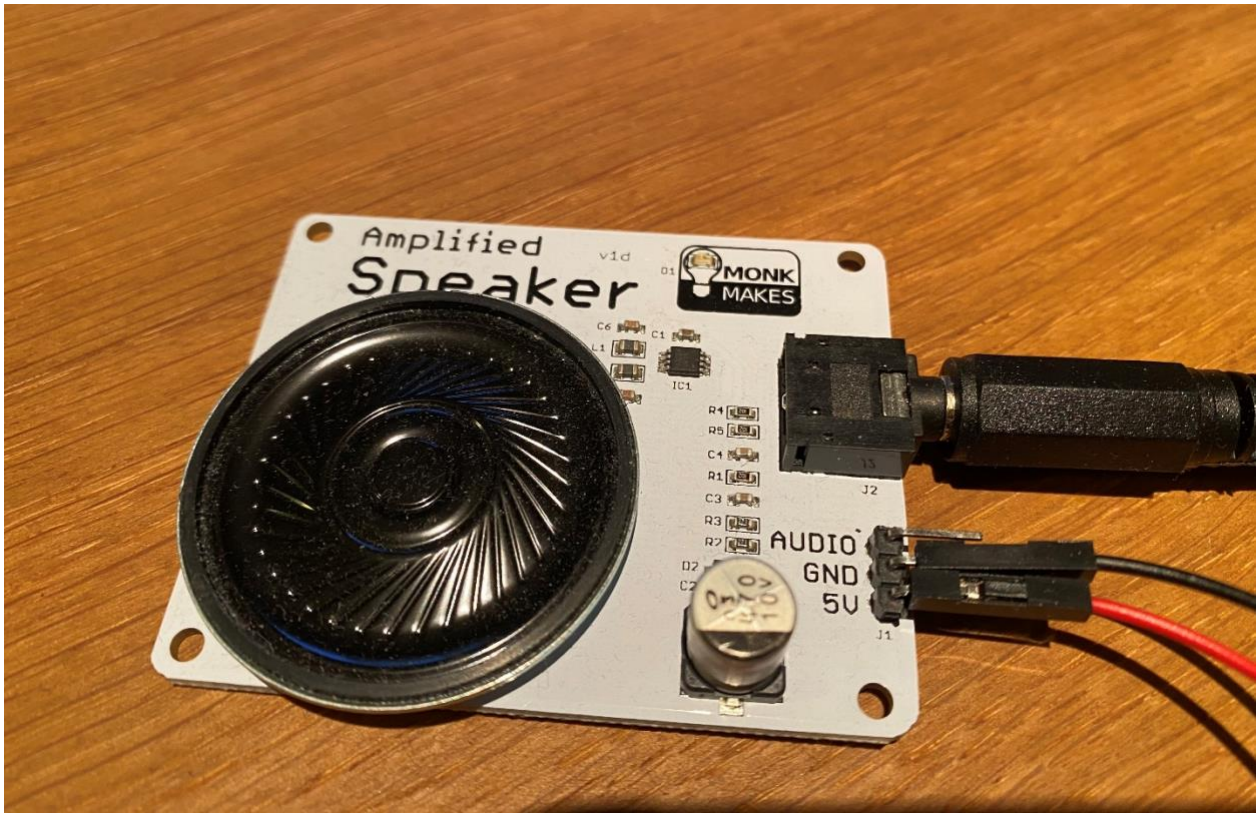
```
"))
else:
    SmartBellID = str(input(
        "\nPlease either enter a new unique name for your SmartBell or,
alternatively, leave this field blank to keep its previous unique name: "))

while response == 'exists':
    dbData_id = {"id": SmartBellID}
    response = (requests.post(serverBaseURL + "/verify_SmartBellID",
dbData_id)).text
    if response == 'notExists' and SmartBellID != '':  #  only store the id if
it is unique and isn't an empty string (i.e. the user wants to keep their
original id)
        r = requests.post(serverBaseURL + "/update_SmartBellIDs", dbData_id)
    else:
        SmartBellID = str(
            input(f"Sorry, the name '{SmartBellID}' already exists. Please
enter another name for your SmartBell: "))

if initialSetup == True or wifi == True:
    ssid = input('\nPlease enter the network SSID you would like to connect
to: ')
    psswd = input(("Please enter the passkey for your network with SSID '{}':
").format(ssid))

data = {}
data['ssid'] = ssid
data['psswd'] = psswd
data["id"] = SmartBellID

with open('SmartBell.json', 'w') as f:
    json.dump(data, f)

print(
    "\nPlease now select 'SmartBell' from the list of available bluetooth
devices on your laptop and then and then press the 'WiFi connect' button on
your doorbell.\nOnce you have successfully paired with your SmartBell, please
transfer the file called 'SmartBell.json' to the SmartBell doorbell.")

thread = threading.Thread(target=wifi_notConnected)
thread.start()

client = mqtt.Client()
client.username_pw_set(username="yrczhohs", password="qPSwbxPDQHEI")
client.on_connect = on_connect  # creates callback for successful connection
with broker
client.connect("hairdresser.cloudmqtt.com", 18973)  # parameters for broker
web address and port number

client.loop_forever()
```

SmartBell Photos




Raspberry Pi Camera
Rev 1.3
FCC ID: 2ABCB-RPI4B
IC: 20953-RPI4B
Made in the
GLOBAL_EN
RUN
DISPLAY




Amplified
Speaker
MONK
MAKES
AUDIO
GND
5V

Database Screenshots

audioMessages

messageID	messageName	messageText	accountID
GvJajwNaR9VPUEzY	Test audio	Null	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
aaxurDZWdGTgtc0A	Test text	This is my test audio re...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
yNAnCKJ79xeynOLS	Hello	Null	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
0mwef1QnLrrqAQjl	Instructions	Hello there. Please leav...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
u1bYDa6b6dV7VtXR	Final message	Goodbye	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
NDEeomkwYcqjl5vn	helloo	Null	i3f6nFtd92J5TI07g3vSgbaQ06FnDfL7lJnbeEpVy8Z=
ukTbMoA8CnexiWW7	brief hello	Hi there!	i3f6nFtd92J5TI07g3vSgbaQ06FnDfL7lJnbeEpVy8Z=
hb6AWhCleC4e3jvG	Waitrose	Null	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...

knownFaces

messageID	messageName	messageText	accountID
GvJajwNaR9VPUEzY	Test audio	Null	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
aaxurDZWdGTgtc0A	Test text	This is my test audio re...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
yNAnCKJ79xeynOLS	Hello	Null	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
0mwef1QnLrrqAQjl	Instructions	Hello there. Please leav...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
u1bYDa6b6dV7VtXR	Final message	Goodbye	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...
NDEeomkwYcqjl5vn	helloo	Null	i3f6nFtd92J5TI07g3vSgbaQ06FnDfL7lJnbeEpVy8Z=
ukTbMoA8CnexiWW7	brief hello	Hi there!	i3f6nFtd92J5TI07g3vSgbaQ06FnDfL7lJnbeEpVy8Z=
hb6AWhCleC4e3jvG	Waitrose	Null	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...

SmartBellIDs

id	accountID
Test	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2RndYCroz...

users

accountID	firstName	surname	email	password
7Tufeg2uAu1vWD7rliBORECXI6HUVCyfDSFEuXf8F...	Orlando	Alexander	aorlando04@gmail.com	e6c513262cc2db19e62ff646d2ff116727662176e...
x2laTtKfo2JwGiHNGBFV1uKEBRBGq3J12Sq2WQI2...	Orlando	Alexander	@.com	e6c513262cc2db19e62ff646d2ff116727662176e...
VOPUV3DNLNAWL2TAeZGLfmKkE4s4L3MIzYCku7...	O	A	@.co	e6c513262cc2db19e62ff646d2ff116727662176e...
gTYNISOi522Zv5kqfs2O5W7yptzniHCFD5LfpBwxN...	orlando	sz	@.con	e6c513262cc2db19e62ff646d2ff116727662176e...
3XtOZvISDbrbAWZSei4zHiLPreWRCBQIxW3fefD5X...	h	b	@.hdhs	4d82cbe10a7285d9950331175302995f14128922...
IXeZzgE2NOqzENckzZDNmiWxtWQUYX7xgDvTSm...	h	b	@.hdhbs	4d82cbe10a7285d9950331175302995f14128922...
yeZq45dy2XcFuUigy8NfTtB2vXxFMA1WJv50Gwys...	u	b	@.£c	2cfa9d47edf3c363f08444e7cd7c43123234f3722...
4HztSjAYLzFgPsJzzF5wff5yxRPbNdfP4HthG96gCPI=	h	b	@.c	e6c513262cc2db19e62ff646d2ff116727662176e...
X8esXF0ycO5jZYHObywSEa8Zmh44oHCXEyliLIs82...	g	b	b@.	685980f543597fade07e005f16abfadf449b087efe...
zwNmlKhsx8g7wMdAlzZmJFkbUaoHoaqjl7Int9iKjG...	h	b	@.jehe	6b9ff38ba91cd4dd996b04820dcd21d7e097513e...
LSaOdbPE1BEHC2kYx3FZktYemq23aTGiFdJsWi3V...	h	b	@.jehen	6b9ff38ba91cd4dd996b04820dcd21d7e097513e...
2TY07Guly79forkBSXsnVTOrXb2UFF8Aboht9KQhA...	bb	bb	@/!37n.	762be92403acb5be8328346a781fb5c227017092...
zvMYqQnUdsodVarXtpBQKMO3Z2wA1f8h8xA9gV...	bb	bb	@/!37n.h	762be92403acb5be8328346a781fb5c227017092...
IyuKToP0tkKoeFNinIgr3fzaDDgMwC5fw47ONWuW...	bb	bb	@/!37n.hh	762be92403acb5be8328346a781fb5c227017092...

visitorLog

visitID	imageTimestamp	faceID	accountID
mfF4jW79OaCEapkhDMTVFmqYSZvpPeJHqzlkf6w...	01.20,1646875245.1...	NO_FACE	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R
ROdHHtQlCMTOwa9NuYfMacySmeG6bdbHa2wOUI...	01.21,1646875283.1...	NO_FACE	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R
GyBI37QOh6xUifE3YSGdYRkvnFqQrUWFffRfxHVxN...	04.10,1646971806.1...	ScQQv3eWBaiUXMcMmdA36jQJKX5KmAj0P5SDnJ...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R
KnFv0w9NmOfZaCxljTCYlB6GHho1A7NtORu3sGov...	05.07,1646975231.5...	x9HLlkQ73CltVVjETT1s28BGepCd7Uc0QVhevhLB...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R
Gw1I73PSGVmdm2LHUvTKe1rWfIbWQgafAbBx5JC...	05.07,1646975272.6...	x9HLlkQ73CltVVjETT1s28BGepCd7Uc0QVhevhLB...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R
hAxl9Jip3CKnjCtXFowu08ek6T32EQ838h3hl5iae2t=	05.08,1646975324.9...	ScQQv3eWBaiUXMcMmdA36jQJKX5KmAj0P5SDnJ...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R
yvf8zvLkWYFTHOtulyqQB7qyES9MLz8b1pKZh2z3l...	05.15,1646975734.6...	ScQQv3eWBaiUXMcMmdA36jQJKX5KmAj0P5SDnJ...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R
6gAS39bMKr5ps9MOTk6TlkHvVMU2vlv99fvgva8o...	05.16,1646975773.1...	V8BplLavmGeDvvyblZGleKLQhKbruNH7VamUMUlo...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R
LfgTL28AnvohUZFRQmZdEhMmlrrt624Y2oYkUwGo...	05.18,1646975916.7...	G8YfHYXY0Vx3xhnbkXSmf5KCZsnCMW7CSULiarH...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R
IKO8gOWxMbXUXH7APX84nJb3mAtUd5gQQCqRQ...	21.23,1647206633.6...	ScQQv3eWBaiUXMcMmdA36jQJKX5KmAj0P5SDnJ...	eSezbXEUclv9g5GK65lAfmeodH8h3KQ2R