

Module 8 Portfolio Project: My Recipe App

Alexander Ricciardi

Colorado State University Global

CSC475: Platform-Based Development

Professor Herbert Pensado

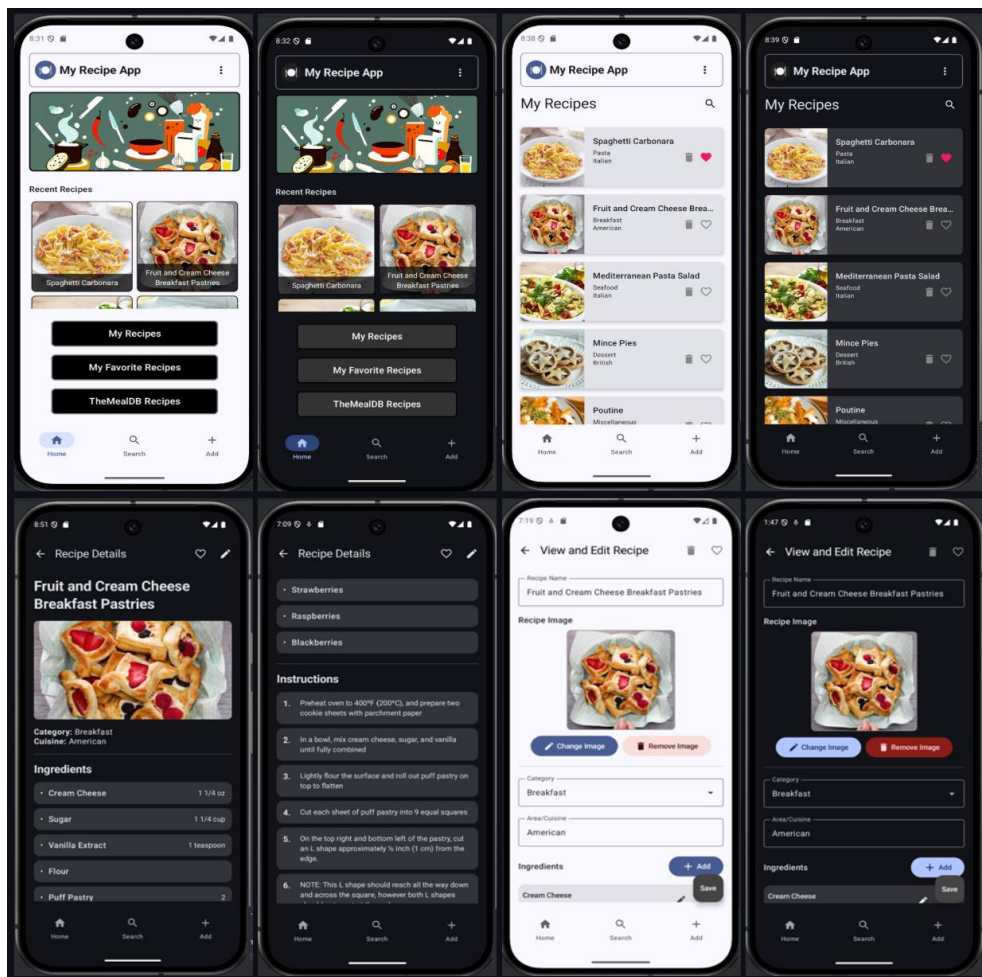
April 6, 2025

Module 8 Portfolio Project: My Recipe App

This document is my portfolio project module from the Platform-Based Development CSC475 course at Colorado State University Global. It is the last step in developing my Android “My Recipe App.” This essay contains a description of the testing process, including details of unit tests and UI tests that were conducted. Additionally, it explains the deployment process of the app and addresses challenges encountered during the deployment phase. Finally, it discusses strategies for ongoing maintenance, bug fixes, and potential future enhancements for the app.

Figure 1

My Recipe App UI Overview



Note: The figure provides an overview of the My Recipe App’s UI.

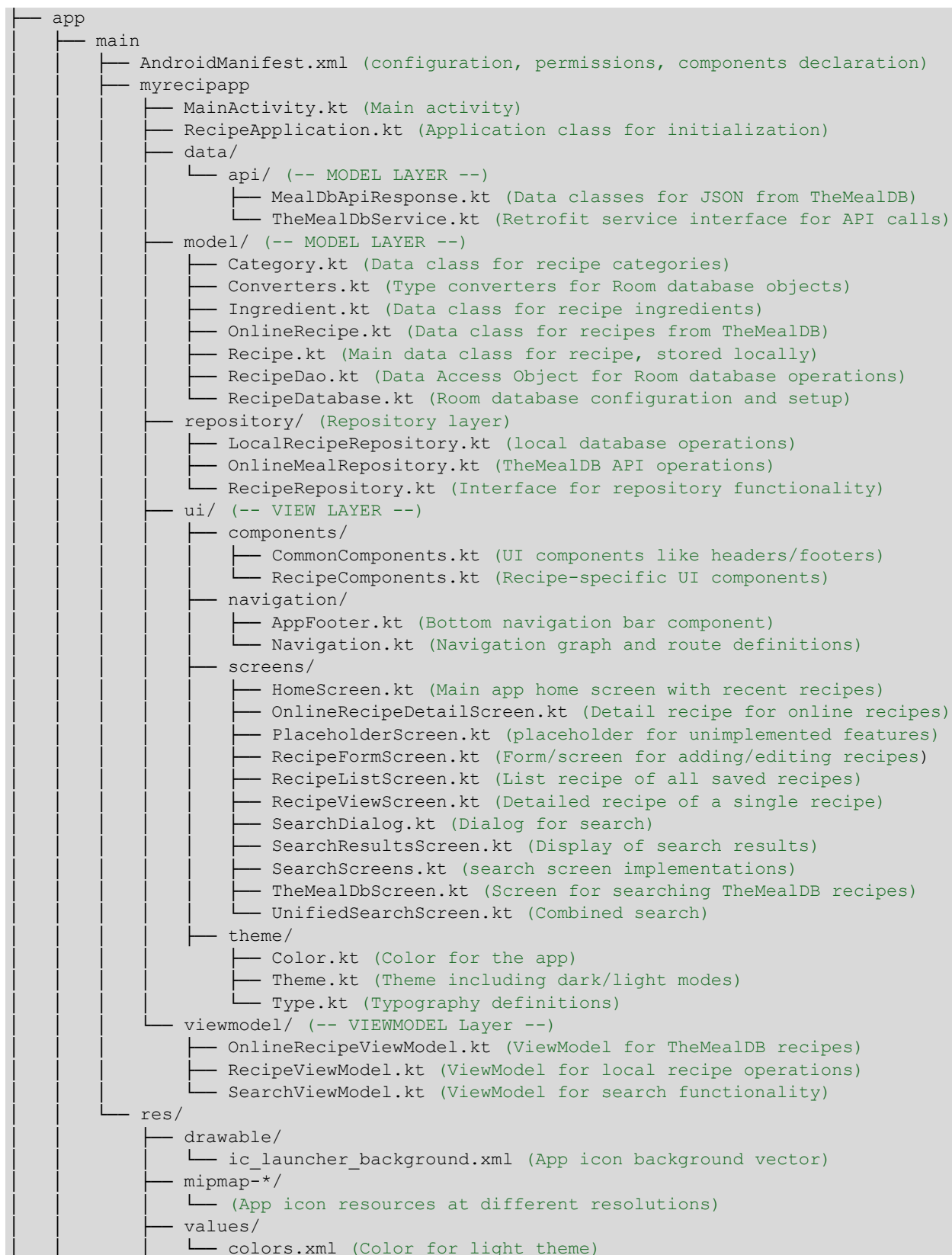
App Description

The app allows a user to access meal recipes. The recipes can be stored on the user's device and fetched from TheMealDB (n.d.) database using API calls. The User Interface (UI) system includes view, search, add, modify, and favorite recipe functionalities, see Figure 1. The app was developed using Kotlin, Jetpack Compose, and the Model-View-ViewModel (MVVM) architecture.

Below is a list of the app requirements:

- Jetpack Compose (2.7.x)
- Kotlin (2.0.21)
- AndroidX Core KTX (1.15.0): Kotlin extensions for core Android functionality
- Navigation Compose (2.7.7): Navigation between screens
- Material 3: Material Design 3 components and theming system
- Room (2.6.1): Local database for storing recipes with SQLite abstraction
- Lifecycle Components (2.8.7): ViewModel and LiveData for MVVM architecture
- Retrofit (2.9.0): Type-safe HTTP client for API
- Moshi (1.15.0): JSON parser for API
- OkHttp (4.12.0): HTTP client and logging
- Coil (2.5.0): Image loading library
- Compose Runtime LiveData (1.6.2)
- Gson (2.10.1): JSON serialization/deserialization library
- Activity Compose (1.10.1): Compose integration with Activity
- Compose BOM: Bill of materials for consistent Compose dependencies

Note that the app requires a TheMealDB API key. Below is the app file structure:



```

├── values-night/
│   └── colors.xml (Color for dark theme)
└── xml/ (Additional XML configuration files)

```

App Testing

The testing was done in phases that followed a functional testing approach. The tests were separated into four categories which are unit testing, integration testing, UI testing, and testing. The tests were implemented using a variety of techniques, such as using JUnit, Hamcrest, Robolectric, and UIAutomated libraries and tools, such as mock web servers, Android Studio virtual devices, and Firebase on the web. To test the app, I created a different branch in my git environment, which I called 'test'. Below is the list of the different tests mimicking the test file structure within the project. Note that the test structure follows best practices, which separates instrumentation tests (UI/integration) and unit tests.

```

├── app
│   └── src
│       ├── androidTest (-- Instrumentation/Integration Tests --)
│       │   ├── AndroidManifest.xml (Test manifest configuration)
│       │   └── java/com/example/myrecipeapp/
│       │       ├── AppNavTest.kt (tests with complex testing scenarios)
│       │       └── ui/
│       │           └── screens/
│       │               ├── RecipeListScreenTest.kt (recipe list screen interactions)
│       │               └── RecipeFormScreenTest.kt (recipe form screen interactions)
│       └── test (-- Unit Tests --)
│           └── java/com/example/myrecipeapp/
│               ├── viewmodel/ (-- VIEWMODEL LAYER TESTS --)
│               │   ├── RecipeViewModelTest.kt (RecipeViewModel-lifecycle-LiveData)
│               │   ├── TestRules.kt (JUnit rules for ViewModel testing)
│               │   └── OnlineRecipeViewModelTest.kt (OnlineRecipeViewModel- API)
│               ├── util/
│               │   ├── UtilityFunctionsTest.kt (utility functions in the app)
│               │   ├── RecipeFilteringTest.kt (recipe filtering and sorting)
│               │   ├── SimpleTest.kt (simple utility test)
│               │   ├── TestHelper.kt (helper functions for tests)
│               │   ├── ConvertersTest.kt (Room database TypeConverters)
│               │   └── MainDispatcherRule.kt (Rule for testing coroutines)
│               ├── repository/ (-- REPOSITORY LAYER TESTS --)
│               │   ├── RepositoryTest.kt (repository CRUD operations)
│               │   └── CombinedRepositoryTest.kt (combined local/online repository)
│               └── api/ (-- API LAYER TESTS --)

```

```

└─ TheMealDbApiTest.kt (TheMealDB API client)
└─ database/ (-- DATABASE LAYER TESTS --)
    └─ RecipeDatabaseTest.kt (Room database operations - migrations)

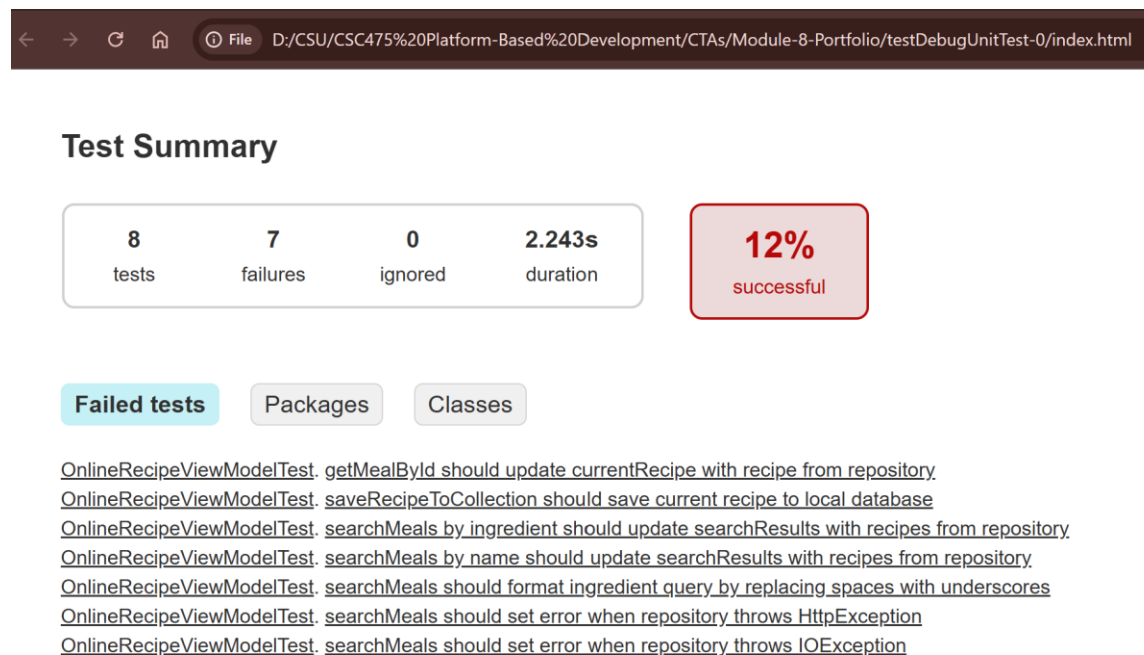
```

Unit Test

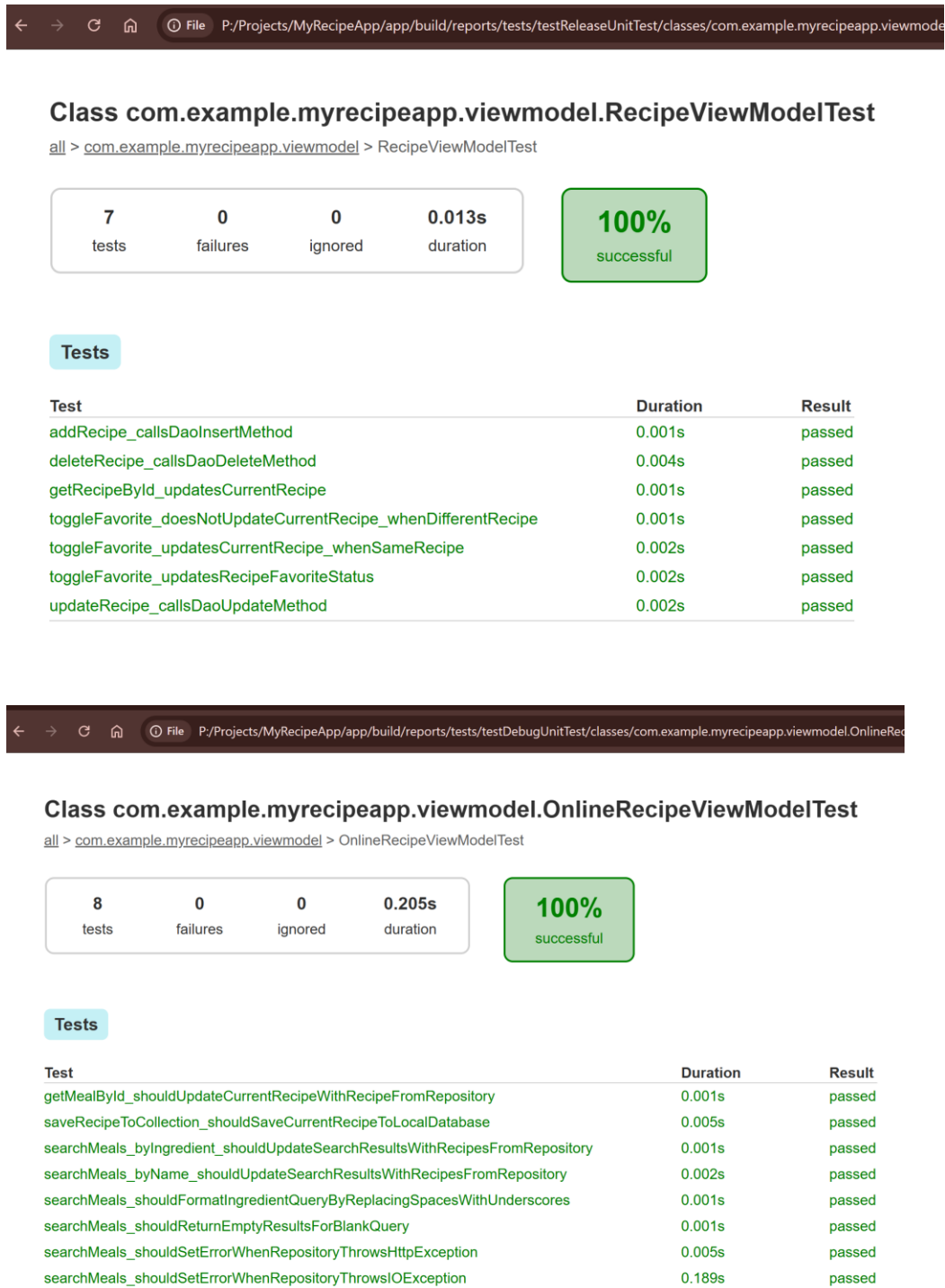
First, I created unit tests to test the ViewModel logic, those tests can be found in the `test/java/com/example/myrecipeapp/viewmodel` directory. Each test focused on one aspect of the ViewModel's functionality. The main issue I encountered was an issue with the `TestOnlineMealRepository` class could not extend the `OnlineMealRepository` class, resulting in the failure of almost all the ViewModel tests, see Figure 2. Making the `OnlineMealRepository` class open so it can be extended, fixed the issue, see Figure 3. This is apparently a common issue in Kotlin, where classes are final by default (unlike Java).

Figure 2

Failing ViewModel Unit Tests



Note: The figure illustrates the failing ViewModel unit tests.

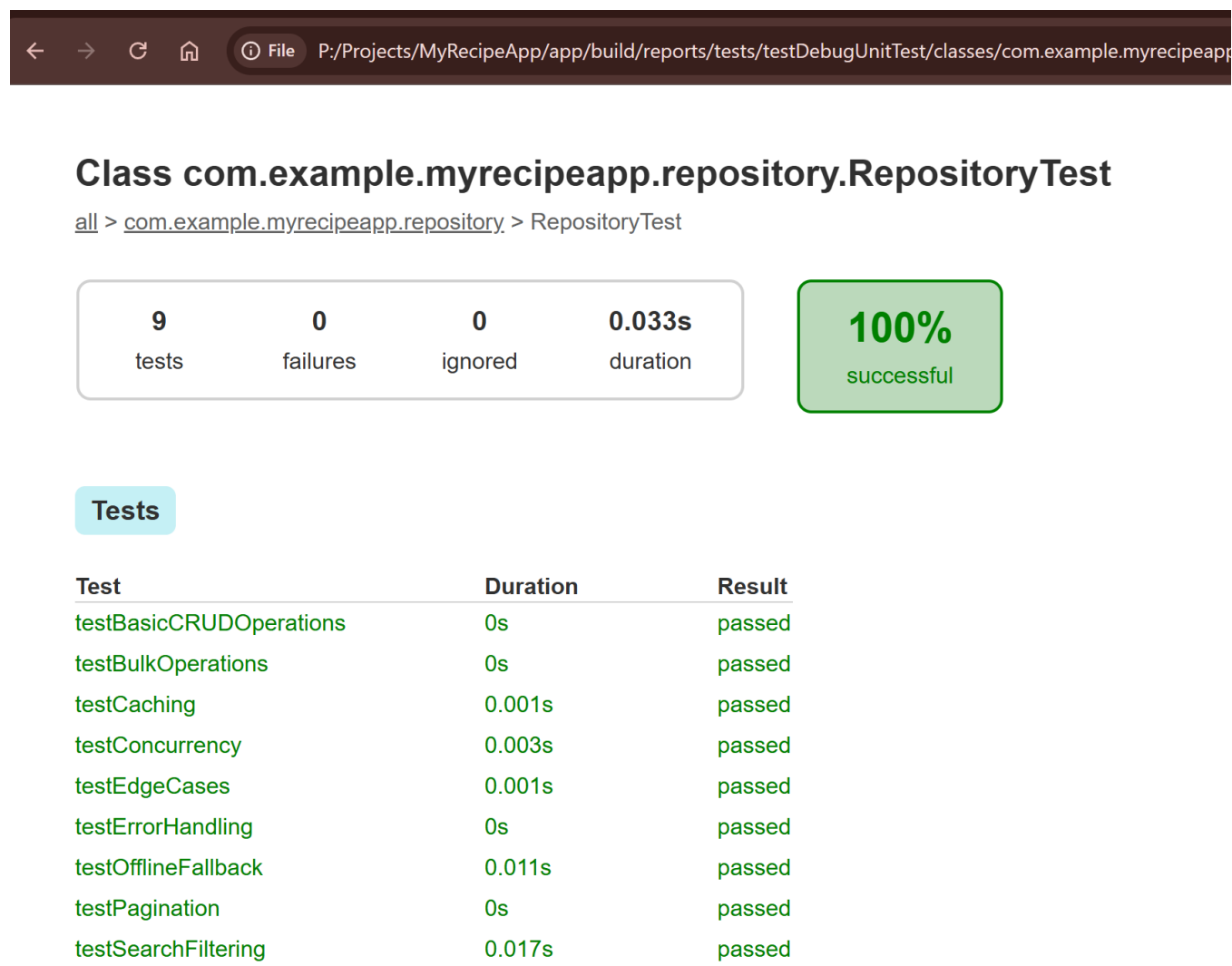
Figure 3*Passing ViewModel Unit Tests*

Note: The figure illustrates the passing ViewModel unit tests.

Next, I created unit tests to test the Repository logic. The tests can be found in the `test/java/com/example/myrecipeapp/repository` directory. These tests verify the functionality of CRUD operations, thread-safe operation, simulated network latency, etc. They verify various aspects of Repository functionality. See Figure 4 for test results.

Figure 4

Passing Repository Unit Tests



Note: The figure illustrates the passing of Repository unit tests.

I also created unit tests to verify the functionality of various utility functions, such as type converters for the Room database, tests for ingredient formatting, string operations, and other basic utility methods. They verify various aspects of various utility methods. The tests can be found in the `test/java/com/example/myrecipeapp/util` directory. See Figure 5 for test results.

Figure 5

Various Utility Methods Unit Test Results

Package com.example.myrecipeapp.util

[all](#) > [com.example.myrecipeapp.util](#)

20
tests

0
failures

0
ignored

0.030s
duration

100%
successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
ConvertersTest	5	0	0	0.025s	100%
RecipeFilteringTest	5	0	0	0.001s	100%
UtilityFunctionsTest	10	0	0	0.004s	100%

Class com.example.myrecipeapp.util.ConvertersTest

[all](#) > [com.example.myrecipeapp.util](#) > [ConvertersTest](#)

5
tests

0
failures

0
ignored

0.025s
duration

100%
successful

Tests

Test	Duration	Result
testCategoryConversion	0s	passed
testIngredientListConversion	0.003s	passed
testNullOrEmptyIngredientsList	0.007s	passed
testStringListConversion	0s	passed
testStringListWithSpecialCharacters	0.015s	passed

Class `com.example.myrecipeapp.util.RecipeFilteringTest`

[all](#) > [com.example.myrecipeapp.util](#) > `RecipeFilteringTest`

5	0	0	0.001s	100% successful
tests	failures	ignored	duration	

Tests

Test	Duration	Result
<code>testCombinedFilters</code>	0.001s	passed
<code>testFilterByCategory</code>	0s	passed
<code>testFilterByIngredients</code>	0s	passed
<code>testFilterByKeyword</code>	0s	passed
<code>testFilterFavorites</code>	0s	passed

Class `com.example.myrecipeapp.util.UtilityFunctionsTest`

[all](#) > [com.example.myrecipeapp.util](#) > `UtilityFunctionsTest`

10	0	0	0.004s	100% successful
tests	failures	ignored	duration	

Tests

Test	Duration	Result
<code>testCategoryConverter</code>	0s	passed
<code>testIngredientFormatWithMultipleScenarios</code>	0s	passed
<code>testIngredientFormattedOutput</code>	0.002s	passed
<code>testIngredientFromString</code>	0s	passed
<code>testInvalidIngredientStrings</code>	0s	passed
<code>testRecipeEmptyIngredients</code>	0.001s	passed
<code>testRecipeGetFormattedIngredients</code>	0s	passed
<code>testRecipeGetIngredientsList</code>	0s	passed
<code>testStringListConverter</code>	0.001s	passed
<code>testValidIngredientStrings</code>	0s	passed

Note: The figure illustrates the passing of the Utilities unit tests.

Integration Testing

Integration tests focus on the database and network components of the app. The app uses the Retrofit library to make calls to the TheMealDB database. Unlike unit tests, integration tests can run on a device/emulator or a simulated Android environment. To run my integration tests, I used the Robolectric library to simulate an Android environment.

First, I created a test class that integrates the Room database that can be found in the `test/java/com/example/myrecipeapp/database` directory, and it is called `RecipeDatabaseTest.kt`. The test covers different aspects of the Room database functionality. See Figure 6 for test results.

Figure 6

Various Room Database Integration Test Results

```

w: file:///P:/Projects/MyRecipeApp/app/src/test/java/com/example/myrecipeapp/viewmodel/TestRules.kt:29:38 'constructor(sc
> Task :app:compileDebugUnitTestJavaWithJavac NO-SOURCE
> Task :app:processDebugUnitTestJavaRes UP-TO-DATE
> Task :app:testDebugUnitTest
RecipeDatabaseTest > getAllRecipes PASSED
RecipeDatabaseTest > searchRecipes PASSED
RecipeDatabaseTest > insertAndRetrieveRecipe PASSED
RecipeDatabaseTest > updateRecipe PASSED
RecipeDatabaseTest > deleteRecipe PASSED
RecipeDatabaseTest > getFavoriteRecipes PASSED
RecipeDatabaseTest > searchByCategory PASSED
BUILD SUCCESSFUL in 14s
33 actionable tasks: 7 executed, 4 from cache, 22 up-to-date

Build Analyzer results available
8:54:39 AM: Execution finished ':app:testDebugUnitTest --tests "com.example.myrecipeapp.database.RecipeDatabaseTest"'.
```

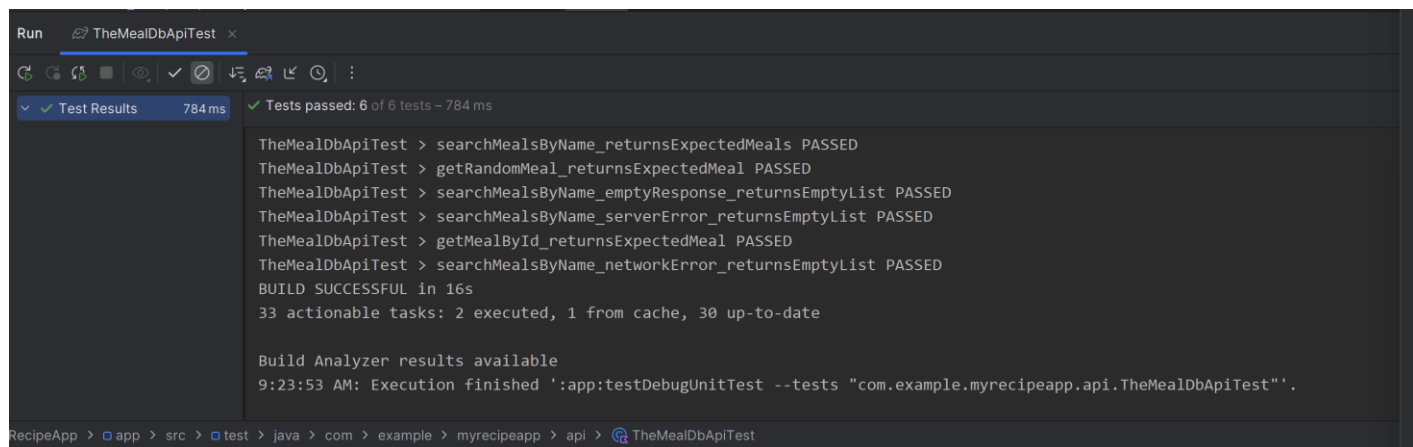
Note: The figure illustrates the passing of the Room Database integration tests.

Next, I created integration tests to verify the app's use of the TheMealDB API. I created a mock server, `MockWebServer`. I set up the server using the `@Before` method, created an instance pointing to the `MockWebServer` URL using Retrofit. The test is named `TheMealDbApiTest.kt`, and it can be found in the `test/java/com/example/myrecipeapp/api` directory. The test covers

different aspects of the app TheMealDB API functionality. See Figure 7 for test results. I encountered an issue with Android Studio triggering a `ConnectException` error with "Connection refused". This took a considerable amount of time to resolve, as I believe that it was an issue with the code, even if the tests were passing. Anyhow, after researching the error, it was fixed by removing the cache and all indexes from Android Studio using the “Invalidate and Restart” command found in the Android Studio menu `File → Invalidate Caches / Invalidate and Restart`.

Figure 7

Various TheMealDB API Integration Test Results



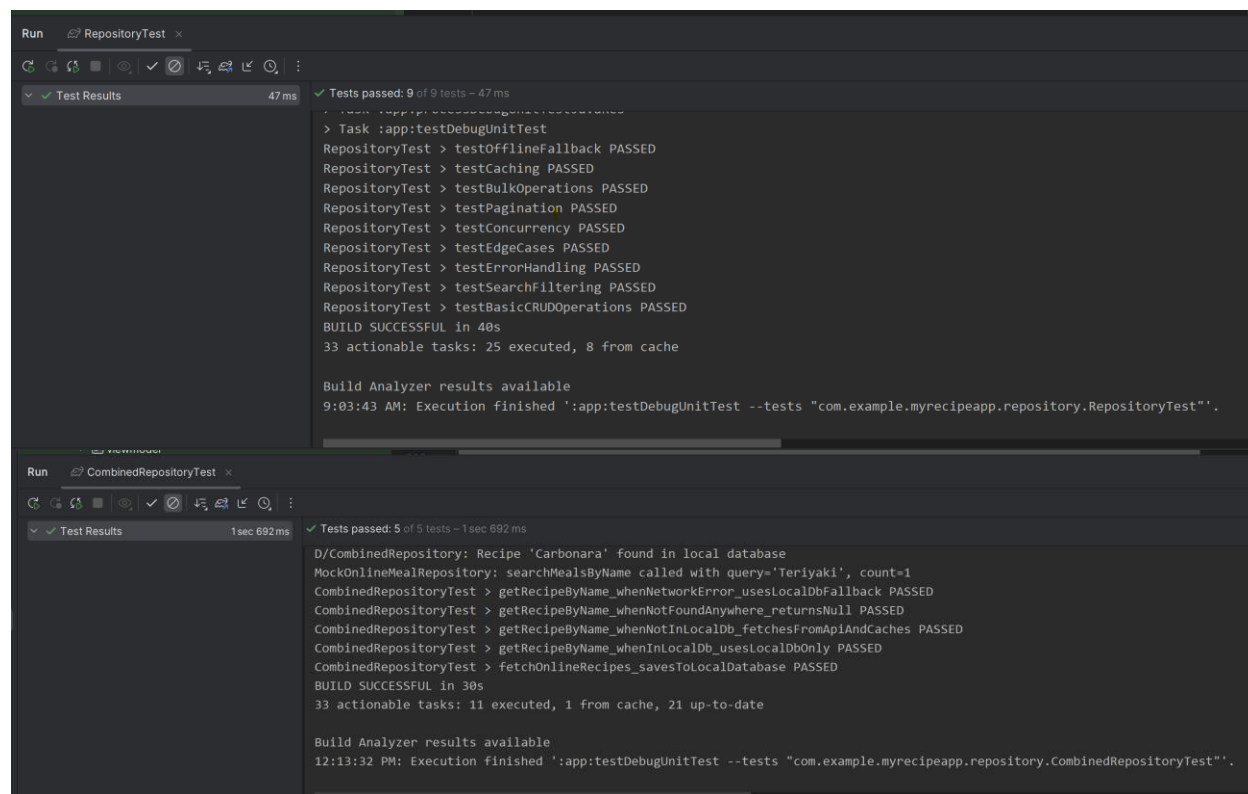
Note: The figure illustrates the passing of the TheMealDB API integration tests.

I also created integration tests to verify the functionality of the Repository layer. To check the functionality of the Repository itself and the Repository combined with both local database and network operations (API). These tests can be found in the `test/java/com/example/myrecipeapp/repository` directory, and they are called `RepositoryTest` and `CombinedRepositoryTest`. Note that the `RepositoryTest` verifies the basic CRUD, search and filter, cache implementation, etc. operations. On the other hand, the `CombinedRepositoryTest`

verifies the Repository combined with both local database and network operations (API) using Robolectric. See Figure 8 for test results.

Figure 8

Various Repository Integration Test Results



Note: The figure illustrates the passing of the Repository integration tests.

UI Testing

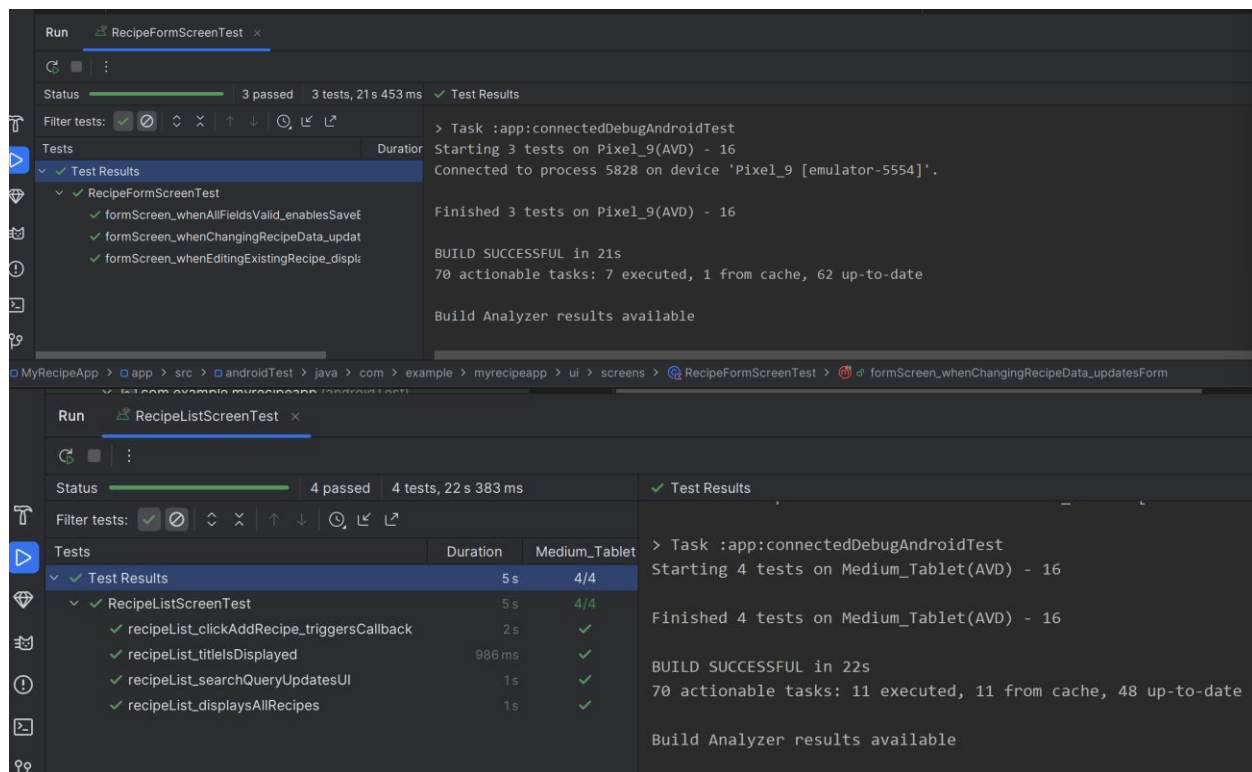
UI tests focus on the Jetpack Compose UI components of the app. The tests use the Composable Test Rules from the JUnit4 library (`androidx.compose.ui.test.junit4.createComposeRule`) to verify the functionality of the app components `RecipeFormScreen` and `RecipeListScreen`. To verify the functionality of the app UI navigation component, the tests use a mix of `UIAutomator`, `Expresso`, and `JUnit4` libraries. I also

use Android Studio virtual device manager and the Firebase tool on the web to test screen sizes and resolutions.

First, I create UI tests to verify the UI components, such as `RecipeFormScreen` and `RecipeListScreen`. These tests can be found in the `androidTest/java/com/example/myrecipeapp/ui/screens` directory, and they are called `RecipeFormScreenTest` and `RecipeListScreenTest`. The major issue I encounter using Hamcrest matcher in JUnit asserts to test user inputs, what ends up working best for my project is using, for example, the assert `assertTextContains("Vanilla Cake")`, which is an extension function provided by the `androidx.compose.ui.test` library. See Figure 9 for test results.

Figure 9

Various Components UI Test Results

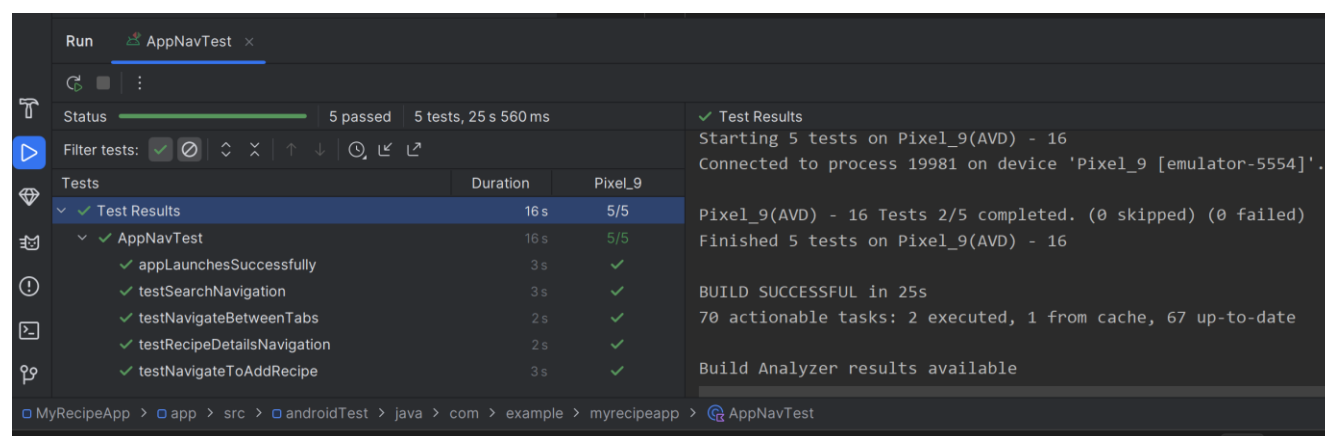


Note: The figure illustrates the passing of the Components UI tests.

Next, I create a UI test to verify the functionality of the UI navigation component of the app. The tests verify various aspects of how the different app UI screens are linked. The tests can be found in the `androidTest/java/com/example/myrecipeapp` directory in the file `AppNavTest.kt`. See Figure 10 for test results.

Figure 10

Various UI Navigation Components UI Test Results



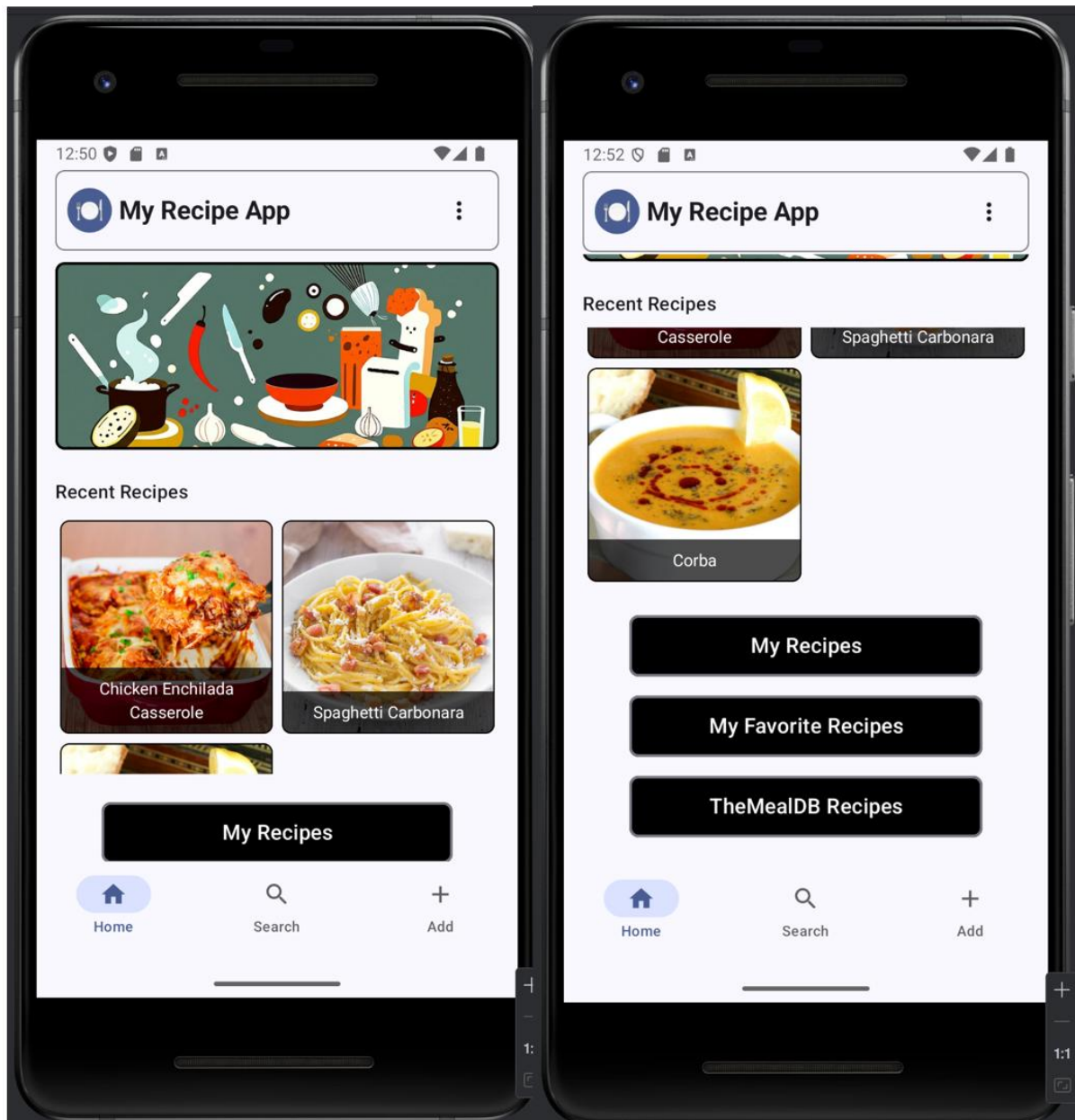
Note: The figure illustrates the passing of the UI Navigation Components UI tests.

I also used the Android Studio virtual device manager to test the UI on different screen sizes and devices. However, the number of virtual tablets was very limited; see Figures 11 and 12 for the UI test performed using Android Studio virtual device manager. The test results show that the images of the meal on buttons on the app's Home Screen, when using an emulated Pixel Tablet Android 16, do not look great. A similar problem exists in the Recipe Detail Screen. These problems can be addressed by implementing responsive design adjustments, mostly for images, for the Home Screen and the Detail/Edit Recipe Screen, which are specific to the device being used, particularly tablets. I also use the Firebase web tool to further test the UI screens on different devices, see Figure 13. But I was also limited on how many devices I could use without

having to change from a free plan to a paid one. I also had to create an Android App Bundle (AAB) file of the project to be used by the Firebase web tool.

Figure 11

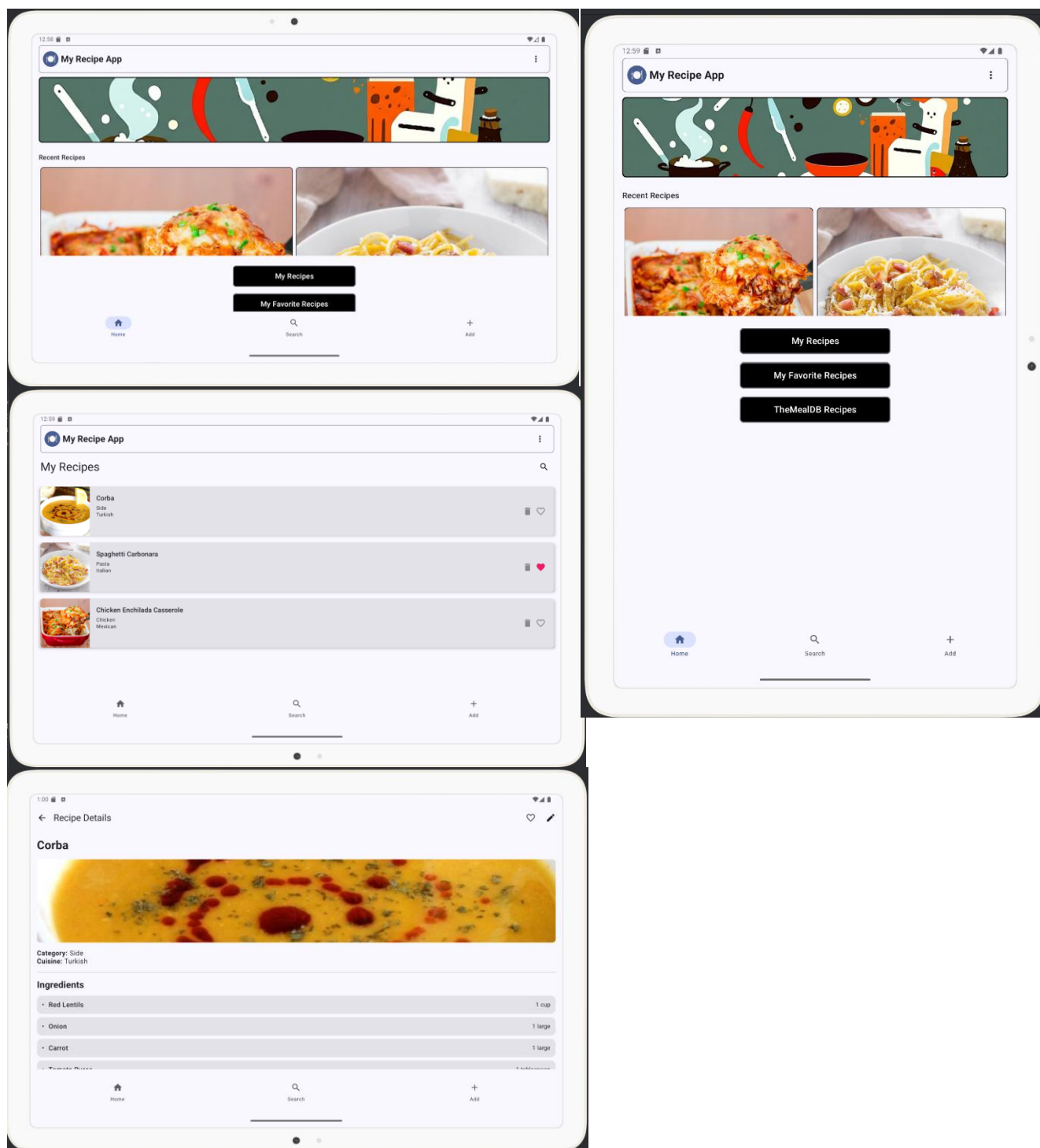
Pixel2 Screen UI Tests



Note: The figure illustrates Pixel2 Screen UI tests.

Figure 12

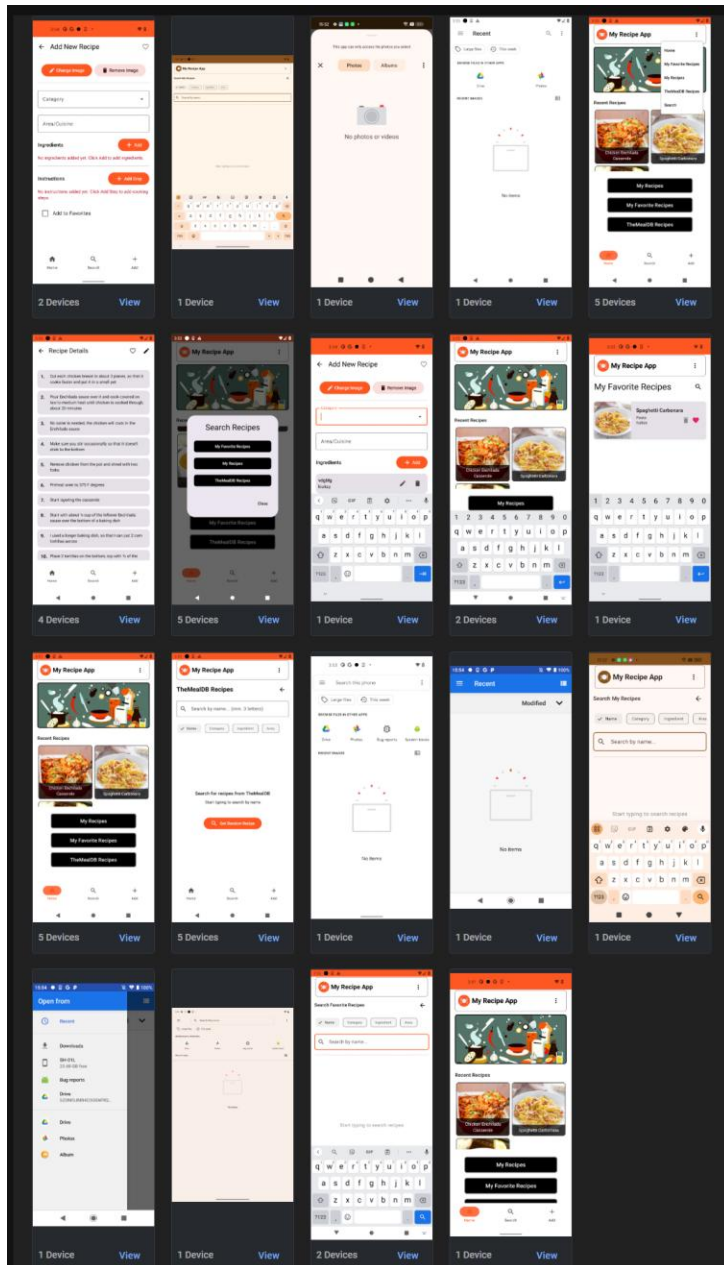
Pixel Tablet Android 16 Screen UI Tests



Note: The figure illustrates Pixel Tablet Android 16 Screen UI tests.

Figure 13

Various Devices UI Tests Using Firebase Web Tool



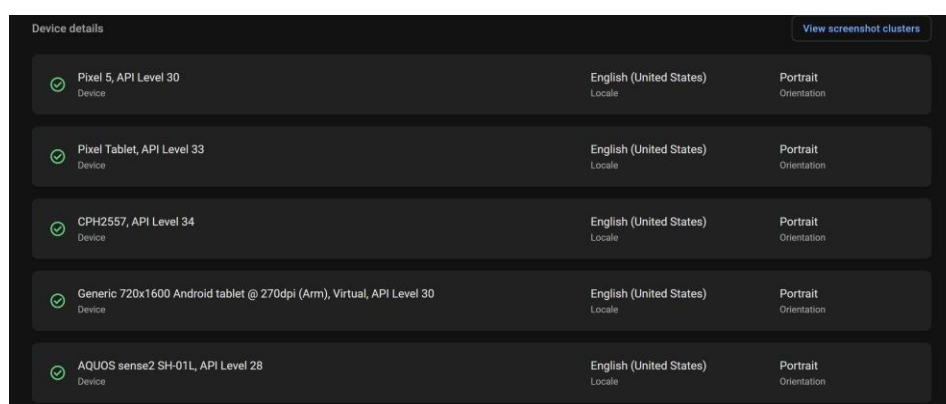
Note: The figure illustrates the different devices used to test the UI on different screens. Clicking on the view option at the bottom of each image will display more images showcasing the app on the devices. Additionally, the images showing a folder icon demonstrate the app accessing only the photo folder of the device; the devices had no images in their photo folders.

End-to-end Testing

The End-to-End (E2E) focuses on simulating real user workflows across the entire app, from launching it to verifying the operation of the UI, ViewModel, Repository, and Model (database) step by step. For that purpose, use the first Firebase Robo test, which is part of the Firebase Test Lab feature. “Robo test analyzes the structure of your app's user interface (UI) and then explores it methodically, automatically simulating user activities” (Firebase, n.d.a). Although the test simulates user activity is not a true E2E test as it mostly interacts with the app UI and does not verify the functionality of the test business logic, data persistence, and interactions with backend systems, see Figure 14 for the Robo test results. Firebase provides many features that allow you to further test the app, such as Test Lab for iOS. However, I was limited by the number I could use with my free plan, and some of the features are only part of the paid plan. Anyhow, after running my Robo tests, I perform the rest of the E2E tests manually, verifying the functionality of the entire app flow.

Figure 14

Robo Test Results



The screenshot shows the 'Device details' section of the Firebase Test Lab interface. It lists five devices, each with a green checkmark indicating a successful test. The devices are: Pixel 5, API Level 30; Pixel Tablet, API Level 33; CPH2557, API Level 34; Generic 720x1600 Android tablet @ 270dpi (Arm), Virtual, API Level 30; and AQUOS sense2 SH-01L, API Level 28. All tests were run on an English (United States) locale in Portrait orientation. A 'View screenshot clusters' button is visible in the top right corner.

Device	Locale	Orientation
Pixel 5, API Level 30	English (United States)	Portrait
Pixel Tablet, API Level 33	English (United States)	Portrait
CPH2557, API Level 34	English (United States)	Portrait
Generic 720x1600 Android tablet @ 270dpi (Arm), Virtual, API Level 30	English (United States)	Portrait
AQUOS sense2 SH-01L, API Level 28	English (United States)	Portrait

Note: The figure illustrates five Firebase Robo tests done using five different devices. All the tests passed (Green Check). The number of tests was limited by my free plan. The “View screenshot cluster” option links to screenshots that are featured in Figure 13.

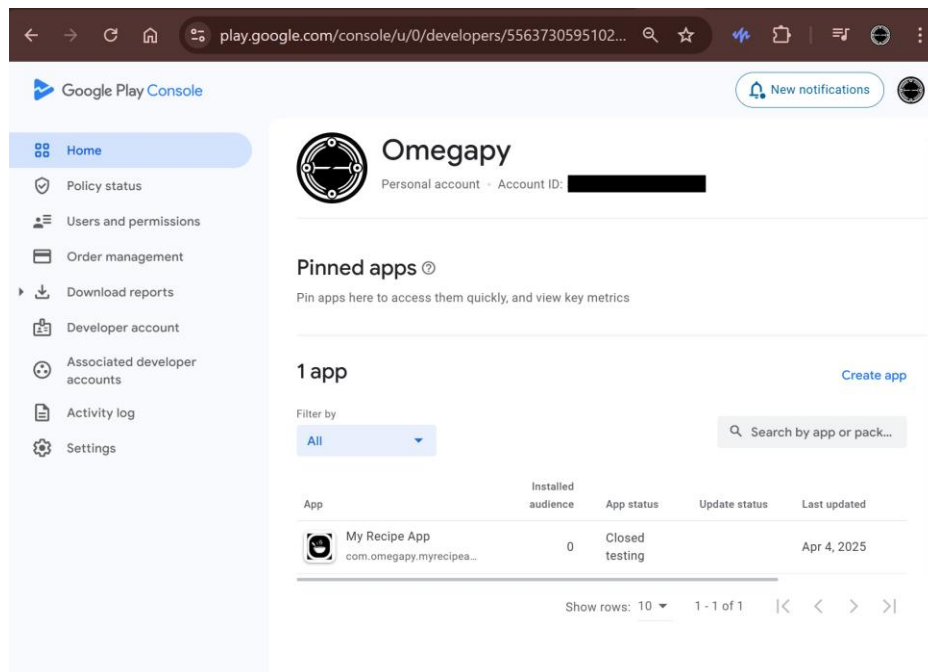
This concludes with the app test section. Implementing the tests was challenging. I struggled the most with implementing the UI navigation tests, it was challenging to understand how to create and how the asserts work when simulating inputs. Nonetheless, the “My Recipe App” passed all the tests; however, the UI tests reveal that the app UI is not well optimized for tablet screen size. This issue will be addressed in further optimization of the app.

Deploying the App

I decided to deploy the app through the Google Play Store. I use the “How to Publish an Android App to Google Play | Updated 2024” YouTube video by MJSD Coding (2024) to guide me through the publishing steps. The process of publishing an app in the Google Play Store requires many steps; this section reflects on these steps and on the challenges that I encountered during the publishing process.

Figure 16

My Google Play Developer Account



Note: The figure illustrates my Google Play Account.

Developer Account

Before pushing the account using the Google Play Console, you need to have a Google Play developer account and pay the creation fee of \$25. This process requires you to prove your identity and address by providing two kinds of ID. Additionally, it requires that you possess at least one Android device. Luckily, I had an old Pixel 9 lying around in my house. I reset the device, and I created a new account for it. After providing all the information needed, I was successful in creating a Google Play Developer account, see Figure 16.

Set up App Content

This step consists of setting the app content, such as:

- Privacy Policy: My policy URL:

<https://1drv.ms/t/c/aa70b8d9fb5f8fc2/EcNhC4xRMx9FiupDxeEyjSwBTAP9O85Hud1NfonhWRUUhw?e=oRAGGy>
- Ads: My app has no ads.
- App Access: My app has no required login credentials.
- Content Rating: My app's age rating is 13.
- Target Audience: My target audience is adults.
- News Apps: My app is a news app.
- Data Safety: My app does not collect or share user data.
- Advertising ID: My does not have ads.
- Government Apps, Financial Features, Health Apps: Not applicable for my app.

Store Listing Setup

This step configures how my app appears on the Play Store. Provides the app's name, contact details, and short/long descriptions. Additionally, it provides the app's icon, banner, and phone/tablet screenshots. The app is also tagged with a food/drink tag.

Build App Release Package

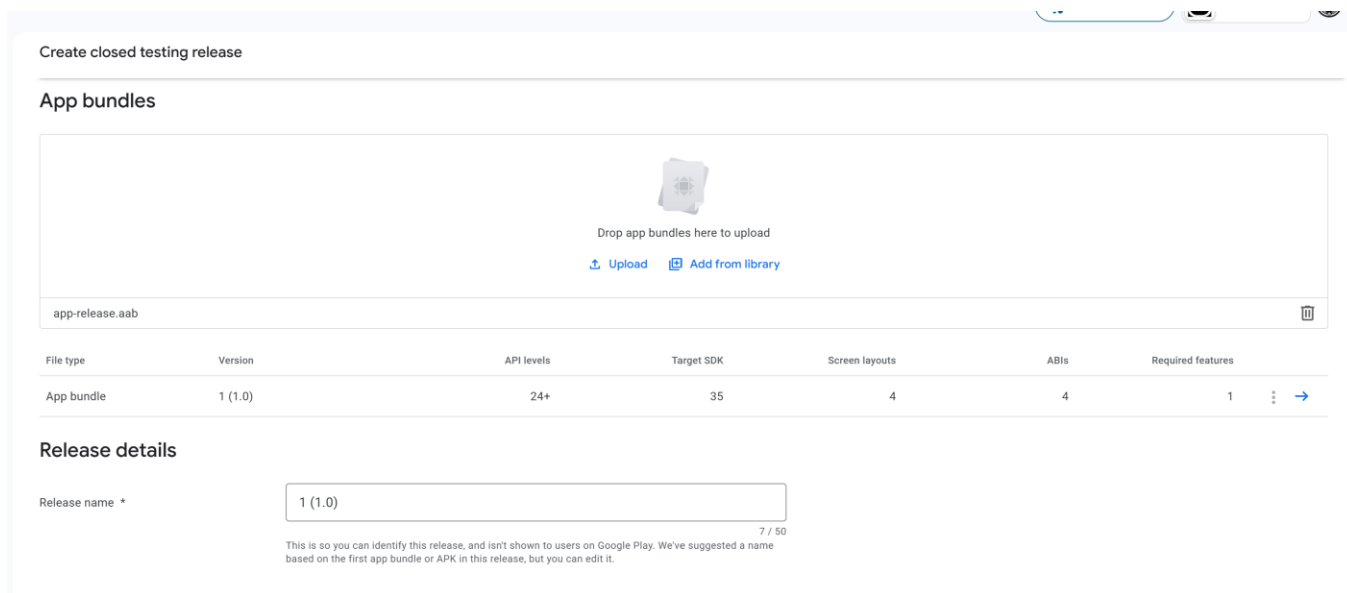
To release the app, I had to build a signed Android App Bundle (AAB) package, you can also build a signed Android Package Kit (APK) file. I chose the AAB as it is recommended by Google when releasing the first version of the app. The app needs to be signed, meaning that the app needs to possess a unique cryptographic key. The process of building a signed AAB file is easily done through Android Studio. The package is used by the Google Play Store to distribute the app. The first time that I tried to submit my app AAB file, the console rejected it and gave the following error: "You need to use a different package name because 'com.example' is restricted." To fix this issue, I had to change the `applicationId` from `com.example` to `com.omegapy`, renaming the directory in the `build.gradle` file (Akhtar, 2015). For more information, see Code Snippet 1. The AAB package was then accepted and given a release name 1(10), see Figure 17.

Code Snippet 1

applicationId

```
defaultConfig {
    applicationId = "com.omegapy.myrecipeapp" // changed from com.example.myrecipeapp
    minSdk = 24
    targetSdk = 35
    versionCode = 1
    versionName = "1.0"
}
```

Note: The code snippet showcases the change made to the `applicationId` to fix the issue with the Google Play Console.

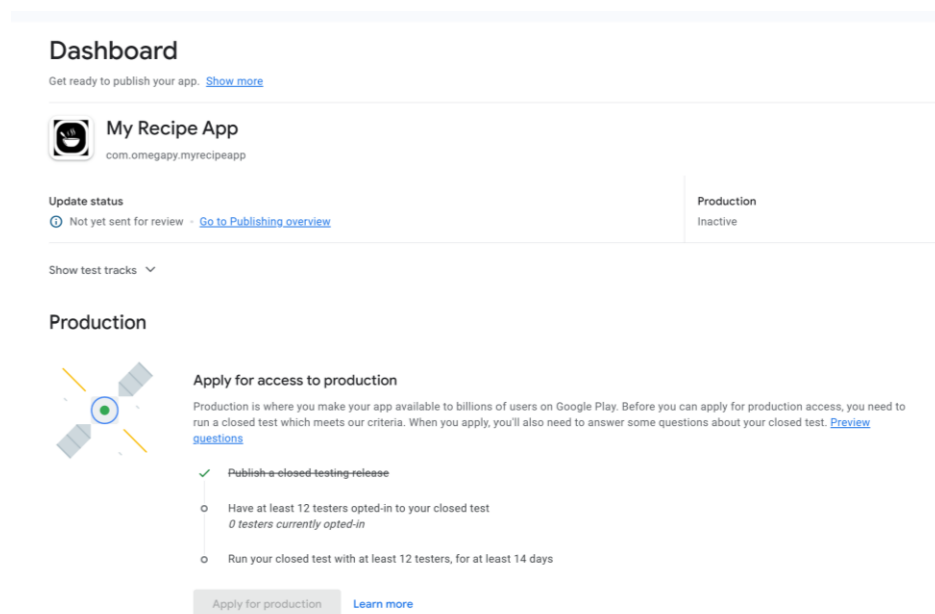
Figure 17*AAB Package Accepted*

Note: The figure illustrates the app's AAB package being accepted by the Google Play Console..

New Developer Account Testing Requirements (Post Nov 2023)

As my account was created after November 13, 2023, I must test my app using the console close test feature. It requires that my app be tested on Android devices (not simulation) by at least 12 testers for 14 days before it can be released to production. My app is now stock close testing until I find 12 testers. As of April 4th, I have 9 testers, finding 12 testers is challenging, as most people in the U.S. use iPhones and iPads instead of Android devices. See Figure 18.

Please see the next page.

Figure 18*App stock in Close Testing*

Note: The figure illustrates my app in the console production step, waiting for 12 testers before running the close test for 14 days.

Production and Maintenance

After the app is closed, tested for 14 days, the Production feature will open, see Figure 17. Then, the production release process involves answering several questions that Google will review for about 7 days (TL;DR Android, 2023). After the app was accepted by Google and published. Google requires that developers maintain their apps by staying compliant Google Play Developer Program Policies (Adnan, 2023). Google also requires that “new apps and app updates MUST target an Android API level within one year of the latest major Android version release. New apps and app updates that fail to meet this requirement will be prevented from app submission in Play Console.” (Google, n.d.). Additionally, developers are required to maintain their app by maintaining the data safety section, providing regular app updates (fixing bugs, adding features), responding to policy violations or security alerts, and keeping developer

account information current. I am planning to follow this maintenance requirement once my app is published. By collecting user feedback, I will fix bugs and add and improve features by providing ongoing maintenance. I also thoroughly test before releasing any new versions of my app, updating or adding features to it. The following is the app publishing overview period of the app being accepted by Google for closed testing.

Figure 19

Publishing Overview

Publishing overview
See an overview of the changes made to your app, and control when changes are sent for review or published. [Show more](#)

Managed publishing [Turn on managed publishing](#)
Managed publishing off
When you send your changes to Google for review, they'll be published automatically as soon as they're approved

Geo-blocking regulation
If you're distributing apps in the EU, see Geo-blocking Regulation (EU) 2018/302
[Learn more](#) [Dismiss](#)

Changes in review [Remove changes](#) [Hide](#)

Your changes are now in review. We may find additional issues when reviewing your app. [Learn more](#)

Item changed	Description	
Closed testing - Alpha		
1 (1/0)	Start full rollout	→
Countries / regions	Add 176 countries / regions: Albania, Algeria, and 174 more	→
Countries / regions	Add rest of world	→
Track status	Resume track	→
Testers	Set testers to be managed by email lists: Alex Ricciardi, Charla Ricciardi, Marianna Ricciardi, Ester Pastore, Claudio Pastore, Kayla Ricciardi, Isaac Delarme, Eryn Ricciardi	→
Default store listing		
English (United States) - en-US	Add language: Provided app name (My Recipe App), and all other required information.	→
App content		
Content Rating	Submit new questionnaire	→
Target audience and content	Update Target audience and content information. Target age is 18 and older.	→
Privacy policy	Set Privacy policy URL to https://1drv.ms/f/c/aa70b8d9fb5f8fc2/EcNHC4xRMv9FupDxeEYj5wB1gWSbzbuGJz-CXH0drPQ7e-ZB5ibB	→
Ads declaration	Update ads declaration	→
Data safety	Complete Data safety questionnaire	→
Store settings		
App category	Select app category (Food & Drink app)	→

What you've told us
Items listed here aren't published, but we'll take them into account when reviewing your app:

- [App content](#): Update App access instructions (All functionality available without special access)
- [App content](#): News apps' declaration updated
- [App content](#): Advertising ID' declaration updated
- [App content](#): Government apps' declaration updated
- [App content](#): Financial features' declaration updated
- [App content](#): Health' declaration updated

Note: The figure showcases the app publishing overview before being accepted by Google for closed testing.

Maintenance, Marketing, and Future Enhancements

As discussed in the previous Production and Maintenance section, I am planning to follow this Google maintenance requirement. Using user feedback from users, crash reports, and ongoing testing, I will fix bugs, add and improve features, and provide ongoing maintenance. I plan to thoroughly review and test all my app's future release versions, including bug fixes, updates, and new features. First in my maintenance agenda is to further optimize my app UI, especially for tablets.

Once my app goes live, I will market my app. This involves App Store Optimization by optimizing app metadata, which includes title, description, and keywords. This improves visibility in app store searches (Colorado State University Global, 2025). I will also use social media platforms to advertise the app and create engagement with users, and promote app features and updates. Additionally, I will utilize in-app marketing by pushing notifications and in-app messages to encourage user retention and engagement. Furthermore, I will encourage users to leave positive reviews and ratings in the app store and on social media, which influences app discoverability and trust with potential new users.

For future app enhancements, I plan to implement user logging and authentication using Firebase authentication. Firebase authentication is a Google service that provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to the app (Firebase, n.d.b). Next, I will implement a feature that will allow the app users to share their recipe online, on a website, with other users if they choose to. The TheMealDB allows patron users to share recipes on their website; however, they do not have API calls that allow users to add recipes or log in to their services through an app. This is a major obstacle to implementing the shared recipe feature. I may have to consider implementing my own NoSQL cloud database

for the app by using Backend-as-a-Service (BaaS) platform such as Cloud Firestore from Google Firebase. I also have a plan to implement a Large Language Model (LLM) through an API that will generate recipes based on provided ingredients while also considering the user-specific dietary needs, like diabetic requirements, food allergies, or vegan preferences.

This ends this section. Deploying my app on the Google Play Store was challenging. From creating a Google Play Store developer account to setting up the app, from app content configuration to preparing the signed AAB release package, it required many steps. Along the way, I encountered technical challenges, such as resolving the application ID conflict, and some administrative hurdles, specifically the recruitment of 12 Android testers for closed testing, which is mandatory for new accounts. As the app awaits completion of this testing, I made plans for its ongoing maintenance following Google's policies and bug fixes. I also made plans for marketing once the app goes live, and for future enhancements to expand the app functionality.

Summary

Testing and deploying "My Recipe App" was a challenge and a very lengthy process. I extensively tested the app functionality. This revealed that the app was robust but needed further UI optimization for tablet devices. Navigating the deployment process was a bit frustrating, mostly as a new Android developer. I encounter a few specific technical and administrative challenges, particularly the closed testing phase, which is required for new developers. Additionally, I formulated strategies for maintenance, marketing, and future enhancements. Overall, this project was extremely rewarding as it allowed me to experience the entire Android app development lifecycle, and it is a great addition to my software engineering portfolio.

References

- Adnan, R. (2023, September 27). *Google Play Developer Program Policies: A guide to avoiding violations and removals*. Medium. <https://medium.com/@rana.adnanali/google-play-developer-program-policies-a-guide-to-avoiding-violations-and-removals-169bb414eab3#:~:text=Google%20Play%20enforces%20its%20policies,even%20permanent%20banishment%20of%20the>
- Akhtar, M. (2015, May 31) *com.example is restricted" when uploading APK to Play Store* [Post Reply]. Stackoverflow. <https://stackoverflow.com/questions/17397195/com-example-is-restricted-when-uploading-apk-to-play-store>
- Colorado State University Global (2025). *Module 8.3: App marketing strategies and post-publishing considerations* [Interactive Lecture]. Canvas. https://csuglobal.instructure.com/courses/105190/pages/8-dot-3-app-marketing-strategies-and-post-publishing-considerations?module_item_id=5453719
- MJSD Coding (2024, September 3). *How to publish an Android app to Google Play | Updated 2024* [Video]. YouTube. <https://www.youtube.com/watch?v=d8uEdeMgikU>
- Firebase (n.d.a). *Run a Robo test (Android)*. Google. <https://firebase.google.com/docs/test-lab/android/robo-ux-test#:~:text=Robo%20test%20is%20a%20testing,methodically%2C%20automatically%20simulating%20user%20activities.>
- Firebase (n.d.b). *Firebase authentication*. Google. <https://firebase.google.com/docs/auth>
- Google (n.d.). *Google play's target API level policy*. Google. <https://support.google.com/googleplay/android->

developer/answer/11917020?hl=en#:~:text=New%20apps%20and%20app%20updates%20that%20fail%20to%20meet%20this,app%20submission%20in%20Play%20Console.

TheMealDB (n.d). *Welcome to TheMealDB*. <https://www.themealdb.com/>

TL;DR Android (2023, December 7). *20 testers? Google Play's new app policy: A Survival masterclass* [Video]. <https://www.youtube.com/watch?v=r7PpboLoamU&t=341s>