50.001: Information Systems & Programming

NoBleaches, a full stack IoT solution (with Android app) by Group 55

1. Members

Student ID	Student Name
1006955	Matthew Lalonde-Low Jiaxin
1007107	Mohammad Saif Zia
1007198	Muhammad Ammar Bin Mohamad Sofian
1007488	Samuel Roshan
1006019	Long Raphael James
1007494	Mithunbalaji Mageswari Ganeshkumar

2. Background

1. Problem

There are 3 housing blocks for the students of SUTD, where each block has a dedicated laundry room for clothes to be washed and dried. Students regularly face inconveniences when they enter the laundry room. Either the machines are in use, or the clothes have not been emptied out, or the machine requires servicing. Thus, the inconclusiveness about the normal functioning of the machines makes planning difficult.

Having experienced these troubles ourselves, our problem statement for our project was framed as follows: “How might we make the laundry system at the SUTD Hostel blocks more convenient for hostel residents?”

2. Solution

Given the reliance SUTD students have on their mobile devices, building an application for users would be favourable since they can be notified right when their clothes are available to pick up, they can find out what machines are free and make reservations 15 minutes ahead to

save time. An application that allows most of the work to be done remotely rather than having to physically enter the vicinity would greatly save time.

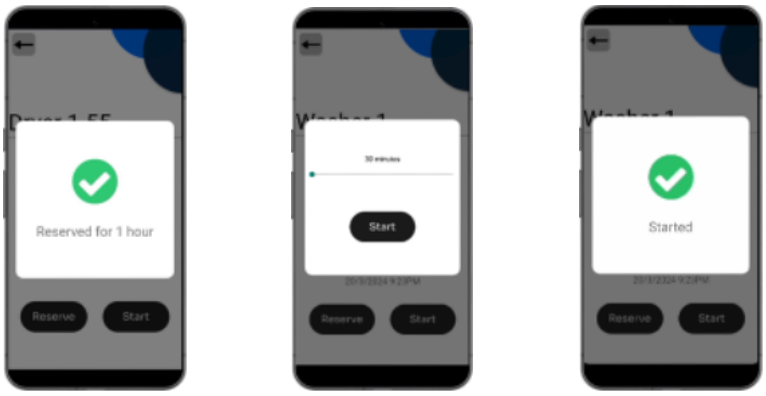
The application, named NoBleaches, was built for mobile devices using Android as their Operating System. Android Studio was used to build and develop the app, using Java for giving the app functionality and XML for enabling user-computer interaction. (Briefly include backend architectures used)

NoBleaches was built to assist with sustainability issues in mind. A reservation history feature was baked in to identify the demand for washers and dryers, recognize which machines required frequent servicing and what machines were avoided. Having the luxury of checking machine status from afar wastes less resources on the user's part, which helps reduce crowding and congestion in each hostel block's laundry room.

Moreover, NoBleaches' User Interface (UI) is simple and has a very low learning curve to use the application's features. This makes the application accessible to even those with minimal technological experience.

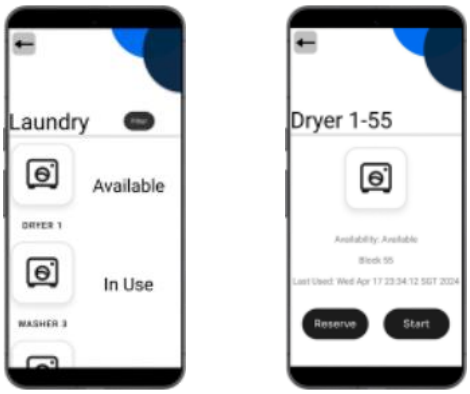
3. Prototype Summary

- Status System



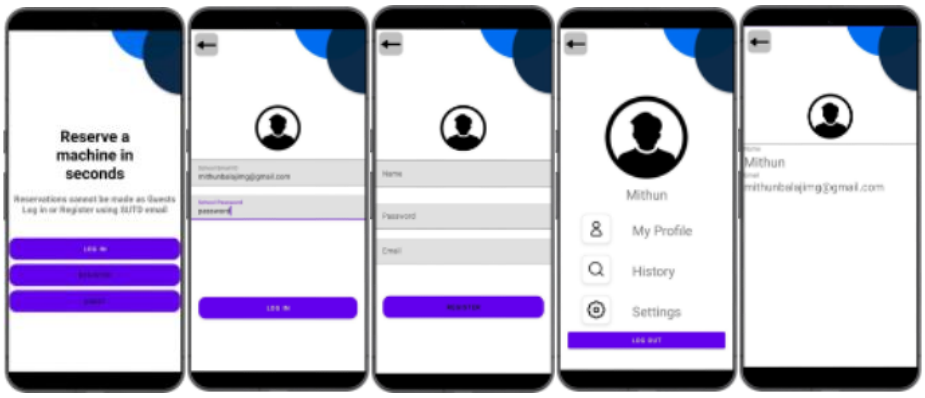
In Figure 1, user can scroll and view the statuses of every machine. In Figure 2, user can click into a machine of choice to get more information of the machine such as, machine's block and the last time it was used.

- Booking System



In Figure 2, User can Reserve or Start an Available machine, if the machine is not available, a toast will inform the user “Machine is not Available”. Figure 3 tells the user when a reservation is successful and the duration its reserved for. Figure 4 uses a line seeker widget for users to adjust the duration of their wash/dry, the duration of the line seeker will be shown in the text above (“30 minutes”) as the user scrolls the line seeker. After the user presses the start button, Figure 5 will be shown informing the user they have successfully started a machine.

- User System



In Figure 6, a new user will open the app to this authentication page. The “LOG IN” button will open Figure 7 the log in page for the users to key in their credentials and log in. The Register button will open Figure 8 for the users to key in their credentials and register their profile into the database, registering will automatically log in to the application. Figure 9 is the in-app profile page of the user; My Profile will open figure 10 showing all profile photo and credentials.

- Filter System

In Figure 4, user can click the “Filter” button to open figure 11 filter page, where they can select the respective blocks and machines they want to view in figure 4 machine statuses.

- History System

In Figure 9, user can click the “History” button to access Figure 12, their history of machines that they have used in the past and its information.

4. Android App Design and Implementation

The following visual should give a comprehensive layout for visualization

Thus, for each block, there is a central PC. (Represented by one MachineDataUpdater thread in the demo code). Several Machines report to this PC via a local MQTT topics (The MQTT portion of the code, along with the ESP32 has been replaced with code for an emulator, to view the demo without an ESP32 microcontroller. The ESP32s were purchased purely for demo purpose)

1. Front End Client (Android App):

- Authentication.java – Code for the user login authenticator
- BookingRequest.java – Booking request object that is sent to the realtime db
- GlobalConfig.java – Global Singleton for setting config (explained later)
- History.java – Activity that has the History page
- HistoryAdapter.java – Adapter code for the list of HistoryRecord objects
- Home.java – Home activity
- Laundry.java – Activity that has the realtime updating list of machines
- Login.java – Login activity
- Machine.java – Activity that shows each individual machine once it's clicked on Laundry.java
- MachineAdapter.java – Adapter code for parsing MachineData objects into the RecyclerView in Laundry.java
- MachineData.java – Each laundry machine as an object. Has variables to store machine data.
- MachineUpdateListener.java – An interface that demands that onMachineDataUpdate be implemented. Serves as the connector joint for MachineUpdateService.java to refresh the adapter in laundry once the list of machines is updated in the background.
- MachineUpdateService.java – Big Boy Chonk Background Service that exists within the lifecycle of Laundry.java. It automatically scans for changes in the realtime database under ./Machines and updates the internal List of MachineData objects if any changes do happen.
- Profile.java – Profile activity
- Register.java – Register activity
- Settings.java – Settings activity (display only – no functionality yet)
- UserData.java – Stores the users data, such as userId etc.
- UserDataManager.java – Shared Preferences implementation, general user data management

- The remaining are deprecated and not used. We forgot they were still there because there's so many things to keep track of.

2. **Firebase Email-linked Authentication and Acessing User Information [1]**

To implement the login via email and password-based authentication, Firebase Authentication SDK was used. Firebase helps provide a secure and scalable authentication system allowing NoBleaches to offload the management of user credentials to Google's infrastructure. By following the Firebase Guide, the ability to sign up new users and sign in existing users via addOnCompleteListener() is achieved. Even accessing the User Information can be done by instantiating a FirebaseUser and checking the currentUser instance inside MyProfile.Java. For security reasons, only the name and the email address are displayed on the app.

Upon sucessful login or signup, appropriate toast messages are displayed to indicate successful authentication into the firebase system. Now the users may navigate through the app by clicking on the Profile or the Laundry buttons.

The difference in app navigation between a guest user and a user signed into our firebase authentication system is that when a guest User tries to access their User Information by clicking on MyProfile inside our app, they are redirected to the Register page as they don't have any information accessible about their user instance. Moreover, their machine reservation history is not stored inside the History page as only logged in Users may book the machines.

3. **Firebase Realtime Database Usage**

Firebase Realtime Database, a noSQL database that represents it's data as a JSON tree - is used by the app in two ways:

- Each machine listed in ./Machine is used as the list of machines to be read by MachineAdapter to show in recycler view.
- Whenever a request to make a booking is made, a BookingRequest object is sent to ./BookingRequests for backend processing.

The RealtimeDB is mainly used for intermediate data transfer and is not meant to be used for long term storage, hence ./BookingRequests doesn't always exist, as booking requests are automatically moved to a Firestore instance once they have been processed. Similarly, if a machine has been removed, it is deleted from ./Machines.

4. **Firestore Database Usage**

The Firestore database is currently used to store booking request history from the user. In the app's current iteration, the History is common for all users, as we lacked the necessary funding

to automate collection creation per user. It reads through a collection of documents, each storing a BookingRequest object that can be read by HistoryAdapter.

(However, user-specific history can be easily implemented, as user UUID is preserved in every BookingRequest object, we will get to this later in backend)

5. **GlobalConfig**

Arguably the most important module in the application, GlobalConfig is a singleton class that facilitates the instantiation, storage and retrieval of key application API object instances. All required API's are called at launch, and only **referenced** for updates later. This helps with memory efficiency, thread consistency, and simplified access in other modules overall.

6. **Realtime Machine Status**

The real-time machine status updates feature is responsible for keeping the list of available machines up-to-date in the app. It involves the following key components:

1. MachineUpdateService class: This is a background service that periodically fetches the latest machine data from the Firebase Realtime Database and updates the app's machine list accordingly.
2. MachineUpdateListener interface: This interface defines a method onMachineDataUpdate that is used to notify the implementing activity or fragment about updates to the machine data.
3. Laundry activity: This activity implements the MachineUpdateListener interface and receives updates to the machine data from the MachineUpdateService.
4. MachineAdapter and machine_adapter.xml: The MachineAdapter is responsible for displaying the list of machines in a RecyclerView. The machine_adapter.xml layout file defines the UI elements for each machine item, such as the machine name and status.

Here's how the real-time machine status updates feature works:

1. When the Laundry activity is created, it starts the MachineUpdateService and binds to it using a ServiceConnection.
2. The MachineUpdateService periodically fetches the latest machine data from the Firebase Realtime Database using the GlobalConfig.getMachineFullList()method.
3. When the machine data is updated, the MachineUpdateService notifies the Laundry activity by calling the onMachineDataUpdate method defined in the MachineUpdateListener interface.

4. The Laundry activity receives the updated machine data in the onMachineDataUpdate method and applies any necessary filtering based on the user's selected filters (e.g., block number, machine type).
5. The filtered machine data is then passed to the MachineAdapter, which updates the RecyclerView with the latest machine information.
6. The MachineAdapter binds the machine data to the corresponding views in the machine_adapter.xml layout file, displaying the machine name and status to the user.
7. When the Laundry activity is destroyed, it unbinds from the MachineUpdateService and stops the service to conserve resources.

The real-time machine status updates feature allows the app to display the most up-to-date information about the available machines, ensuring that users can make informed decisions when selecting a machine for their laundry needs.

7. Booking Requests

The booking request feature is responsible for creating a new booking request for a specific machine. It involves the following key components:

1. BookingRequest class: This class represents a booking request and contains the following properties:
    - bookingUUID: A unique identifier for the booking request.
    - userUUID: The UUID of the user making the booking request.
    - machine: The MachineData object representing the machine being booked.
    - requestTime: The date and time when the booking request was made.
2. GlobalConfig class: This class is a singleton that holds global configurations and references for the application. It includes a reference to the Firebase Realtime Database instance and a reference to the "BookingRequests" node where new booking requests are stored.

When a user wants to make a booking request for a machine, a new BookingRequest object is created with the selected MachineData. The BookingRequest object is then serialized to JSON using Gson and stored in the "BookingRequests" node of the Firebase Realtime Database using the bookingUUID as the key.

8. **History Feature**

The history feature displays a list of previous booking requests made by the user, along with their status and other relevant information. It involves the following key components:

1. HistoryRecord class: This class represents a single history record and contains the following properties:
    - bookingUUID: The unique identifier of the booking request.
    - userUUID: The UUID of the user who made the booking request.
    - machine: The MachineData object representing the booked machine.
    - requestTime: The date and time when the booking request was made.
    - processedOn: The date and time when the booking request was processed (optional).
2. History activity: This activity retrieves the booking history data from the Firestore database and displays it in a RecyclerView using the HistoryAdapter.
3. HistoryAdapter and item_history.xml: The HistoryAdapter is responsible for binding the data from each HistoryRecord object to the corresponding views in the item_history.xml layout file. This layout file defines the UI elements for each history item, such as the machine name, request date and time, processed status, and reserved time.

When the History activity is launched, it fetches all the documents from the "booking-history" collection in Firestore. For each document, it creates a HistoryRecord object and populates its properties with the data from the document. These HistoryRecord objects are added to a list and passed to the HistoryAdapter, which then binds the data to the corresponding views in the RecyclerView.

The HistoryAdapter handles formatting and displaying the various date and time values, as well as the machine name and other relevant information for each history record.

Overall, the booking request feature allows users to create new booking requests for machines, while the history feature retrieves and displays the previous booking requests made by the user, along with their status and other relevant details.

9. **Encapsulation**

Encapsulation is heavily evident in the MachineData.java class. Its responsibility was to represent a machine. Therefore, having getter and setter methods to retrieve and update the attributes of each machine makes the code in this file reusable for other IoT-related applications in retrieving status of other products.

Getter Methods like getName(), getStatus() and getBlock() and setter methods like setStatus() and setLastUsed() are fairly self-explanatory. Other classes only need to know how to use the methods of MachineData, providing an understandable level of abstraction.

10. Design Principles and Patterns

Here are some of the design principles and patterns observed in the provided code:

1. **Singleton Pattern**: The GlobalConfig and UserDataManager classes are implemented as singletons, ensuring that only one instance of each class exists throughout the application's lifecycle.
2. **Separation of Concerns**: The codebase follows the principle of separating concerns by dividing functionality into different classes and activities. For example, the BookingRequest class encapsulates the data related to a booking request, while the HistoryRecord class represents a history record.
3. **Model-View-Controller (MVC) Pattern**: The application follows a loose implementation of the MVC pattern. The activities (Home, Laundry, Login, Register, etc.) act as the controllers, managing user interactions and updating the corresponding views. The data models (UserData, MachineData, BookingRequest, HistoryRecord) represent the application's data entities.
4. **Observer Pattern**: The MachineUpdateService and MachineUpdateListenerinterface demonstrate the Observer pattern. The Laundry activity implements the MachineUpdateListener interface and receives updates from the MachineUpdateService whenever the machine data changes.
5. **Firebase Integration**: The application heavily relies on Firebase services, such as Firebase Authentication, Firebase Realtime Database, and Firebase Firestore, for user authentication, data storage, and real-time updates.
6. **Dependency Injection**: The UserDataManager class is injected into various activities and classes, allowing for better testability and maintainability.
7. **Adapters and Recycler Views**: The MachineAdapter and HistoryAdapterclasses are used in conjunction with RecyclerViews to efficiently display lists of data.

However, there certainly are areas for improvement, such as better separation of concerns between the activities and the data models, and adherence to coding standards and naming conventions. Additionally, the application could benefit from better error handling and input validation. (Even though text inputs after login are basically nil)

11. Logcat Statements and Toast Notifications

Throughout the stages of NoBleaches' development, Logcat statements were used to verify whether the program flows as expected and whether the values of attributes were updated correctly.

Toast Notifications were used as a primary form of providing feedback as users interact with NoBleaches. The notifications are gentle, non-intrusive and encourages users to interact with NoBleaches more since the feeling of confusion is minimized.

5. Backend Implementation

The backend server, also a gradle application interacts with a Kafka Cluster (which the block computers listen to) and the database, effectively serving as a central 'middle man'. The following are it's assigned tasks:

1. **Tasks:**

- **Polling Machine Data from Kafka Topics:**

The backend server polls three Kafka topics (TOPIC1, TOPIC2, and TOPIC3) representing different blocks (55, 59, and 57) for machine data updates.

It uses a ScheduledExecutorService to periodically poll these topics at a fixed rate (every 2 seconds).

The polled machine data is stored in a local machineDataMap for further processing.

- **Updating Firebase Realtime Database:**

After polling the Kafka topics, the backend server updates the "Machines" node in the Firebase Realtime Database with the latest machine data from the machineDataMap.

This operation ensures that the mobile app can retrieve the most up-to-date machine information from the Realtime Database.

- **Listening for Booking Requests:**

The backend server listens for new booking requests in the "BookingRequests" node of the Firebase Realtime Database.

When a new booking request is detected, the server performs the following tasks:

Schedules a message to be sent to the "booking-requests" Kafka topic after a delay (15 seconds in the provided code).

Adds the booking request to the "booking-history" collection in Firebase Firestore.

Removes the booking request from the Firebase Realtime Database after processing.

- **Parsing and Processing Booking Requests:**

The backend server deserializes the booking request data received from the Realtime Database using Gson.

It creates a BookingRequest object and sets the processedOn field with the current timestamp.

The BookingRequest object is then scheduled to be sent to the "booking-requests" Kafka topic after a specified delay.

- **Sending Messages to Kafka:**

The backend server sends messages to the "booking-requests" Kafka topic using a KafkaProducer.

These messages contain a custom KafkaBookingRequest object, which includes the booking UUID, machine block, and machine name.

2. **Server Scalability**

After comments from Singtel Engineers that concerned server scalability during deployment, strategies were also made for scaling in a real-life deployment scenario.

- The backend server can be *vertically scaled* by increasing the computational resources (CPU, RAM, etc.) of the machine or server instance it runs on.
- This approach can handle an increased load to a certain extent, but it has limitations due to the physical constraints of a single machine.
- The backend server can also be *horizontally scaled* by running multiple instances of the server across multiple machines or cloud instances, essentially providing more cores to the same service.
- This can be achieved using load balancers or cluster management tools to distribute the incoming requests across multiple server instances, such as how a Kafka cluster runs inside a Apache Zookeeper container by default.
- Speaking of which, Kafka provides built-in support for horizontal scaling by allowing the addition of more brokers and partitions to handle increased message throughput.
- Firebase Realtime Database and Firestore also support horizontal scaling by automatically distributing data across multiple nodes and providing automatic partitioning and replication.

To accommodate more users and handle increased load, a combination of vertical and horizontal scaling techniques can be employed. However, it's essential to consider factors such as the expected load, traffic patterns, and cost-effectiveness when deciding on the appropriate scaling strategy.

6. Laundry Machine Emulator:

Since ESP32 cannot be used for anything other than a demonstration, and setting up a local MQTT server is a hassle of its own, an emulator that emulates the endpoints of a machine that runs entire on java is in the repo under ./machinedemo. Here's what it essentially does:

- **Polling Machine Data from Kafka Topics:**

The MachineDemo component serves as a conduit between the Kafka Cluster, where block computers communicate, and the database. It actively polls three distinct Kafka topics, namely TOPIC1, TOPIC2, and TOPIC3, representing machine data updates from blocks 55, 59, and 57, respectively. Leveraging a ScheduledExecutorService, it initiates periodic polling at a fixed rate of every 2 seconds. The retrieved machine data is stored locally within a machineDataMap for further processing.

- **Updating Firebase Realtime Database:**

Upon successfully polling the Kafka topics, the MachineDemo component promptly updates the "Machines" node within the Firebase Realtime Database. This operation ensures the synchronization of machine data with the Realtime Database, facilitating access to the latest information for associated services and applications.

- **Listening for Booking Requests:**

In tandem with its other responsibilities, the MachineDemo diligently listens for new booking requests within the Firebase Realtime Database's "BookingRequests" node. Upon detection of a new request, it executes a series of actions:

- Schedules a message to be dispatched to the "booking-requests" Kafka topic after a predetermined delay (as specified, 15 seconds).
- Records the booking request in the "booking-history" collection within Firebase Firestore for archival purposes.
- Subsequently removes the processed booking request from the Firebase Realtime Database, ensuring data cleanliness and efficient handling.

- **Parsing and Processing Booking Requests:** Upon receiving booking request data from the Realtime Database, the MachineDemo employs Gson for deserialization purposes. It constructs a BookingRequest object and annotates it with the current timestamp as the processedOn field. The finalized BookingRequest object is then scheduled for dispatch to the "booking-requests" Kafka topic following a predefined delay.
- **Sending Messages to Kafka:** Equipped with a KafkaProducer, the MachineDemo component efficiently dispatches messages to the "booking-requests" Kafka topic. These messages encapsulate a custom KafkaBookingRequest object, containing crucial booking details such as the UUID, machine block, and machine name.

7. Division of Responsibilities

Student Name	Responsibility
Matthew Lalonde-Low Jiaxin	Application UI, History feature
Mohammad Saif Zia	User Authentication feature, Backend Development (Firebase), presentation
Muhammad Ammar Bin Mohamad Sofian	Machine Status feature, report and presentation
Samuel Roshan	Filtering features, report and documentation
Long Raphael James	Application UI, Filtering feature
Mithunbalaji Mageswari Ganeshkumar	Backend Development (Kafka/Firebase/Custom server), Realtime Updates, Ecosystem Integration

8. Financial Report

Product	Quantity	Overall Cost	Purpose
ESP32	6	SG$ 33.30	Suitable for IoT due to Wi-Fi & Bluetooth capabilities built in, unlike its competitors
Micro-USB Charging Cable	6	SG$ 10.91	Supply power to each ESP32

Out of the $80 granted to our group, $44.21 was used for this project while the rest of the funds were diverted to our 50.001 Computational Structures 1D project. Our group aimed to allocate most of our funds for the latter project since the nature of this project is mainly software-oriented and required no use of paid services. The event streaming platform of our choice (Apache Kafka) was free to use.

9. Conclusion

10. References

[1] Firebase. "Firebase Authentication for Android." Firebase Documentation. [Online]. Available: https://firebase.google.com/docs/auth/android/start. [Accessed: 4/10/2024].