Report

Mobila Project

Supervisor: Fabien Lagriffoul
Laborant: Tri Dao Nguyen

Abstract

Each time people come up with an idea or a thought, most often they try to write it down as quickly as possible. This app provides a solution to such a problem, by capturing a picture and sending, the information will be directly delivered to relevant parties. During the development of this app, challenges arise like data persistency and database complexity. The app's key feature is customizable templates and supporting the writing process with auto-filled content. Template auto-filling consists of recipient management and updating important information such as time and location. While a well-designed user interface and a structured data layer with Kotlin coroutines provides smooth navigations and parallel executions. The content provider, intent, broadcast, notification, and online API functionalities are implemented. On top of that, the UI layer takes advantage of the powerful adapters and view models for data presentation and interaction. Overall, this report provides an overview of the app's architecture, functionalities, and implementation details.

Table of content

Introduction

Using the app, the user can generate reports and quick notes while on the move. However, the development process has encountered several challenges. One major obstacle is data persistency, with a portion of the implementation being undertaken in Kotlin. Additionally, it is required to create Entity-Relationship (ER) due to the complex database for clarity. Android Studio has proven to be the most suitable platform, offering a well-designed interface that includes features such as word completion, a compiler package and an overall helpful user interface. The initial development stages involved Java, but the useful feature of Kotlin has proven to be more preferred. Particularly the coroutines for parallel execution, introduced by Google.

Functions clarification

Eventhough the app's functionality had been described in the project proposal, it is no where near a proper explanation. Therefore in this report, the implemented functionality of the app will be explained.

Customizable Templates

The core of the app is customizable templates, designed to simplify the reporting process. These templates come pre-populated with relevant content, so the users only have to fill in specific details. Currently there are 2 template types in total, one has less content than the other for every day use. Even so, it is possible to pre fill the title and the email fields.

Recipient Management

The users can predefine recipients for each of their customized templates. By setting up recipient lists in advance, each template use automatically fill in the recipients' emails. This feature aids the sharing process, makes it possible to document and update the relevant parties with particularly beneficial in time-sensitive situations.

Detailed Informations

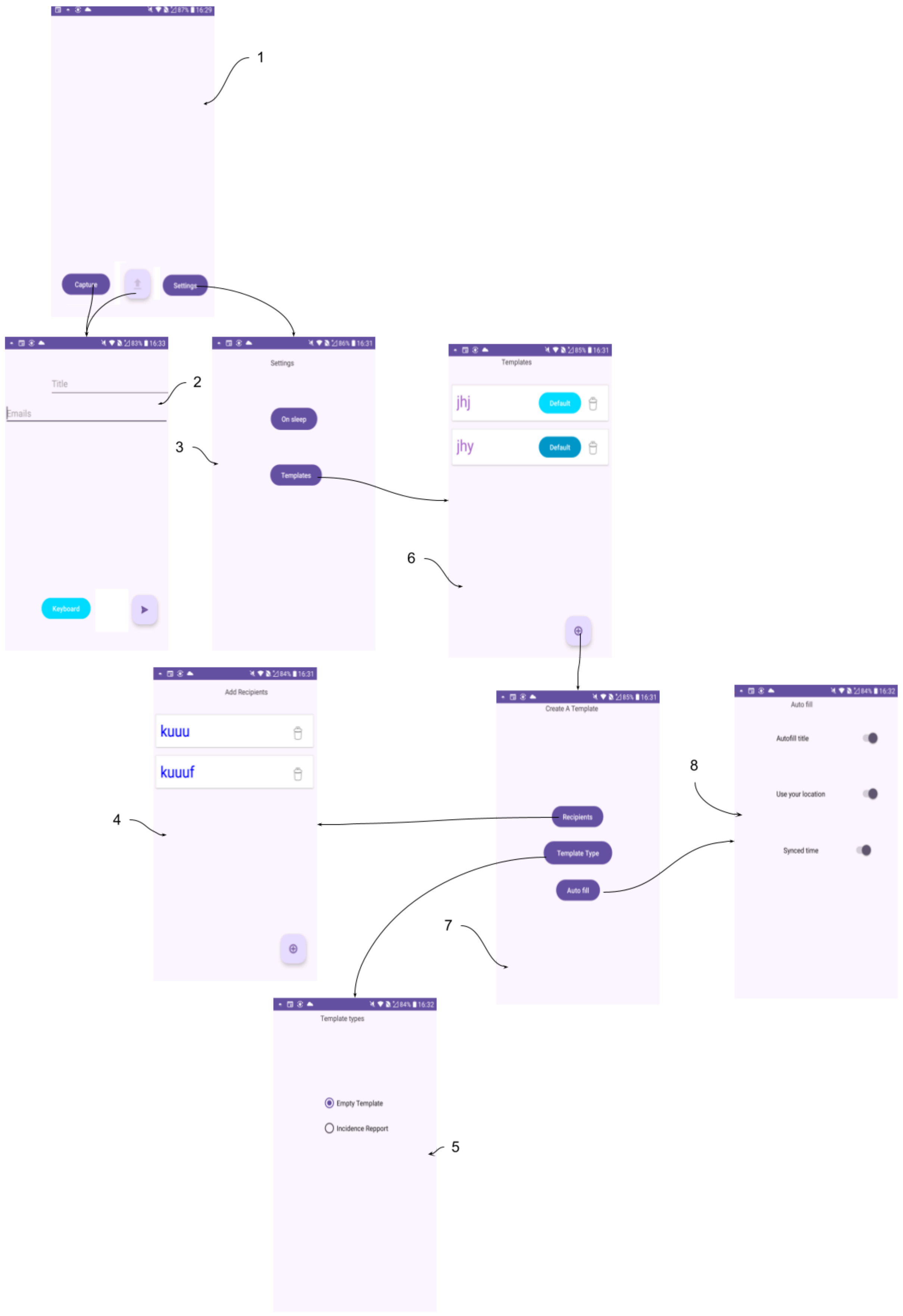
To speed up the documenting process even quicker, users have the option to enable template auto filling feature. This feature, like the name says, fill in the details such as location, date, and time. Be default, these features are always on when a new template is created. However, it is worth to meantion that each time a template is used, the information is fetched anew.

Using the Template

Just by taking a picture, the chosen template will be used. Only one template is marked with deep blue, signifying that it is the chosen one for the next camera capture. By default, each newly created template is always the chosen one.

Annotated Flowchart

Below here is a navigation tree between the activities of the app, starting with the first activity as the root.


1
Capture
Settings
2
Title
Emails
Keyboard
3
Settings
On sleep
Templates
Templates
jhj
Default
jhy
Default
6
Add Recipients
kuuu
kuuuf
4
Create A Template
Recipients
Template Type
Auto fill
7
Auto fill
Autofill title
Use your location
Synced time
8
Template types
Empty Template
Incidence Repport
5

1. **First Activity**
   - Capturing an image or pressing the load button, directs the user to an edit activity (2nd).
   - Access to app settings and more available from this activity (3rd).

2. **Second Activity**
   - Serves as a leaf node, which means the user does not go to any other activities in the app then back to the previous.

3. **Third Activity**
   - Displays app options; currently, only a button marked "Templates" is available.
   - Leads to the sixth activity.

4. **Fourth Activity**
   - Offers only the option to go back, functioning as a leaf node.

5. **Fifth Activity**
   - Similar to the fourth activity, it serves as a leaf node.

6. **Sixth Activity**
   - Pressing the hover button with the addition sign directs the user to the seventh activity.

7. **Seventh Activity**
   - Acts as a hub, providing access to three leaf node activities.

8. **Eighth Activity**
   - A leaf node, similar to the fourth and fifth activities.

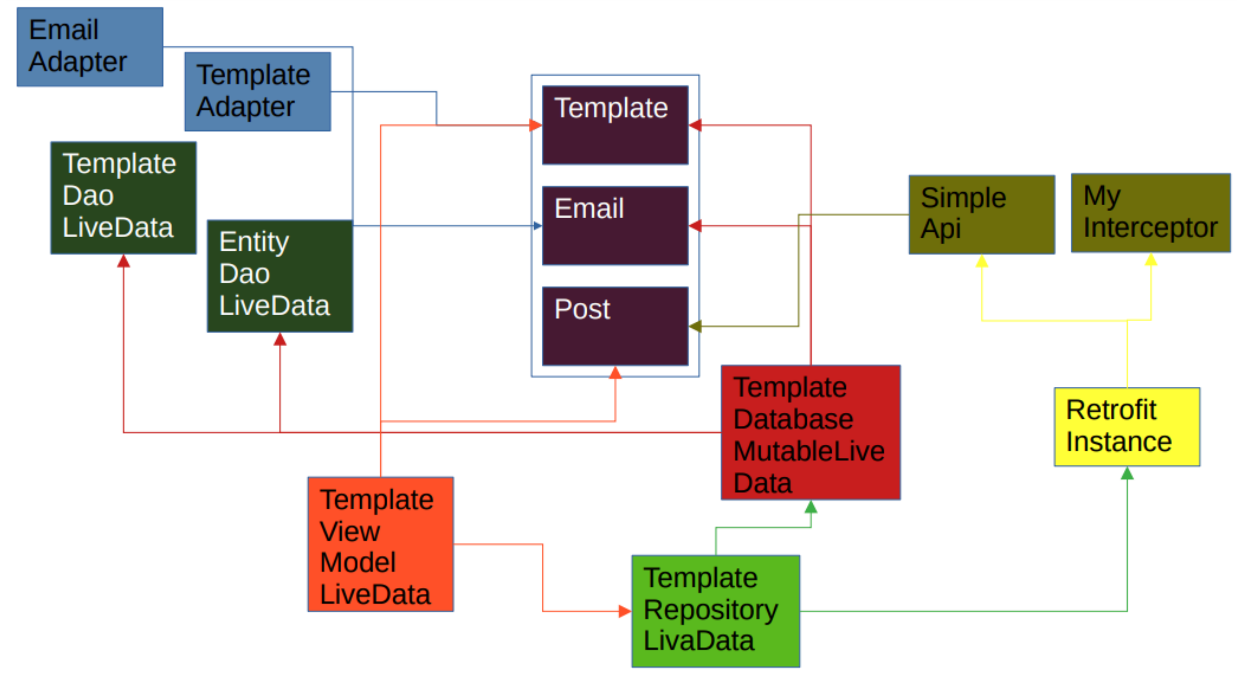
User interface

Most of the buttons are in color purple to indicate that the button will navigate the user elsewhere. In the process of designing the app, a key consideration was the speed at which it aids users in swiftly transmitting their essential information to relevant parties.
first activity: first time using the app, a pop up will show up to direct the user to creating a template. the app will not be able to do its intended function if there is no template. After creating one, the user can now capture an image and the app will follow the template, creating a draft for the user. However it is not finished and needs the user to fill out the rest, so the user will be shown with the draft on the second activity. Next time opening the app, the user will have the option to edit their templates by pressing the right most button. On top of that any previous safe can be loaded using the middle button.
Second activity: In the second activity, the previously captured image is displayed in a smaller size. Here, users can choose to send the displayed data by pressing the "Send" button, enabling them to select an appropriate application. There are only three fields—title, text, and

email—accompanied by an image view which closely represents what an email app may display.
Third activity: Currently, I have only implemented a button in the third activity, enabling users to both create and select their self-designed form. The alternative button serves the purpose of guiding users to other app options, such as using the volume button to initiate the app. Unfortunately, such a feature has not yet been implemented.
Fourth activity: Upon tapping the add icon, users will encounter a popup dialog prompting them to input an email. They have the flexibility to add multiple recipients as needed. By tapping on the card, users can modify the entered email address. Alternatively, pressing the trashcan icon removes the card both from the list and the database.
Fifth activity: For now there are only 2 types of template being implemented. The first type is the simplest one, the user can get help choosing a title and filling out the emails. The second type can do all that and even help with writing some content like writing the structure of an incident report.
Sixth activity: In the sixth activity, a list of user-created templates is displayed, with the one marked as "default" or highlighted in deep blue being the selected template for activity 2. Templates shown in light blue are unchosen and will not be used for activity 2. Upon tapping the add icon, users encounter a popup dialog to enter the name for their newly created template. Selecting the template or completing the name, directs the user to activity 7. Additionally, users can remove a created template by tapping the trashcan icon.
Seventh activity: The seventh activity comprises multiple buttons, each leading to distinct leaf node activities. This activity acts as a crossroad to different template properties.
Last activity: Currently, this marks the final activity within the app. Selecting every available option enables users to speed up the process of providing essential information. As indicated on the buttons, the app is designed to automatically populate certain fields, such as title or location.

Implementation

Data Layer



This application allows users to both store and retrieve data from local sources on their devices, as well as access external data through online APIs. Notably, the retrieval of data from online APIs involves a detailed HTTP process. This layer is often called the data layer since it mostly associates with data.

In terms of data management, the application utilizes different classes to encapsulate various information. The Template class is responsible for storing essential template properties such as autofill locations, which become relevant in camera captures later on. The Email class, on the other hand, serves as a holds the emails associated with each template. Additionally, the Post and PostArg classes are designed to store JSON information upon decoding.

To streamline data organization, two tables suffice for the Template and Email classes. However, the Post and PostArg classes, which are utilized during JSON decoding are not stored. Data Access Objects (DAOs) are implemented to easily access each table.

To fetch data and listen to responses from the online API, SimpleAPI is implemented. Retrofit transforms the API into a Java interface, simplifying the execution of HTTP requests.

All the DAOs and tables are managed via a central database. It is worth mentioning that on any modification to the database, the user's data will not be saved. Finally, the repository adds an additional abstract level to the architecture, housing both the database and Retrofit instance.

This encapsulation ensures a systematic and well-structured implementation of the application's functionalities.

All the elements of the layer are implemented in Kotlin to take advantage of the coroutines and its capabilities for parallel execution.

Content Provider

The process begins by adding a provider in the manifest file. This is important for storage and retrieval of images within the application. Then, a resource file is created. This file essentially acts as a guide, specifying the designated file path.

A URI is generated and then provided to the ActivityResultCaller. When successful writing of the file to the URI, callback mechanism verifies the with a 'true' indicating a successful operation.

Intent

The objectives are to send an intent to a mailing app and navigation between activities. Previously, handling permissions was a somewhat tedious process to send email. Permissions are declared in the manifest and seek approval before actual usage.

On top of sending messages in text form, image files were also attached to the intent for communication with the mailing app.

Broadcast

A notification is configured to appear whenever the battery level reaches a low threshold. A static broadcast receiver named 'HardwareButtonReceiver' was implemented, inheriting from the Broadcast Receiver class. Finally, specific filter was added to listen exclusively for low battery events.

However, this endeavor required requesting permission to access battery information. Fully setting up the Broadcast Receiver, it is declared in the manifest with the 'export' attribute set to true.

Notification

To ensure the successful delivery of notification, a dedicated channel is created and permission is needed. Beyond the technical setup, some working hours were also spent to design the notification itself. Most importantly, a broadcast receiver will invoke the notify function from ManagerCompat upon receiving a battery low intent.

Online Api

When using an online API, authentication is implemented through an interceptor. This interceptor encapsulates the essential header components such as content type and platform, required by the server. Authorization information such as client ID and user secret are transformed and included into the header.

The POST and GET methods for HTTP requests, found within the SimpleApi interface. This interface serves as a blueprint for structuring and executing these requests.

Retrofit provides a client for the interactions. This client incorporates the previously mentioned interceptor and interfaces with the SimpleApi.

UI Layer

The ui layer is composed of all the UI elements and ViewModels. Here, the most elements are the adapters and the activities themselves. Most of the UI are implemented in java since the project originally was written in java.

Adapter

A structured approach is followed, where each RecyclerView is associated with an activity. The adapter creation process involves the making of a CardClickedListener class, responsible for detecting card clicks, and a Holder class to manage the content within each card.

When instantiating the CardClickedListener, all its methods must be overridden, specifying the actions to be taken when a card is clicked. The Holder sets the content for each newly created card and in the same time invokes methods from the CardClickedListener.

Activities such as the number 4 and 6 take advantage of this adapter architecture approach to create new templates or introduce new recipients. Each card is dynamically added to the RecyclerView by only calling the method notifyItemChanged.

ViewModel

The viewmodel offers an abstract layer that ties together data and user interface elements. It operates as a mediator, encapsulating data and provides their interaction with the UI.

Some of the viewmodel's attributes are liveTemplate and myResponse, both of which are LiveData, containing values observable by the UI. Furthermore, the viewmodel establishes a connection to the underlying layer through its repository property, enabling it to retrieve and manage data.

Another aspect of the viewmodel is its extended lifecycle, which is longer than both activities and fragments. During events like screen rotations or when the app loses focus, the viewmodel ensures data persistence by retaining it in the cache. This eliminates the need to re-fetch data from the underlying layer, contributing to operational efficiency.

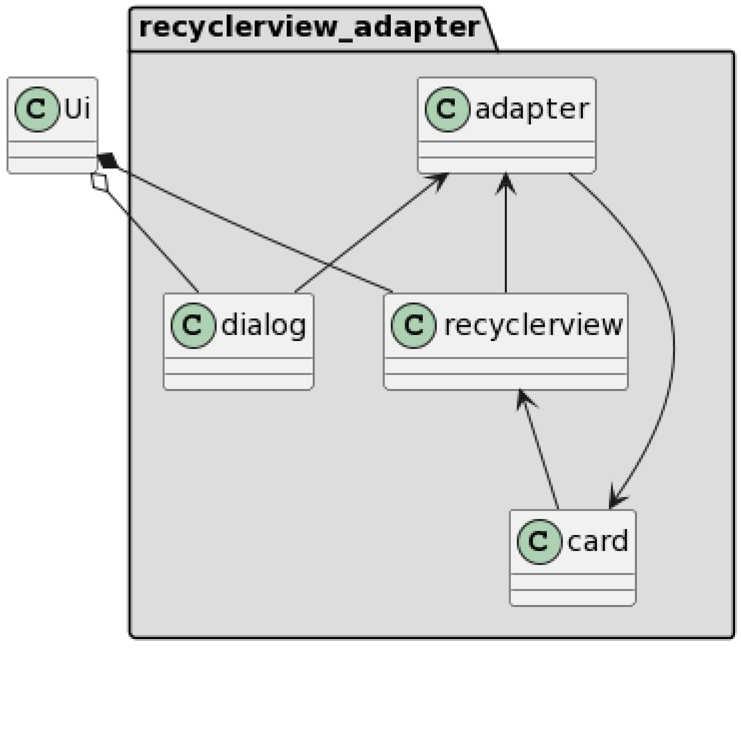
Many viewmodel's methods can modify data and storage. Given the asynchronous nature of many data layer methods, the viewmodelscope is used. This means that even when associated fragments or activities are destroyed, the asynchronous work continues.

It is often recommended to approach a more abstract approach like that which involves LiveData. LiveData serves as a data holder that updates the UI only when it is active. This abstraction starts with the DAO, which returns LiveData that passes along to the repository, viewmodel, and finally, the UI which observes the values.

Furthermore, the introduction of switchmap makes it possible to immediately invoke repository methods upon changes to a specified LiveData. This functionality supports transition between two LiveData instances. Since when one receives the necessary value, this value will be then used as an argument for the next method. The result of this transition is a livedata hence the transition property.

Activities

Viewmodels methods are being used by activities, particularly those involved in data presentation or user input. However, not every activity dedicated to data display needs adapters, recycler views, or cards. Some activities take a different approach, using popup dialogs to present information or gather user input.


recyclerview_adapter
Ui
adapter
dialog
recyclerview
card