Assessment 2: Report

COMP2000 CW2, Corey Richardson

Introduction

This report showcases the design, development and implementation of an employee management application designed to be used by employees and administrators of a company. The application was developed as coursework for COMP2000: Software Engineering II. It provides functionality for employee users to update their personal information and submit Paid Time Off (PTO) / Holiday requests, while administrators can perform a wider set of actions such as managing and updating other employees records and actioning holiday requests by either approval, rejection or resubmission.

Background

The produced application is a management tool for employees and admins within a company. It can be used by Employees to update their personal details, and submit Holiday (PTO) Requests, whereas administrative users can carry out a wider set of features in addition to those mentioned above. These features include viewing and managing employee records via CRUD operations and reviewing and actioning employee's holiday requests.

When employee details or PTO request statuses are updated, a notification is pushed to the user. These notifications ensure users are informed of updates to their information, promoting communication.

The application relies on a backend RESTful API Web Service.

Assessment Materials

GitHub Repository: github.com/Plymouth-COMP2000/coursework-report-corey-richardson

GitHub Release: github.com/Plymouth-COMP2000/coursework-report-corey-richardson/releases/tag/cw2

Video Link: https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/Video/ReportVideo.mp4

WakaTime Project: https://wakatime.com/@coreyrichardson/projects/rwcthymumd?start=2024-12-10&end=2025-01-07

Design and Storyboarding

Class Diagrams

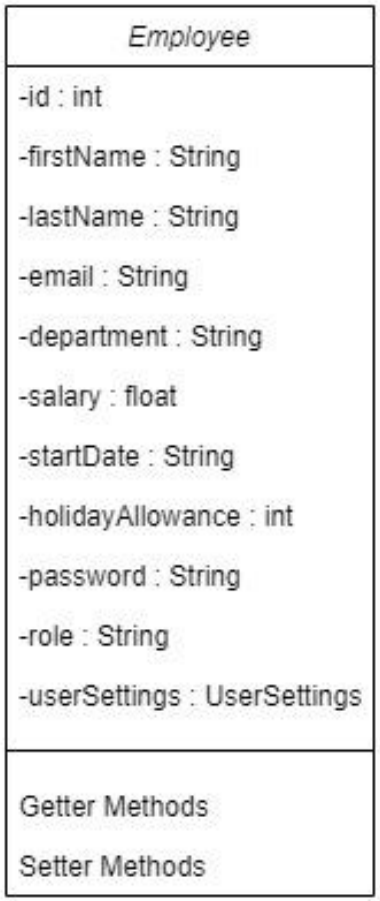


Figure 1: UML Diagram for Employee Class

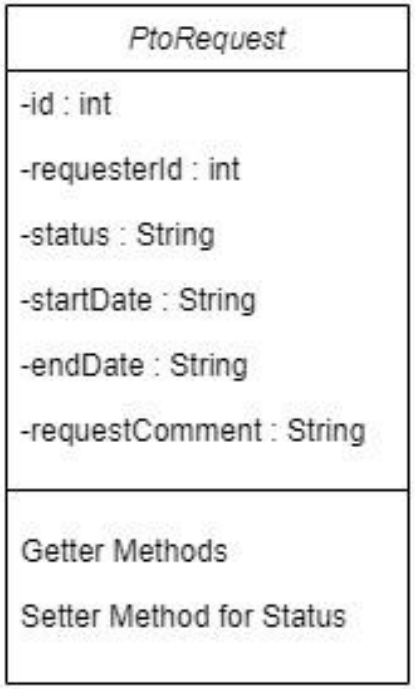


Figure 2: UML Diagram for PtoRequest Class

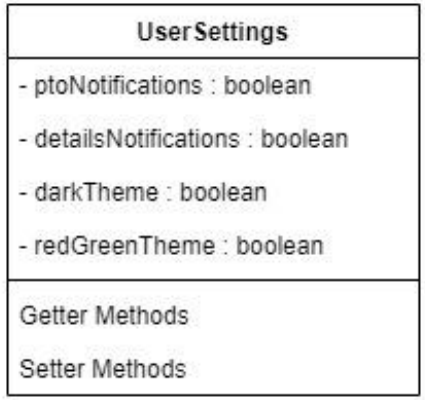


Figure 3: UML Diagram for UserSettings Class

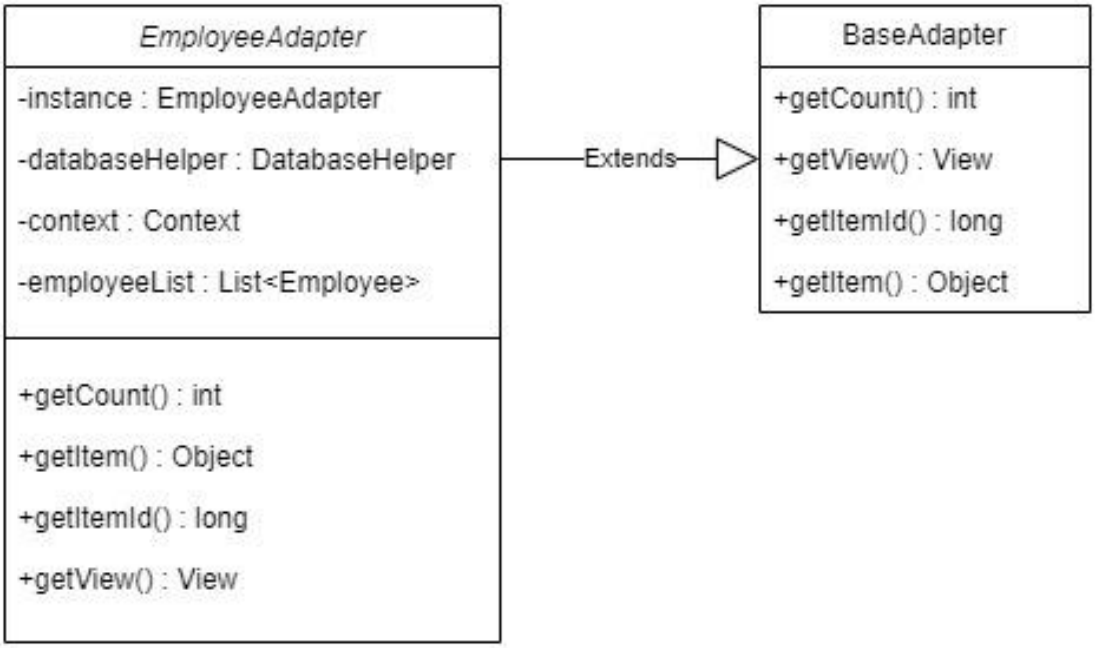


Figure 4: UML Diagram for EmployeeAdapter ListView Adapter

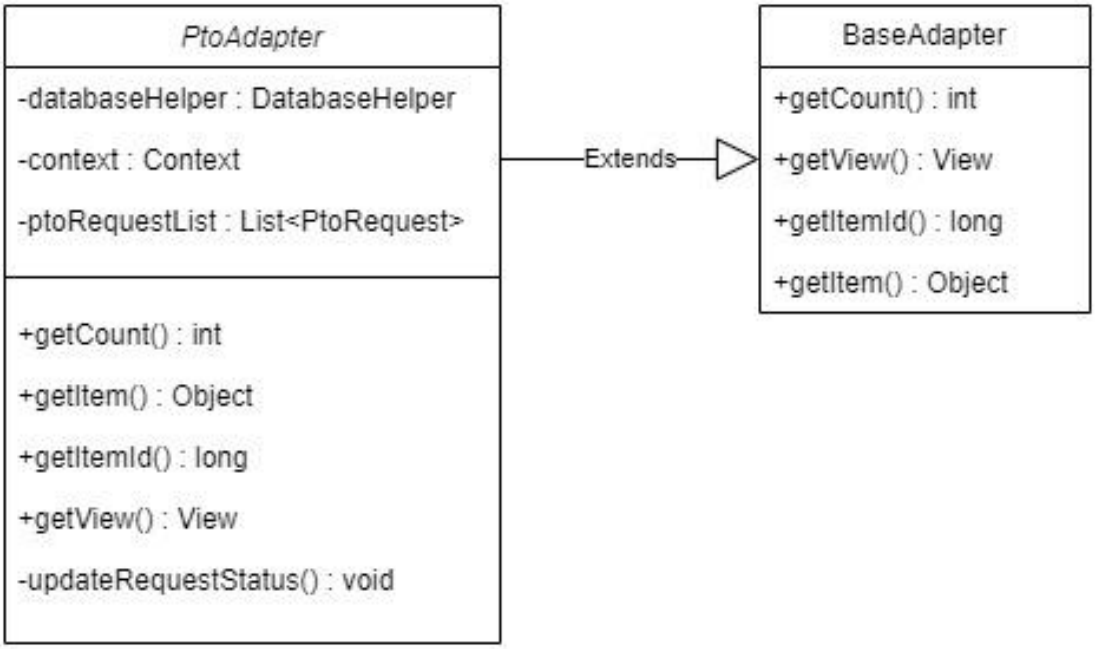


Figure 5: UML Diagram for PtoAdapter ListView Adapter

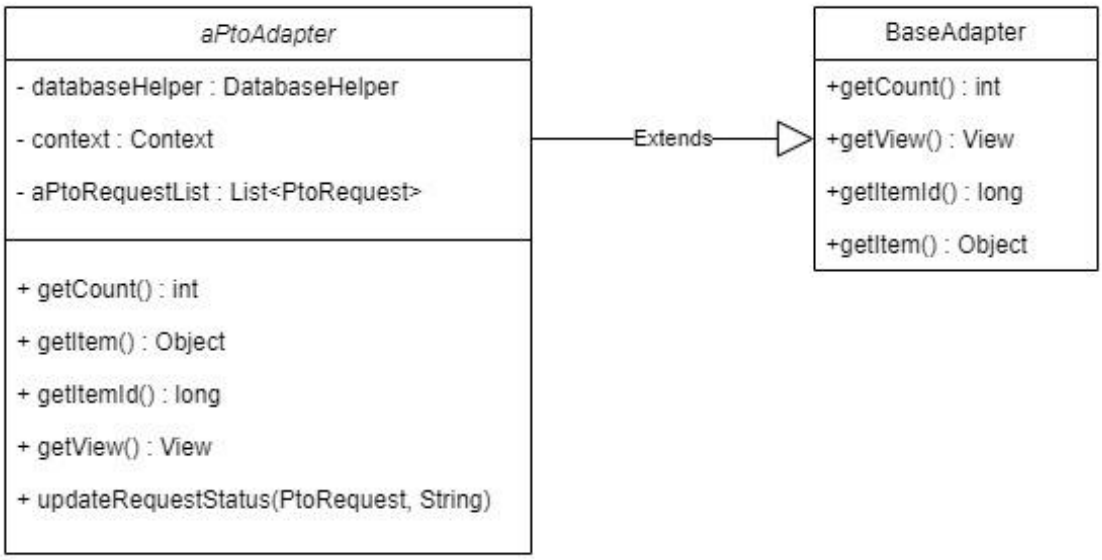


Figure 6: UML Diagram for aPtoAdapter ListView Adapter

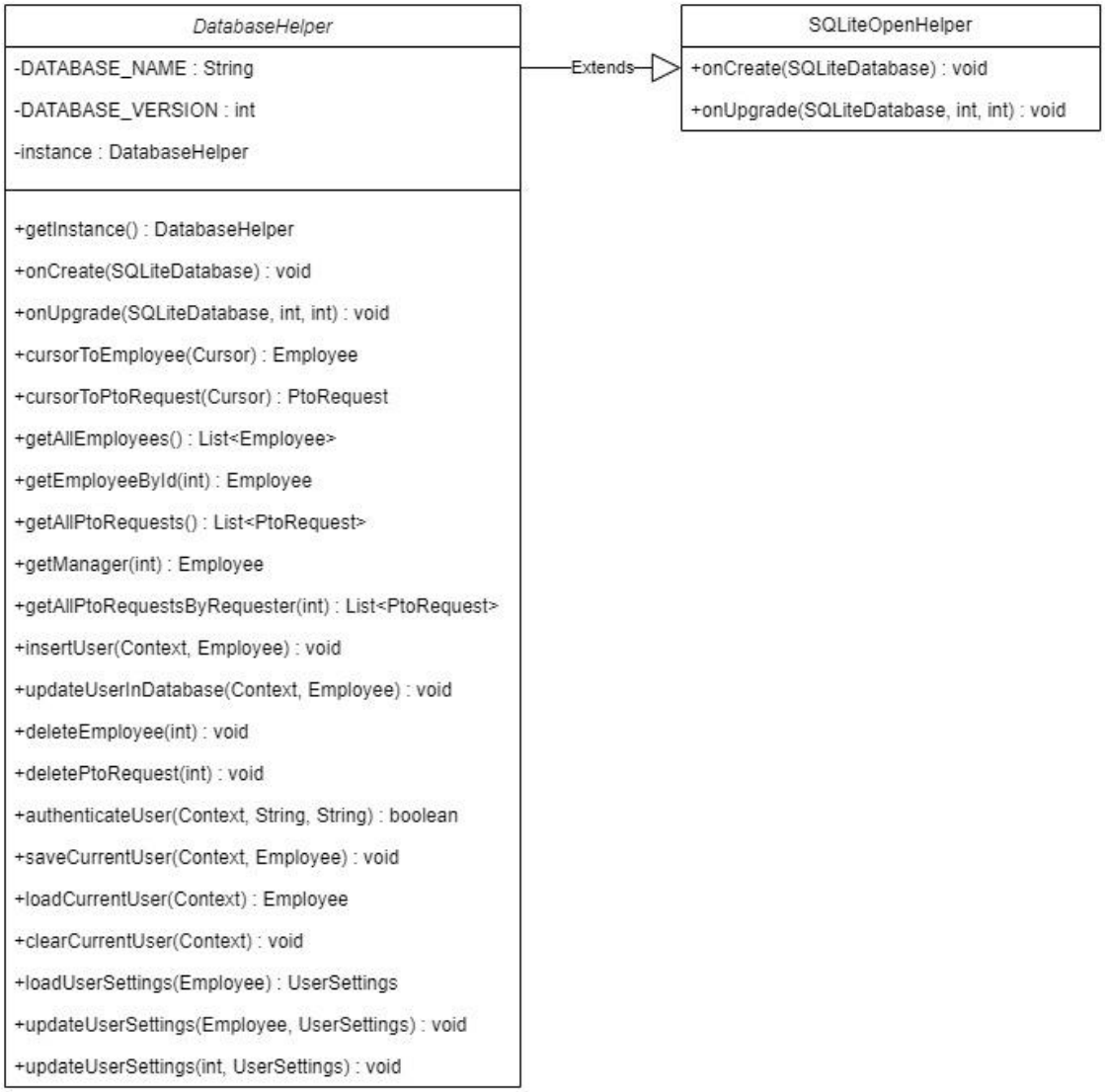


Figure 7: UML Diagram for DatabaseHelper Class

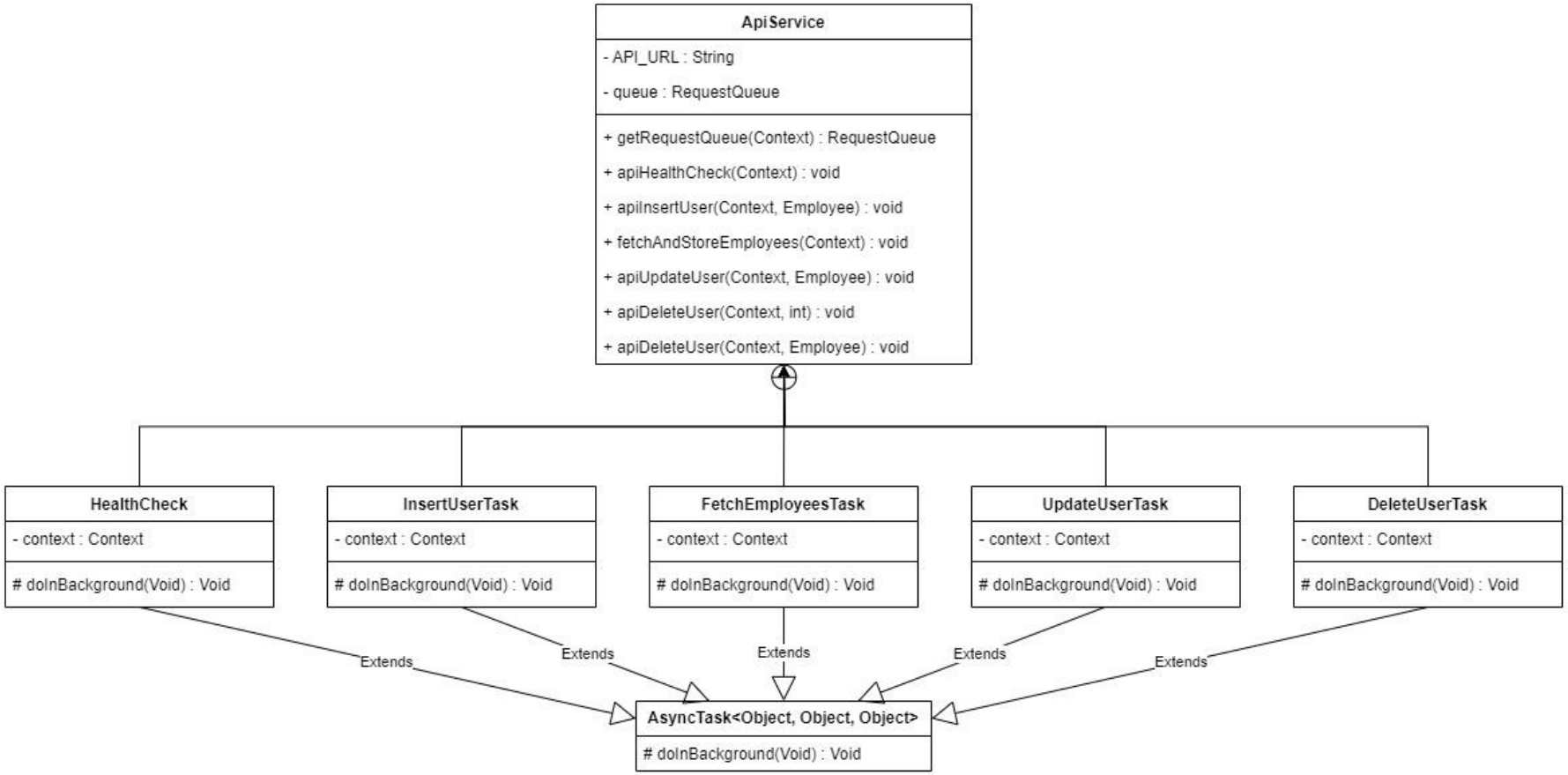


Figure 8: UML Diagram for ApiService Class

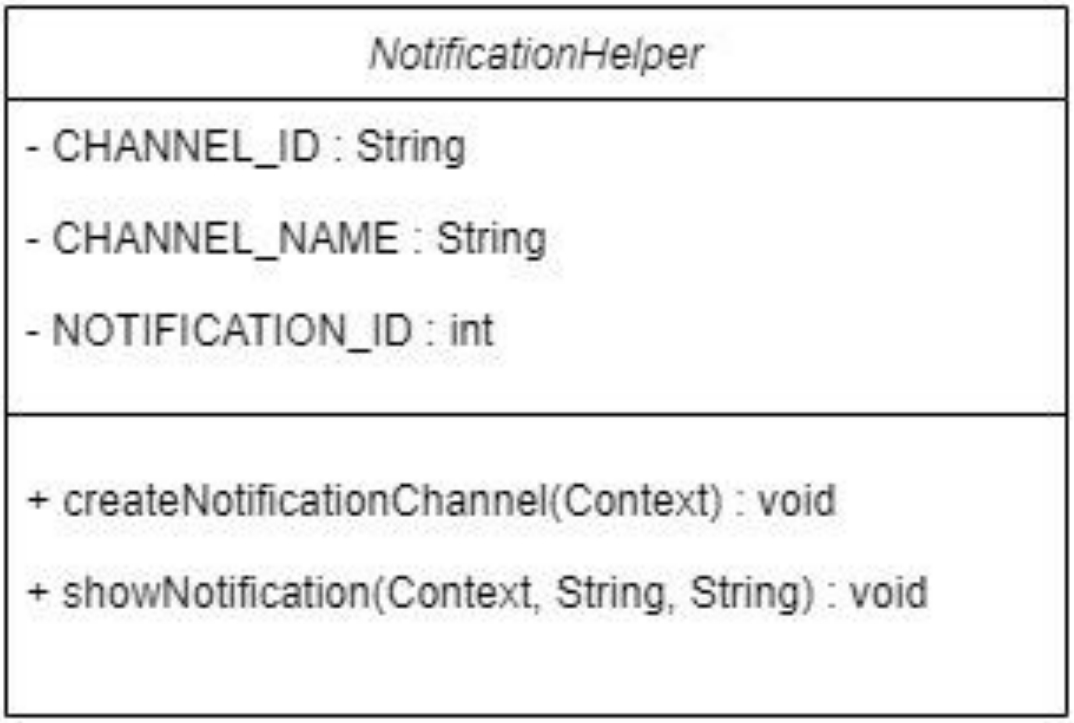


Figure 9: UML Diagram for NotificationHelper Class

Storyboards

This section details how the application could be used by employee and admin users to perform tasks such as updating their personal details and requesting holiday.

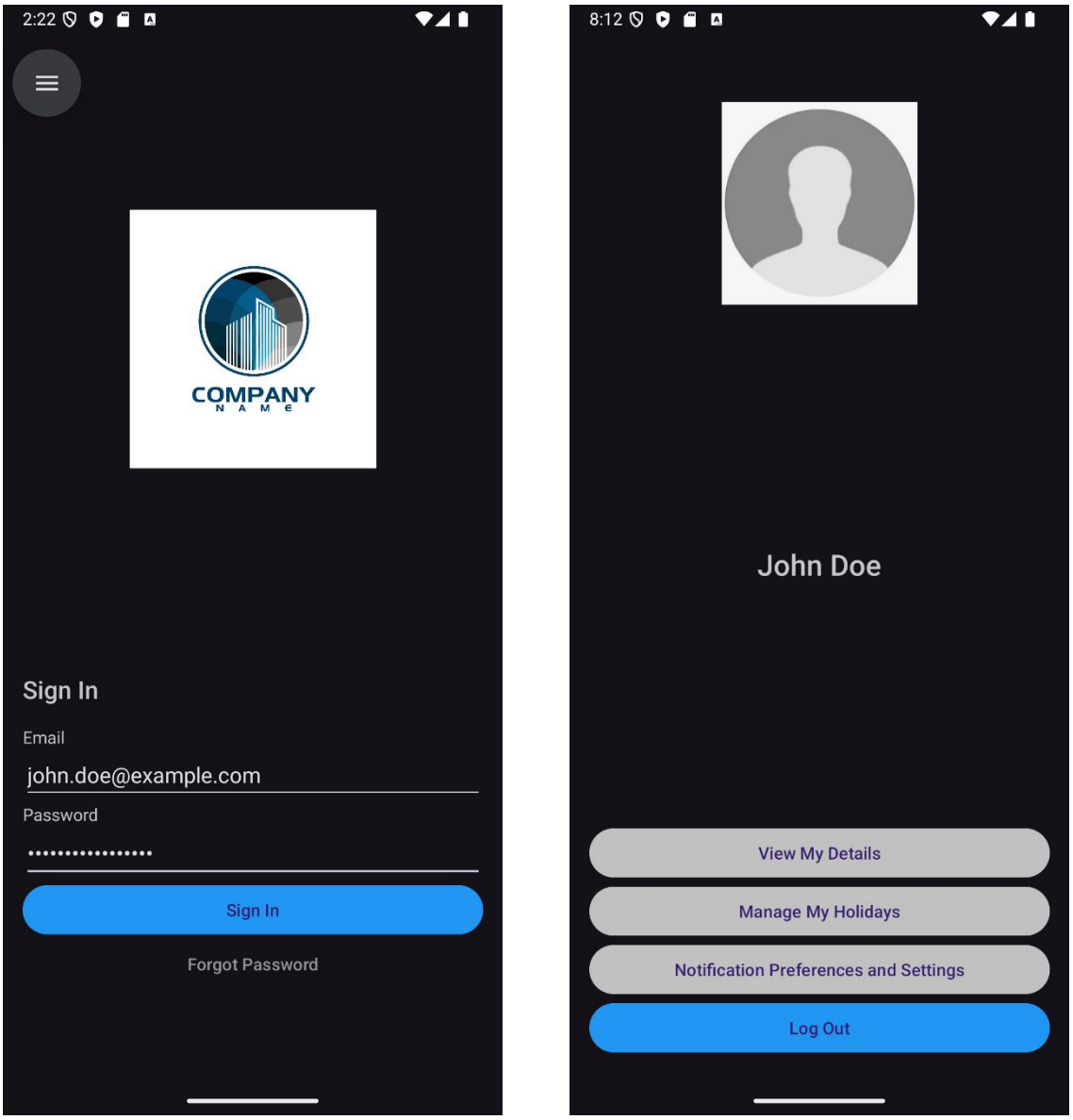


Figure 10: Storyboard for Logging In as an Employee

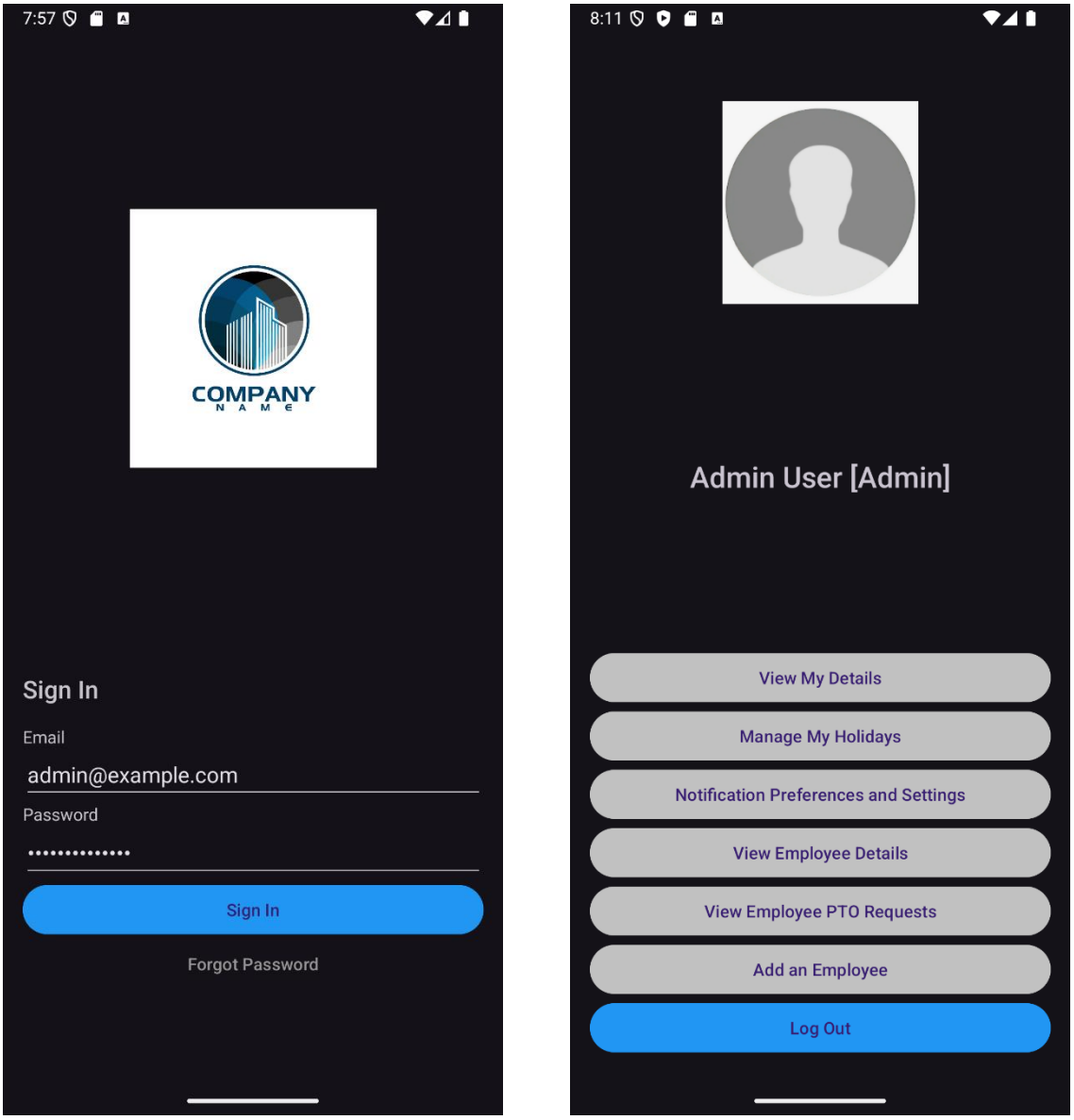


Figure 11: Storyboard for Logging In as an Admin

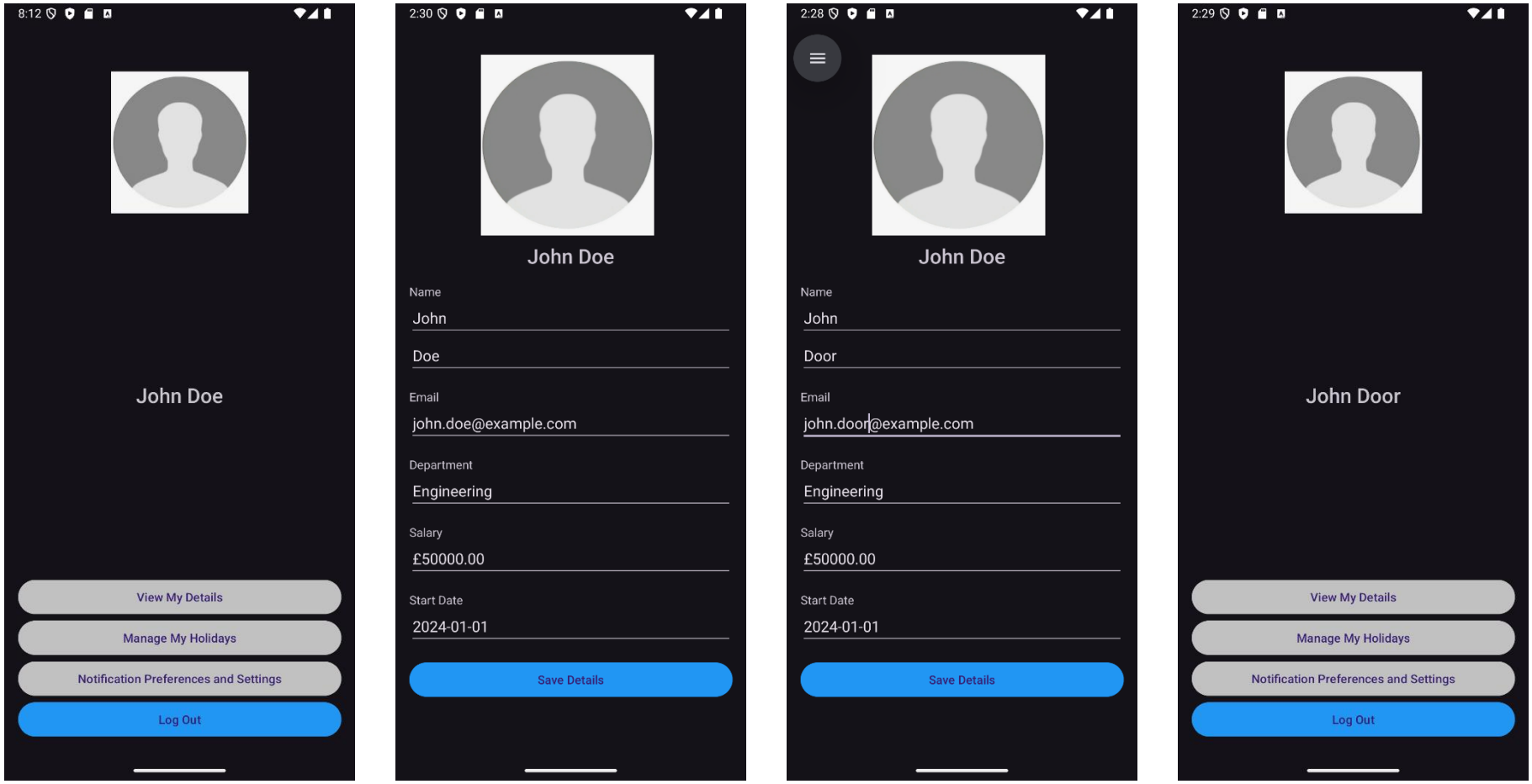


Figure 12: Storyboard for an Employee Updating their Personal Details

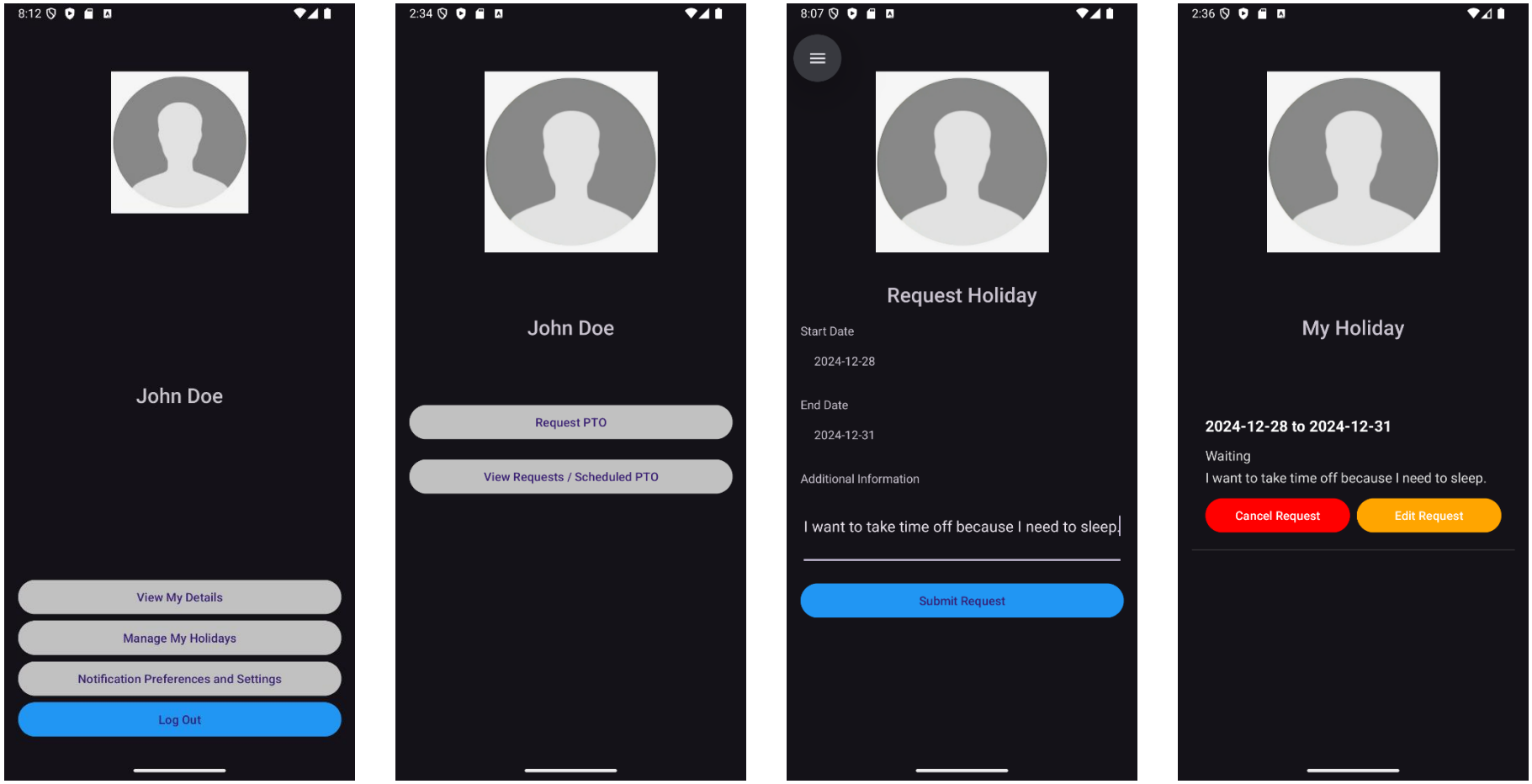


Figure 13: Storyboard for an Employee Requesting Holiday

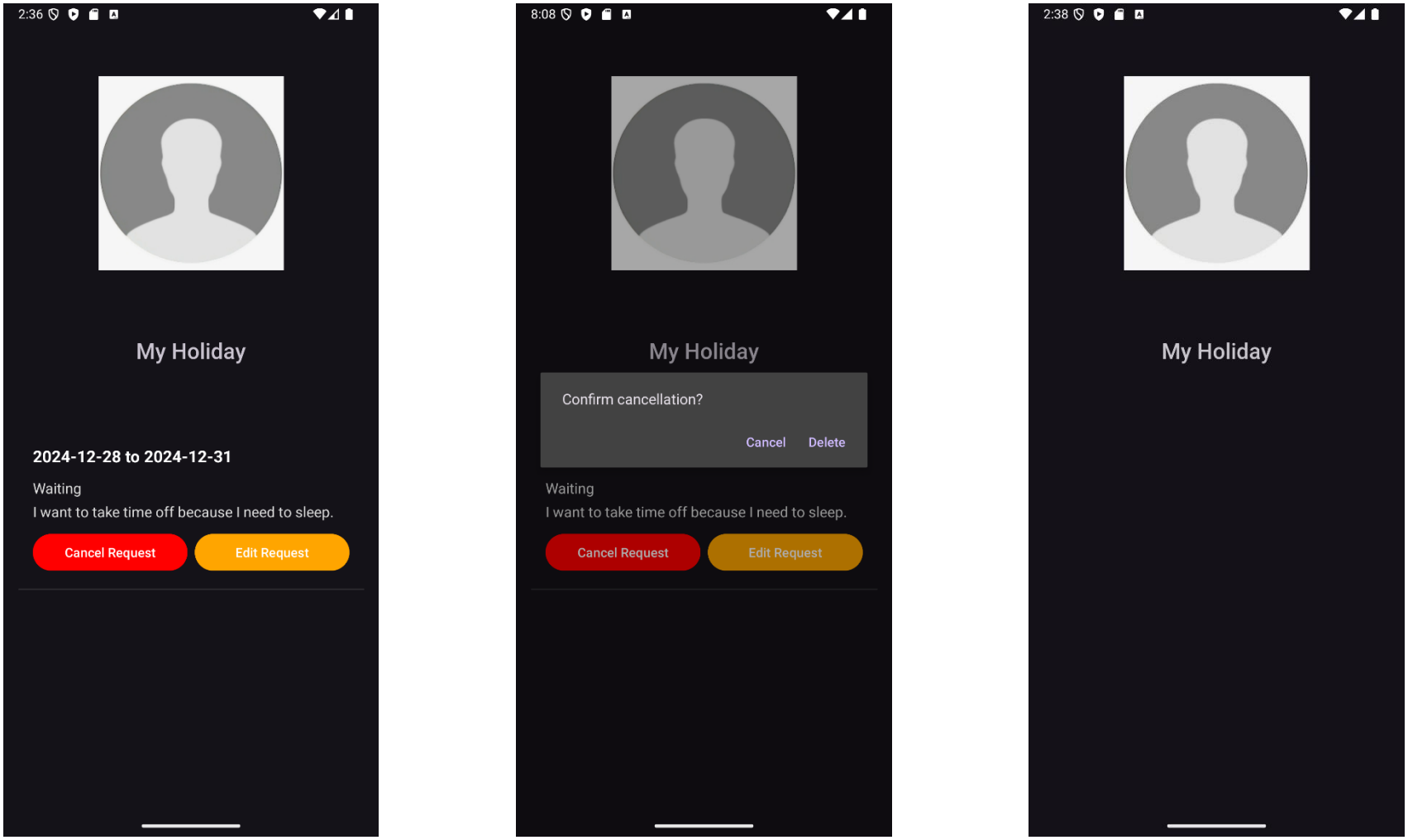


Figure 14: Storyboard for an Employee Cancelling a Holiday Request

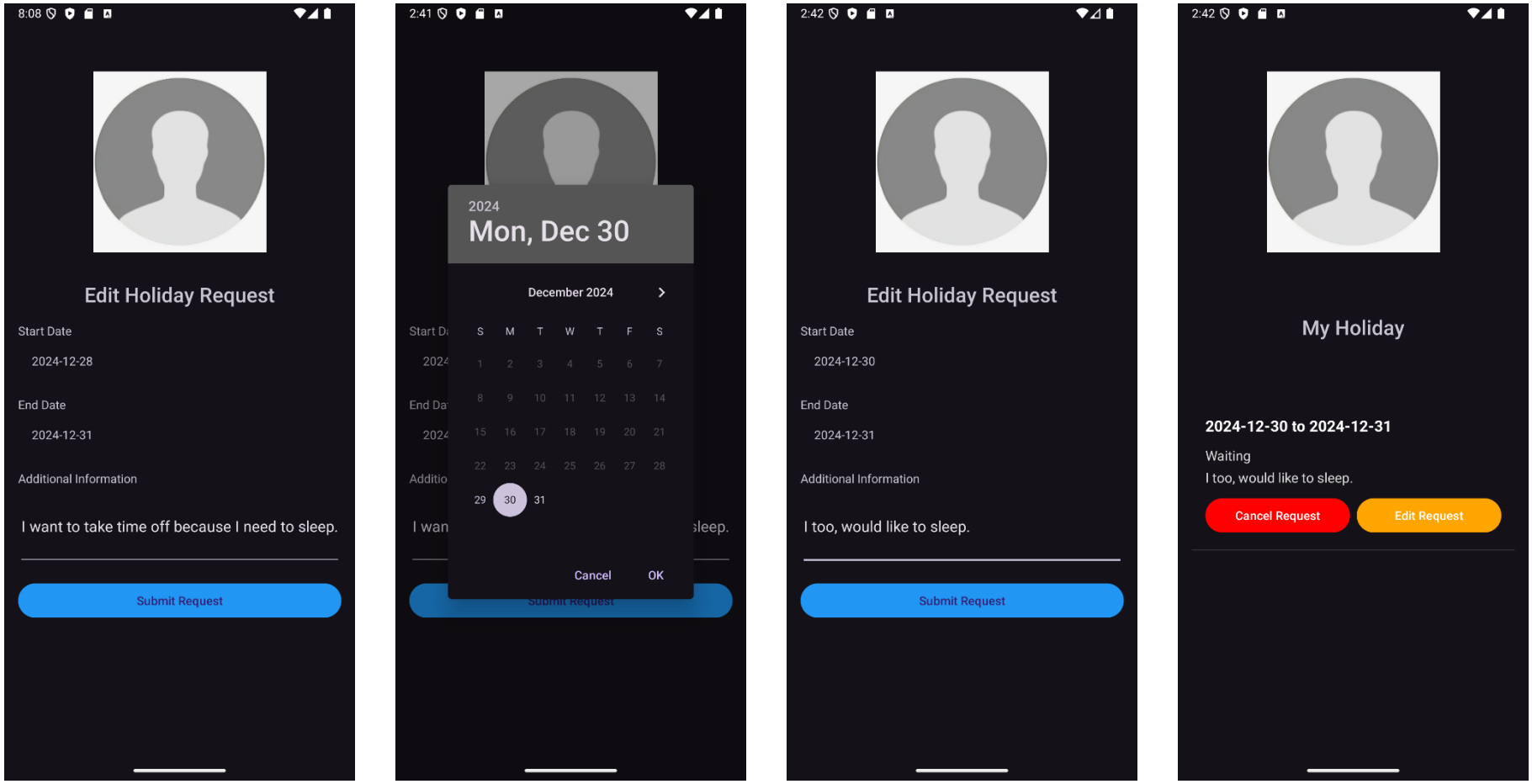


Figure 15: Storyboard for Updating a PTO Request

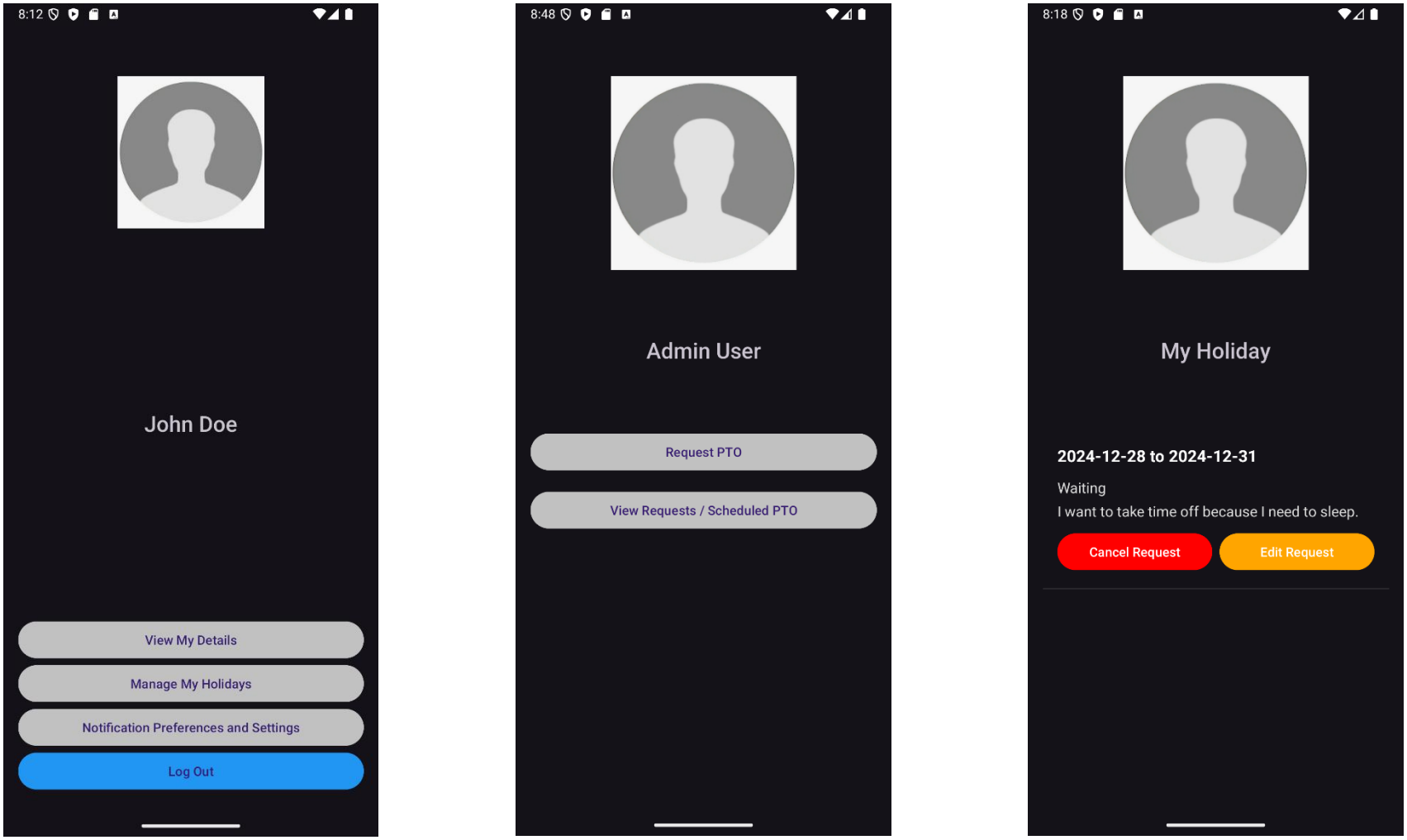


Figure 16: Storyboard for Viewing PTO Requests and Scheduled Holiday

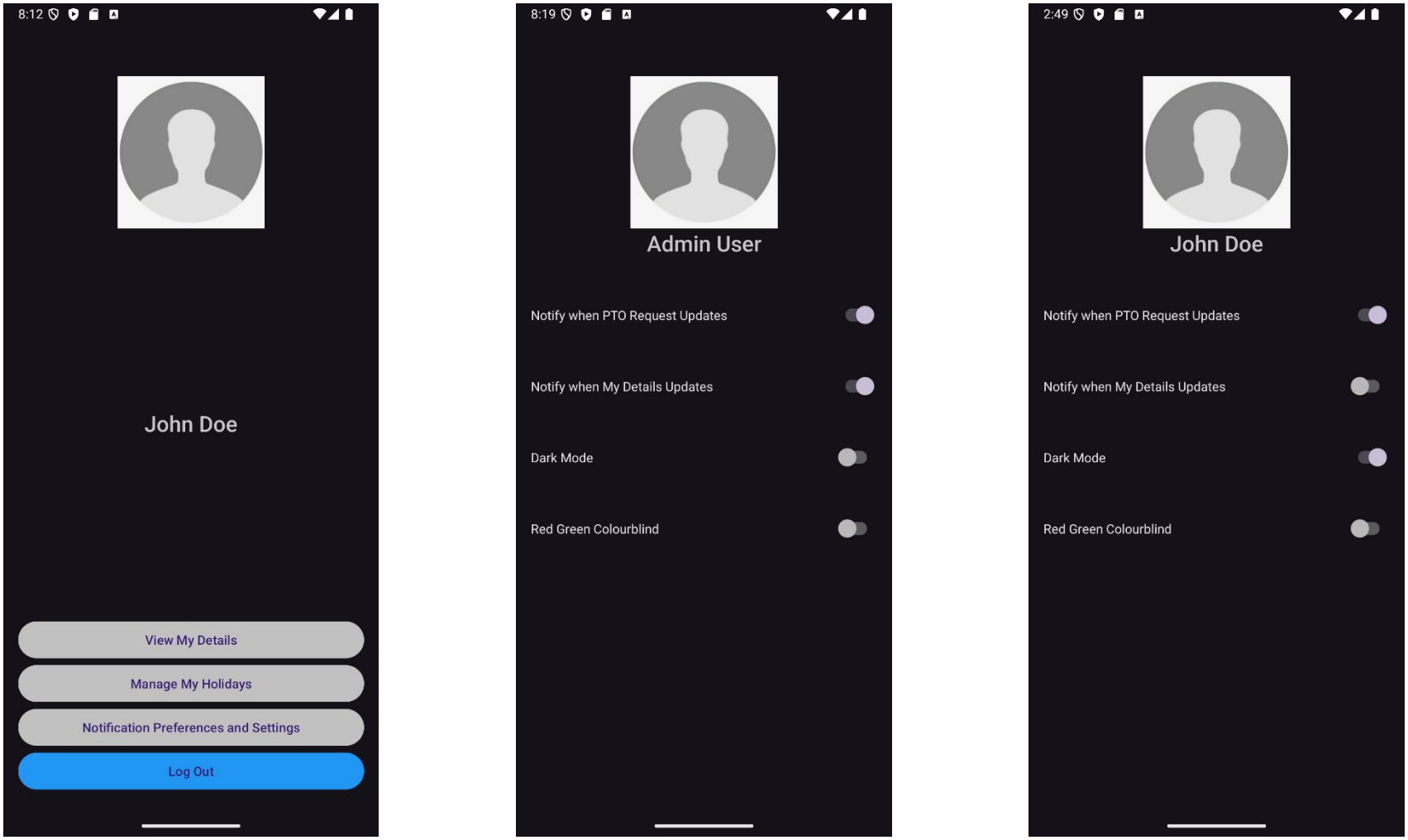
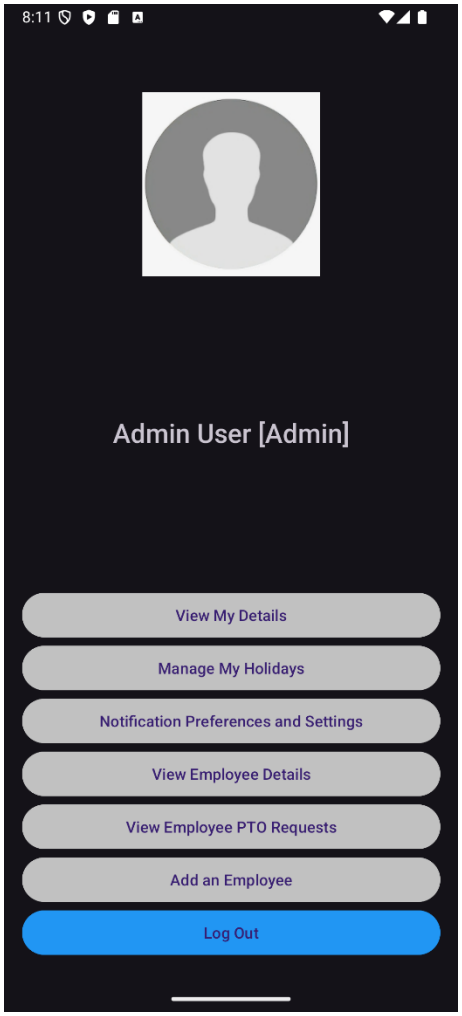


Figure 17: Storyboard for Updating a User Settings



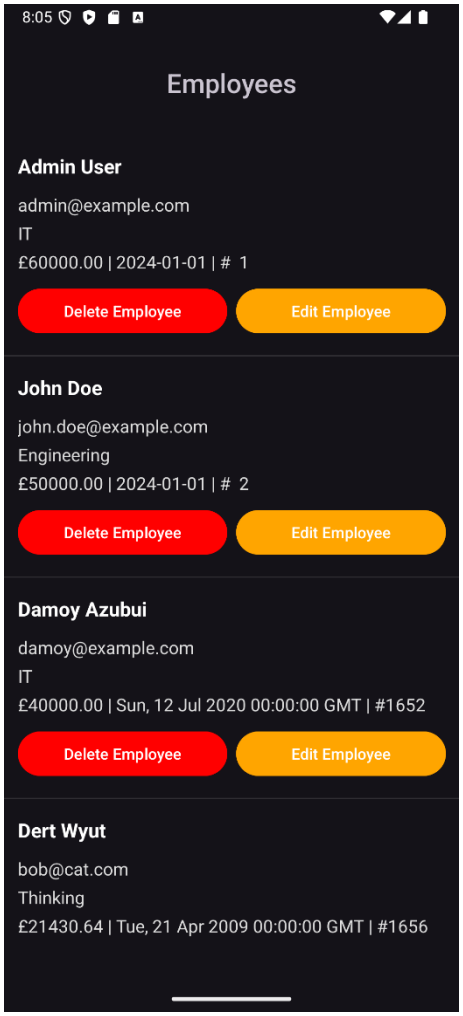


Figure 18: Storyboard for an Admin User Viewing Employee Details

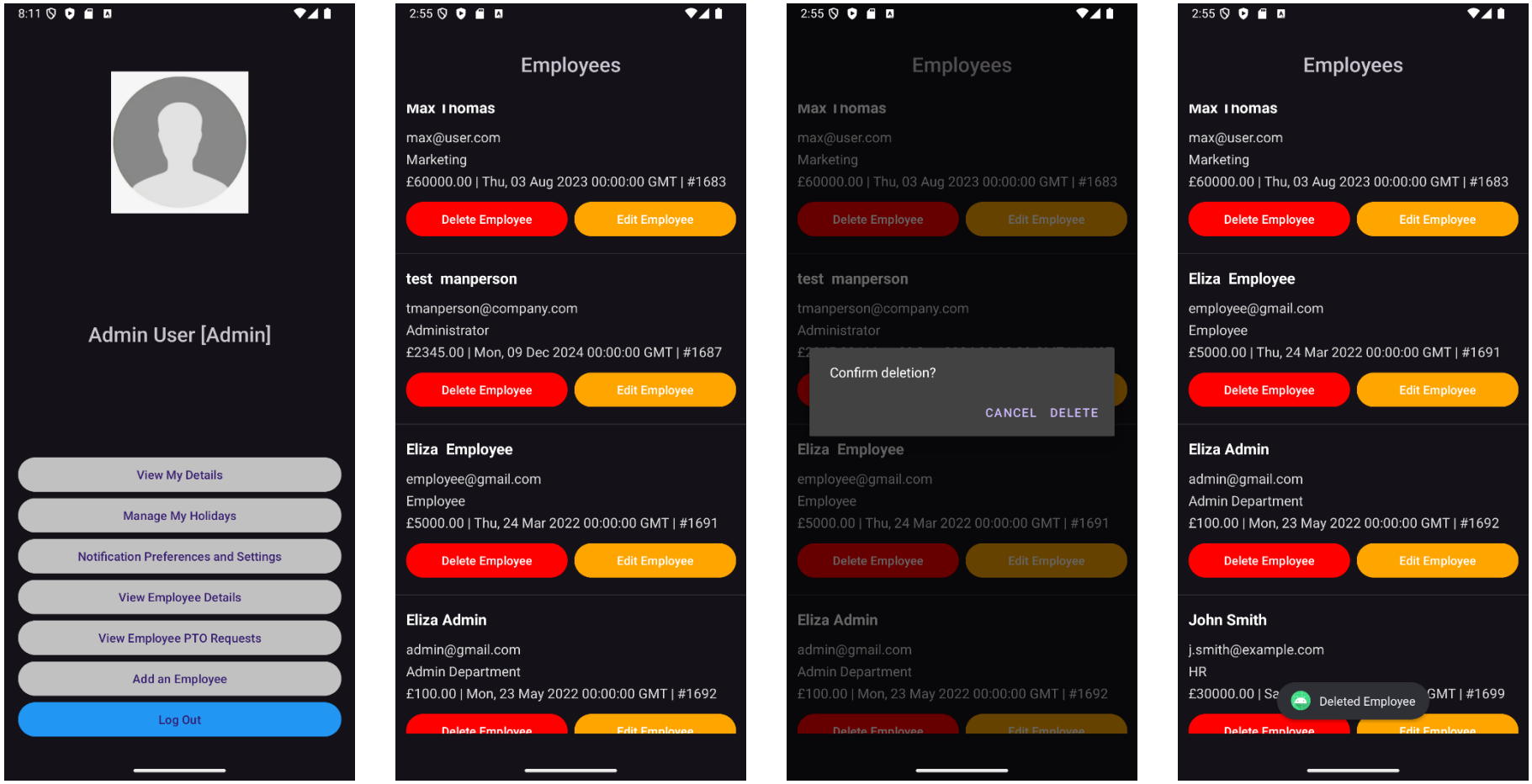


Figure 19: Storyboard for Admin User Deleting an Employee Record

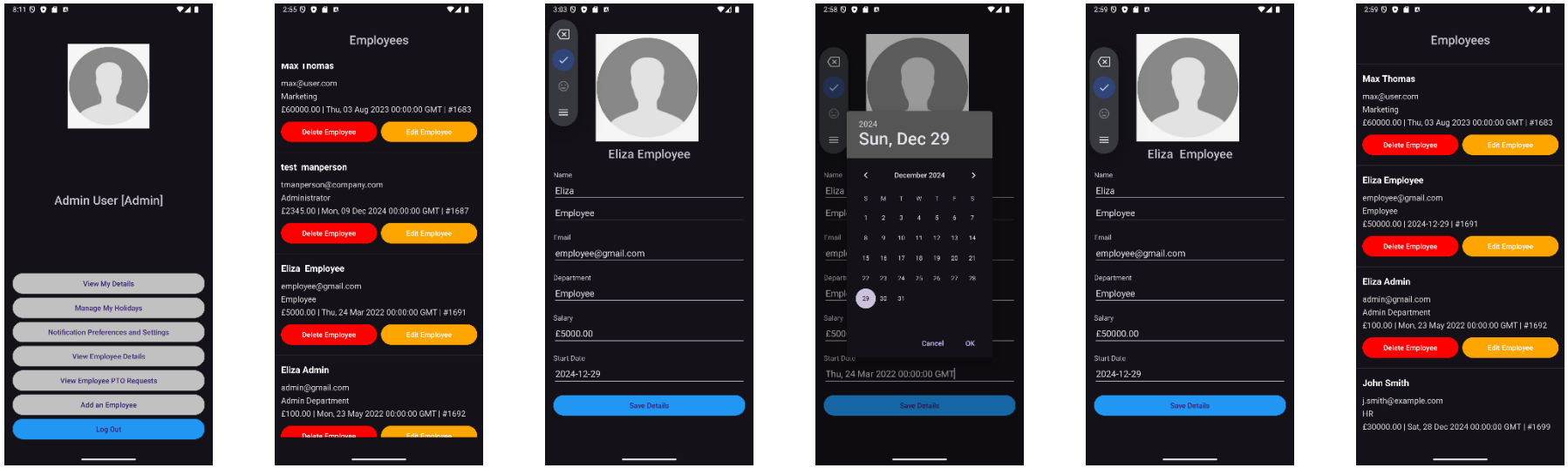


Figure 20: Storyboard for an Admin User Updating an Employees Details

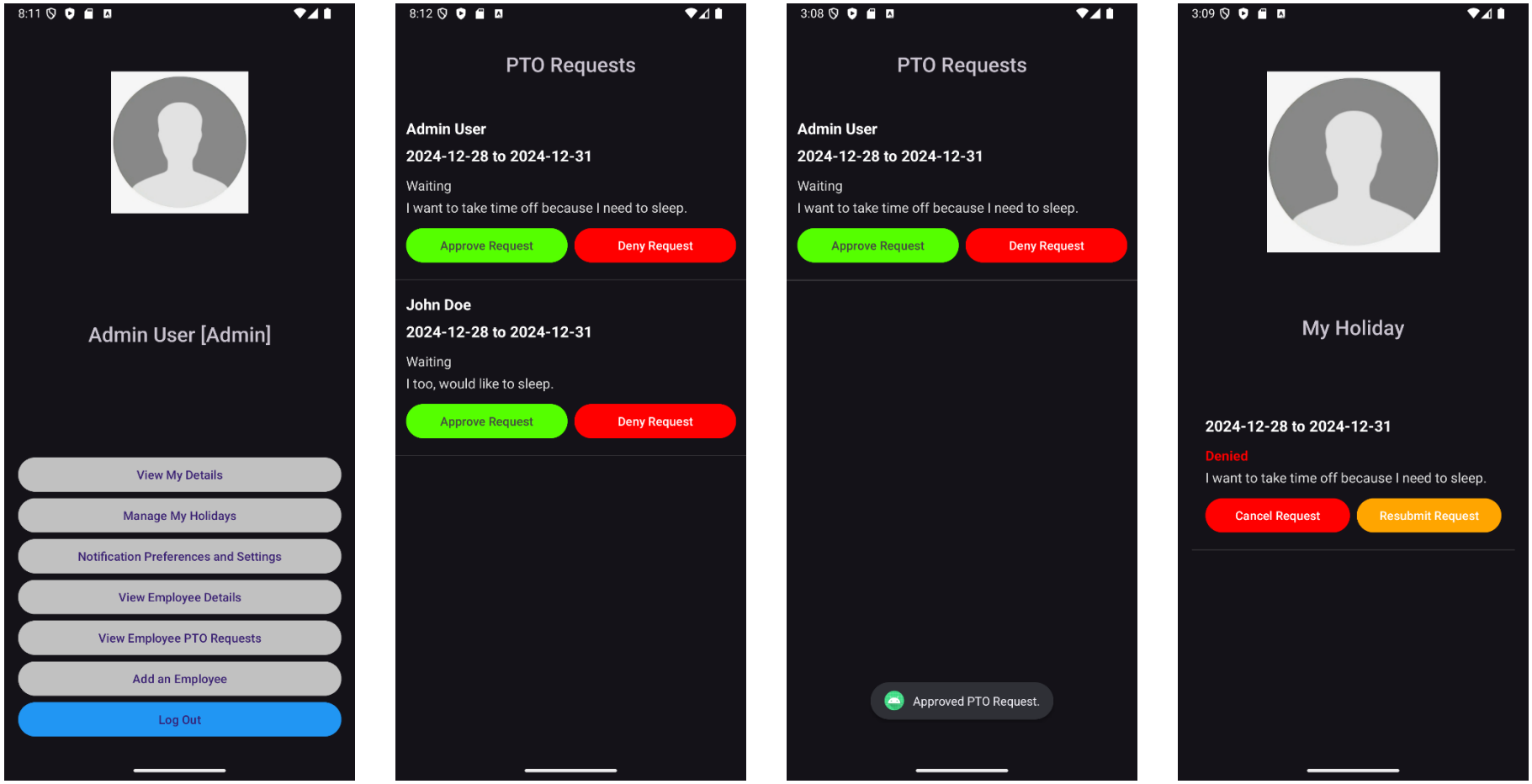


Figure 21: Storyboard for Admin User Actioning a PTO Requests

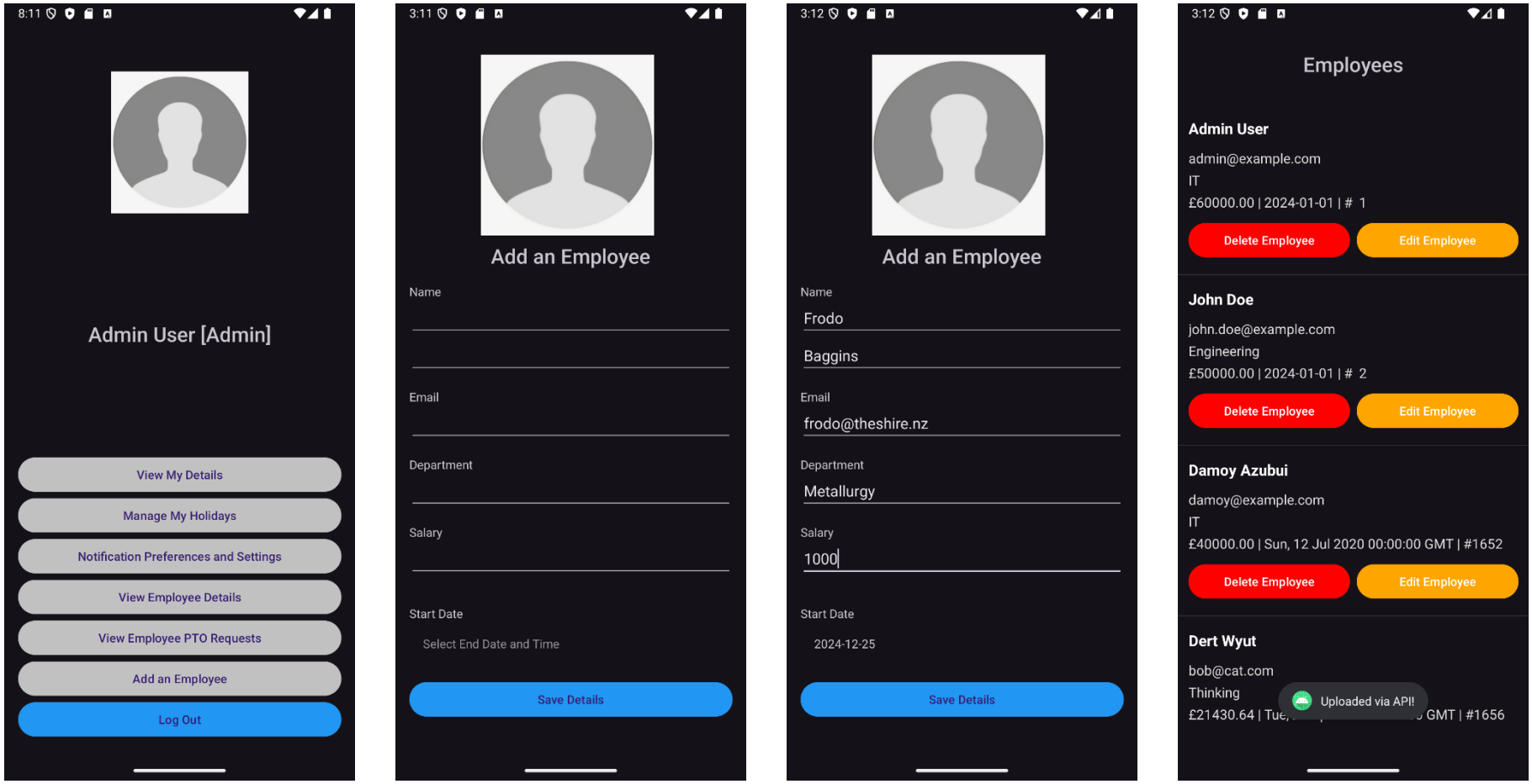


Figure 22: Storyboard for an Admin User Adding a New Employee to the Database and API

Legal, Social, Ethical and Professional Considerations

Protecting user's personal information would be an important aspect of the applications development. The local database stores private information such as the user's password (in plaintext!) and both the local database and the web service API stores Personally Identifiable Information(PII) such as full names, email addresses and salaries. Passwords are stored in plaintext, as encryption falls out of scope of this module's coursework, but this approach is not suitable for a production application.

SQL data transactions are used to ensure that data integrity is maintained; updates to the database are atomic, consistent, isolated and durable (ACID), mitigating the risk of a partial database update compromising data integrity.

The API Web service does not provide a Secure Sockets Layer (SSL) certificate and so communication with the API can only be done over an insecure and unencrypted HTTP connection, exposing the application to data interception attacks.

Ideally, all communication would be handled over HTTPS connections, and passwords would be encrypted using strong encryption algorithms such as SHA-256.

Interface Design

This section displays the XML files, viewed in Android Studio, with a screenshot taken from the **Medium Phone API 35** emulator alongside it. Also included is a link to the XML file in the GitHub repository.

Consistent design was used across activities to improve the user experience of the application by ensuring that users can navigate the layout of the application intuitively. The uniformity of layouts was achieved using common elements like navigation buttons and input fields using a consistent size, style and positioning across all pages.

While not yet implemented, view-accessibility features such as a Dark Theme and a Red Green Colourblind theme were planned, accommodating users with disabilities and light sensitivities. The inclusion of these features in a future iteration would enhance the applications usability for a larger market audience.

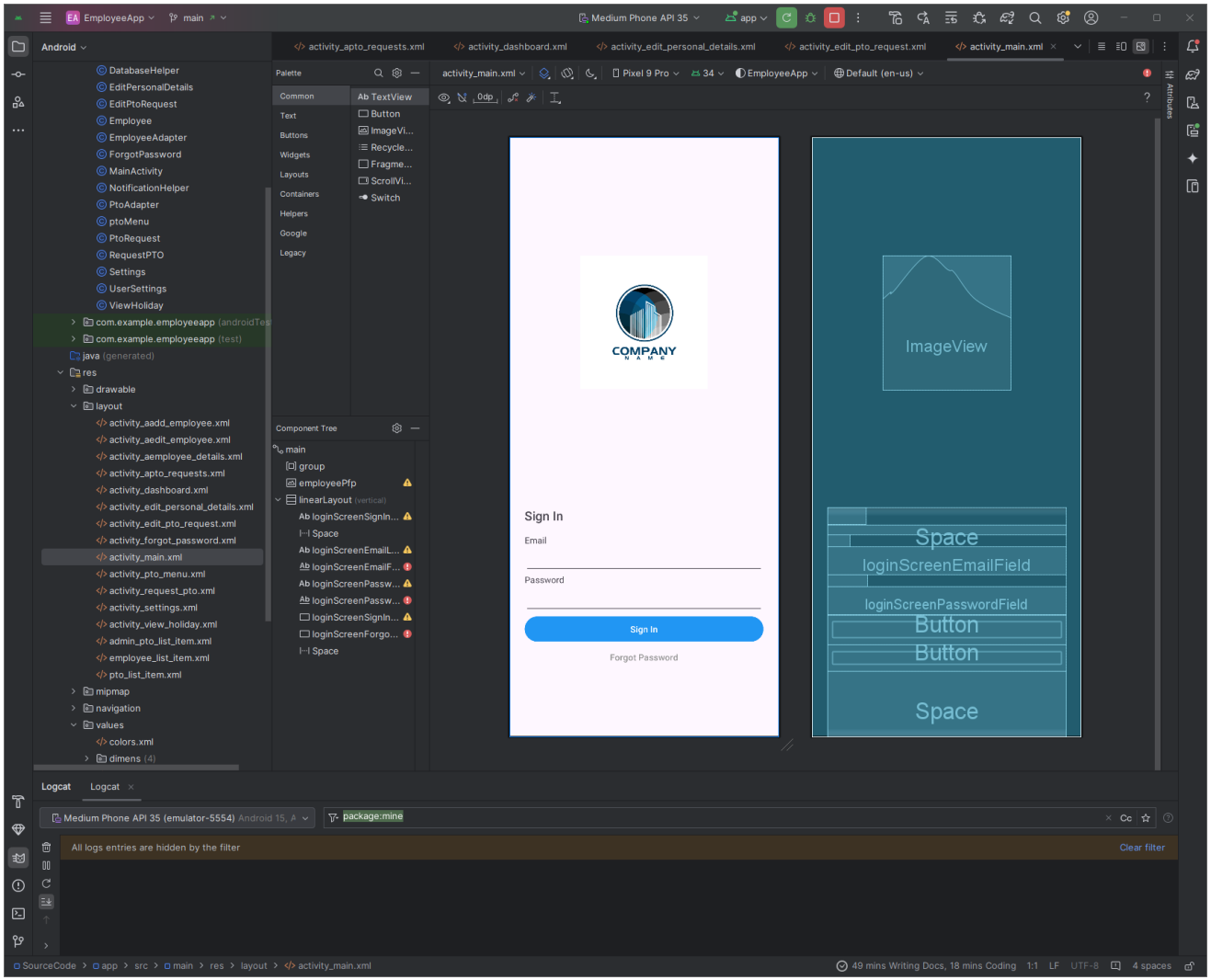


Figure 23: activity_main.xml

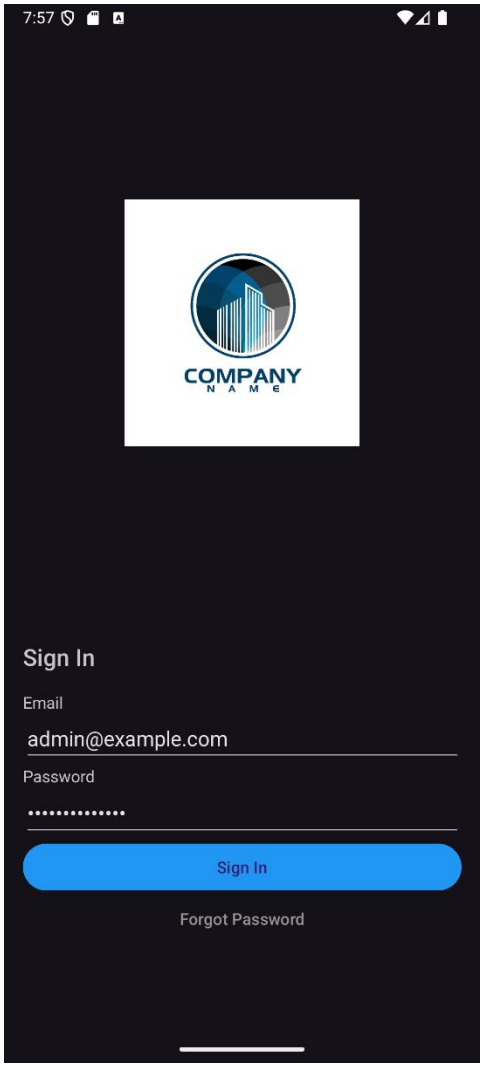


Figure 24: MainActivity

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_main.xml

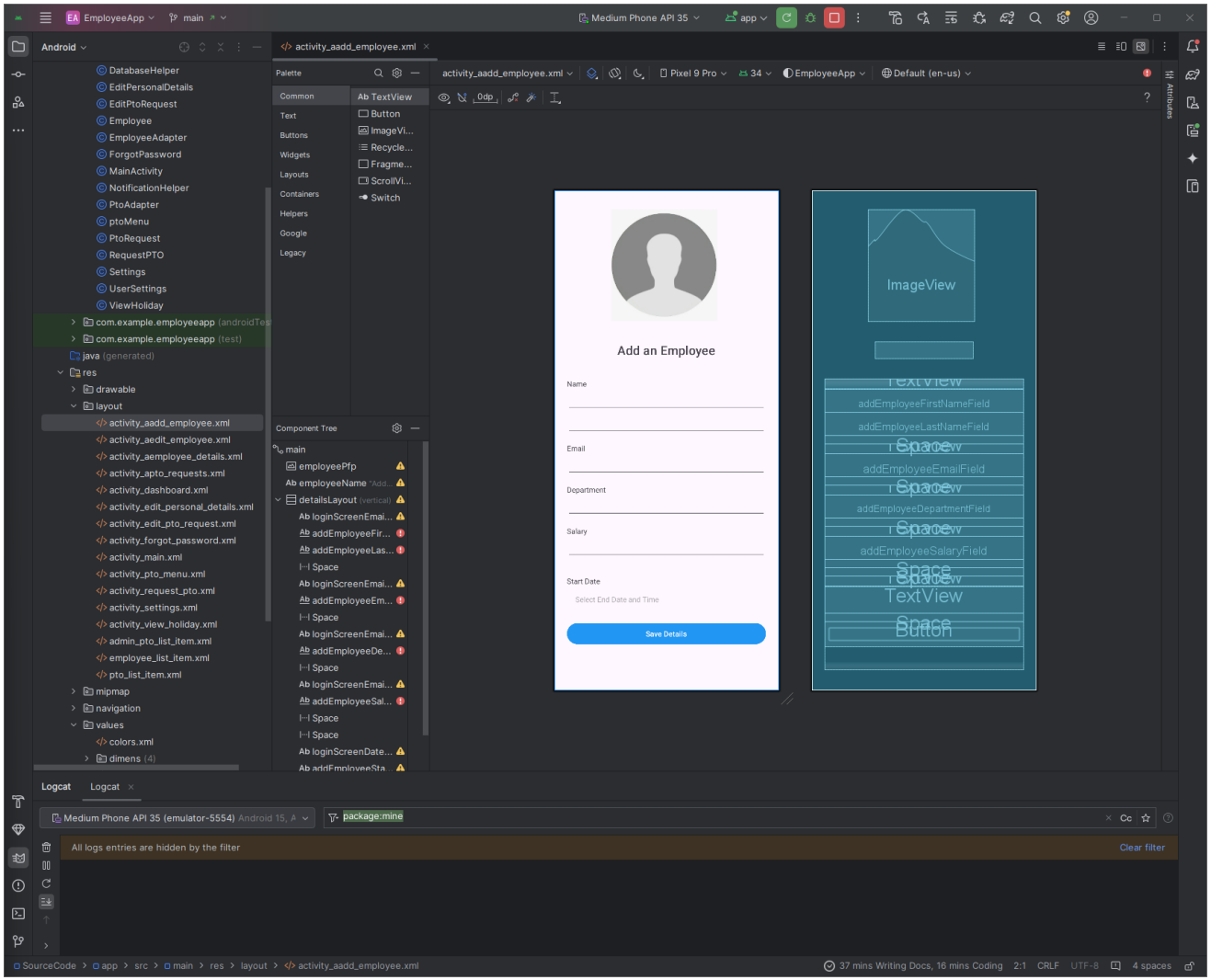


Figure 25: activity_aadd_employee.xml

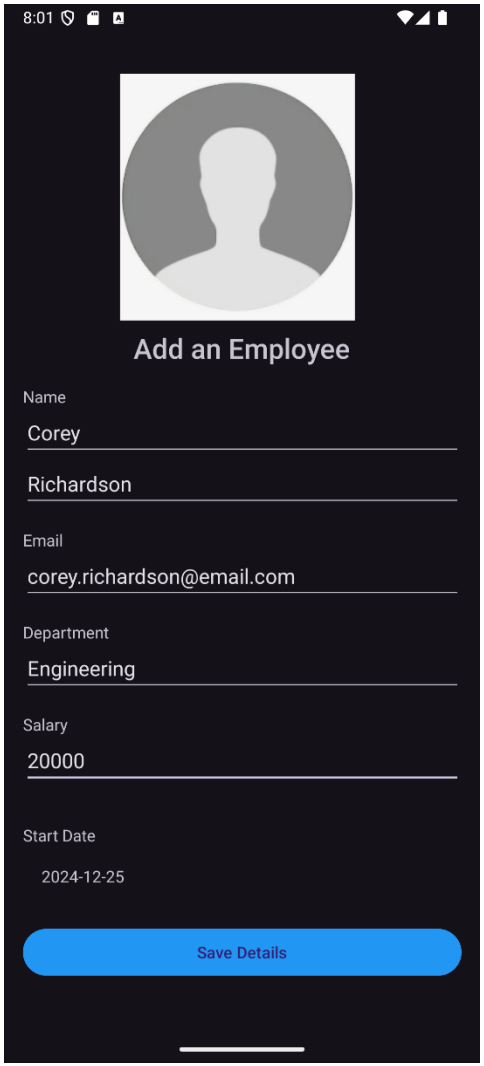


Figure 26: aAddEmployee

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_aadd_employee.xml

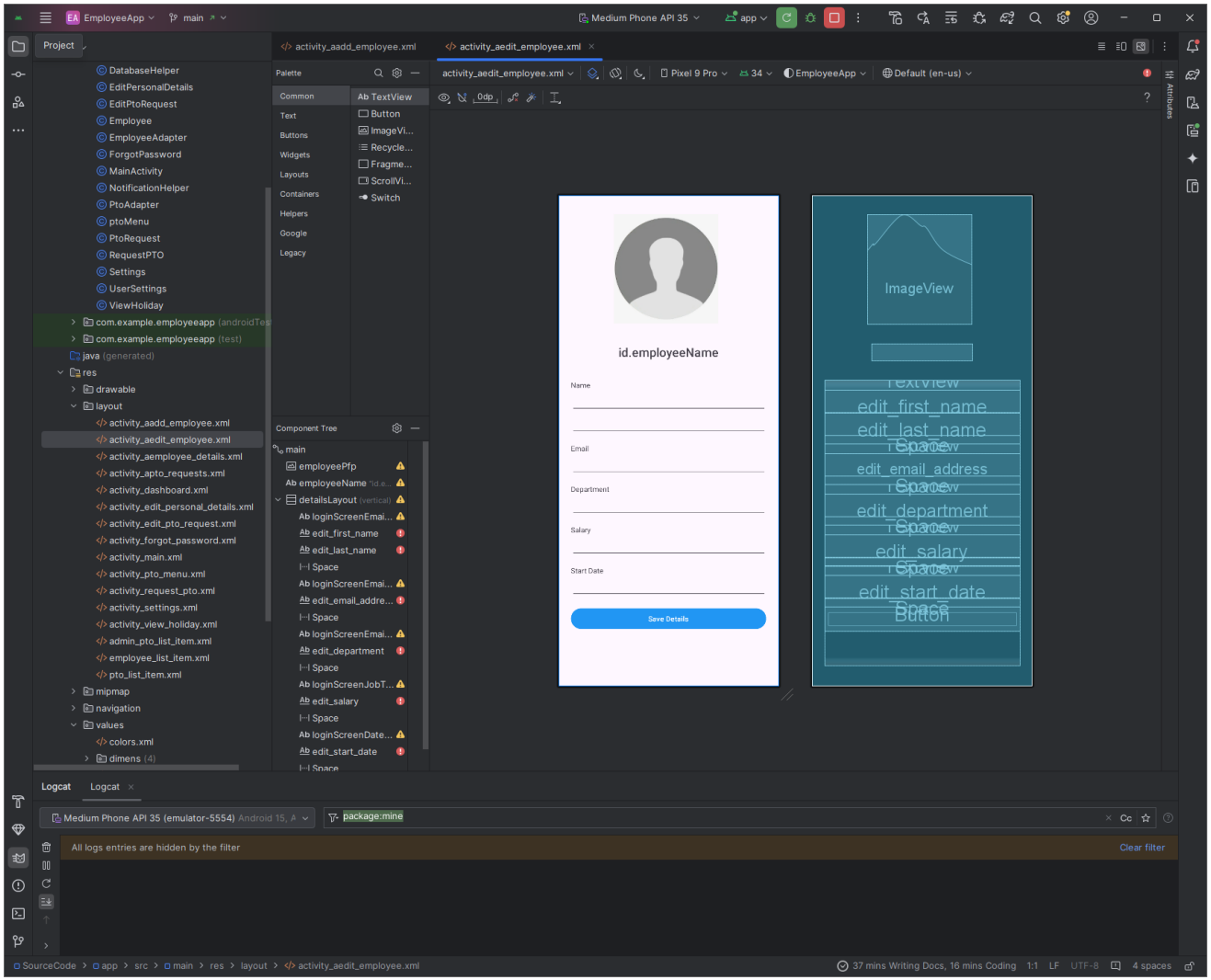


Figure 27: activity_aedit_employee.xml

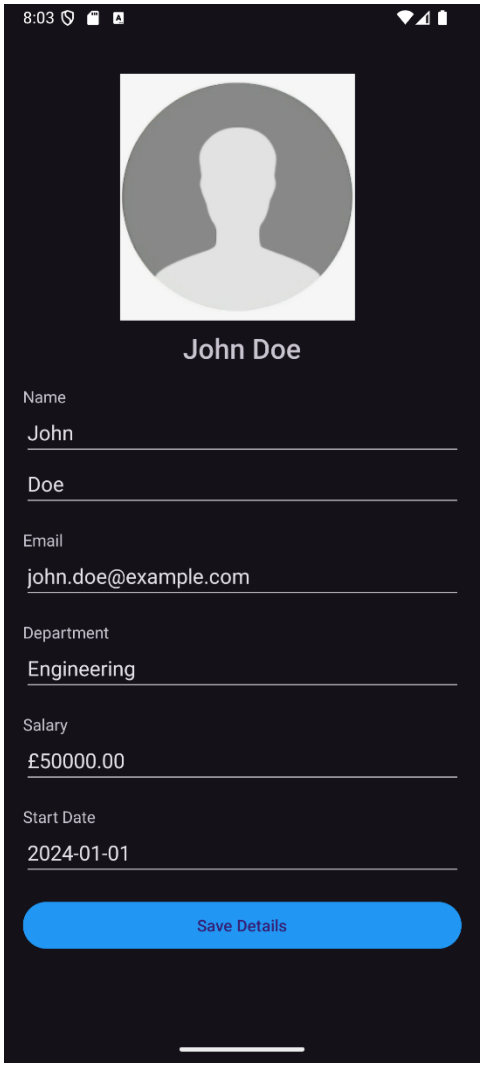


Figure 28: aEditEmployee

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_aedit_employee.xml

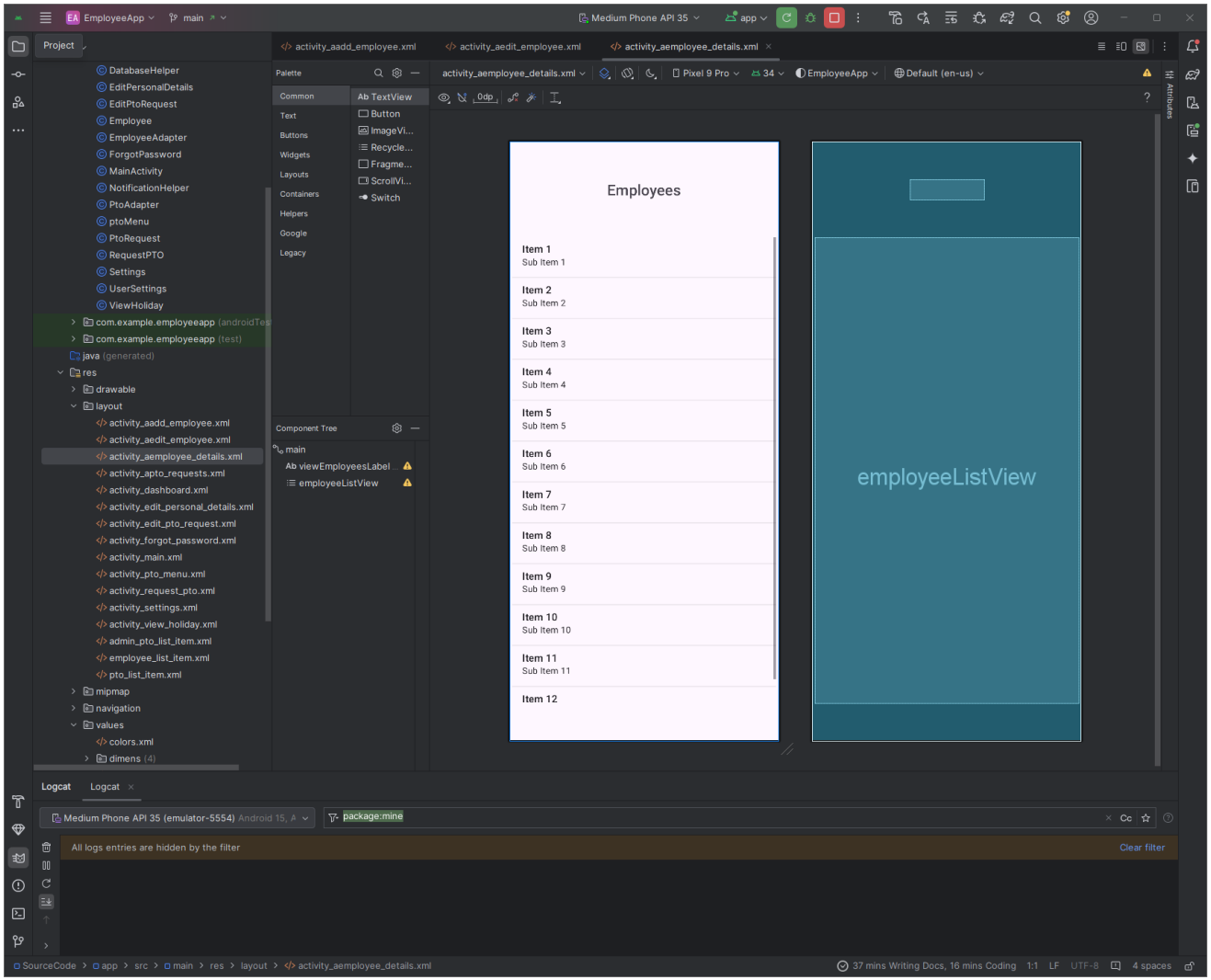


Figure 29: aemployee_details.xml

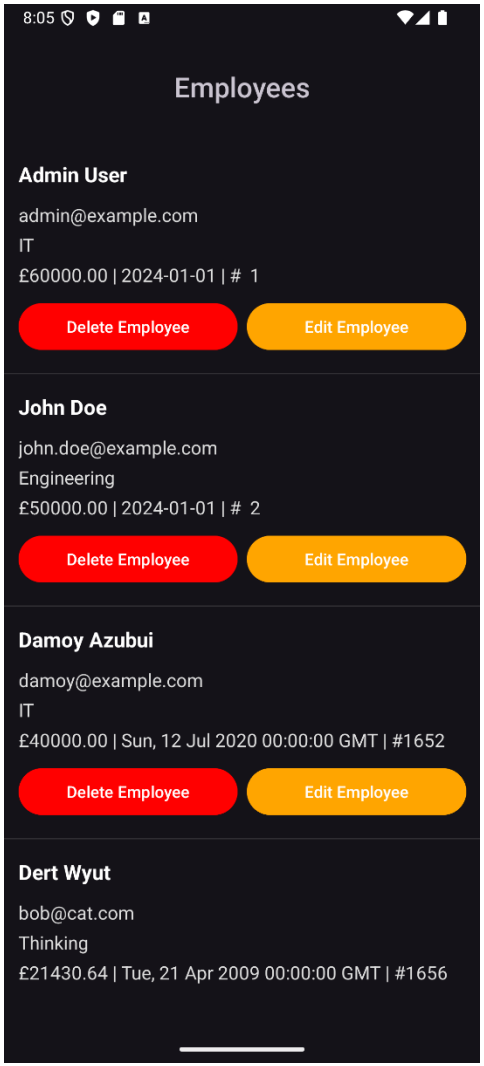


Figure 30: aEmployeeDetails

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_aemployee_details.xml

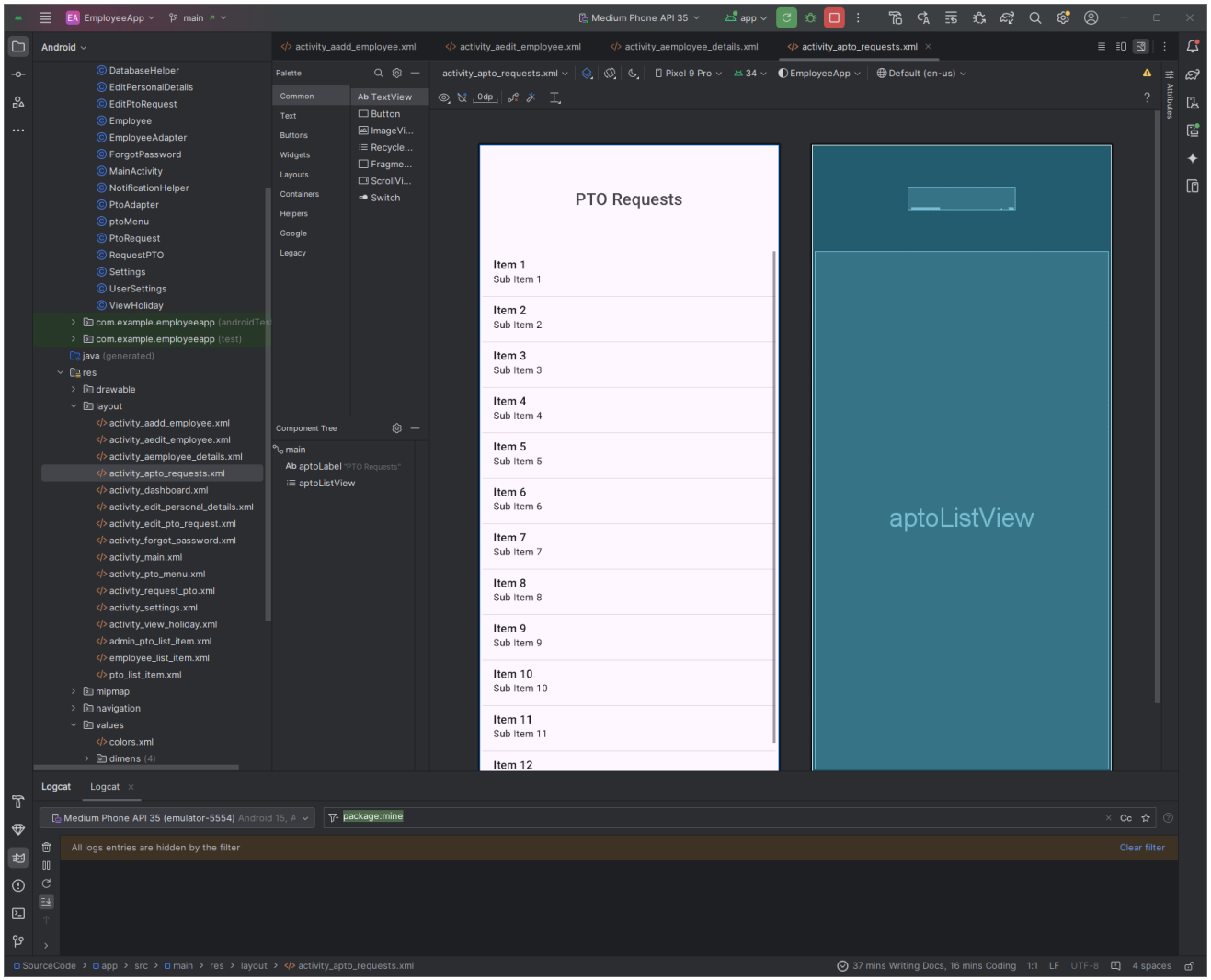


Figure 31: activity_apto_requests.xml

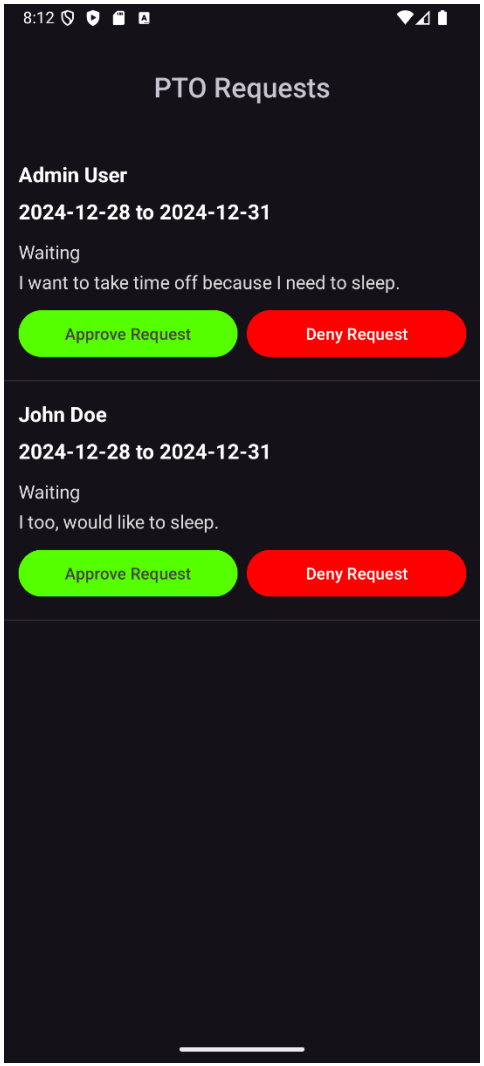


Figure 32: aPtoRequests

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_apto_requests.xml

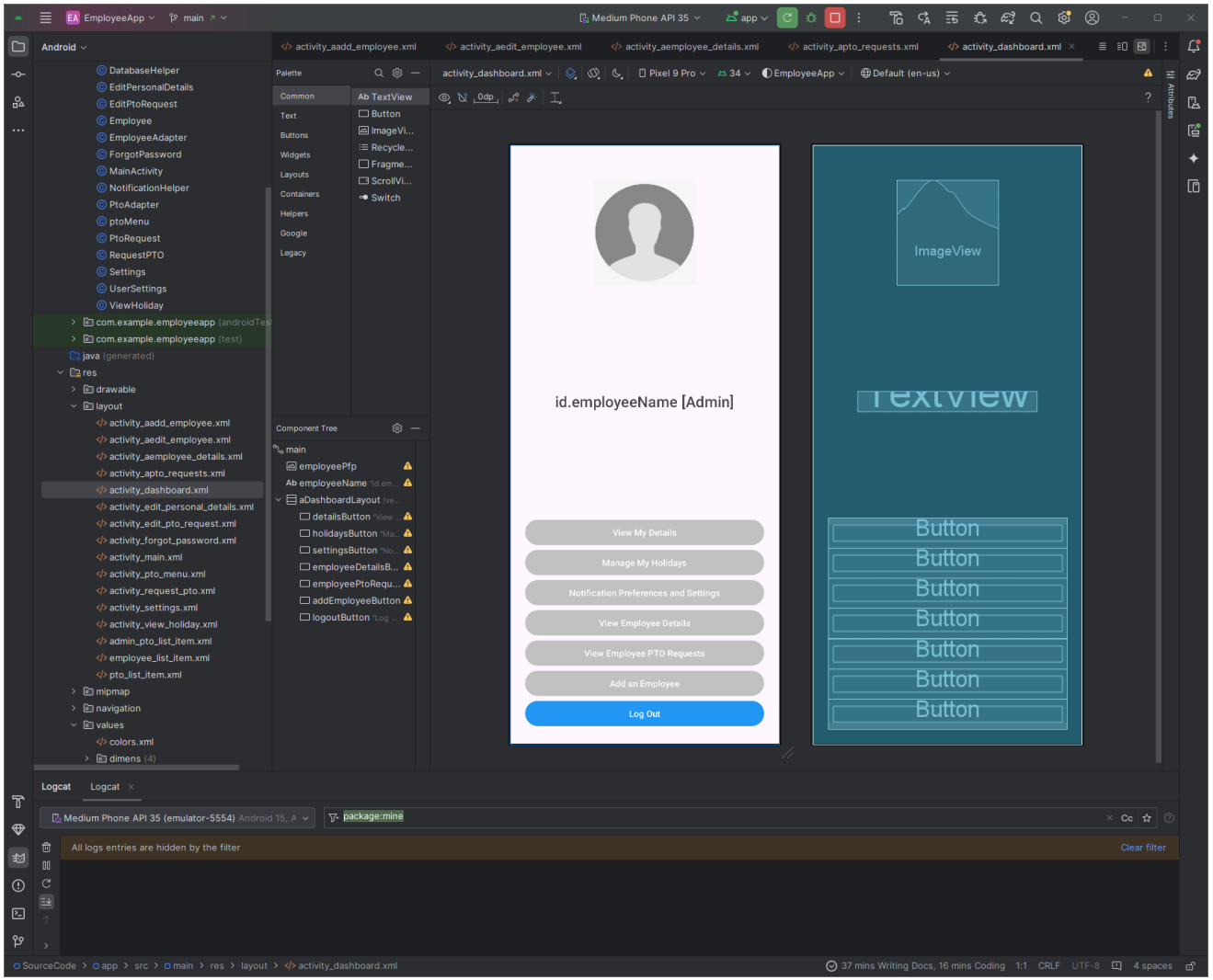


Figure 33: activity_dashboard.xml

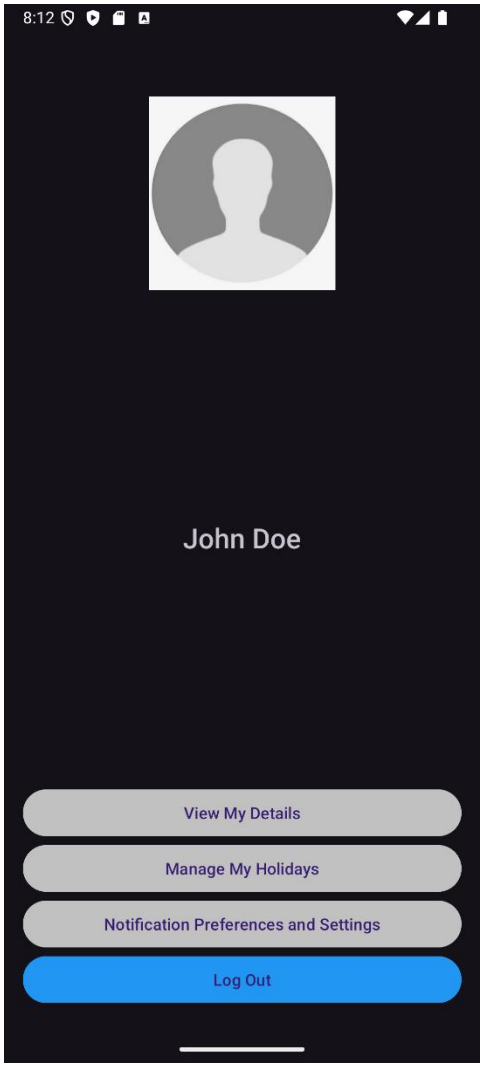


Figure 34: Dashboard (Employee)

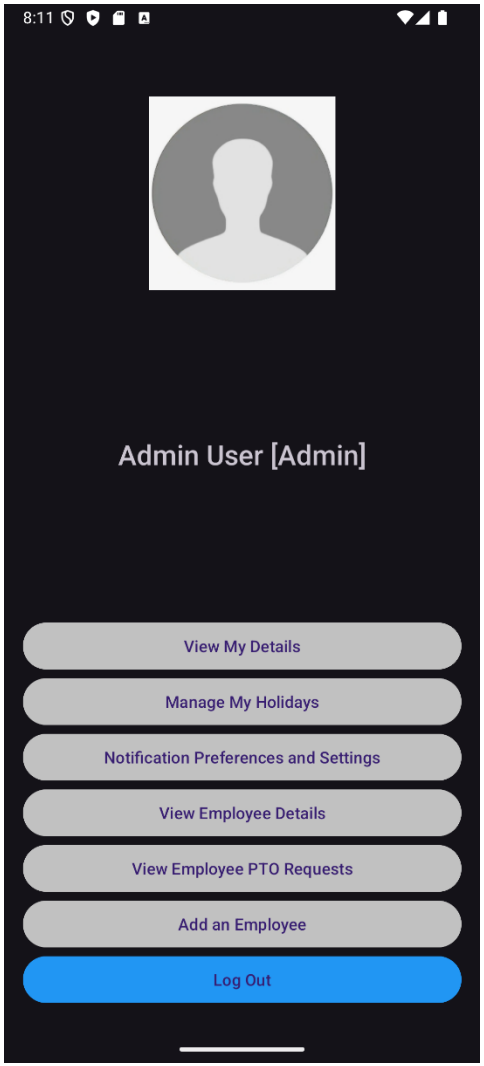


Figure 35: Dashboard (Admin)

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_dashboard.xml

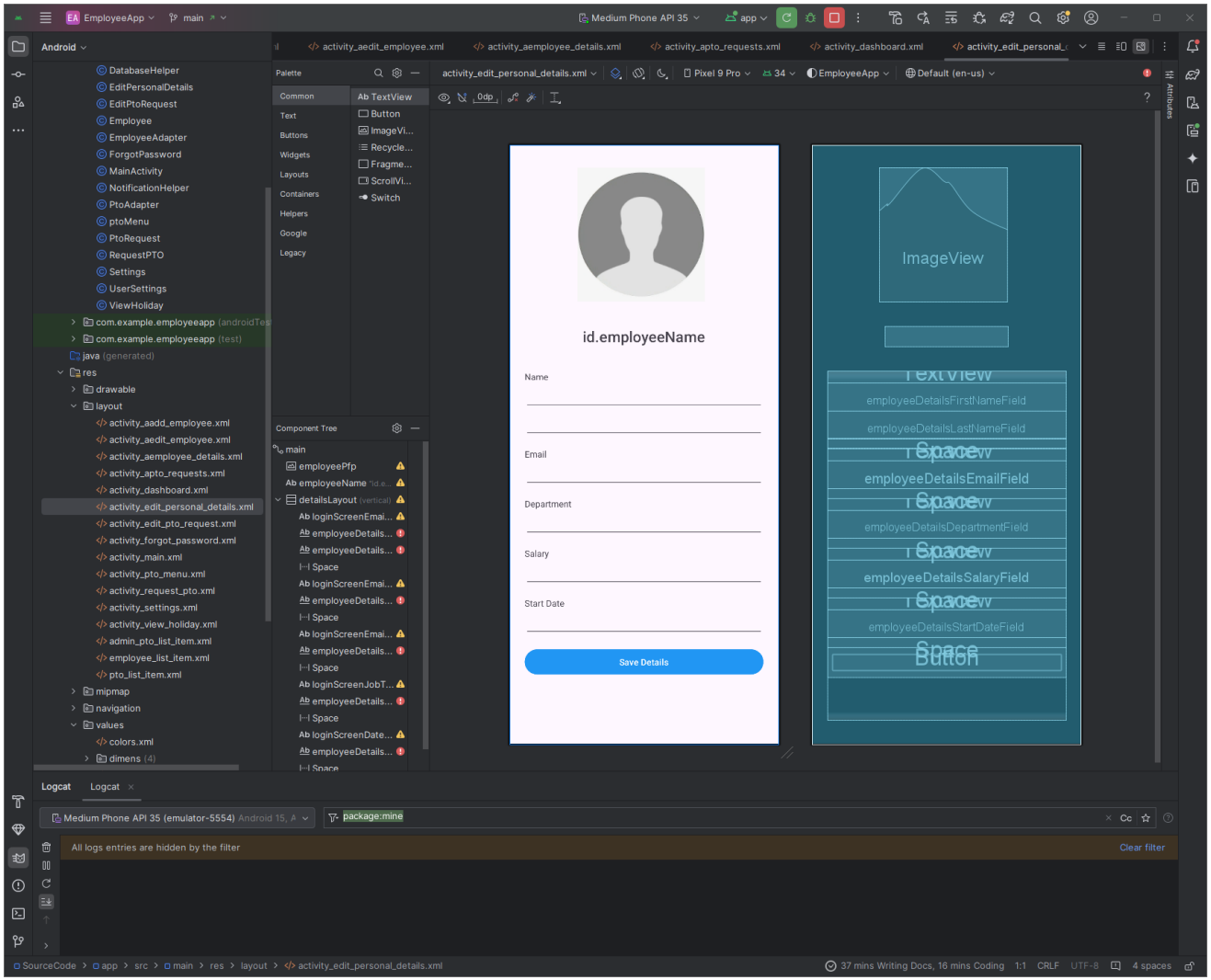


Figure 36: activity_edit_personal_details.xml

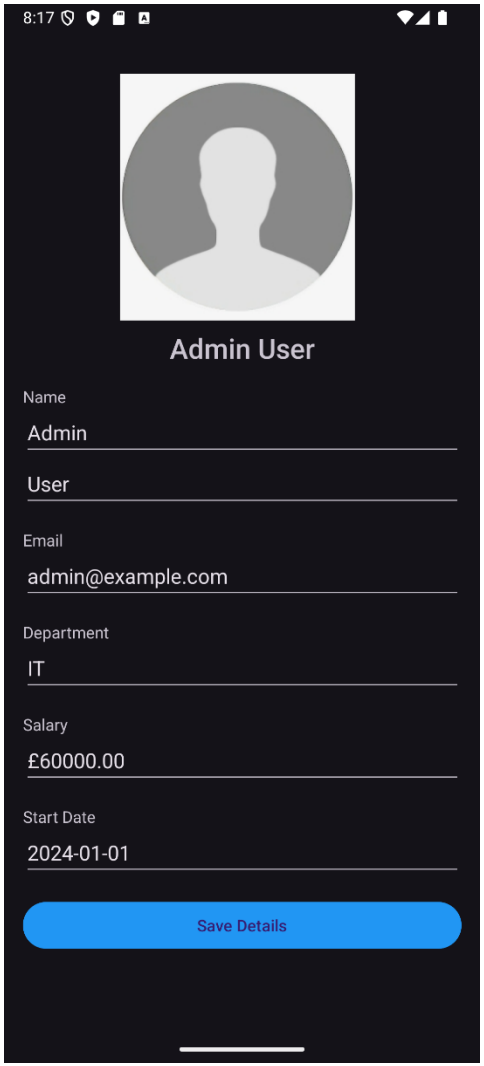


Figure 37: EditPersonalDetails

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_edit_personal_details.xml

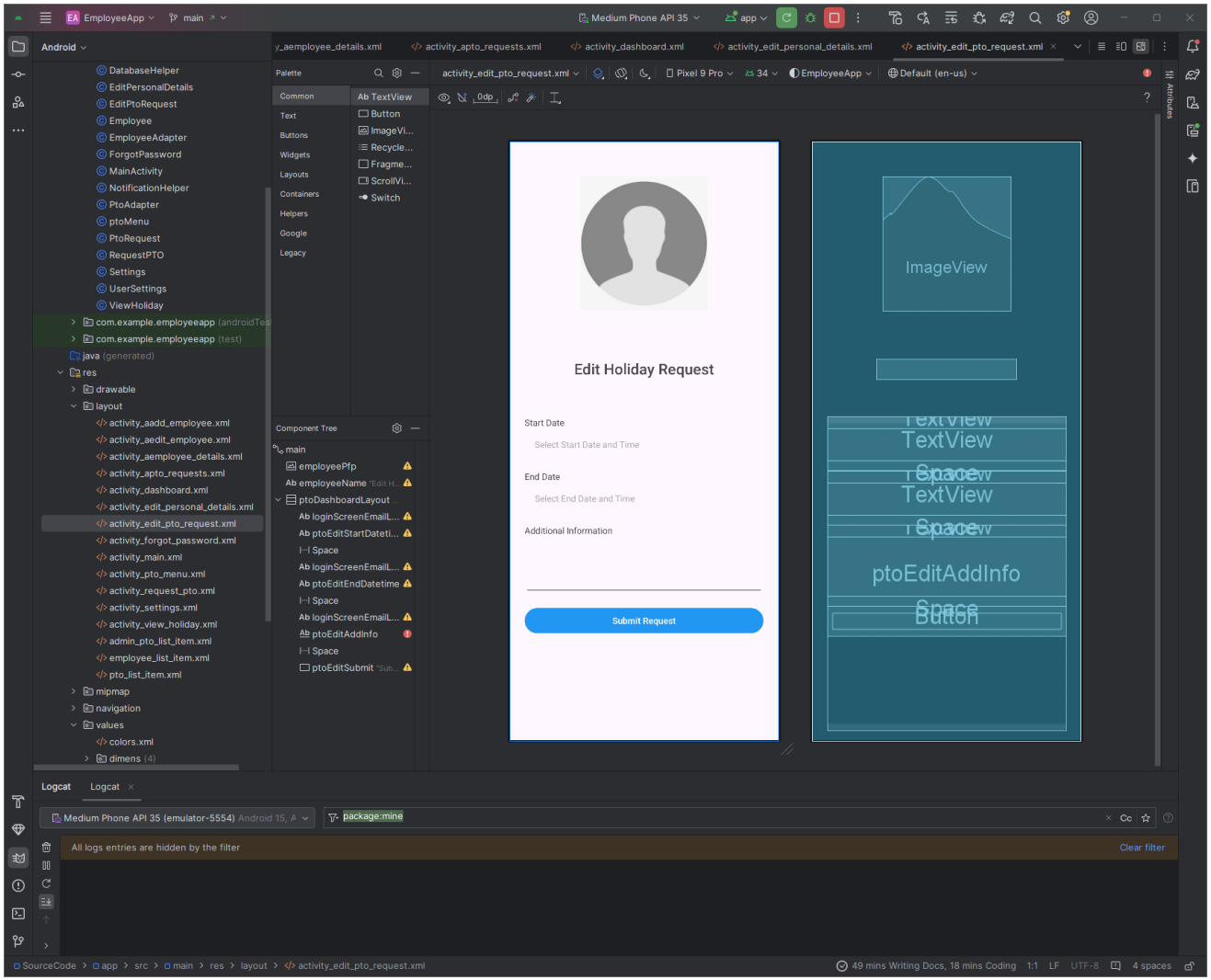


Figure 38: activity_edit_pto_request.xml

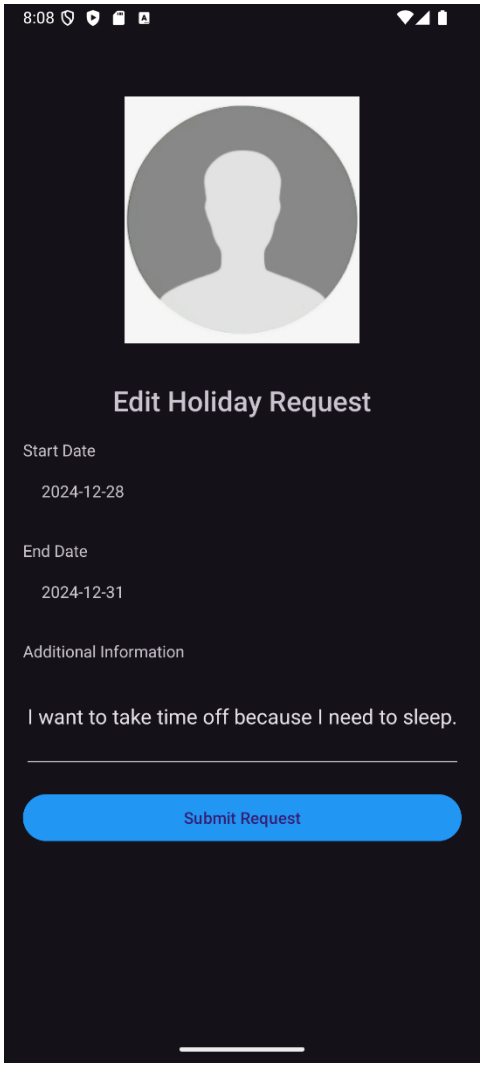


Figure 39: EditPtoRequest

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_edit_pto_request.xml

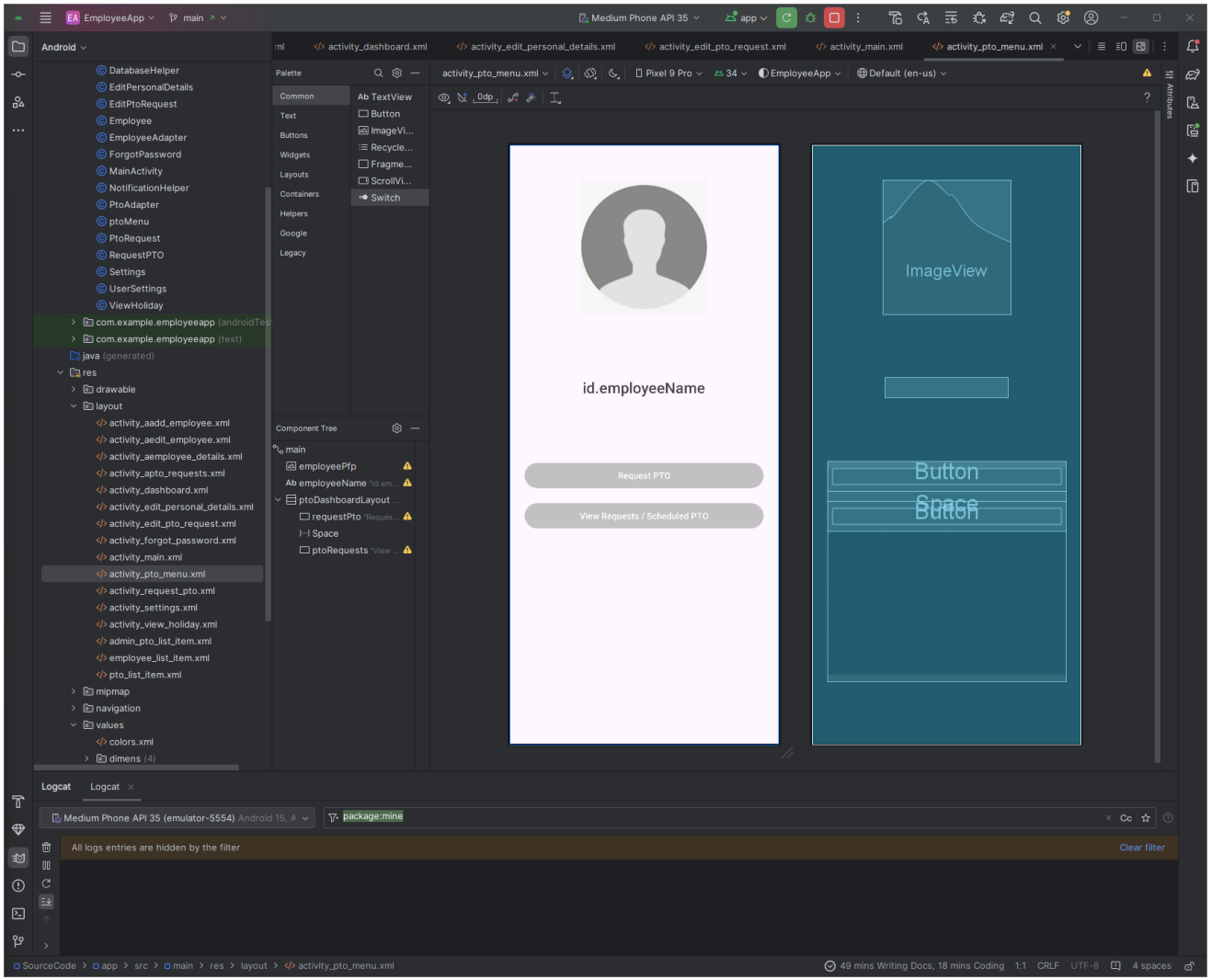


Figure 40: activity_pto_menu.xml

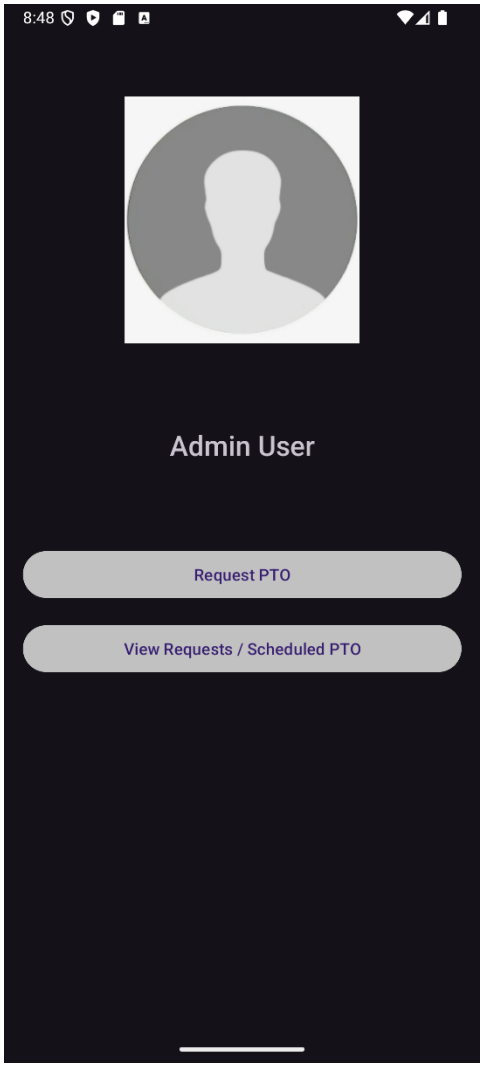


Figure 41: PtoMenu

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_pto_menu.xml

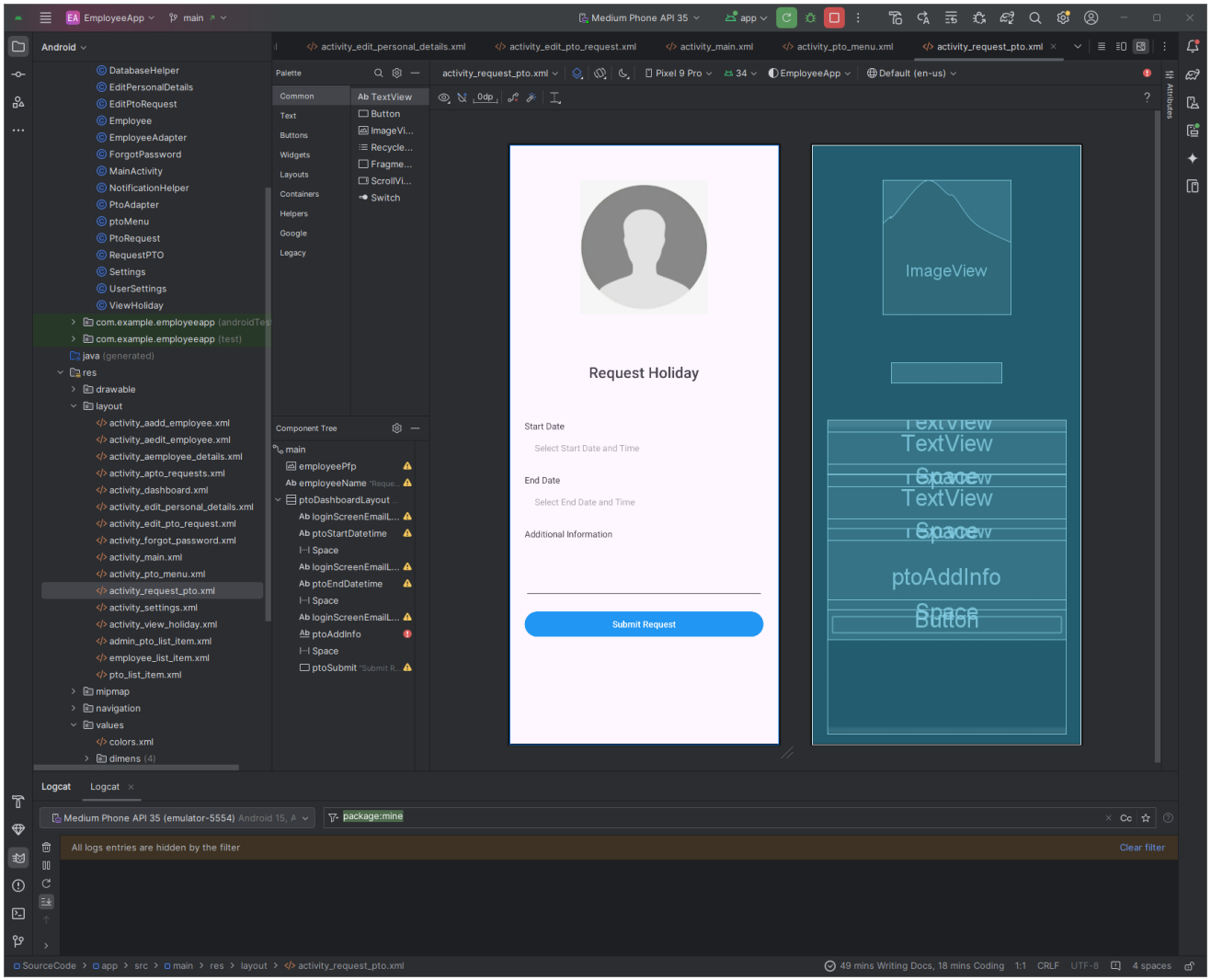


Figure 42: activity_request_pto.xml

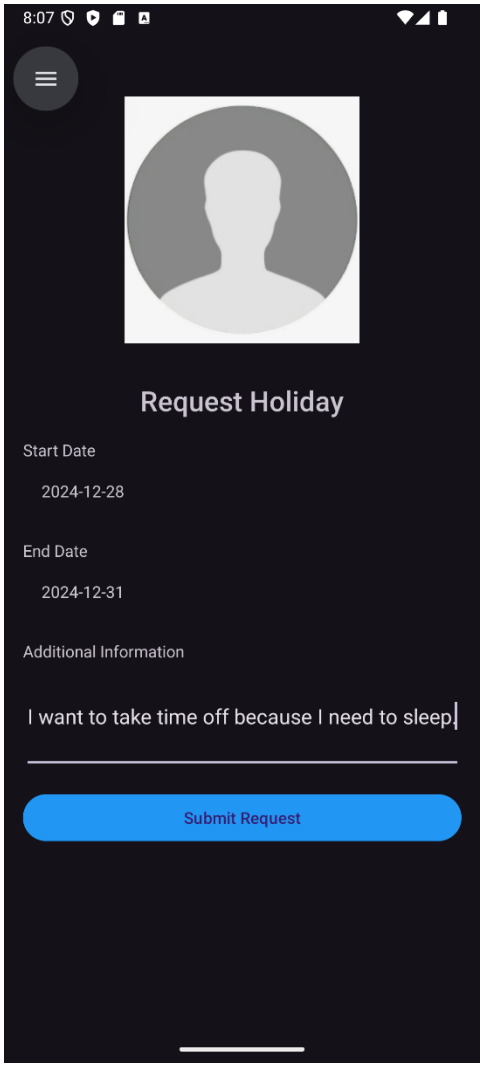


Figure 43: RequestPTO

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_request_pto.xml

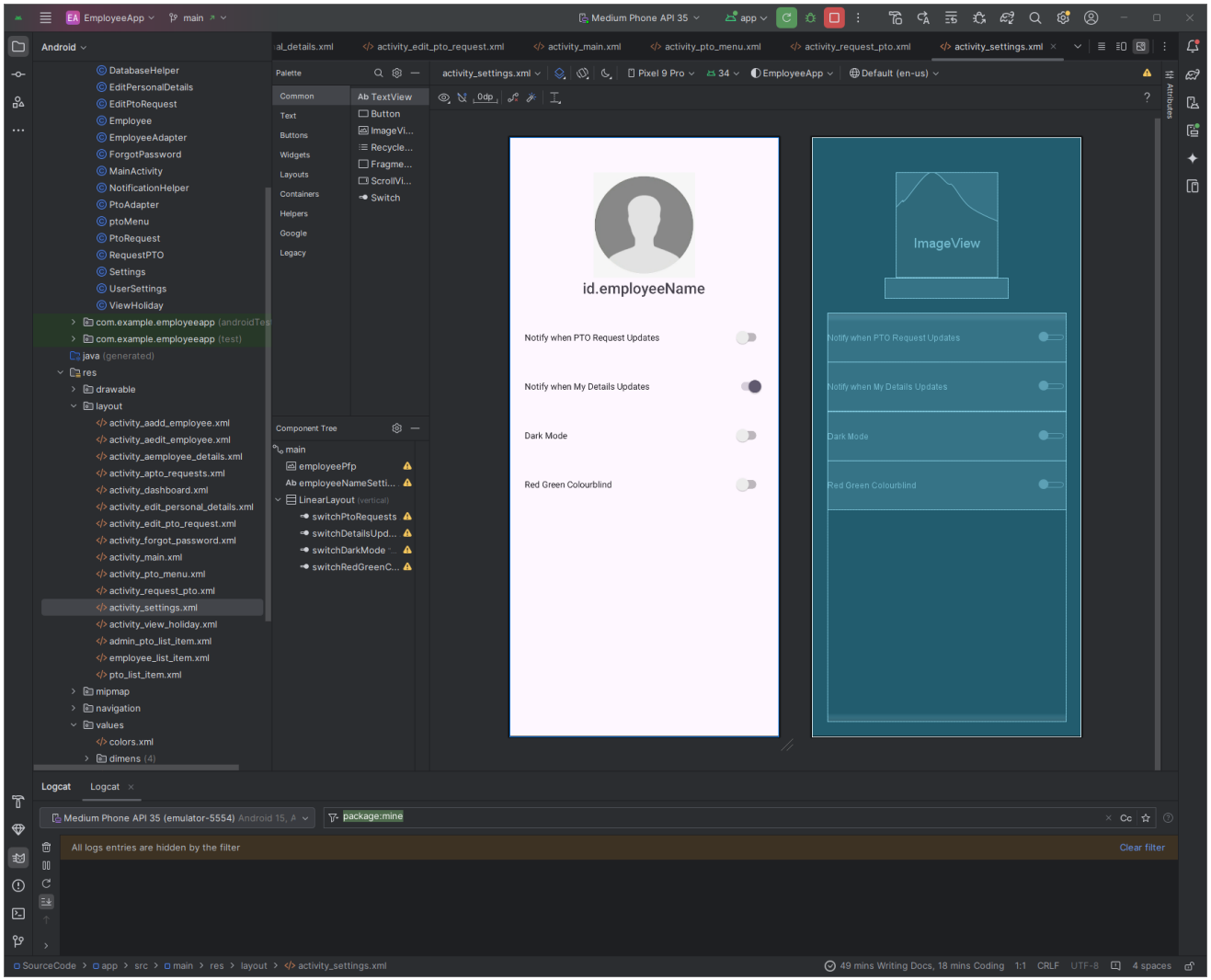


Figure 44: activity_settings.xml

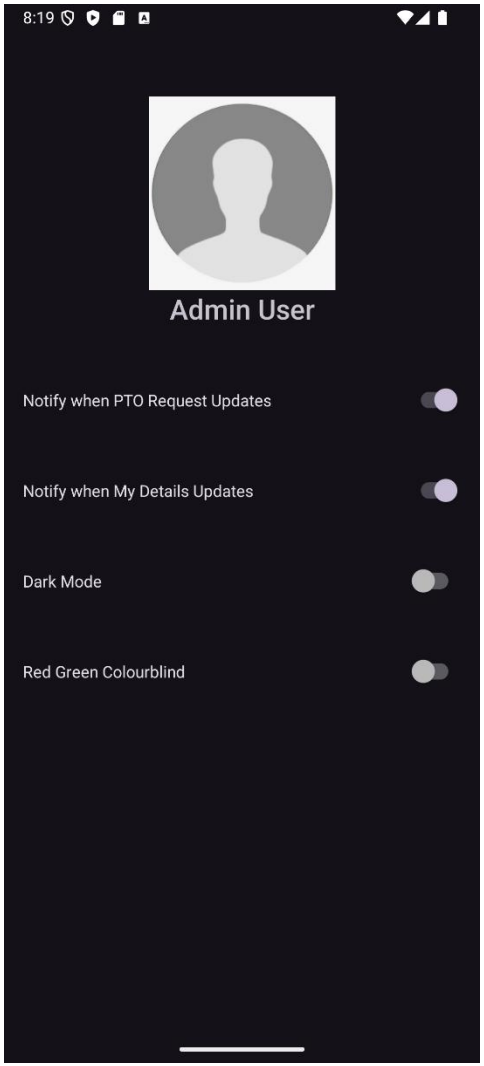


Figure 45: UserSettings

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_settings.xml

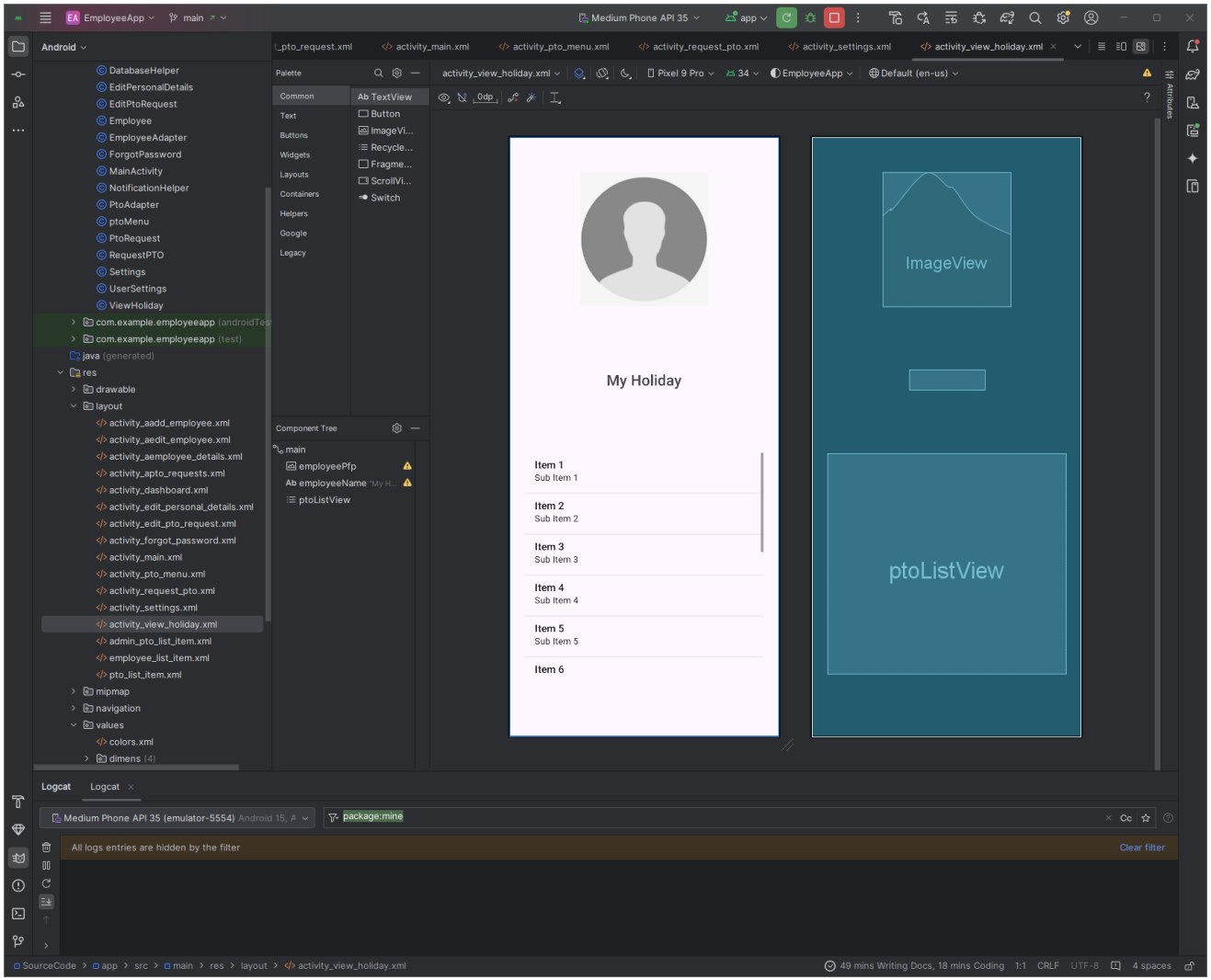


Figure 46: activity_view_holiday.xml

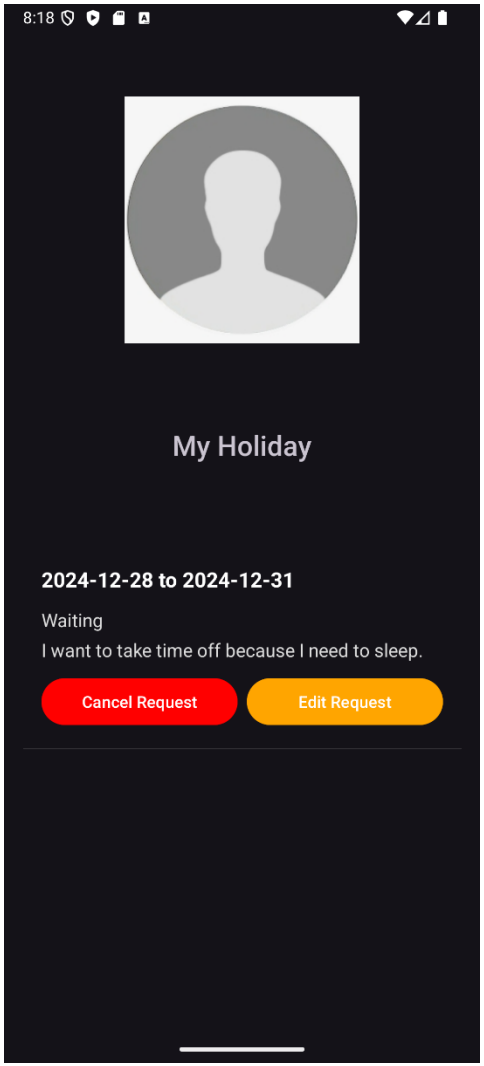


Figure 47: ViewHoliday

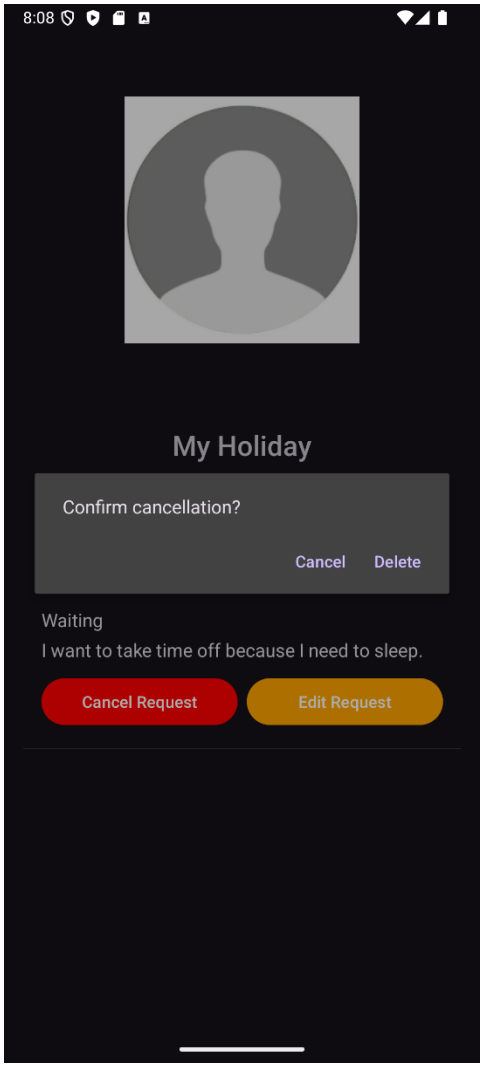


Figure 48: ViewHoliday (Confirm cancellation?)

https://github.com/Plymouth-COMP2000/coursework-report-corey-richardson/blob/main/SourceCode/app/src/main/res/layout/activity_view_holiday.xml

Implementation

Functional Requirements of the Application

Admin Users	
As an administrator I wish to add employees' details.	Implemented
As an administrator I wish to check the employees' details have been uploaded.	Implemented
As an administrator I wish to edit employees' details.	Implemented
As an administrator I wish to delete an employee's details.	Implemented
The app should also apply an automatic increment by 5% on the employee salary when they complete one year.	Not implemented

Employee Users	
As an employee I wish to view my details.	Implemented
As an employee I wish to edit my details.	Implemented
As an employee I wish to book/ manage my annual leave/ holiday (the allowance is 30 days per year).	Implemented

Activities

Activities are created with a combination of an XML file defining the layout of page to display, and a Java file which handles the dynamic content-generation. The application starts at the **MainActivity**.

Activity	XML File	Java File	Explained
MainActivity	activity_main.xml	MainActivity.java	This activity holds the Sign In screen. The tester can use the two pre-defined user accounts (**admin@example.com**, **admin_password** or **john.doe@example.com**, **employee_password**) to test the applications functionality.
aAddEmployee	activity_aadd_employee.xml	aAddEmployee.java	This activity can be accessed by Admin users and contains a form to add new users into the local database and API service.
aEditEmployee	activity_aedit_employee.xml	aEditEmployee.java	This activity can be accessed by Admin users and contains a prefilled form with the selected user's information. Here, the admin can edit the information and save the changes to the local database and API service.
aEmployeeDetails	activity_aemployee_details.xml	aEmployeeDetails.java	This activity holds a ListView item using the EmployeeAdapter which displays user information along with options for opening the aEditEmployee activity, and user record deletion.
aPtoRequests	activity_apto_requests.xml	aPtoRequests.java	This activity holds a ListView item using the aPtoAdapter which allows the Admin user to action employee's holiday requests.
Dashboard	activity_dashboard.xml	Dashboard.java	This is the main navigation hub of the application.
EditPersonalDetails	activity_edit_personal_details.xml	EditPersonalDetails.java	This activity contains a prefilled form with the user's information. The user can edit their information and save the changes to the local database and API service.
EditPtoRequests	activity_edit_pto_request.xml	EditPtoRequests.java	This activity contains a prefilled form with the selected holiday requests information. The user can edit their requests details and save the changes to the local database.

ForgotPassword	activity_forgot_password.xml	ForgotPassword.java	Not functional.
ptoMenu	activity_pto_menu.xml	ptoMenu.java	This activity is a navigation page for selecting activities to Request PTO or Viewing Requests.
RequestPto	activity_request_pto.xml	RequestPTO.java	This activity contains a form where the user can submit new PTO requests.
Settings	activity_settings.xml	Settings.java	This activity lets the user modify the setting Booleans stored in their userSettings attribute.
ViewHoliday	activity_view_holiday.xml	ViewHoliday.java	This activity holds a ListView item using the ptoAdapter which allows the user to see their requested and scheduled holiday.

ListView Items

Adapter Utilised By	**XML File**
aPtoAdapter.java	admin_pto_list_item.xml
EmployeeAdapter.java	employee_list_item.xml
PtoAdapter.java	pto_list_item.xml

Local Database Implementation

The application uses a local SQLite database for storing employee records, along with additional information not held by the API such as the Employee's role and login information, as a short-term but persistent storage medium to minimise the number of GET requests sent to the API Web Service, reducing reliance for repetitive retrievals. This allows users read-access to their data even without an internet connection. SQLite was the chosen technology here due to its simple and lightweight implementation due to native support for Android.

The schema consists of three tables: User, UserSettings and PtoRequest.

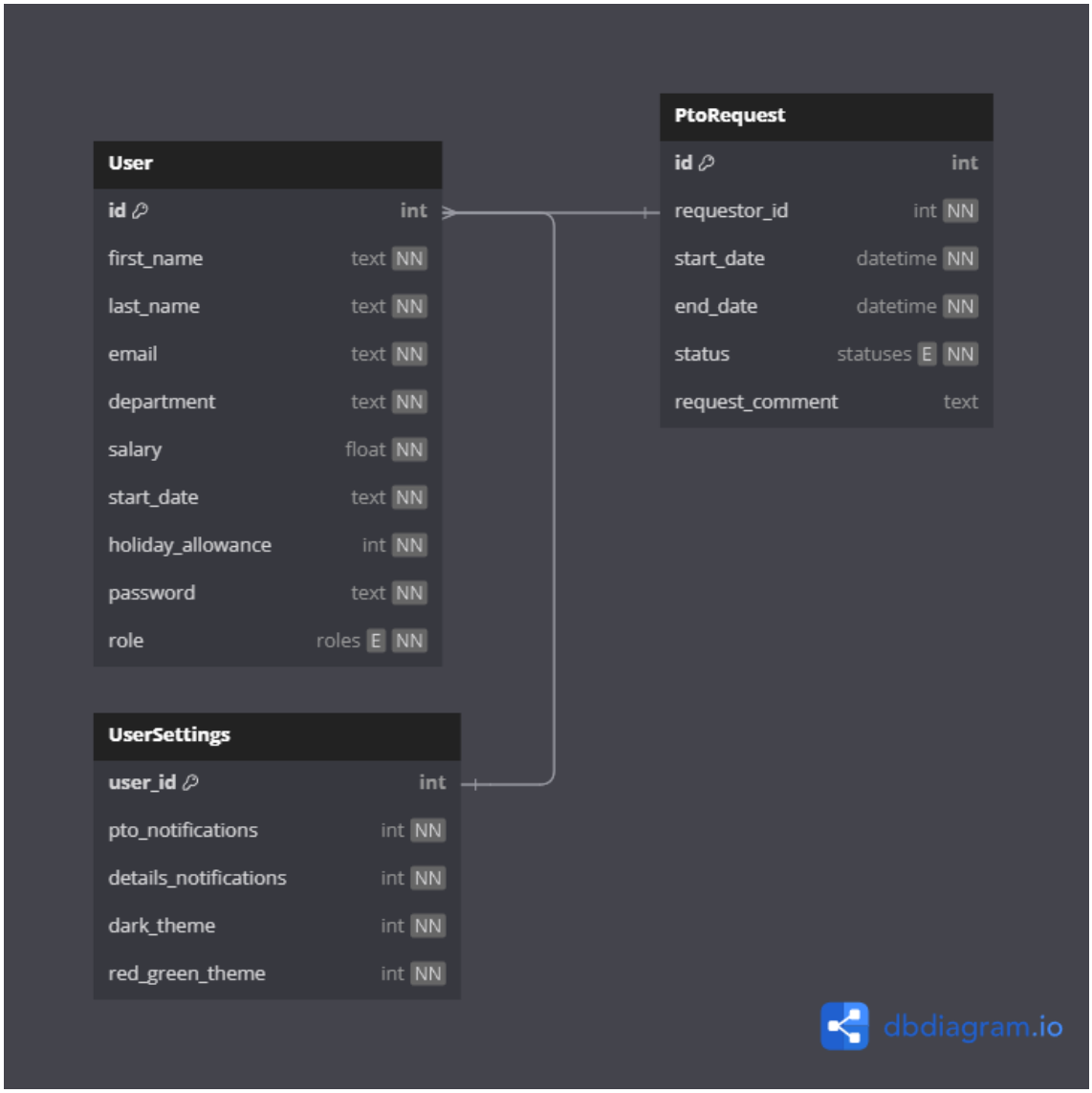


Figure 49: ERD Diagram of Local Database Schema

```sql
CREATE TABLE User (
    id INTEGER PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,+
    email TEXT UNIQUE NOT NULL,
    department TEXT NOT NULL,
    salary REAL NOT NULL CHECK (salary > 0) DEFAULT 0,
    start_date TEXT NOT NULL,
    holiday_allowance INT NOT NULL CHECK (holiday_allowance > 0) DEFAULT 0,
    password TEXT NOT NULL,
    role TEXT NOT NULL CHECK (role IN ('Admin', 'Employee'))
);

CREATE TABLE UserSettings (
    user_id INTEGER PRIMARY KEY,
    pto_notifications INTEGER NOT NULL,
    details_notifications INTEGER NOT NULL,
    dark_theme INTEGER NOT NULL,
    red_green_theme INTEGER NOT NULL,
    FOREIGN KEY (user_id) REFERENCES User (id)
);

CREATE TABLE PtoRequest (
    id INTEGER PRIMARY KEY,
    requester_id INTEGER NOT NULL,
    start_date TEXT NOT NULL,
    end_date TEXT NOT NULL,
    status TEXT NOT NULL CHECK (status IN ('Approved', 'Waiting', 'Denied')),
    request_comment TEXT,
    FOREIGN KEY (requester_id) REFERENCES User (id),
    UNIQUE(requester_id, start_date, end_date)
);

CREATE INDEX idx_user_first_name ON User (first_name);
CREATE INDEX idx_user_last_name ON User (last_name);
CREATE INDEX idx_user_email ON User (email);
CREATE INDEX idx_user_id_settings ON UserSettings (user_id);
CREATE INDEX idx_pto_request_requester ON PtoRequest (requester_id);

```

Figure 50: SQLite Database Schema

The SQLiteOpenHelper class is extended by DatabaseHelper to manage database creation, upgrade and functions.

```java
public class DatabaseHelper extends SQLiteOpenHelper {

    // Class attributes
    private static final String DATABASE_NAME = "database.db";
    private static final int DATABASE_VERSION = /* CURRENT_VERSION */;
    private static DatabaseHelper instance;

    // Constructor
    public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    // Singleton pattern to prevent multiple instances of the database helper
    public static synchronized DatabaseHelper getInstance(Context context) {
        if (instance == null) {
            instance = new DatabaseHelper(context.getApplicationContext());
            Log.d("DatabaseInstance", "Instance created.");
        } else {
            Log.d("DatabaseInstance", "Instance carried forward.");
        }
        return instance;
    }

    // Database Creation and Upgrade methods
    @Override
    public void onCreate(SQLiteDatabase db) {
        String createUserTable = // ...
        String createUserSettingsTable = // ...
        String createPtoRequestTable = // ...

        try {
            db.execSQL(createUserTable);
            db.execSQL(createUserSettingsTable);
            db.execSQL(createPtoRequestTable);
        } catch (Exception e) {
            Log.d("DatabaseCreationError", Log.getStackTraceString(e));
        }

        db.execSQL("CREATE INDEX idx_user_first_name ON User (first_name);");
        db.execSQL("CREATE INDEX idx_user_last_name ON User (last_name);");
        db.execSQL("CREATE INDEX idx_user_email ON User (email);");
        db.execSQL("CREATE INDEX idx_user_id_settings ON UserSettings (user_id);");
        db.execSQL("CREATE INDEX idx_pto_request_requester ON PtoRequest (requester_id);");

        db.execSQL("INSERT INTO User (first_name, last_name, email, department, salary, start_date, " +
                "holiday_allowance, password, role) " +
                "VALUES ('Admin', 'User', 'admin@example.com', 'IT', 60000.00, '2024-01-01', 30, " +
                "'admin_password', 'Admin');");
        db.execSQL("INSERT INTO UserSettings (user_id, pto_notifications, details_notifications, " +
                "dark_theme, red_green_theme)" +
                "VALUES (1, 1, 1, 0, 0);");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS User;");
        db.execSQL("DROP TABLE IF EXISTS UserSettings;");
        db.execSQL("DROP TABLE IF EXISTS PtoRequest;");
        onCreate(db);
    }

    // Database helper and logic methods
}
```

Figure 51: DatabaseHelper extends SQLiteOpenHelper

Data operations are handled through Data Access Object methods to Create, Read, Update or Delete records across the three tables.

As examples, here are the CRUD methods for the **Employee** table.

```java
public void insertUser(Context context, Employee newEmployee) {
    SQLiteDatabase db = getWritableDatabase();
    ContentValues userValues = new ContentValues();
    ContentValues settingsValues = new ContentValues();

    db.beginTransaction();
    try {

        if (newEmployee.getId() > 0) { // if ID is meaningful
            userValues.put("id", newEmployee.getId());
        }

        userValues.put("first_name", newEmployee.getFirstName());
        userValues.put("last_name", newEmployee.getLastName());
        userValues.put("email", newEmployee.getEmail());
        userValues.put("department", newEmployee.getDepartment()); // New field
        userValues.put("salary", newEmployee.getSalary()); // New field
        userValues.put("start_date", newEmployee.getStartDate());
        userValues.put("holiday_allowance", newEmployee.getHolidayAllowance());
        userValues.put("password", newEmployee.getPassword());
        userValues.put("role", newEmployee.getRole());

        long userId = db.insert("User", null, userValues);
        if (userId == -1) {
            throw new SQLException("Failed to insert new user.");
        }

        settingsValues.put("user_id", userId);
        settingsValues.put("pto_notifications", 1);
        settingsValues.put("details_notifications", 1);
        settingsValues.put("dark_theme", 1);
        settingsValues.put("red_green_theme", 0);

        long settingsId = db.insert("UserSettings", null, settingsValues);
        if (settingsId == -1) { throw new SQLException("Failed to insert new user's settings."); }

        db.setTransactionSuccessful();
    } catch (Exception e) {
        Log.e("InsertUser", "Failed to insert new user.");
        throw new SQLException("Failed to insert new user.");
    } finally {
        db.endTransaction();
        db.close();
    }

    List<Employee> updatedEmployeeList = getAllEmployees();
    EmployeeAdapter employeeAdapter = new EmployeeAdapter(context, updatedEmployeeList);
    employeeAdapter.notifyDataSetChanged();
}
```

Figure 52: Example CREATE Operation - insertUser()

```java
public List<Employee> getAllEmployees() {
    List<Employee> employeeList = new ArrayList<>();
    SQLiteDatabase db = this.getReadableDatabase();

    Cursor cursor = db.rawQuery("SELECT * FROM User", null);

    if (cursor.moveToFirst()) {
        do {
            employeeList.add(cursorToEmployee(cursor));
        } while (cursor.moveToNext());
    }

    cursor.close();
    db.close();

    return employeeList;
}
```

Figure 53: Example READ Operation - getAllEmployees()

```java
public void updateUserInDatabase(Context context, Employee employee) {
    ContentValues values = new ContentValues();
    values.put("first_name", employee.getFirstName());
    values.put("last_name", employee.getLastName());
    values.put("email", employee.getEmail());
    values.put("department", employee.getDepartment());
    values.put("salary", employee.getSalary());
    values.put("start_date", employee.getStartDate());
    values.put("holiday_allowance", employee.getHolidayAllowance());
    values.put("password", employee.getPassword());
    values.put("role", employee.getRole());

    try (SQLiteDatabase db = getWritableDatabase()) {
        db.update("User", values, "id = ?",
                new String[]{ Integer.toString(employee.getId()) });
    } catch (SQLException e) {
        Log.e("DatabaseHelper", "Error updating employee details.");
        throw e;
    }

    List<Employee> updatedEmployeeList = getAllEmployees();
    EmployeeAdapter employeeAdapter = new EmployeeAdapter(context, updatedEmployeeList);
    employeeAdapter.notifyDataSetChanged();
}
```

Figure 54: Example UPDATE Operation - updateUserInDatabase()

```java
public void deleteEmployee(int id) {
    try (SQLiteDatabase db = getWritableDatabase()) { // AndroidStudio suggested this \_O_/
        db.delete("User", "id = ?", new String[]{Integer.toString(id)});
        db.delete("UserSettings", "user_id = ?", new String[]{Integer.toString(id)});
    } catch (SQLException e) {
        Log.e("DatabaseHelper", "Error deleting employee " + id, e);
        throw e; // Propagates the error to PtoAdapter::cancelPtoRequest to Toast in context
    }
}
```

Figure 55: Example DELETE Operation - deleteEmployee()

API Integration

The integration of a RESTful API enables data management and synchronisation across devices and users. Periodic fetches from the API to the Local Database ensure the data is consistent and up to date.

On app startup, employee records are fetched from the API and stored locally. These records are then re-fetched, and new records are inserted into the local database each time the admin user adds a new user to the database. This uses lazy loading, meaning that new data is only loaded on demand to reduce the background memory usage of the application, at the cost of slowing the loading of this operation.

This approach comes with a few issues, namely, how to ensure a consistent data sync with the API when the user switches between online and offline web connections, how to manage data migrations between different schema versions and how to handle other users of the API deleting employee records.

```
public static void fetchAndStoreEmployees(Context context) {
    new FetchEmployeesTask(context).execute();
}

@Override
protected Void FetchEmployeesTask::doInBackground(Void... voids) {
    String fetchEmployeesUrl = API_URL + "/employees";
    DatabaseHelper databaseHelper = DatabaseHelper.getInstance(context);

    JsonArrayRequest jsonArrayRequest = new JsonArrayRequest(
            Request.Method.GET, fetchEmployeesUrl, null,
            new Response.Listener<JSONArray>() {
                @Override
                public void onResponse(JSONArray response) {
                    for (int i = 0; i < response.length(); i++) {

                        try {
                            JSONObject jsonObject = response.getJSONObject(i);

                            // Skip if employee already exists
                            if (databaseHelper.getEmployeeById(jsonObject.getInt("id")) != null) {
                                databaseHelper.deleteEmployee(jsonObject.getInt("id"));
                            }

                            // Create Employee object
                            Employee employee = new Employee(
                                    jsonObject.getInt("id"),
                                    jsonObject.getString("firstname"),
                                    jsonObject.getString("lastname"),
                                    jsonObject.getString("email"),
                                    jsonObject.getString("department"),
                                    (float) jsonObject.getDouble("salary"),
                                    jsonObject.getString("joiningdate"),
                                    jsonObject.getInt("leaves"),
                                    "employee_password",
                                    "Employee"
                            );

                            try {
                                // Insert employee into database
                                databaseHelper.insertUser(context, employee);
                                Log.d("AddedEmployee", employee.getFullName());
                            } catch (Exception e) {
                                Log.d("FailedEmployeeInsert", employee.getFullName());
                            }

                        } catch (Exception e) {
                            Log.e("FetchAndStore", "Error processing employee");
                        }
                    }
                }
            },
            new Response.ErrorListener() {
                @Override
                public void onErrorResponse(VolleyError error) {
                    Log.e("FetchEmployees", "Error fetching employees: " + error.getMessage());
                    Toast.makeText(context, "Failed to fetch employees!", Toast.LENGTH_SHORT).show();
                }
            }
    );

    queue = getRequestQueue(context);
    queue.add(jsonArrayRequest);

    return null;
}
```

Figure 56: fetchAndStoreEmployees()

Base URL:	http://10.224.41.11/comp2000	
Endpoint	**Method**	**Description**
/employees	GET	Retrieve a list of all employees in JSON format.
/employees/get/<int:id>	GET	Retrieve details of a specific employee by their ID.
/employees/add	POST	Add a new employee to the system. 'firstname', 'lastname', 'email', 'department', 'salary' and 'joiningdate' fields must be present in the request body.
/employees/edit/<int:id>	PUT	Updates an existing employee's information. 'firstname', 'lastname', 'email', 'department', 'salary' and 'joiningdate' fields must be present in the request body.
/employees/delete/<int:id>	DELETE	Delete an employee by their ID. The ID of the employee to delete must be present in the request body.
/health	GET	Check if the API is running. On a 200 OK response, the API returns a status message "API is working".

Key Design Features

SOLID Principles

I used SOLID principles to ensure that my application was maintainable, scalable and robust.

Following Single Responsibility Principle, each class in the application is designed to have a single responsibility. The **DatabaseHelper** class handles all operations relating to the local SQLite database, such as the CRUD operations for employees and holiday requests. The **ApiHelper** class manages communication with the API and utilises methods from the DatabaseHelper class to ensure data synchronisation between the two stores. The **NotificationHelper** class handles the creation and display of notifications. This separation of concerns ensures loose coupling between classes, meaning that changes to one 'responsibility', does not have secondary impacts of unrelated areas of the codebase. This use of abstracted layers also means that the codebase follows the Open-Closed Principle, as extending these functionalities and classes does not change the core implementation of the application.

As an aspect of Liskov Substitution Principle, any object of a superclass can be replaced by an object of a subclass without impacting the program. This could be useful, especially during testing, to substitute the **ApiHelper** class with an interface mocking the implementation.

The Interface Segregation Principle ensures interfaces/classes are designed to be specific and only have methods relevant to their needs, to avoid bloated classes with unnecessary methods.

Use of Worker Threads via AsyncTask

The application uses worker threads using **AsyncTasks** to handle API operations in the background, ensuring that the Android UI remains responsive even during intensive, long running operations. If intensive tasks are performed on the main thread, the UI can freeze, degrading user experience or leading to application crashes. AsyncTasks execute tasks in the

background and updates the UI upon completion, enabling the application to handle these intensive tasks without affecting performance.

Fetching all the employee records stored on the API takes a longer period, therefore an **AsyncTask** process was used to perform this network request on the background thread. Loading this dataset, or performing many CRUD operations at a similar time, could block the UI thread if it wasn't executed in the background.

AsyncTask has been deprecated from the Android API and so ExecutorService should be preferred in future projects. Unlike AsyncTask, ExecutorService can handle multiple threads allowing it to handle multiple concurrent tasks.

Summary

This report has covered the design, development and implementation of a employee and holiday request management tool for employees and administrators, via the use of CRUD operations on employee data, submission and actioning of holiday requests and synchronisation with a RESTful API Web Service. Key aspects covered include the Design and Storyboarding of how the application can be used, the Legal Ethical Security and Privacy considerations and concerns and how SOLID design principles have been implemented.

The project successfully delivered a functional, user-friendly application for employee and PTO management. The adoption of best practices ensured scalability and maintainability.

References

DatePickerDialog
https://developer.android.com/reference/android/app/DatePickerDialog

Dialogs - used for deletion cancellation confirmation
https://developer.android.com/develop/ui/views/components/dialogs

CRUD Operations with Android SQLite
https://www.geeksforgeeks.org/how-to-delete-data-in-sqlite-database-in-android/