Jarvis: A Virtual Assistant That Won't Sell Your Data

Developer Documentation

Emmy Shi, Eric Ahn, Gina Wang, Jason Park, Sam Feng, Will Kohn

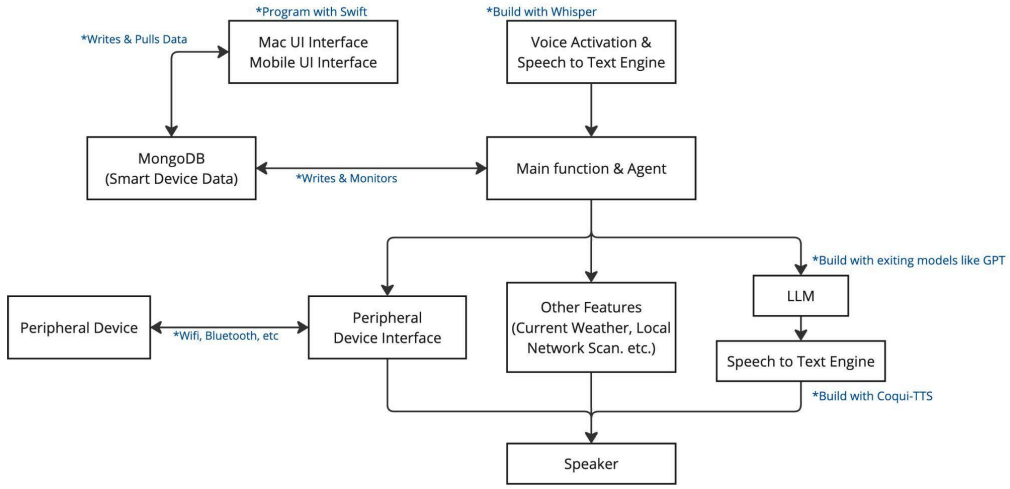
This document serves as the developer documentation for Jarvis the Virtual Assistant. It contains information crucial for developers who want to expand this project further. The documentation is separated into Project Overview, frontend documentation (MacOS Interface documentation, iOS Interface documentation), and backend documentation. For the code base, please visit this GitHub repo.

Project Overview

Jarvis is a product similar to preexisting Smart Home managers and virtual assistants but with one major distinction: your data isn't being collected and sold to advertisers. Once the product is activated, users can speak to Jarvis, and it will respond, doing its best to answer user queries. Jarvis can also manage smart devices in your home without you ever having to touch a keyboard. The entire system will be customizable including the voice and the name of the virtual assistant, named Jarvis by default.

Backend Documentation

This section includes everything on the backend that powers Jarvis and will be divided into the following subsections: MacOS Version & Windows Version, LLM-Powered Assistant, Smart Devices Control, and Other Features. Jarvis requires a Python version equal to or above 3.10 and lower than 3.12 for anyone running the program locally and wants the full functionality. The rest of the dependencies depend on the user's operating system, which we will talk about later in the documentation. The workflow for the backend is generally as shown in the below image:



**Mac UI Interface & Mobile UI Interface are Frontend components and can be disregarded in this section

The rest of this section dives deeper into the purpose of each part shown in the image, and the components they contain.

1. **MacOS Version & Windows Version**

    a. Dependencies

    Due to the structure mentioned above, users and developers need to download different dependencies. There are two files supporting the easy installation of dependencies: **"requirement-mac.txt"** and **"requirement-win.txt"**. Based on the operating system the user has, users can install all dependencies via one of the following commands:

    - pip install -r requirements-mac.txt
    - pip install -r requirements-win.txt

    The virtual assistant also provides functions to scan the user's local network and to connect to real sense depth cameras. For the assistant to scan the user's local network, the nmap software is needed and requires the user to independently download on their local machine as of now. If users wish to connect to real sense depth cameras, users will need to obtain the hardware and connect to their local device independently. This is a beta function and is currently still under testing and development.

    b. Purpose & Functions

    The virtual assistant connects to the user's local device's input and output device, which vary between Operating Systems. As a result, we have developed the following folder structure. The root folder "jarvis" contains the "MacVersion" folder and the "Windows" folder. It also includes main.py, updateDb.py, and watchDb.py. updateDb.py and watchDb.py are related to controlling smart devices and will be disregarded in this section. The main function in the root folder is responsible for detecting the type of user's operating system, then directing the program toward the correct folder - "MacVersion" or "Windows". Any other operating system is not supported in the current version.

    The main.py file in the "MacVersion" and the "Windows" folder achieves the same things. The functions provided via the main.py file for Mac users include the beta version of local network scanning and camera connecting, whereas those for Windows users are fully developed.

    c. Components

    In the MacVersion and the Windows folder, there are several folders: clips, temp_chunks, and yolov5. The clips folder stores pre-generating wave files utilized in the main function. The temp_chunks folder stores wave files that get recorded and documented while the virtual assistant is operating. The audio files stored in the temp_chunks folder are then deleted while the virtual assistant shuts down. The yolov5 folder contains the object detection model utilized in this project and can be disregarded by developers and users.

    For main.py in MacVersion:

    i. main()

    This is the main function for the user if the user is using a MacOS. Once entered,

it first plays two wave files to signal the user that Jarvis is getting activated. It then initiates other components Jarvis utilizes and connects to MongoDB through the watchMongo function. After that, the program enters an infinite while loop which keeps on listening to its surroundings via the listen() function and executes the corresponding case the user requests via the execute(prompt, response_num) function. The while only exists if the virtual agent hears “Jarvis, shut down” with Jarvis representing the activate word in this case. Once the while loop exists, the program starts shutting down the program, including clearing out any wave files written and stored in the temp_chunks folder under the MacVersion folder.

ii. play_audio(file_path)
This is a helper function that allows the virtual assistant to work with MacOS’s output systems. By taking in a parameter file_path that contains the file path to the wave file the user wants the virtual assistant to output, the play_audio function allows the virtual assistant to “speak” out what was in the file.

iii. chunk_tts(n)
This is a helper function that takes in a wave file and then transcribes the wave file into text. The function is utilized in the listen function in the case that the db variable (local variable in the listen function) is larger than a certain threshold.

iv. listen()
This is the function that automatically gets activated when the virtual assistant (Jarvis) is started. This function takes care of listening to its surrounding sounds and also takes care of detecting the activate word, the name parameter, which is by default set to “Jarvis”. In this function, the program will not enter another while loop if the activated word is detected.

v. get_response(prompt)
This function is responsible for detecting which action the virtual assistant should activate. Possible actions (cases) include: starting a video feed, scanning the user’s local network, controlling smart home devices, checking the current weather of instructed locations, and answering any general user prompts such as “How many colors does a rainbow have?”. Some functions are currently its beta version. This function then gets called in the execution(prompt, n) file.

vi. execution(prompt, n)
This function is responsible for playing either a pre-generated wave file or a custom response wave file via the play_audio(file_path) function after calling the get_response(prompt).

For main.py in Windows:

i. main()
This generally serves the same purpose as the main.py file in the MacVersion folder. The difference is that it doesn’t start listen to surroundings and call listen() by default. Instead, this part of the functionality is outsourced to the function

main_backend() and main() checks if the users defined a UI variable, meaning that the likely have a video camera attached to the system.

ii. main_backend()
This is the function that contains the while loop which contains the listen function, the respond function, and also detects whether the user wants to shut the virtual assistant down. Same as the main function in the MacVerison folder, this function cleans up the wave files created under the temp_chunks folder.

iii. respond(prompt, n)
This function is the equivalent of the execution(prompt, n) function in the Mac version. Please see the documentation above for what it does.

iv. get_response(prompt)
This function serves the same purpose as that in the MacVersion folder. However, all cases and possible actions under this Windows version is fully developed.

v. listen()
Again, This function generally serves the same purpose as that in the MacVersion folder. Please see the above documentation for what it does. However, the function detects whether the name of the virtual assistant (activate word) was said first. If the name of the virtual assistant is detected, the function prints "listening for prompt", and then starts listening to the user command or query.

vi. get_chunk(l)
This is a helper function that is called in the listen() function. This function takes in a parameter indicating the number of audio segments to read, then collect audio data into the variable "frames", joins all frames into a single byte chunk, then checks the volume of decibels. It then returns two variables: "chunk" containing the audio data and "dec" containing the decibel level of the chunk.

vii. name_tts(n)
This is a helper function that is called in the listen() function and is responsible for detecting whether the activate word (defaulted to Jarvis) was said via translating audio data into text.

viii. prompt_tts()
This is a helper function that is called in the listen() function and is responsible of translating audio data into text to capture the user prompt after this function is called.

2. LLM-Powered Assistant

a. Purpose & Functions: This is the default case whenever a query is detected from the user. A query is detected via Whisper, an OpenAI product designed to convert speech to text, and is identified whenever the name of the virtual assistant is spoken. By default that

name is Jarvis. Once the query is identified, it's passed through GPT 3.5 Turbo alongside an explanation that it is serving as a virtual assistant. That result is converted back to audio using Coqui-TTS and played aloud to the user. This process is standard throughout except for the use of the LLM. In certain edge cases, the LLM is skipped and the queries are handled separately as described in the **Other Features** section.

b. Prerequisites:
   i. **OpenAI API Key**: the API key is obtainable via the website. The API key should then be stored as your environment variable named "OPENAI" via this command:

      Mac: export OPENAI="YOUR-API-KEY"

      Windows: setx OPENAI YOUR-API-KEY

c. Components:
   i. **listen** is the function responsible for getting the prompt. First, it waits for the name of the virtual assistant to be called. It constantly listens to chunks of audio using **pyaudio**, converting them to text using **Whisper** and checking if the virtual assistant name has been said. To limit the computational cost, it also checks that the chunk has a decibel level greater than 45 before evaluating it. Once the name has been said, an audio indicator is played via **playsound** and it begins listening for the prompt. It listens until the decibel level drops below 45 then translates the audio to text and returns it.
   ii. **get_response** is in charge of creating the audio reply. Given the prompt, it searches for keywords to see if it meets the criteria described in **Other Features**. If it doesn't hit any of the special cases, it is passed to a remote GPT-3.5-Turbo model. The model's response is then extracted from the query and returned.
   iii. **respond** is the parent method of **get_response**. It calls **get_response** and captures the reply. It then converts the text response back into speech using **Coqui-TTS**, and plays it aloud using **playsound**.

3. Smart Device Control

a. Environment Variables
   i. **Connection String**: the connection string is obtainable via website. The connection string should then be stored as your environment variable named "MONGO" via this command:

      Mac: export MONGO="YOUR-STRING"

      Windows: set MONGO="YOUR-STRING"

b. Purpose & Functions
the purpose of this script is to manage smart device controls within the Jarvis virtual

assistant system. It provides voice-activated and app-based controls, updates the device status in a MongoDB database, and synchronizes actions with real-time database changes. The script enables seamless interaction between user commands, database operations, and device responses.

c. Components:
    i. **device_control** allows us to control smart devices with two methods
        1. Core Methods
            a. **device_control_voice** processes voice commands to control devices (lightbulb). When prompted with keywords about home devices, the method calls the appropriate function(light_turn_on or light_turn_off), updates the Jarvis database with updateDb, and returns the final status of the device
            b. **device_control_app** controls devices based on the input from the database document. It turns the lightbulb on or off based on the status field. For unsupported actions, it returns an appropriate message to notify the user.
    ii. **updateDb** allows us to update the status of devices stored in a MongoDB database as part of the Jarvis virtual assistant system. The main method, updateDb(id, status), accepts two parameters: the id, a string representing the unique ObjectId of a device document in the MongoDB collection, and status, which must be either "on" or "off". The function updates the device's status in the database. It returns true if the update is successful or false if the update fails due to invalid inputs, no matching document, or connection errors. The function also handles exceptions such as invalid ObjectId formatting, connection issues, or unexpected errors.
    iii. **watchDb** enables real-time synchronization between database changes and device control actions, ensuring that the Jarvis system responds promptly to updates. Specifically, this method allows developers to access the Jarvis database by connecting to the MongoDB Atlas cluster using a connection string, such as "Devices" and "Rooms" collections. When a change is detected and the status field has been updated, the function retrieves the updated device document and calls device_control_app to execute any changes made
d. Limitations: the script's functionality is limited to controlling a lightbulb. The device_control functions and watchDb monitoring process are designed specifically for managing the status of lights. Commands and database updates support turning the light on or off. Expanding support for additional devices would require modifying the device_control logic and updating the database.

4. Other Features

a. **Shut down**: searches for either "shut down" or "shutdown" in the prompt. If found, it exits the entire program after scrubbing all temporary audio files to maintain the user's privacy.

b. **Activate video feed**: searches for “video feed” in the prompt. If the UI is set up with a compatible Intel RealSense Depth Camera and the key word is found, it enables the UI and streams a live feed of the RGB camera, Depth camera, and YOLO model identifying objects in the image. When this mode is activated, it becomes the main thread since tkinter refuses to work otherwise.

c. **Tell me about yourself**: searches for “tell me about yourself” in the prompt. If found, it plays a paragraph long audio recording describing what Jarvis is.

d. **Scan network**: searches for “scan network” in the prompt. If found, it runs an nmap scan on your local network, identifying how many devices are currently active. This can be easily edited to run a port scan and provide more specific information. This requires you to download nmap from its official website and restart your computer.

e. **Turn on/off the light**: searches for “light” in the prompt. If found, it changes the light to either on or off based on the rest of the prompt using a request sent to **lifx** (the lightbulb brand). It also updates the current status of the lightbulb to the MongoDB database.

f. **Get the weather**: searches for “weather” in the prompt. If found, it extracts the city name by looking for it after “in” or “for”. This may seem pretty weak, but it hasn’t yet failed. If it does, it just asks you to reiterate. Otherwise, it queries **openweathermap**, which returns a comprehensive json containing all relevant information. This also can be easily altered to include different information beyond temperature and current conditions. For this feature to work correctly, you need an OpenWeatherApp API Key. The API key is obtainable via the OpenWeatherMap website. The API key should then be stored as your environment variable named “WEATHER” via the below command:

   Mac: export WEATHER="YOUR-API-KEY"
   Windows: setx WEATHER YOUR-API-KEY

g. **Other possibilities in development**: We currently are developing options to find the current value of a stock ticker and plan out routes from one location to another. They’re both pretty feasible, but likely won’t be completed within the semester.

5. **Future Development**

The main short-term future goals for the team include establishing a stable connection with the iOS mobile application and continuing development of the smart device connection. More specifically, in the current version, the connections to specific smart devices are hard-coded. In the next version, we plan on prompting the user for their specific device ID to increase customization. We also plan on increasing the number of actions users can take regarding smart devices. For example, in the current version, users can turn the light bulb on and off, and we plan on introducing other features such as changing the color of the light bulb and scheduling a time to turn on or off the light bulb. In addition, we plan on increasing compatibility with other smart home device. Currently, the system is compatible with the LFINX lightbulb. We plan on

expanding the connection to other smart devices provided by LFINX, such as the LFINX smart switch. We also plan on expanding our connection to other brands, such as Kasa Smart.

Frontend Documentation

This section includes both the MacOS App Documentation and the iOS App Documentation. The technology used in both components includes Xcode, Swift, and JSON.

Part 1: MacOS App Documentation

1. Version Requirement: MacOS 14.6 or Higher

2. Purpose and Functions:
   The purpose of the macOS-based frontend app is to seamlessly integrate with a MongoDB database to visualize and help users manage their rooms and devices. The core functionality includes the ability to **add**, **edit**, and **delete** rooms and devices, and **synchronize** data with the database. Users can create new rooms, associate devices with them, and edit device attributes such as name, type, and status via a user-friendly interface. The application leverages alerts and contextual menus for intuitive interactions while maintaining robust data handling, ensuring that all operations, including syncing to and from MongoDB, are secure and error-free.

3. Components:
   The macOS app includes 5 main components that are integrated together to manage the visualization and manipulation of data.

   a. **Data**
      The Data component includes the files **rooms.json** and **devices.json**. These JSON files store all the app's room and device information locally, acting as intermediaries between the macOS app and the MongoDB database. They are critical for persisting data changes locally and enabling smooth synchronization during import and export operations.

   b. **Models**
      The Models component includes the **Room.swift** and **Device.swift** files. These define the data structures for rooms and devices, including attributes like room names, device types, statuses, and associated relationships. These models ensure consistent formatting for data, facilitating smooth serialization to and from JSON and compatibility with the MongoDB schema.

   c. **Utilities**
      The Utilities component includes the following files:

      i. **export_mongo.sh** and **import_mongo.sh**
         These shell scripts enable synchronization with the MongoDB database by importing and exporting room and device data to/from the local JSON files.

ii. **JSONHelper.swift**:
This helper file provides utility functions, such as loadJSON, for reading and decoding JSON data from local files. It simplifies data handling throughout the app and ensures compatibility with the Codable protocol for models.

d. **ViewModels**
The ViewModels component includes **RoomViewModel.swift**. This file manages the business logic of the app, such as adding, editing, and deleting rooms and devices, and synchronizing data with MongoDB. It uses utilities such as **JSONHelper.swift** to load and save data, keeping the UI responsive and the data consistent.

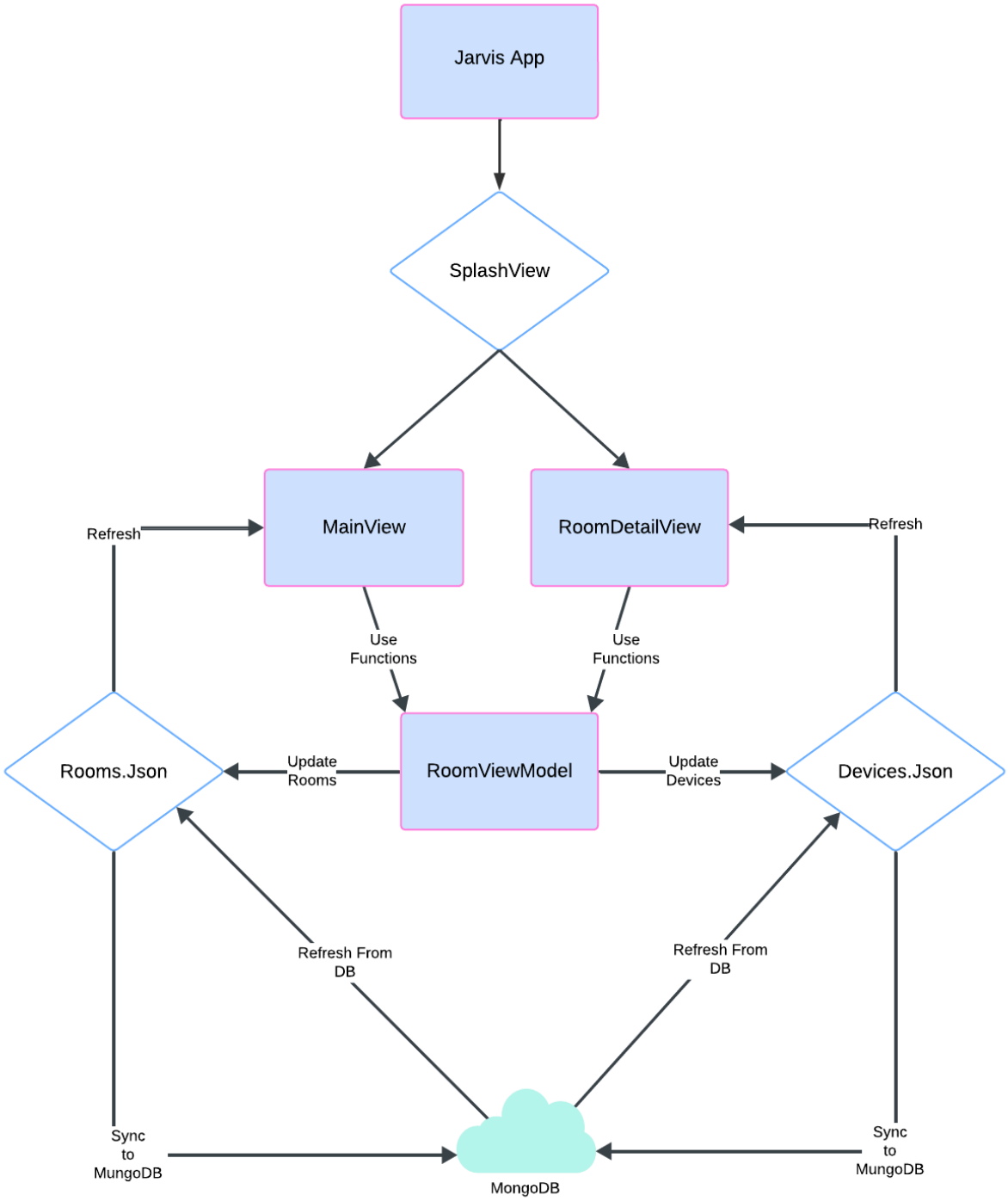
e. Views
The Views component includes **MainView.swift, SplashView.swift,** and **RoomDetailView.swift**.

i. **MainView.swift** provides the primary interface for listing and managing rooms.

ii. **SplashView.swift** provides a welcome page to users while loading data from the MungoDB.

iii. **RoomDetailView.swift** allows users to manage devices within a specific room. Together, these views create an interactive platform for visualizing and modifying data, tightly integrating with the ViewModels and Utilities.

4. Workflow Prototype:



The workflow of the Jarvis app begins with **SplashView**, which transitions the user to the **MainView** for managing rooms or the **RoomDetailView** for managing devices. Both views utilize the **RoomViewModel** to handle business logic, such as adding, editing, and deleting rooms or devices. Data changes made in the views are reflected in the local **Rooms.json** and **Devices.json** files through the **RoomViewModel**. The app synchronizes with the MongoDB database via shell scripts, enabling the import and export of data.

Users can refresh the views to load the latest data from MongoDB or sync their local changes to the database, ensuring consistency across the system. This interconnected architecture efficiently manages data visualization and manipulation.

5. Future Direction:

   Customizable Status Edit Function: Currently the macOS version provide only text edit for status, which provide maximum flexibility to users. However, that is also less intuitive and user friendly. We will implement customizable status edit function where users can declare their devices' capability and their status options, so they can later change the status in more user friendly ways, instead of entering text.

Part 2: iOS App Documentation

1. Version Requirement: iOS 16 or Higher

2. Purpose and Functions:
   The purpose of the iOS-based frontend app is to be used as an extension version for more potential users who prefer using mobile phones to control their home devices. This app contains all the features and user interfaces; however, the current version lacks connectivity with the backend database. Our immediate future goal is to connect the iOS app with the backend to further help more users employ Jarvis in a flexible manner.

3. Components:
   Since the iOS version lacks connectivity with the database, so in this technical document, we mainly focus on the View components. The Views components contain the primary user interface files that define how users interact with the app. There are mainly 8 view components:

   a. **MainView.swift** is the primary dashboard for managing rooms and devices. It could display a list of rooms, provide options to add, edit, or delete rooms, and navigate users to access detailed views.

   b. **RoomDetailView.swift** allows management for devices within a selected room. It lists all devices in the selected room and allows users to add or remove devices.

   c. **AddDeviceView.swift** allows users to add new devices to specific rooms. It contains a form for specifying device attributes.

   d. **DeviceDetailView.swift** provides detailed information to users on and settings for a single device. It would display device-specific data such as the status and type. It would also allow users to edit device attributes or have specific actions.

   e. **DeviceManagementView.swift** would help device-level operations across the app. Its function, along with RoomDetailView, would allow for a comprehensive device management experience.

   f. **SplashView.swift** guides the users starting from a welcome screen while initializing the app.

g. **WelcomeView.swift** serves as an information loading page, allowing the users to have access to room view and device view.

h. **RoomListView.swift** helps the users to manage the display and interaction of the list of rooms within the app

4. Workflow:
   The workflow of the Jarvis iOS app begins with the SplashView, which serves as a welcome screen for users while initializing the app. Once the initialization is complete, the app transitions to the MainView, acting as the central hub for managing rooms. In the MainView, users can view a list of rooms displayed using the RoomListView, and they can select a room to navigate to its details or perform operations like adding or deleting rooms. When a specific room is selected, the app directs the user to the RoomDetailView, where all devices within the selected room are displayed. Here, users can add new devices, edit existing devices, or delete devices as needed. If the user chooses to add a new device, they are navigated to the AddDeviceView, where they can input the required details, such as the device name and type, and save the device to the room. Additionally, users can access detailed information about individual devices in the DeviceDetailView, allowing them to view or modify device-specific attributes.

5. **Future Goals & Limitations**: Below we list out the limitations and future goals of the current iOS version.

   a. Lack of Backend Connectivity
      The iOS app currently is not connected to a MongoDB backend. Developers will need to refine based on the current version to develop an advanced version of the iOS app. Therefore, for the future development, we are going to focus more on connecting iOS app with the database to provide a flexible environment for users

   b. No User Session
      The current app needs more features to set user accounts to allow for a more customized experience for individual users. For future goals, the app developers are going to create account features for users to have a more customized user experience.