Mobile Device Programming (COMP3040 UNMC)

Design of ‘Instatt’

Name: Liqi Tang
Student ID: 20259468
Lecturer: Dr Nabil El Ioini

Content

Introduction:

Instatt is a comprehensive mobile app specifically designed to enhance the process of managing attendance in educational institutions. Its purpose is to offer an intuitive, efficient, and precise solution for instructors to log attendance and for students to keep track of their attendance record. This reduces the likelihood of manual errors and saves valuable time.

In educational environments, tracking attendance is crucial but often comes with challenges. Conventional systems can be inefficient, error-prone, and time-consuming, which can hinder the main goals of education. Instatt addresses these issues head-on. The app aims to transform the way attendance is managed in schools and colleges. By leveraging digital technology, Instatt overcomes the drawbacks of traditional attendance methods. It offers an easy-to-use and accurate way to monitor attendance, aiming to lessen the workload for teachers and ensure students have instant access to their attendance data. This promotes a transparent and responsible educational atmosphere.

Instatt is a harmonious fusion of simplicity and advanced technology, making it accessible to everyone, regardless of their technological proficiency. For educators, Instatt simplifies the attendance recording process, and for students, it provides up-to-the-minute attendance information. This not only streamlines the attendance tracking process but also supports a more structured and efficient learning environment. Instatt is more than a mere attendance recording tool; it is a thoughtfully designed response to a long-standing educational challenge. Embracing mobile technology, Instatt aims to redefine the standards in attendance management, positioning itself as a dependable, time-saving component in contemporary education systems.

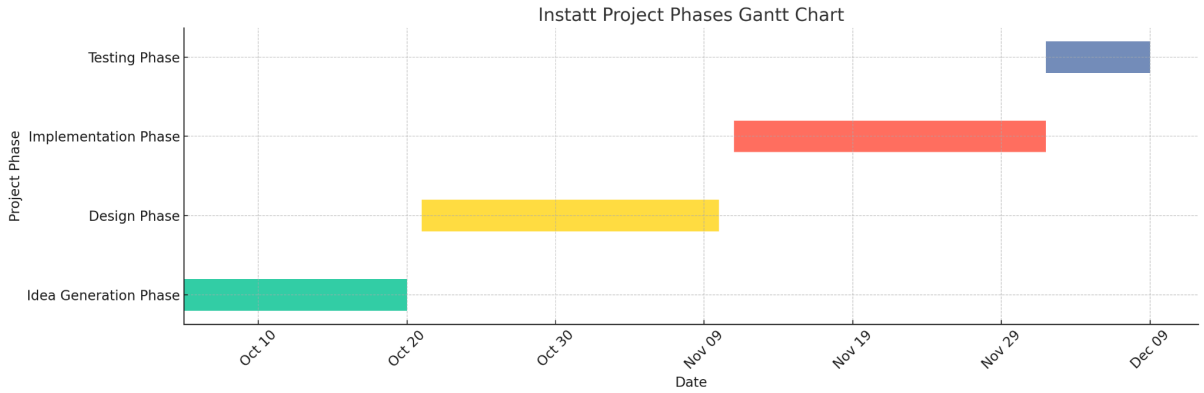
In this application, I utilised a simulated Pixel 2 device, emulating Android API 14.0.

Project Planning and Management:

1. Idea Generation Phase (October 5, 2023 - October 20, 2023):
    a. Milestone 1: Identifying a Real-World Problem
    b. Milestone 2: Analysing Instatt for Potential Improvements and Issues
    c. Milestone 3: Finalising Enhancement Strategies and Overcoming Challenges for Instatt
2. Design Phase (October 21, 2023 - November 10, 2023):
    a. Milestone 4: Completing the Analysis of Proposed Improvements
    b. Milestone 5: Defining and Describing the Functionality of Each Component
    c. Milestone 6: Formulation of User Stories

d. Milestone 7: Securing Approval for UML Diagrams and UI Wireframes
e. Milestone 8: Designing the User Interface with Figma
f. Milestone 9: Establishing Assumptions for App Utilisation

3. Implementation Phase (November 11, 2023 - December 2, 2023):
    a. Milestone 10: Constructing Core Features like User Authentication and Class Management
    b. Milestone 11: Incorporating Firebase for Secure Data Storage and Authentication
    c. Milestone 12: Implementing the User Interface
4. Testing Phase (December 2, 2023 - December 9, 2023):
    a. Milestone 13: Executing Unit Tests for Individual Elements
    b. Milestone 14: Conducting Integration Tests and Addressing Bugs
    c. Milestone 15: Performing UI Testing and Gathering User Feedback



Requirements Analysis:

Describe requirements

User Authentication (Log In and Log Out):

- Log In:
    - Users must be able to log in with their email and password.
    - The system should validate the user's input and provide appropriate error messages.
    - Successful login should redirect users to the homepage.
- Log Out:
    - Users must have the option to log out.
    - Logging out should clear the user's session and redirect them to the login screen.

Navigation Drawer:

- The app must include a navigation drawer for easy access to different parts of the application.

- It should contain items for navigating to various screens and performing actions like viewing attendance history and personal details.

Homepage:

- The homepage should display the user's class schedule in a calendar view.
- Users should be able to switch between weekly and monthly calendar views.
- It should provide options for navigating to different weeks or months.
- Displaying user-specific information such as name and email is required.

Calendar and Event Management:

- The calendar should display class events, including date, time, room, type, and status.
- Events should be colour-coded or visually distinct based on their type or status.
- Users should be able to click on events to view details or mark attendance.

Module Information (AttendanceHistory):

- Users should be able to view their module information.
- The module information should include details such as module name, module code, and year.
- Real-time updates should reflect changes in module information.

Change Password:

- Users should have the option to change their password.
- Password change should require re-authentication.
- Proper password validation should be implemented.

Feedback Submission (Feedback):

- Users should be able to provide feedback through the app.
- Feedback should be sent via email or a similar communication method.
- User input should be validated to ensure all required information is provided.

Notification Settings (NotificationAlarm):

- Users should be able to set and manage notification alarms.
- The app should request and handle necessary permissions for notifications.
- Alarms should trigger notifications at the specified times.

Export Data to PDF (ModuleDetails):

- Users should be able to export class data to a PDF file.
- The exported PDF should include relevant class details.

Absence Reporting (AbsentForm):

- Users should be able to report absences.
- The form should include date selection, file attachment, and optional comments.
- The system should handle the attachment of documents.

User Profile (PersonalDetail):

- Users should be able to view their personal details.
- The app should fetch and display user-specific information.
- The user should have the option to log out from the profile screen.

Database Integration (Firebase):

- The app should integrate with Firebase for user authentication and real-time data updates.
- Firebase Realtime Database should be used to store and retrieve class and module data.
- Firebase Authentication should be used for user login and session management.

Data Models (ClassItem, ClassType, Status, Module, Event):

- Proper data models should be defined to represent class items, class types, statuses, modules, and events.
- These models should facilitate data retrieval and display.

User Experience (UX) and User Interface (UI):

- The app's UI should be user-friendly and intuitive.
- Proper layout and design should be implemented to enhance the user experience.

Security and Privacy:

- User data, including passwords, should be securely stored and transmitted.
- Proper authentication and authorization mechanisms should be in place.

Permissions:

- The app should request and handle necessary permissions, such as camera and storage access, for features like file attachments.

Error Handling:

- The app should handle errors gracefully and provide informative error messages to users.

Testing and Quality Assurance:

- The app should undergo testing, including unit testing and user testing, to ensure functionality and quality.

Documentation:

- Documentation should be provided for code readability and maintainability.

Performance:

- The app should be optimised for performance, especially when dealing with real-time data updates.

User Story

User Story 1: Secure Login As a student, I want to securely log in to the `Instatt` app using my Campus ID and password. This will allow me to access

my personalised academic information, manage my schedule, and mark attendance securely.
Task 1: Enter Email Address and Password.
Task 2: Tap the "LogIn" button.
Task 3: Verify the Email Address and password against the credentials stored in the database.
Task 4: Upon successful verification, navigate to the "Homepage" user interface.

User Story 2: Daily Class Schedule As a student, I want to view my daily class schedule on the mobile app. This helps me manage my time efficiently and keeps me informed about the status of each class (whether it's over, in progress, or upcoming).
Task 1: Unlock the class if it is in progress.
Task 2: Tap the "Sign In" button.
Task 3: Update the class attendance record in the database.
Task 4: Refresh the user interface to reflect the current class status.

User Story 3: Past and Future Class Schedule As a student, I want to view both my past and future class schedules on the mobile app. This enables me to track completed classes and prepare for upcoming ones.
Task 1: Tap the date on the calendar on the homepage.
Task 2: Retrieve the user's class schedule from the database.
Task 3: Display the class schedule on the user interface.

User Story 4: App Navigation As a user of the mobile app, I want an easy-to-use navigation menu. This allows me to quickly access the Home page, view Attendance History, Personal Detail, adjust settings without hassle and log out.
Task 1: Tap the "Menu" button on the top left of each page.
Task 2: Select the desired page from the menu.
Task 3: Navigate to the corresponding page on the user interface.

User Story 6: Secure Logout As a user of the mobile app, I want a straightforward way to log out of my account. This ensures my account's security when I'm not using the app.
Task 1: Tap the "LogOut" button.
Task 2: Display a confirmation prompt upon clicking "Log Out."
Task 3: Upon confirmation, terminate the user session and require login on the next use.
Task 4: Provide a notification of successful logout.
Task 5: Clear cached data from the database.
Task 6: Redirect to the Login Screen.
Task 7: Update the user interface.

User Story 7: Attendance History As a student, I want to view my attendance history for each module. This helps me keep track of my participation and ensures I meet attendance requirements.
Task 1: Tap the "Attendance History" button on the "Menu" page.
Task 2: Navigate to the "Attendance History" page.
Task 3: Retrieve module details and attendance percentage from the dataset.
Task 4: Update the user interface to display the attendance history.

User Story 8: Detailed Module Attendance As a student, I want to view detailed attendance records for a specific course module. This provides me with an in-depth understanding of my attendance pattern and helps identify discrepancies.
Task 1: Select the desired module for detailed information.
Task 2: Retrieve and display specific details of the chosen module.
Task 3: Refresh the user interface to reflect the latest information.
User Story 9: Absence Verification As a student, I want to report and verify my absence from a module. This ensures it is officially acknowledged and does not negatively affect my attendance record.
Task 1: Locate the class marked as absent within the module's detail interface.
Task 2: Click the “Report” link corresponding to the absent class.
Task 3: Transition to the “Student Absence Verification Form” interface.
Task 4: Enter mandatory fields as requested by the form.
Task 5: Record the entered information in the system's database.
Task 6: Click the “Submit” button to finalise the report.
Task 7: Dispatch an email notification to the module's coordinator.
Task 8: Display a confirmation message indicating successful submission.
Task 9: Refresh the user interface to reflect the reported absence.
Task 10: Update the module detail user interface.

User Story 10: Export Attendance History As a student, I want to export my module attendance history for personal tracking or proof of attendance if required.
Task 1: Click the “Export” symbol in the module details interface.
Task 2: Prompt the user to specify the desired filename for the export file.
Task 3: Confirm the export and initiate the file download.
Task 4: Retrieve and compile the module's attendance history from the database. Task 5: Return to the module details page post-export.

User Story 11: Settings and Feedback As a user, I want to customise my app experience, receive relevant notifications, and provide feedback to improve the app.
Task 1: Navigate to the main menu.

Task 2: Select the option to adjust notification preferences or submit feedback.
Task 3: Update the user interface to reflect any changes.

User Story 12: Notification Preferences As a user, I want to customise my notification settings to choose the types of alerts I receive and their sound. This ensures I only receive relevant updates.
Task 1: Tap the “Notification Settings” button on the menu page.
Task 2: Select the notification time for every day.
Task 3: Confirm alarm set and exit settings.

User Story 13: Feedback Submission As a user, I want to submit feedback about the application to help improve the user experience.
Task 1: Navigate to the "Feedback" section through the main menu.
Task 2: Enter feedback details into the provided form.
Task 3: Temporarily store the feedback and dispatch it to the university's feedback management system via email.
Task 4: Update the user interface.
User Story 14: Absence Email Notification As a student who has missed a class, I want to receive an email notification prompting me to fill out an "Absence Verification Form" so that I can provide a valid reason for my absence.
Task 1: Mark the student as absent from class.
Task 2: Send an email notification to the student with a link to the "Absence Verification Form."
Task 3: Update the user interface to reflect the absence status.

User Story 15: Class Schedule Reminder As a student, I want to receive a reminder notification about my next day’s class schedule. This helps me prepare in advance and manage my time effectively.
Task 1: Load the schedule for the next day from the dataset.
Task 2: Send a reminder notification to the student.

Design and Architecture:

Overall Architecture — MVVM/MVC

Model Layer:

- The Model layer is responsible for managing data and business logic.
- Firebase Realtime Database and Firebase Authentication serve as the primary data sources and authentication mechanisms.
- Model classes like ClassItem, Module, and User encapsulate data logic and provide methods for data manipulation.

- Firebase queries and operations related to data storage and retrieval are abstracted within this layer.
- Utility classes like CalendarUtils are used for common tasks, such as date formatting and event handling.

View Layer:

- The View layer is responsible for rendering the user interface and handling user interactions.
- Activities, Fragments, and XML layout files represent the UI elements.
- RecyclerViews, TextViews, EditTexts, ImageButtons, and other UI components are used for user interaction.
- Views like AttendanceHistory, ModuleDetails, and PersonalDetail display data to the user and capture user input.

ViewModel/Controller Layer (MVVM/MVC):

- ViewModel classes, such as ModuleAdapter, ClassAdapter, and others, act as intermediaries between the View and Model.
- These classes manage the application logic, data retrieval, and updating UI elements.
- LiveData and Data Binding are used to bind data from ViewModel to UI components and ensure that the UI remains synchronised with the underlying data.
- User interactions and actions are handled within ViewModel classes.
- Firebase interactions and asynchronous data retrieval are managed in the ViewModel layer, ensuring responsive UI updates.

Firebase Integration:

- Firebase is integrated into the Model layer for user authentication and real-time data storage.
- Firebase Authentication (FirebaseAuth) is used for user login/logout functionality.
- Firebase Realtime Database (FirebaseDatabase) is used to store and retrieve data such as user information, class details, and attendance records.
- Firebase queries and listeners are employed to fetch and update data in real-time, ensuring that the app remains up-to-date.

Separation of Concerns:

- Each class has a distinct responsibility, adhering to the principle of separation of concerns.
- UI-related logic is handled in Activities and Fragments (View).
- Data logic and interactions with Firebase are encapsulated within Model and ViewModel/Controller classes.
- Utility classes provide reusable methods for common tasks.

Reusability and Modularity:

- Common functionalities, such as string manipulation, date formatting, and status lookup, are encapsulated in utility classes, promoting reusability.

- The use of RecyclerView adapters and modular components (e.g., ModuleAdapter, ClassAdapter) allows for easy extension and modification of functionality.

Navigation:

- Navigation between different screens is facilitated through activities and intents.
- A navigation drawer (NavigationView) is implemented for app navigation in several activities.
- Button listeners and intents are used to switch between different activities and fragments.

Security Considerations:

- Firebase Authentication is used to secure user data and authenticate users.
- User authentication is required before accessing certain parts of the app, ensuring data privacy and security.
- Error handling and validation are implemented to provide a secure and user-friendly experience.

User Interaction:

- The app provides user-friendly interfaces for tasks like logging in, viewing module details, managing attendance, providing feedback, and changing passwords.
- User inputs are captured and validated to ensure a smooth user experience.

Asynchronous Data Handling:

- Firebase asynchronous data retrieval is managed using callbacks, listeners, and addOnCompleteListener to ensure that the UI remains responsive.

PDF Generation:

- The app includes functionality to generate PDF documents, which can be useful for exporting class and attendance data.

Permissions Handling:

- The app requests and handles necessary permissions, such as media permissions for file attachments and notification permissions.

Design Principles

DRY (Don't Repeat Yourself):

- Avoid duplicating code or logic in multiple places within your codebase.
- Encapsulate repetitive code into functions, methods, or classes to promote reusability and easier maintenance.

KISS (Keep It Simple, Stupid):

- Favour simplicity in design and implementation.
- Avoid unnecessary complexity and over-engineering.
- Simpler solutions are often easier to understand, test, and maintain.

YAGNI (You Ain't Gonna Need It):

- Avoid adding functionality, features, or code that you don't currently need.
- Focus on solving the problems at hand and avoid speculative development for future requirements.

Single Responsibility Principle (SRP):

- Each class or module should have only one reason to change.
- Divide responsibilities and concerns in a way that a class has only one job or responsibility.

Open/Closed Principle (OCP):

- Software entities (classes, modules, functions) should be open for extension but closed for modification.
- New functionality should be added through extensions or subclasses without changing existing code.

Liskov Substitution Principle (LSP):

- Subtypes (derived classes) must be substitutable for their base types (parent classes) without altering the correctness of the program.
- Ensure that derived classes adhere to the contract defined by their base classes.

Interface Segregation Principle (ISP):

- Clients should not be forced to depend on interfaces they do not use.
- Split large, monolithic interfaces into smaller, specific ones to avoid unnecessary dependencies.

Dependency Inversion Principle (DIP):

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details; details should depend on abstractions.
- Promote loose coupling between components by depending on interfaces or abstract classes rather than concrete implementations.

Composition Over Inheritance:

- Prefer composing objects over inheriting from classes to achieve code reuse.
- Composition allows for more flexibility and avoids issues related to class hierarchies.

Law of Demeter (LoD) or Principle of Least Knowledge:

- Each unit (class or method) should have limited knowledge of other units, reducing dependencies.
- Encapsulate interactions with other objects and avoid chaining method calls.

Separation of Concerns (SoC):

- Divide a software system into distinct modules or classes, each addressing a specific concern or responsibility.

- Enhances maintainability, testability, and understanding of the codebase.

Principle of Least Astonishment (POLA):

- Code should behave in a way that's intuitive and consistent with the user's expectations.
- Avoid unexpected behaviour or surprising side effects in your code.

Keep Consistency:

- Maintain consistent naming conventions, coding style, and design patterns throughout your codebase.
- Consistency improves code readability and collaboration.

Fail Fast:

- Identify errors and issues as early as possible in the development process.
- Fail fast and provide meaningful error messages or logging to aid debugging.

Documentation and Comments:

- Write clear, concise, and meaningful comments and documentation to explain the code's purpose and functionality.
- Documentation should be kept up to date as code evolves.

Testing:

- Implement testing practices such as unit tests, integration tests, and automated testing to verify code correctness.
- Test-driven development (TDD) can help ensure that code is testable and meets requirements.

Performance Optimization:

- Optimise code for performance only when necessary, and based on measured performance bottlenecks.
- Avoid premature optimization, as it can lead to code complexity and reduced maintainability.

Security Considerations:

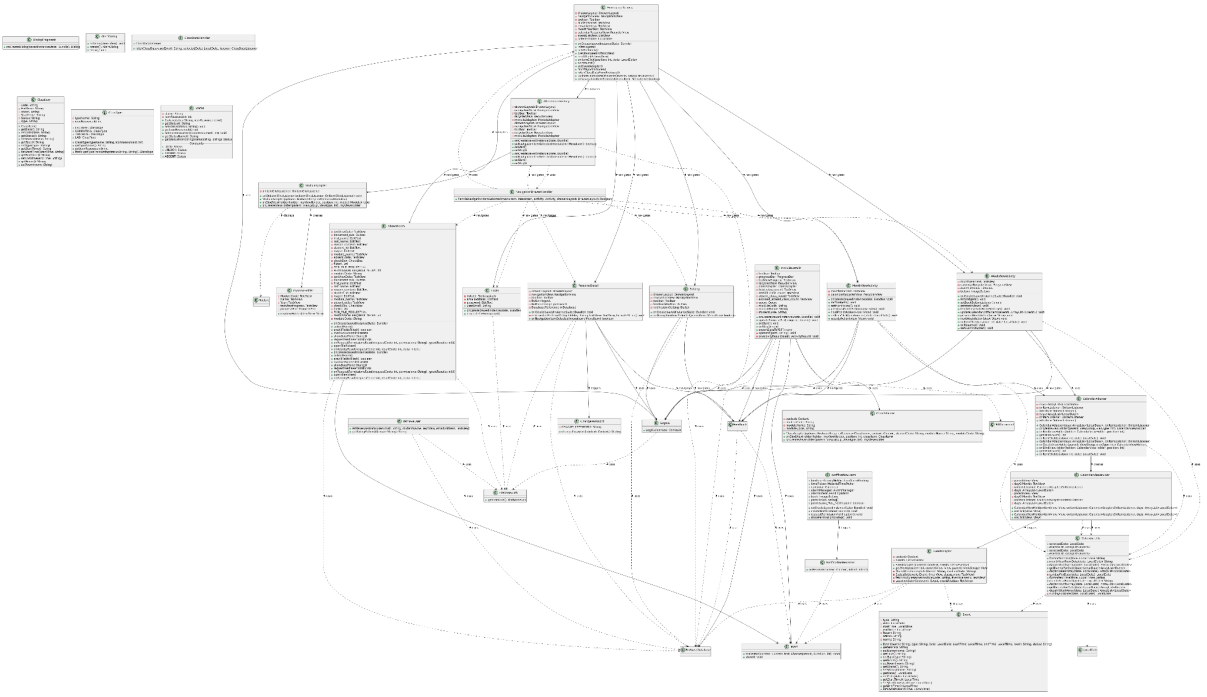
- Design with security in mind, including input validation, authentication, and authorization mechanisms.
- Protect against common security vulnerabilities, such as SQL injection and cross-site scripting (XSS).

Scalability and Maintainability:

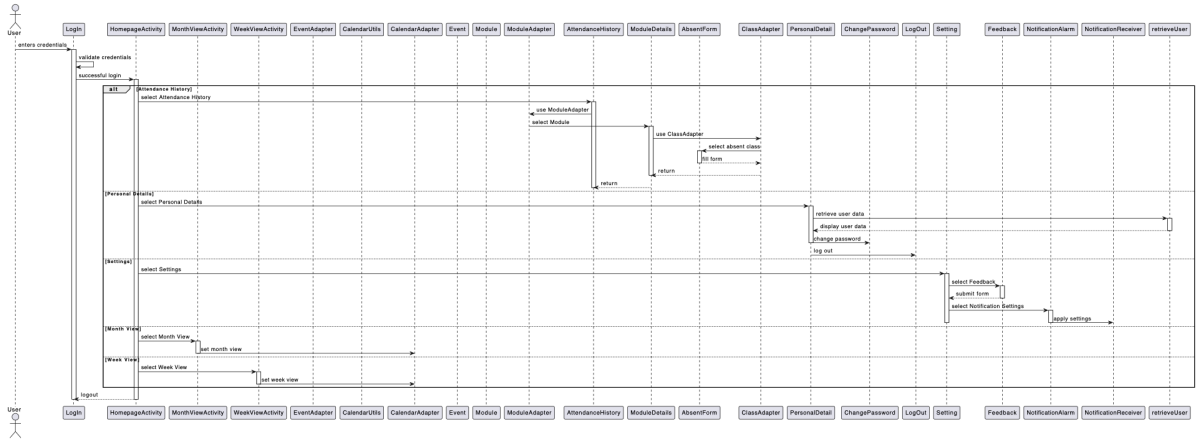
- Design systems with scalability and maintainability in mind.
- Consider factors like database scalability, code modularity, and architectural patterns (e.g., microservices) for long-term growth.

UML Diagrams

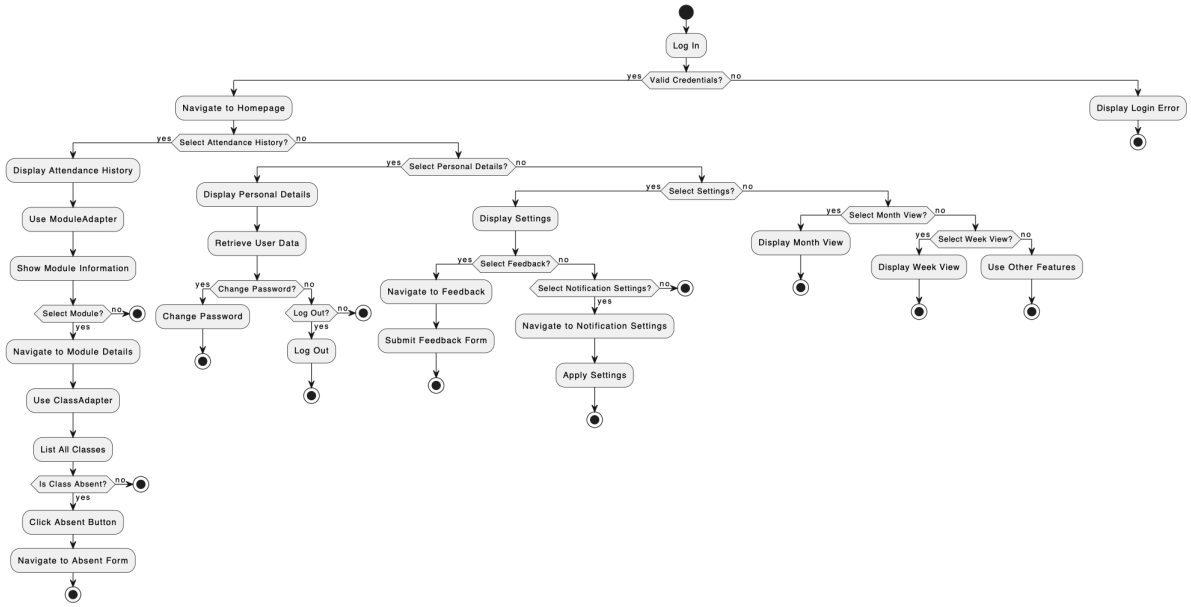
Class Diagram



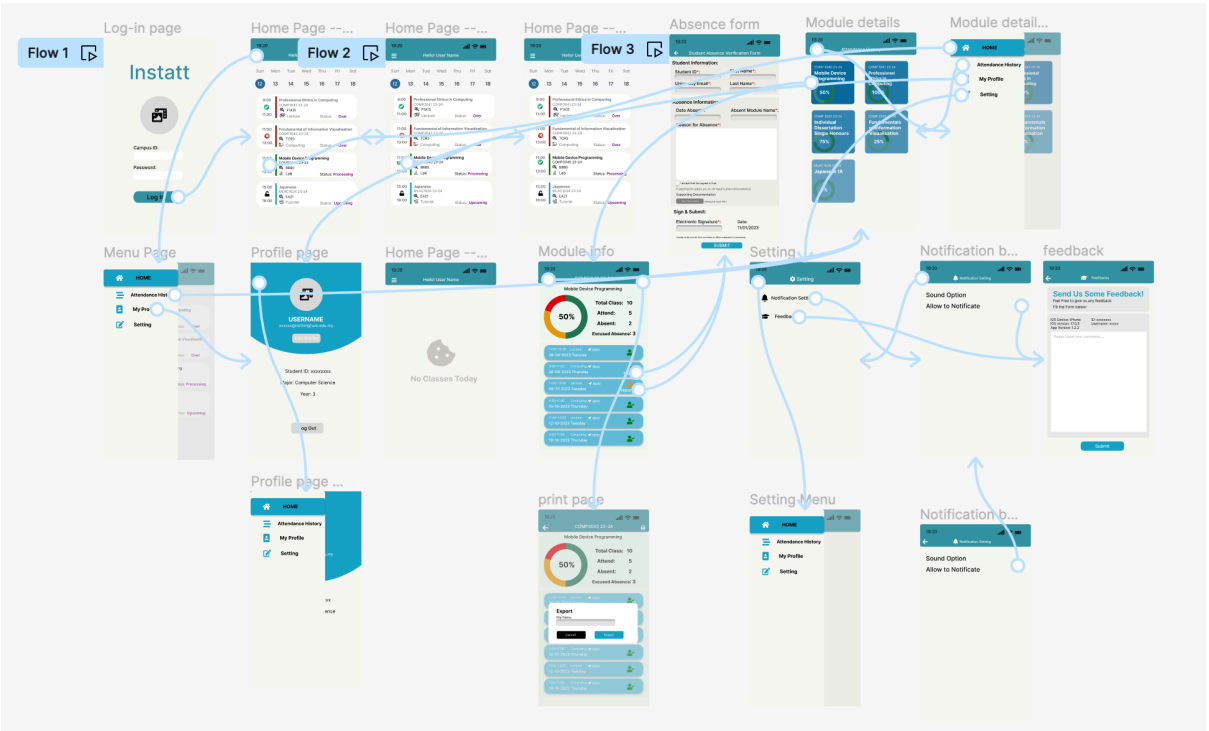
Sequence Diagram



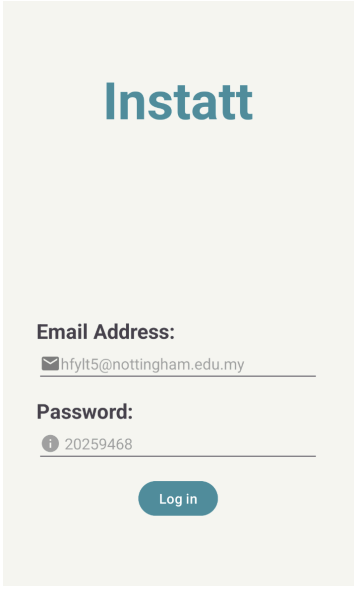
Activity Diagram

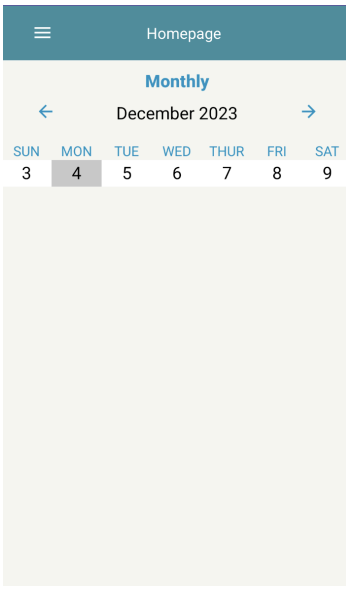


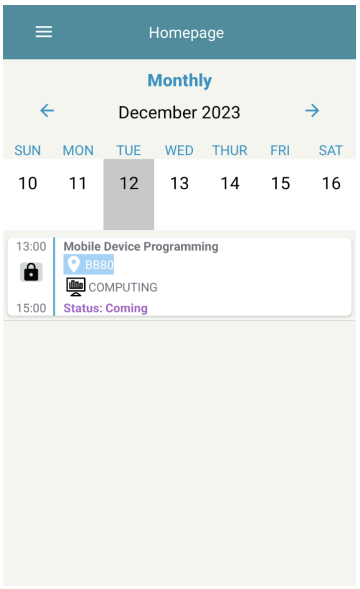
Wireframes

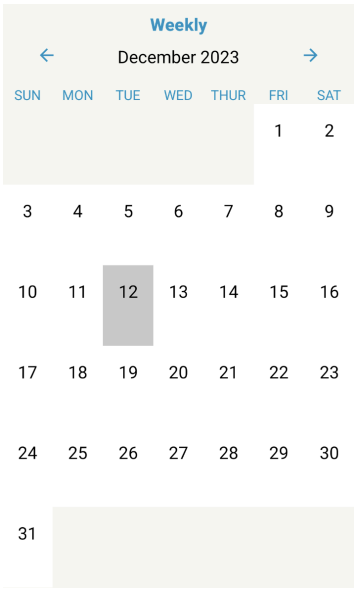


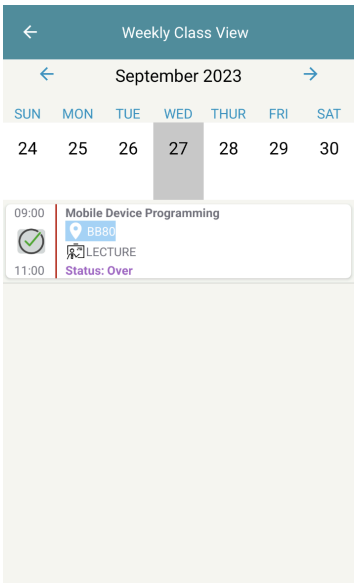
UI

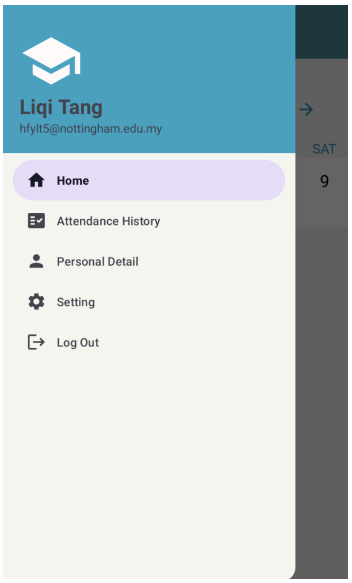

Instatt
Email Address:
hfylt5@nottingham.edu.my
Password:
20259468
Log in


Homepage
Monthly
December 2023
SUN MON TUE WED THUR FRI SAT
3 4 5 6 7 8 9

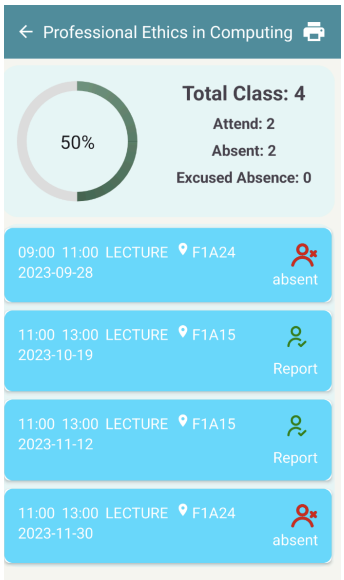

Homepage
Monthly
December 2023
SUN MON TUE WED THUR FRI SAT
10 11 12 13 14 15 16
13:00 Mobile Device Programming
BB80
COMPUTING
15:00 Status: Coming

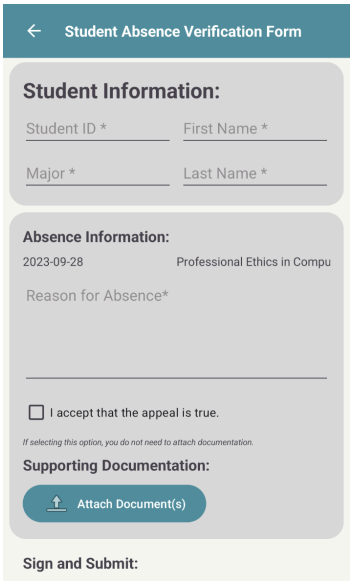

Weekly
December 2023
SUN MON TUE WED THUR FRI SAT
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

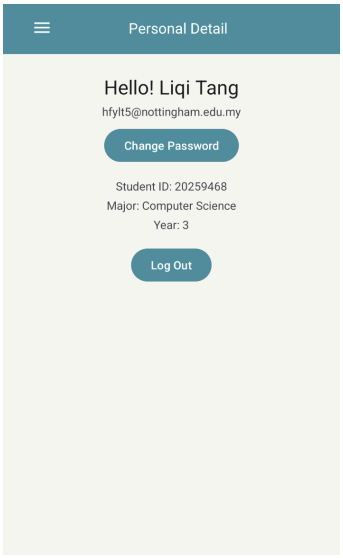

Weekly Class View
September 2023
SUN MON TUE WED THUR FRI SAT
24 25 26 27 28 29 30
09:00 Mobile Device Programming
BB80
LECTURE
11:00 Status: Over

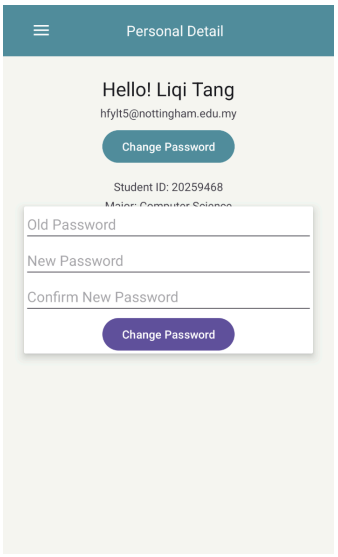

Liqi Tang
hfylt5@nottingham.edu.my
Home
Attendance History
Personal Detail
Setting
Log Out
SAT
9

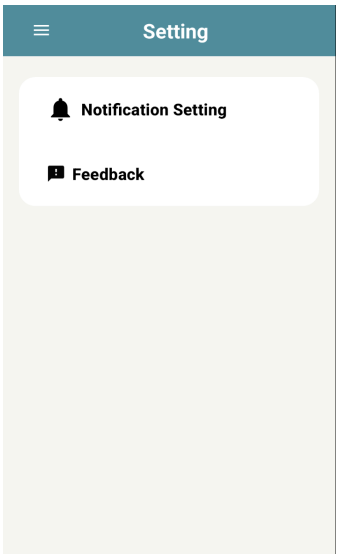

Attendance History
COMP3040 23-24
33%
Mobile Device Programming
COMP3041 23-24
50%
Professional Ethics in Computing
MLAC1034 23-24
100%
Japanese 1A


Professional Ethics in Computing
50%
Total Class: 4
Attend: 2
Absent: 2
Excused Absence: 0
09:00 11:00 LECTURE F1A24
2023-09-28
absent
11:00 13:00 LECTURE F1A15
2023-10-19
Report
11:00 13:00 LECTURE F1A15
2023-11-12
Report
11:00 13:00 LECTURE F1A24
2023-11-30
absent

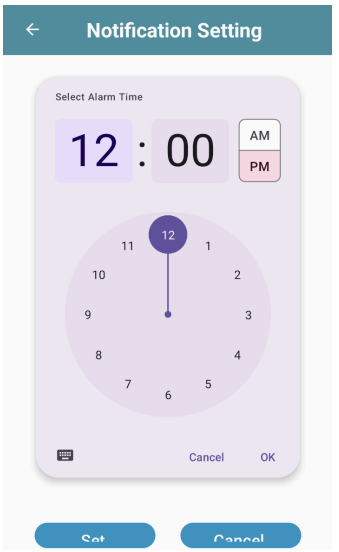

Student Absence Verification Form
Student Information:
Student ID *
First Name *
Major *
Last Name *
Absence Information:
2023-09-28
Professional Ethics in Compu
Reason for Absence*
I accept that the appeal is true.
If selecting this option, you do not need to attach documentation.
Supporting Documentation:
Attach Document(s)
Sign and Submit:

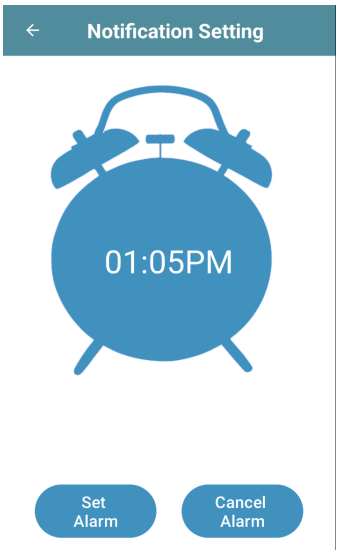

Personal Detail
Hello! Liqi Tang
hfylt5@nottingham.edu.my
Change Password
Student ID: 20259468
Major: Computer Science
Year: 3
Log Out

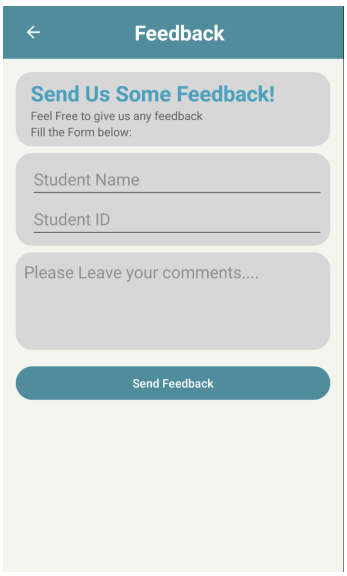

Personal Detail
Hello! Liqi Tang
hfylt5@nottingham.edu.my
Change Password
Student ID: 20259468
Old Password
New Password
Confirm New Password
Change Password

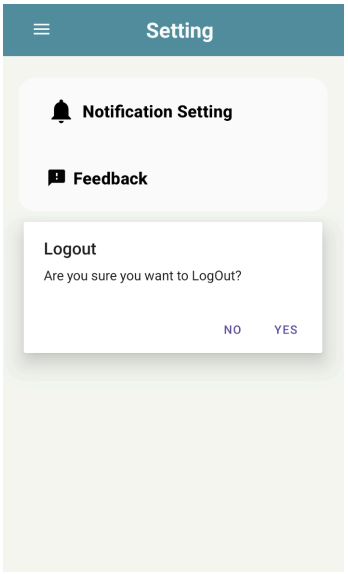

Setting
Notification Setting
Feedback

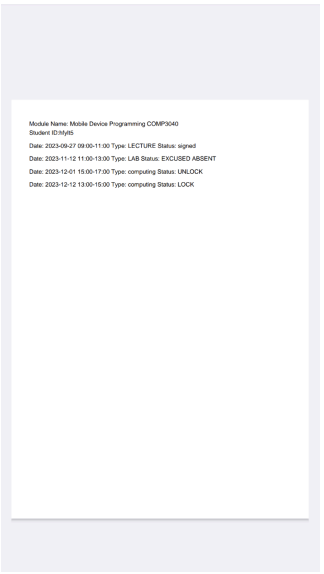

Notification Setting
Select Time


Notification Setting
Select Alarm Time
12 : 00
AM
PM
Cancel
OK


Notification Setting
01:05PM
Set Alarm
Cancel Alarm


Feedback
Send Us Some Feedback!
Feel Free to give us any feedback
Fill the Form below:
Student Name
Student ID
Please Leave your comments....
Send Feedback


Setting
Notification Setting
Feedback
Logout
Are you sure you want to LogOut?
NO
YES



Rationale Behind Design Decisions

Controller/ViewModel Use:

- Separation of Concerns: By using Controllers (in MVC) or ViewModels (in MVVM), the logic for handling user interactions is separated from the UI logic. This separation makes the code easier to manage and test.
- Data Presentation Logic: ViewModels act as a data preparer for the View, which means that the ViewModel holds the responsibility for the presentation logic. It converts the data from the Model in a way that the View can easily display it.
- Lifecycle Management: ViewModels are designed to be lifecycle-aware, meaning they survive configuration changes such as screen rotations, thus preventing common issues like data loss on UI changes.

Firebase Integration:

- Serverless Backend: Firebase provides a suite of cloud-based tools that removes the need for traditional server-side development. This accelerates development time and reduces the cost of maintaining servers.
- Real-Time Updates: Firebase's real-time database allows the app to update the UI instantly when data changes, which is key for dynamic content and collaborative features.
- Authentication: Using Firebase Auth simplifies the process of building a secure authentication system, providing various methods like email/password, social logins, and more.

Modular Design:

- Code Reusability: Creating modular components like ClassItem and utility classes such as CalendarUtils enables code reuse across different parts of the application.
- Easier Maintenance: Smaller, well-defined components are easier to debug and maintain compared to large, monolithic codebases.

Utility Classes:

- Common Functionality: Utility classes encapsulate common functionality that can be used throughout the application, adhering to the DRY principle.
- Simplifies Changes: When a common functionality needs to be updated, it can be done in a single place, reflecting everywhere it's used.

Static Members:

- Ease of Access: Static members like userEmail are easily accessible from different parts of the application without needing to pass around objects.

- State Sharing: Useful for sharing state across components, especially when dealing with data that doesn't change frequently.

Listeners and Interfaces:

- Decoupling: Interfaces allow for decoupling the event handling code from the main business logic, which simplifies the code structure and improves modularity.
- Flexibility: Listeners can be implemented by any class, allowing for flexible event handling and easy swapping of functionality.

Ethical Considerations:

- Ethical design decisions involve considerations related to user privacy, data security, and the potential impact of the design on society.
- Ethical design aims to protect users and adhere to ethical standards.

Implementation:

`retrieveUser` Class

This class is designed to retrieve user data from Firebase:

1. ***`retrieveUserData`***: Fetches user's name based on the provided email address and updates the provided `TextViews` with the user's name and email.
    a. Utilises `FirebaseDatabase` to access the Firebase Realtime Database.
    b. Extracts the part of the email before '@' for database querying.
    c. Implements `addOnCompleteListener` to handle Firebase asynchronous data retrieval.
2. ***`getStringBeforeAt`***: Utility method to extract the string before the '@' symbol in an email address.

`ClassItem` Class

A simple model class representing a class item:

1. Properties: Includes `iconSourceID`, and `typeName`.
2. Empty Constructor: Allows Firebase to directly map data from the database to an object of this class.
3. Getter and Setter Methods: For each property, facilitating access and modification.

`ClassType Class

A simple model class representing a class type:

1. Properties: Includes `Date`, `EndTime`, `Room`, `StartTime`, `Status`, and `Type`.
2. Empty Constructor: Allows Firebase to directly map data from the database to an object of this class.

3. Getter and Setter Methods: For each property, facilitating access and modification.
4. Static Final Instances: Defines constant instances for different statuses (`LECTURE`, `COMPUTING`, `LAB`, `TUTORIAL`).
5. ***`getTypeFromString`***: A static method to return a `classType` object based on a given status string.

`Status` Class

Manages status types and associated icon resources for class items:

1. Properties: `status` (name of the status) and `iconResourceId` (resource ID of the associated icon).
2. Constructor: Initialises a `Status` object with the given status name and icon resource ID.
3. Static Final Instances: Defines constant instances for different statuses (`LOCK`, `UNLOCK`, `SIGNED`, `ABSENT`).
4. ***`getStatusFromString`***: A static method to return a `Status` object based on a given status string.

Key Features of `retrieveUser` Class, `ClassItem` Class, `Status` Class:

1. Firebase Integration: `retrieveUser` integrates with Firebase Realtime Database for fetching user data.
2. Model-View Separation: `ClassItem` and `Status` serve as model classes, separating data representation from view logic.
3. Utility Methods: Includes methods for common tasks like string processing and status lookup.

`NavigationDrawerHandler` class

1. Navigation Drawer: The navigation drawer is a common UI pattern in Android apps that provides a side menu for navigation. It typically contains a list of items that the user can select to perform various actions or navigate to different parts of the app.
2. MenuItem: The `MenuItem` parameter represents the item that the user selected in the navigation drawer. Each item corresponds to a specific action or destination within the app. The `itemId` is used to identify which item was selected.
3. Activity: The `Activity` parameter represents the current activity from which the navigation drawer item was selected. An activity represents a single screen with a user interface. Different activities can be launched based on the selected item.
4. DrawerLayout: The `DrawerLayout` parameter represents the layout that contains the navigation drawer. It is responsible for displaying and hiding the navigation drawer when the user interacts with it.

5. Intent: Intents are used to launch different activities within the Android app. Depending on the selected item, a specific `Intent` is created to start the corresponding activity. For example, when the "Attendance History" item is selected, an intent to start the `AttendanceHistory` activity is created.
6. Activities: The app likely consists of multiple activities, each representing a different screen or part of the app. Activities are launched to display different content or functionality to the user. Examples of activities mentioned in the code include `AttendanceHistory`, `PersonalDetail`, `Setting`, `HomepageActivity`, and `logout`.
7. Logout: The `logout` class/module seems to be responsible for handling the logout functionality. When the "Logout" item is selected, an instance of the `logout` class is created, and the `LogOut` method is called to perform the logout action. This module likely handles user authentication and session management.
8. Navigation: The primary purpose of this code is to handle navigation within the app. It allows users to move between different parts of the app by selecting items in the navigation drawer. Depending on the item selected, a specific activity is launched to provide the corresponding functionality.
9. Drawer Closure: After selecting an item from the navigation drawer, the code ensures that the drawer is closed (if it's not null) to provide a smooth user experience.

Overall, the `NavigationDrawerHandler` class plays a crucial role in managing the navigation and interaction flow of the app, allowing users to access different parts of the application's functionality by selecting items from the navigation drawer.

`Log In` Class

1. Variables:
    a. `private FirebaseAuth mAuth`: This is an instance of FirebaseAuth, which is used to handle all authentication-related functions.
    b. `private EditText emailAddress, password`: These are EditText variables to get the user's input for email and password.
    c. `public static String userEmail`: A static variable to store the user's email after successful login.
2. ***`onCreate`***:
    a. This method is called when the activity is starting.
    b. `setContentView(R.layout.activity_main)`: Sets the user interface layout for this Activity.
    c. The UI elements (email address, password fields, and login button) are initialised.
    d. `mAuth = FirebaseAuth.getInstance()`: Initialises the FirebaseAuth instance.
3. ***`Login Button Click Listener`***:

    a. When the login button is clicked, it retrieves the text from the email and password fields.
    b. It checks if the email or password fields are empty and shows a toast message if either is.
    c. If both fields are filled, it attempts to sign in using `mAuth.signInWithEmailAndPassword`.
    d. Upon successful sign-in, it stores the user's email in `userEmail` and navigates to the homepage activity.
    e. If sign-in fails, it displays a toast message with the error.
4. ***`navigateToHomepage`***:
    a. This method creates an Intent to start the `HomepageActivity`.
    b. It then finishes the current `LogIn` activity, removing it from the back stack.
5. Error Handling:
    a. The class includes basic error handling with toast messages for empty fields or failed sign-ins.
6. Firebase Integration:
    a. The class integrates Firebase Authentication for user sign-in, demonstrating how Firebase can be used for managing user authentication in Android apps.

This code is a good example of implementing a basic login functionality in an Android app with Firebase. However, it's always recommended to handle passwords with care and consider adding more robust error handling and security features in a production environment.

`HomepageActivity` class

1. Extending AppCompatActivity and Implementing Interfaces:
    a. `extends AppCompatActivity`: This indicates that `HomepageActivity` is an Android activity.
    b. `implements NavigationView.OnNavigationItemSelectedListener, CalendarAdapter.OnItemListener`: The class implements two interfaces to handle navigation drawer item selections and calendar item clicks.
2. UI Elements:
    a. DrawerLayout, NavigationView, Toolbar for the navigation drawer setup.
    b. TextViews for displaying the student name and email address.
    c. RecyclerView for the calendar and ListView for events.
3. Initialization in **`onCreate`**:
    a. Set up the UI layout from `R.layout.activity_homepage`.
    b. Initialises and configures the navigation drawer and toolbar.

    c. Retrieves and sets the student's name and email address in the navigation header.
4. Calendar and Event Management:
    a. ***`initWidgets()`***: Initialises the calendar and event list views.
    b. ***`setWeekView()`***: Sets up the calendar to display a `week view`, fetching and displaying class data from Firebase.
    c. ***`previousWeekAction`*** and ***`nextWeekAction`***: Methods to navigate between weeks in the calendar.
    d. ***`onItemClick`***: Handles calendar item clicks to update the selected date.
5. Lifecycle Management:
    a. ***`onResume`***: Refreshes the event adapter and fetches class data from Firebase when the activity resumes.
6. Event Handling and Data Fetching:
    a. ***`setEventAdapter`***: Sets the adapter for the event list view, displaying events for the selected date.
    b. ***`fetchClassDataFromFirebase`***: Fetches class data related to the selected date from Firebase.
    c. ***`updateCalendarWithEvents`***: Updates the calendar view with fetched events.
7. Navigation Drawer Handling:
    a. ***`onNavigationItemSelected`***: Handles item selections in the navigation drawer.
8. Firebase Integration:
    a. The class integrates Firebase to fetch class data, demonstrating how Firebase can be used for data retrieval in real-time applications.

This activity serves as a central hub for `instatt`, providing a comprehensive view of the weekly calendar, event management, and navigation functionalities.

`Event` Class

This class represents an event with various properties and functionalities:

1. Properties: Includes `type`, `date`, `startTime`, `endTime`, `Room`, `Status`, and `name`.
2. Constructor: Initialises an event object with the given properties.
3. Getter and Setter Methods: Provide access to the event's properties.
4. Static List: `eventsList` holds all event instances.
5. Static Method ***`eventsForDate`***: Filters events by a specific date.

`EventAdapter` Class

This class is an adapter for displaying `Event` objects in a `ListView`:

1. Constructor: Inherits from `ArrayAdapter<Event>`, initialising with a context and list of events.

2. ***`getView`***: Customises the view for each event item in the list.
   It includes:
   a. Inflating a layout for the event item if not already done.
   b. Finding and setting up views from the layout (e.g., `TextViews`, `ImageViews`).
   c. Setting event details and handling different event types and statuses.
   d. Implementing logic for marking attendance and updating event status.

Key Features and Implementations of `Event` class and `EventAdapter` class:

1. Event Management: Both classes work together to manage and display event data.
2. Custom ListView Adapter: `EventAdapter` is a customised adapter for a `ListView`, allowing for complex layouts and interactions within each list item.
3. Dynamic UI Updates: The adapter updates the UI based on the event's properties, such as changing colours or icons based on the event's type or status.
4. Firebase Integration: The adapter interacts with Firebase to retrieve and update data, demonstrating real-time data handling in an Android app.
5. Complex Logic for UI Display: Includes logic for setting text, images, and colours based on event attributes, enhancing the user experience.

`CalendarUtils` Class

This utility class provides various methods for handling dates and events:

1. Static Fields:
   a. ***`selectedDate`***: Stores the currently selected date.
   b. ***`eventsList`***: Stores a list of `Event` objects.
2. Date Formatting Methods:
   a. ***`formattedTime`***: Formats a `LocalTime` object into a string in "HH:mm" format.
   b. ***`monthYearFromDate`***: Formats a `LocalDate` object into a string in "MMMM yyyy" format.
3. Date Arrays Generation:
   a. ***`daysInMonthArray`***: Generates an `ArrayList` of `LocalDate` for all days in the month of a given date.
   b. ***`daysInWeekArray`***: Generates an `ArrayList` of `LocalDate` for all days in the week of a given date.
4. Event Retrieval:
   a. ***`getEventsForDate`***: Retrieves events for a specific date from `eventsList`.
5. Helper Methods:
   a. ***`sundayForDate`***: Finds the Sunday of the week for a given date.

`CalendarAdapter` Class

This class is a `RecyclerView.Adapter` for managing calendar views:

1. Fields:
    a. `days`: An `ArrayList` of `LocalDate` representing the dates to display.
    b. `onItemListener`: An interface for handling item clicks.
2. Adapter Methods:
    a. `onCreateViewHolder`: Inflates the view for each calendar cell and sets the height based on the view type (month or week).
    b. `onBindViewHolder`: Sets up each calendar cell with the correct date and background colour based on whether the date is selected or has events.
    c. `getItemCount`: Returns the total number of items (dates) in the adapter.

`CalendarViewHolder` Class (Nested in `CalendarAdapter`)

This class is a `RecyclerView.ViewHolder` for individual calendar cells:

1. Fields:
    a. `days`: The list of dates associated with the calendar.
    b. `parentView`: The parent view of each item.
    c. dayOfMonth`: A `TextView` to display the day of the month.
    d. onItemListener`: Listener for item click events.
2. Methods:
    a. onClick`: Handles click events on each calendar cell.

Key Features and Implementations of `CalendarUtils` Class, `CalendarAdapter` Class and `CalendarViewHolder` Class

1. Calendar Management: The combination of `CalendarUtils` and `CalendarAdapter` allows for robust handling and display of a calendar with events.
2. Dynamic UI Updates: The adapter updates the calendar view dynamically based on events and the selected date.
3. Event Handling: Incorporates an interface for handling item clicks, enabling interaction with calendar dates.
4. Flexible Date Handling: The utility class provides methods to work with both weekly and monthly views.

`WeekViewActivity` Class

This class provides a weekly view of the calendar:

1. UI Elements: A `RecyclerView` for the calendar, a `ListView` for events, a `TextView` for displaying the month and year, and an `ImageButton` for navigation.
2. ***`onCreate`*** Method: Sets up the initial week view and fetches class data from Firebase.
3. ***`initWidgets`***: Initialises UI elements.

4. ***`setBackButtonListener`***: Sets up a listener for the back button to navigate to `HomepageActivity`.
5. ***`setWeekView`***: Updates the week view based on the selected date.
6. ***`fetchClassDataFromFirebase`***: Fetches class data from Firebase and updates the calendar with these events.
7. ***`updateCalendarWithEvents`***: Updates the event list view with fetched events.
8. Navigation Methods: ***`previousWeekAction`*** and ***`nextWeekAction`*** for navigating between weeks.
9. ***`onItemClick`*** Method: Implements `CalendarAdapter.OnItemListener` to handle calendar item clicks.
10. ***`onResume`***: Refreshes the event adapter and fetches class data when the activity resumes.
11. ***`setEventAdapter`***: Sets the adapter for the event list view.

`MonthViewActivity` Class

This class provides a monthly view of the calendar:

1. UI Elements: A `RecyclerView` for the calendar and a `TextView` for displaying the month and year.
2. ***`onCreate`*** Method: Sets up the initial month view.
3. ***`initWidgets`*** : Initialises UI elements.
4. ***`setMonthView`***: Updates the month view based on the selected date.
5. Navigation Methods: ***`previousMonthAction`*** and ***`nextMonthAction`*** for navigating between months.
6. ***`onItemClick`*** Method: Implements `CalendarAdapter.OnItemListener` to handle calendar item clicks.
7. ***`weeklyAction`***: Starts `WeekViewActivity` to switch to the week view.

Both activities use the `CalendarAdapter` and rely on `CalendarUtils` for date handling.

Key Features and Implementations of `WeekViewActivity` class and `MonthViewActivity` class

1. Flexible Calendar Views: Both activities offer different calendar views (week and month) with navigation options.
2. Dynamic UI Updates: Calendar views are updated based on user interactions like selecting a different week or month.
3. Firebase Integration: Class data is fetched from Firebase, demonstrating real-time data retrieval.
4. Event Management: Events are displayed and updated in the `ListView` in `WeekViewActivity`.

`Module` Class

This class represents a module with properties and getter/setter methods:

1. Properties: `Name`, `ModuleCode`, `Year`.
2. Constructor: An empty constructor is provided, which is a requirement for Firebase's data snapshot to model conversion.
3. Getter and Setter Methods: For each property, to access and modify the module's data.

`ModuleAdapter` Class

This class extends `FirebaseRecyclerAdapter` and is used to display module information in a RecyclerView:

1. OnItemClickListener Interface: A custom interface to handle item clicks within the adapter.
2. Constructor: Initialises the adapter with FirebaseRecyclerOptions.
3. `onBindViewHolder` Method: Binds each module to a ViewHolder. It fetches additional class data from Firebase and updates UI elements like ProgressBar based on the attendance status.
4. `onCreateViewHolder`: Inflates the layout for each module item.
5. `myViewHolder` Class: A ViewHolder class that holds UI elements for each module item.
6. ***`updateProgressBar`***: Updates the progress bar based on the calculated attendance progress.

`AttendanceHistory` Class

An `AppCompatActivity` class that displays the modules' information:

1. UI Elements: Includes a `DrawerLayout`, `NavigationView`, `Toolbar`, and `RecyclerView`.
2. `onCreate` Method: Sets up the UI elements, initialises the `ModuleAdapter`, and sets up the navigation drawer.
3. ***`onNavigationItemSelected`***: Handles navigation drawer item selection.
4. ***`onStart`*** and ***`onStop`*** Methods: Manage the start and stop of listening to Firebase data changes in `ModuleAdapter`.

`Module`, `ModuleAdapter`, and `AttendanceHistory` are designed to work together in the `instatt` to manage and display module information.

Key Features and Implementations `ModuleAdapter` class and `AttendanceHistory` class

1. Firebase Integration: `ModuleAdapter` uses `FirebaseRecyclerAdapter` for real-time data updates from Firebase.
2. Custom RecyclerView Adapter: `ModuleAdapter` provides a custom implementation for displaying module data and handling user interactions.

3. Dynamic UI Updates: The UI, especially the progress bar in each module item, updates dynamically based on Firebase data.
4. Navigation Drawer: `AttendanceHistory` incorporates a navigation drawer for app navigation.

`ModuleDetails` Class

This class provides detailed information about a specific module with corresponding classes' details and manages attendance-related data:

1. UI Elements: A `Toolbar`, `ProgressBar`, `TextViews` for progress and class counts, a `RecyclerView` for class details, and `ImageButtons` for navigation and actions.
2. `onCreate` Method: Sets up the UI elements, initialises Firebase queries, and sets up button listeners.
3. Firebase Query: Fetches class data from Firebase, calculates attendance statistics, and updates UI elements accordingly.
4. Exporting Data to PDF: Includes functionality to export class data to a PDF file and view it.
5. ***`updateProgressBar`***: Updates the progress bar based on attendance statistics.
6. ***`onStart`*** and ***`onStop`*** Methods: Manage the start and stop listening to Firebase data changes in `classAdapter`.
7. ***`exportDataToPDF`***: Exports class data to a PDF file.
8. ***`openPdf`***: Opens a PDF file using an `ActivityResultLauncher`.

`AbsentForm` Class

This class provides a form for students to submit an absence report directly through `instatt`:

1. UI Elements: `TextViews`, `EditTexts`, `Buttons`, and a `CheckBox` for form inputs and actions.
2. `onCreate` Method: Sets up the UI elements and button listeners.
3. `markAsExcusedAbsent` : Updates the Firebase database to mark the absence as excused.
4. File Picker and Permissions: Implements functionality to attach documents with the form and request necessary permissions.
5. `showDatePickerDialog`: Shows a date picker for selecting the absent date.
6. ***`requestMediaPermissions`*** and ***`onRequestPermissionsResult`*** : Request and handle media access permissions for attaching files.
7. ***`openFilePicker`*** and ***`onActivityResult`*** : Open a file picker to select a document and handle the result.

Key Features and Implementations of `ModuleDetails` Class and `AbsentForm` Class

1. Firebase Integration: Both classes interact with Firebase to fetch and update data.
2. PDF Generation and Viewing: `ModuleDetails` includes functionality to generate and view PDF documents.
3. Dynamic UI Updates: The UI updates based on the data fetched from Firebase and user interactions.
4. Email Intent Creation: `AbsentForm` creates an email intent to submit the absence form via email, including file attachments.
5. Form Validations and Permissions Handling: `AbsentForm` checks if all fields are filled and handles media permissions for file attachments.

`ClassAdapter` class:

1. Overview of `ClassAdapter`
    a. Extends FirebaseRecyclerAdapter: `ClassAdapter` extends `FirebaseRecyclerAdapter`, a specialised adapter which directly integrates with a Firebase Realtime Database query.
    b. Constructor: Accepts `FirebaseRecyclerOptions<ClassItem>`, `Context`, `studentCode`, `moduleName`, and `moduleCode`. These are used for configuring the adapter and handling item interactions.
2. ***`onBindViewHolder`***
    a. Sets text for various TextViews from the `ClassItem` object (date, start time, end time, room, type, and status).
    b. Changes the status icon and text based on the status of the class (e.g., "EXCUSED ABSENT", "ABSENT", "SIGNED", "LOCK", "UNLOCK").
    c. For classes with the status "ABSENT", the status icon becomes clickable, leading to the `AbsentForm` activity when clicked.
3. ***`onCreateViewHolder`***
    a. Inflates the layout for each class item.
    b. Uses `LayoutInflater` to inflate `class_history` layout.
4. ***`myViewHolder`*** Static Inner Class
    a. Acts as a ViewHolder for the RecyclerView items.
    b. Contains TextViews for class details (type, end time, start time, status, room, date) and an ImageButton for status.

The `ClassAdapter` class provided is a custom adapter for a RecyclerView in an Android application, designed to display and manage class items (`ClassItem` objects) using Firebase.

`PersonalDetail` Class

This class displays and manages the user's personal details:

1. UI Elements: Includes a `DrawerLayout`, `NavigationView`, `Toolbar`, `Buttons`, and `TextViews` for displaying user information.
2. `onCreate` Method: Initialises the UI elements, sets up the navigation drawer, and fetches user details from Firebase.
3. Firebase Database Reference: Uses `FirebaseDatabase` to fetch user details.
4. ***`retrieveAndSetUserData`***: Retrieves specific user data from Firebase and sets it to the corresponding `TextView`.
5. `Logout` and `Change Password` Buttons: Implements click listeners for the logout and change password functionalities.
6. ***`onNavigationItemSelected`***: Handles navigation item selection in the drawer.

`ChangePassword` Class

This class extends `DialogFragment` and is responsible for changing the user's password:

1. `onCreateDialog` : Creates a dialog for changing the password.
2. ***`changePassword`***: Sets up the dialog with layout `change_pawssword` and includes logic for changing the password.
3. Firebase Authentication: Utilises `FirebaseAuth` and `FirebaseUser` to authenticate and update the user's password.
4. Validation and Firebase Operations: Performs password validation and updates the password in Firebase after successful reauthentication.

Key Features and Implementations of `PersonalDetail` class and `ChangePassword` class

1. User Data Management: `PersonalDetail` fetches and displays user data from Firebase.
2. Firebase Integration: Both classes use Firebase services - `PersonalDetail` for fetching user data, and `ChangePassword` for password management.
3. Dynamic UI Updates: The UI updates based on user interactions and Firebase data retrieval.
4. DialogFragment for Changing Password: `ChangePassword` uses a custom dialog to facilitate password changes.
5. Navigation Drawer: Implements a navigation drawer for app navigation in `PersonalDetail`.

`Setting` Class

1. UI Elements: Includes a `DrawerLayout`, `NavigationView`, `Toolbar`, and buttons for feedback and notification settings.
2. Navigation Drawer: Implements a navigation drawer with an action bar toggle for menu interaction.

3. Button Listeners: Sets up listeners for feedback and notification setting buttons, navigating to respective activities (`Feedback` and `NotificationAlarm`).

`Feedback` Class

1. Functionality: Handles the feedback submission process.
2. UI Elements: Consists of `EditText` fields for user input and a `Button` for sending feedback.
3. Email Intent: On clicking the send button, an email intent is created with feedback details, which opens an email client for sending the feedback.

`NotificationAlarm` Class

1. Functionality: Manages notification alarm settings.

- **Time Picker**: Includes a Material Design time picker for setting an alarm time.
- **Alarm Management**: Sets and cancels alarms using the `AlarmManager` service.
- **Notification Channel**: Creates a notification channel for posting notifications.
- **Permission Handling**: Requests and checks notification posting permissions.
- **Alert Dialog for Permission**: Displays an alert dialog guiding the user to enable permissions in the app settings if necessary.

`NotificationReceiver` Class

- **Functionality**: Acts as a broadcast receiver to handle alarm events.
- **Notification Creation**: Creates and displays a notification when the alarm is triggered.

`Log Out` Class

1. ***`LogOut(Context context)`***
    a. Parameter: Accepts a `Context` object, which is used to display the `AlertDialog` and to create an `Intent` for navigation.
    b. AlertDialog Creation: Constructs an `AlertDialog` using the `Builder` pattern, setting a title and a message to confirm the user's intention to log out.
    c. Positive Button ("Yes"):
        i. Sign out the user from Firebase Authentication.
        ii. Displays a toast message confirming successful logout.
        iii. Redirects the user to the `LogIn` activity using an `Intent`.
    d. Negative Button ("No"):
        i. Dismisses the dialog, allowing the user to remain logged in if they choose not to log out.

The `logout` class you provided is a utility class designed to handle the logout functionality in an Android application. It creates and displays an `AlertDialog` to confirm the user's intention to log out.

Key Features of `Log out` class

1. Firebase Authentication Integration: Utilises `FirebaseAuth.getInstance().signOut()` to sign out the user.
2. User Confirmation: Asks for user confirmation before logging out to prevent accidental logouts.
3. Context-Dependent: Takes a `Context` parameter to be adaptable for use in different activities.
4. Navigation on Logout: Redirects to the `LogIn` activity upon successful logout.

Testing:

ClassTypeTest

1. Test: testConstructorAndGetters()

Purpose: This test verifies the constructor and getter methods of the ClassType class. It checks whether the class correctly initialises its properties and whether the getter methods return the expected values.

Implementation:

- It creates an instance of ClassType with a specific type name and icon resource ID.
- It asserts that the getTypeName() method returns the correct type name.
- It asserts that the getIconResourceId() method returns the correct icon resource ID.

Expected Result:

- The test should pass if the constructor correctly assigns the values to the class fields and the getter methods return these values accurately.
- If the test fails, it indicates an issue with either the constructor's initialization logic or the getter methods not returning the correct values.

Fixing Issues:

- Ensure that the constructor is correctly assigning the provided parameters to the class fields.
- Verify that the getter methods are correctly returning the respective field values.

2. Test: testGetTypeFromString()

Purpose: This test checks the static method getTypeFromString of the ClassType class. It ensures that the method correctly returns a ClassType instance based on a given string, and returns null for unknown types.

Implementation:

- It calls getTypeFromString with different string values that represent class types.

- It asserts that the method returns the correct ClassType instance for each known type.
- It also asserts that the method returns null for an unknown type string.

Expected Result:

- The test should pass if the method returns the correct ClassType instance for known type strings and null for an unknown type string.
- If the test fails, it suggests an issue with the getTypeFromString method's logic in handling the provided strings.

Fixing Issues:

- Review the switch or conditional logic in the getTypeFromString method to ensure it correctly matches the provided strings to the corresponding ClassType instances.
- Ensure that the method handles unknown or unexpected strings appropriately, typically by returning null.

HomepageActivityTest

1. Test: testNavigationDrawerInteraction()

Purpose: This test verifies the functionality of the navigation drawer in HomepageActivity. It specifically tests opening the drawer, selecting an item, and then checking if the drawer closes correctly.

Implementation:

- Open the navigation drawer.
- Selects a navigation item (R.id.nav_home).
- Check if the drawer is closed after the selection.

Expected Result:

- The test should pass if the drawer opens, allows selection, and closes properly.
- A failure indicates a problem with the drawer's interaction, such as not opening, not responding to item selections, or not closing after a selection.

Fixing Issues:

- Ensure the drawer layout and navigation view are correctly set up in the activity.
- Verify that the navigation items are correctly defined and selectable.
- Check that the drawer's state changes (open/close) are properly handled.

2. Test: testRecyclerViewIsPopulated()

Purpose: This test checks whether the RecyclerView in HomepageActivity is properly populated with data.

Implementation:

- Waits for the data to load (this might require idling resources or a delay).
- Checks if the RecyclerView is displayed.
- Scrolls to a position (e.g., position 0) in the RecyclerView.
- Optionally checks for specific content at a particular position.

Expected Result:

- The test should pass if the RecyclerView is visible and can be scrolled, indicating it is populated with data.
- A failure might occur if the RecyclerView is not visible, not populated, or not scrollable.

Fixing Issues:

- Ensure that the RecyclerView is correctly set up with an adapter and layout manager.
- Verify that the data source feeding the RecyclerView is properly initialised and populated.
- If using asynchronous data loading, ensure synchronisation with the test execution (using Espresso Idling Resources is recommended).

LogInTest

1. Test: testSuccessfulLogin()

Purpose: This test verifies the successful login process. It inputs valid credentials and checks whether the HomepageActivity is opened upon successful login.

Implementation:

- Inputs a valid email and password.
- Click the login button.
- Waits for 2 seconds (using Thread.sleep) to allow asynchronous operations (like network requests) to complete.
- Verifies that an intent to start HomepageActivity is sent.

Expected Result:

- The test should pass if the correct intent is fired after a successful login.
- A failure indicates that the HomepageActivity was not started as expected, which could be due to incorrect login logic, issues in intent firing, or asynchronous operations not completing in time.

Fixing Issues:

- Ensure the login logic in LogIn is correctly implemented.
- Replace Thread.sleep with Espresso Idling Resources for more reliable synchronisation.
- Verify that HomepageActivity is correctly started upon successful login.

2. Test: testInvalidLoginAttempt()

Purpose: This test checks the behaviour of the application when an invalid login attempt is made.

Implementation:

- Inputs an incorrect email and password.
- Click the login button.

Expected Result:

- The test currently lacks assertions to verify the outcome of an invalid login attempt.

- To be effective, it should check for a failure message or other indications of a failed login.

Fixing Issues:

- Add assertions to check for the presence of error messages or UI changes that indicate a failed login.
- Ensure that the LogIn activity handles invalid credentials correctly and provides user feedback.

3. Test: testUIElementsVisibility()

Purpose: This test verifies the visibility of essential UI elements in the LogIn activity.

Implementation:

- Checks if the email address input, password input, and login button are displayed.

Expected Result:

- The test should pass if all the UI elements are visible.
- A failure indicates a problem with the layout or visibility of these elements.

Fixing Issues:

- Ensure that all UI elements are correctly initialised in the LogIn activity.
- Check the layout file for LogIn to ensure that all elements have the correct visibility settings.

SettingTest

1. Test: testNotificationSettingButton()

Purpose: This test checks the functionality of the notification settings button within the Setting activity. It verifies that clicking this button opens the NotificationAlarm activity.

Implementation:

- Simulates a click on the notification settings button (R.id.btn_notification).
- Verifies if the NotificationAlarm activity is started using intended(hasComponent(NotificationAlarm.class.getName())).
- Checks if the Setting activity finishes after this action.

Expected Result:

- The test should pass if the NotificationAlarm activity is correctly started upon clicking the button.
- If the test fails, it could be due to the intent not being fired as expected, or the target activity (NotificationAlarm) not being correctly specified.

Fixing Issues:

- Ensure that the correct intent is fired in the Setting activity when the notification settings button is clicked.
- Confirm that the NotificationAlarm class name is correctly referenced in the test.

2. Test: testFeedbackSettingButton()

Purpose: Similar to the first test, this one checks the functionality of the feedback settings button in the Setting activity. It validates that clicking this button opens the Feedback activity.

Implementation:

- Simulates a click on the feedback settings button (R.id.btn_feedback).
- Verifies if the Feedback activity is started using intended(hasComponent(Feedback.class.getName())).
- Checks if the Setting activity finishes after this action.

Expected Result:

- The test should pass if the Feedback activity is correctly started upon clicking the button.
- A failure indicates an issue with the intent firing or the specified target activity (Feedback).

Fixing Issues:

- Ensure that the Setting activity correctly fires an intent to start the Feedback activity when the feedback button is clicked.
- Check that the Feedback class name is correctly used in the test.

User Documentation for Instatt:

1. Logging In

Initiating the Instatt App: Instructions on how to open Instatt on your device, ensuring a smooth start to your attendance management journey.

a. Inputting Login Credentials: Step-by-step guidance on entering your email and password for accessing the app, ensuring secure and accurate login.
b. Confirming a Successful Login: Understanding what to expect post-login, including initial steps and functionalities accessible to the user.

2. App Navigation

Overview of the Main Menu: A detailed walkthrough of the main menu, explaining how to navigate through various features and sections of Instatt.
Utilising the Navigation Drawer: Discover the functionality and options within the navigation drawer, enhancing your app experience.

a. Features of the Homepage: An overview of the homepage's capabilities, highlighting key actions and information available to users.
b. Personal Details: An overview of all information of your personal details.
c. Attendance History: Guidance on accessing and understanding the attendance history, including detailed module information.
d. Settings: Instructions on adjusting app settings, including feedback submission and notification preferences.
e. Log Out: Log Out the account.

3. Sign Attendance

Instatt streamlines the process of signing class attendance for students. The attendance can be signed only when the class status is changed to "unlock" by an admin. Here's a step-by-step guide on how to sign your attendance using the Instatt app.

a. Monitor Class Status:
   - Regularly check your homepage in the InstAtt app for any updates to your class statuses.
   - Classes available for signing attendance will have their status updated to "unlock" by an admin.

b. Identifying the Sign Icon:
   - Once a class is unlocked, you will notice a "sign" icon appearing next to the class status on your homepage.

c. Signing Your Attendance:
   - Tap on the "sign" icon for the unlocked class.
   - Upon clicking, your attendance for that class will be automatically recorded.
   - A confirmation message or indication might appear to signify that your attendance has been successfully signed.

4. Viewing Class History in `Instatt`: Accessing Class Information by using Monthly and Weekly Calendars

Instatt makes it easy for students and teachers to view the history of classes based on specific dates. This feature is particularly useful for keeping track of past classes and their relevant details. Here's how you can use the `Instatt` to view class information for a selected day using the calendar function on the homepage.

a. Navigating the Weekly Calendar
   - Accessing the Weekly View: Open the InstAtt app and go to the homepage. The default calendar view is usually set to 'Weekly.'
   - Browsing Weekly Schedules: Use the arrow or 'Next' and 'Previous' buttons to move between weeks. This allows you to view classes scheduled in the past or upcoming weeks.

b. Switching to the Monthly Calendar
   - Changing to Monthly View: Tap on the 'Monthly' button, usually located at the top or near the calendar view. This will switch the display to a full month's calendar.
   - Selecting a Month and Date: Browse through the months to find the one you are interested in. Tap on a specific day in the month to view the classes scheduled for that date.

c. Returning to Weekly View for Specific Date
   - Accessing Weekly View for Selected Date: After selecting a day in the monthly view, tap the 'Weekly' button at the top of the

calendar. This action will bring you back to the weekly view, focusing on the week of the selected day.

5. Password Modification

a. Accessing the Password Reset Function: To start updating your password, go to the app's navigation menu and select the "Personal Detail" page. This is where the password reset feature is located.
b. Procedure for Password Update: Begin the password change process by first entering your existing password. Then, proceed to input your new password, following the provided guidelines for a secure update.
c. Finalising Your Password Change: Understand the final steps in the password update process, including the confirmation mechanisms that ensure your password has been securely and successfully changed.

6. Attendance Module Review

Accessing Your Attendance Records: Explore your attendance records with an emphasis on specific course details.

This includes:

- Module Code: Each course's unique identifier.
- Module Name: The official title of the course.
- Class Attendance Percentage: The proportion of classes attended out of the total classes conducted for each module.

7. Navigating to Detailed Module Views

a. Upon selecting a specific module in the "Attendance History" section, you will be taken to a detailed view of that module. This comprehensive view provides an in-depth analysis of your class participation with the following details:
    - Total Classes: Displays the complete count of scheduled classes for the chosen module.
    - Absent Classes: Indicates the number of classes you were marked absent.
    - Attended Classes: Shows the tally of classes you have attended.
    - Excused Classes: Lists the classes you missed but have been excused from.
    - Class Attendance Percentage: Represents your overall attendance in percentage form, calculated against the total classes.
b. Viewing Class List and Details: You can also view specific details about each class session within the module:
    - Class Date: This field shows the date on which each class was conducted.
    - Class Status: Indicates your attendance status for each class, such as 'Present', 'Absent', or 'Excused'.

- Class Duration: Details the length of each class session.
- Class Location: Provides information on where each class was held, be it a physical location or a virtual platform.

8. Exporting Attendance History from Module Details

The Instatt app provides a convenient feature for users to export their attendance history. This function is particularly useful for maintaining personal records, academic reviews, or sharing attendance data. The following steps guide you through exporting attendance history from the module details page in the `Instatt` application.

a. Accessing Module Details: Open the `Instatt` app and navigate to the 'Attendance History' section. Select the specific module whose attendance details you wish to export.
b. Locating the Export Option: In the module details page, look for the 'Export' button. This is typically located at the top of the page or within a dropdown menu.
c. Initiating the Export Process: Tap on the 'Export' button. You may be prompted to choose the preferred format for the export, such as CSV, Excel, or PDF.
d. Choosing the Export Format: Select the format that best suits your needs. CSV or Excel is ideal for data manipulation, while PDF is suitable for viewing and printing.
e. Confirming and Downloading the File: Confirm your choice, and the app will generate the file, which will then be automatically downloaded to your device.

9. Sending Absentee Report from Module Details

`Instatt` includes a feature that allows users to send an absentee report directly from the module details. This functionality is essential for maintaining accurate records and notifying relevant parties about attendance discrepancies. Follow these steps to send an absentee report from within a specific module in the `Instatt` application.

a. Accessing the Module Details: Launch the `Instatt` app and navigate to the 'Attendance History' section. Select the module from which you want to send an absentee report.
b. Finding the Absentee Report Option: On the module details page, look for the option labelled 'Report' button with red image icon in the list of classes.
c. Selecting Specific Absences: Choose the specific dates or classes for which you want to report an absence. You may be able to fill the form following the detailed content of the form, you can also upload a file as attachment from doctors or anywhere as support materials as well for each absence.

d. Confirming and Sending the Report: Review the information that you filled in the form. Confirm the details and tap on the 'Send' or 'Submit' button to dispatch the absentee report.

10. Understanding Background Colour Coding for Class Locations

Instatt simplifies the process of finding your class location through an intuitive colour-coded system. Each class in the app is assigned a specific background colour that corresponds to different areas or buildings within the educational institution. This section explains how to interpret these colours for easy navigation to your classes.

11. Log Out

To ensure the security of your account and personal information, InstAtt provides convenient logout options both in the 'Personal Detail' page and the side navigation drawer. Here's how you can securely log out from either of these locations within the app.

a. Logging Out from the Personal Detail Page
   - Navigate to Personal Detail: Access the 'Personal Detail' page from the side navigation drawer.
   - Find the Logout Option: Scroll through your personal details to find the 'Logout' or 'Sign Out' button, typically located at the bottom of the page.
   - Confirm Logout: Tap on 'Logout' or 'Sign Out' and confirm your action if prompted. This will securely end your session and log you out of the app.

b. Logging Out from the Side Navigation Drawer
   - Access the Side Drawer: Tap on the hamburger icon (three horizontal lines) or swipe right from the left edge of the screen to open the side navigation drawer.
   - Locate the Logout Button: The 'Logout' or 'Sign Out' option is usually found at the bottom of the navigation drawer.
   - Complete the Logout: Select 'Logout' or 'Sign Out' and confirm if asked. The app will log you out and redirect you to the login screen.

12. Application Settings

Learn how to access and modify settings within Instatt to tailor the app to your preferences.

a. How to Provide User Feedback: Details on submitting feedback through the app, contributing to continuous improvement and user satisfaction.

b. Managing Notification Preferences: Instructions for customising notification settings to suit your personal needs and preferences.

Deployment:

Finalise App Features and Testing:

- Ensure that all features are complete and working as expected.
- Perform thorough testing, including unit tests, integration tests, and UI tests.
- Conduct beta testing with a group of users to gather feedback and catch any last-minute issues.

Prepare the App for Release:

- Remove all debugging code and logs from the app.
- Update version information in the app's build.gradle file (versionCode and versionName).
- Optimise resources, such as images and layouts, for different screen sizes and densities.
- Confirm that all app permissions are justified and explained in the app's listing.

Configure ProGuard/R8 for Code Obfuscation:

- Enable ProGuard or R8 in your build.gradle file to obfuscate and minimise your APK, which helps to protect your code and reduce APK size.

Generate a Signed APK or App Bundle:

- Create a keystore and a key if you haven't already.
- Use Android Studio's 'Build > Generate Signed Bundle / APK' option to generate a signed APK or an AAB (Android App Bundle) which is now recommended by Google.

Set Up Production Environment:

- If using services like Firebase, ensure you switch to the production environment with appropriate database rules, storage buckets, and cloud functions.

Prepare the App Store Listing:

- Create a compelling app store listing with screenshots, a detailed description, feature graphics, and videos if necessary.
- Provide a privacy policy, especially if your app handles user data.

Publish the App on the Google Play Store:

- Follow the steps in the Google Play Console to create a new app listing.
- Upload the APK or AAB to the Google Play Console.
- Set the app's content rating and pricing options.

Post-Launch Monitoring and Updates:

- Monitor crash reports and analytics to understand how users are interacting with the app.
- Respond to user feedback and reviews to improve the app.
- Plan for and implement regular updates to keep the app fresh and secure.

Marketing and Promotion:

- Develop and execute a marketing plan to increase visibility.
- Leverage social media, content marketing, and other channels to attract users.

Legal and Compliance:

- Ensure that the app complies with all legal requirements in your target markets, including data privacy laws and intellectual property rights. Maintain Continuous Integration/Continuous Deployment (CI/CD):
- Set up a CI/CD pipeline to automate building, testing, and deploying the app.
- Utilise beta tracks in Google Play for staged rollouts to catch any potential issues in production.

Assumptions

1. Students have access to devices that meet the app's operating system and hardware requirements.
2. Students possess a basic understanding of how to operate a smartphone and navigate through apps.
3. The university provides all users (students, teachers, administrators) with unique Campus IDs and passwords which are necessary to log in to the app.
4. Students have a stable internet connection to access real-time data, such as class schedules, and to perform tasks like marking attendance or submitting forms.
5. The university's IT infrastructure can support the app, especially during peak times like class check-ins, ensuring system responsiveness and reliability.
6. The geofencing technology used for marking attendance is accurate and the students' devices are capable of supporting this feature.
7. Students attend classes in-person; thus, the need for physical presence to mark attendance is valid and enforceable.
8. The university's database is up to date with all necessary student and class schedule information, and there's a robust synchronisation mechanism between the database and the app.

9. Students are informed and consent to the collection of personal data required for the app's operation, such as geolocation data for attendance marking.
10. The app’s notification system can interface with the university's email system to send out alerts, such as absence notifications.
11. Students will report their absences truthfully and provide necessary documentation where required.
12. The feedback and reporting features are monitored, and the information provided by students will be used to improve the app's functionality and student experience.
13. All students have access to an email account that is regularly checked so they can receive notifications from the app in a timely manner.
14. Students will log out after each session to maintain account security, especially when using shared or public devices.