# Assessment 2: Report

COMP2000 CW2, Corey Richardson

## Introduction

## 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.

## Design and Storyboard

* Class Diagrams
  + Employee
  + PtoRequest
  + EmployeeAdapter, PtoAdapter
  + DatabaseHelper
  + ApiService
  + NotificationHelper
* Design carried forward from CW1
  + XML
* Storyboards illustrating the functionality of the submitted solution
  + Updating Personal Details
  + Updating Employee’s Details, including Delete
  + Submitting a PTO Request
  + Reviewing and Approving PTO Requests as an Admin

### Class Diagrams

|  |  |  |
| --- | --- | --- |
| Figure : UML Diagram for Employee Class | Figure : UML Diagram for PtoRequest Class | Figure : UML Diagram for UserSettings Class |

|  |
| --- |
| Figure : UML Diagram for EmployeeAdapter ListView Adapter |
| Figure : UML Diagram for PtoAdapter ListView Adapter |
| Figure : UML Diagram for aPtoAdapter ListView Adapter |

|  |
| --- |
| Figure 7: UML Diagram for DatabaseHelper Class |

|  |
| --- |
| Figure : UML Diagram for ApiService Class |

|  |
| --- |
| Figure : UML Diagram for NotificationHelper Class |

## Legal, Social, Ethical and Professional Considerations

* Privacy, Integrity, Security
  + Privacy: Passwords NOT encrypted, out of scope for this CW
  + Integrity: DB Transactions, Input Validation and Sanitisation
  + Security: API calls over HTTP! API doesn’t provide SSL certificates
* How were the issues addressed?

## Interface Design

* UI Screenshots with links to XML files in repository
* Design Principles:
  + Consistency across pages
  + Accessibility
    - Light/Dark theme NOT IMPLEMENTED
    - Red/Green theme NOT IMPLEMENTED

## Implementation

* Code snippets in relevant sections
* CRUD
* API Integration
* NotificationService
* Technologies used
  + SQLite
  + Volley for API
  + NotificationManager and NotificationCompat

### 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 |  |
| aAddEmployee | activity\_aadd\_employee.xml | aAddEmployee.java |  |
| aEditEmployee | activity\_aedit\_employee.xml | aEditEmployee.java |  |
| aEmployeeDetails | activity\_aemployee\_details.xml | aEmployeeDetails.java |  |
| aPtoRequests | activity\_apto\_requests.xml | aPtoRequests.java |  |
| Dashboard | activity\_dashboard.xml | Dashboard.java |  |
| EditPersonalDetails | activity\_edit\_personal\_details.xml | EditPersonalDetails.java |  |
| EditPtoRequests | activity\_edit\_pto\_request.xml | EditPtoRequests.java |  |
| ForgotPassword | activity\_forgot\_password.xml | ForgotPassword.java |  |
| ptoMenu | activity\_pto\_menu.xml | ptoMenu.java |  |
| RequestPto | activity\_request\_pto.xml | RequestPTO.java |  |
| Settings | activity\_settings.xml | Settings.java |  |
| ViewHoliday | activity\_view\_holiday.xml | ViewHoliday.java |  |

### ListView Items

|  |  |  |
| --- | --- | --- |
| **Adapter Utilised By** | **XML File** | **Explained** |
| 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 : SQLite Database Schema |

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

|  |
| --- |
| Figure : 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:

|  |
| --- |
| Figure : Example CREATE Operation - insertUser() |

|  |
| --- |
| Figure : Example READ Operation - getAllEmployees() |

|  |
| --- |
| Figure : Example UPDATE Operation - updateUserInDatabase() |

|  |
| --- |
| Figure : 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.

|  |
| --- |
| Figure : 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”. |

* How each endpoint was used by Helper and Service classes

## Key Design Features

### SOLID Principles

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

* Single Responsibility Principle
* Open-Closed Principle
* Liskov Substitution Principle
* Interface Segregation Principle
* Dependency Inversion Principle

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, in an effort 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 of the employee records stored on the API takes a longer time 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.

## Evaluation

* Post-design evaluation on usability and functionality
* Testing?
* Recommendations for future improvements
  + True role based access, rather than hiding activity navigation links

## Summary

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

## References

* Include some Android documentation used here? Currently included as code comments
* Volley documentation
* AlertDialog

## Assessment Materials

* GitHub Repository
* GitHub Release
* Video Link (also include MP4 in submission!)
* WakaTime Project