WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: Computer Science in English

BACHELOR THESIS

SUPERVISOR:
Lect. Dr. Mafteiu-Scai Liviu Octavian

GRADUATE:
Șova Dumitru Ștefan Andrei

TIMIŞOARA
2022

WEST UNIVERSITY OF TIMIŞOARA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
BACHELOR STUDY PROGRAM: Computer Science in English

Data Manager

SUPERVISOR:
Lect. Dr. Mafteiu-Scai Liviu Octavian

GRADUATE:
Șova Dumitru Ștefan Andrei

TIMIŞOARA
2022

Abstract

The addressed problem is comprised of the storage of an Android phone, the proposed thesis comprises an application that proposes a way for its users to save storage space with minimal input.

The way this app is going to save up on storage space is by looking at specific file types that the user can filter through, as such, the app will ask the user for the required permissions to be able to scan the phone and detect said unnecessary data/ applications. The following options will be available for the user: the ability to delete the cache memory, the option to check if there are currently files/folders in the trash, and an option to empty it if the user wants to or not, the option to delete the leftover data of an application that has been deleted, the option to delete empty folders, archive files, and Android Package (APK) files, and the option to check the list of applications of the user and see the last time the user accessed them, with the ability to go to the Application (app) info and delete/disable them or leave them alone.

Such options represent the more hidden parts of data management, most users being unaware that their storage space is being drained by such files, forgetting about apps that they haven't used in a while, or simply forgetting to empty their trash folders.

Keywords: Android Studio, Kotlin, data, management, cache, cleaning, files, delete, XML, memory.

Rezumat

Problema abordată constă în memoria unui telefon Android, lucrarea propusă cuprinde o aplicație ce propune utilizatorilor săi o modalitate de-a economisi spațiu de stocare cu un input minim.

Modul în care această aplicație va economisi spațiul de stocare este prin a analiza anumite tipuri de fișiere pe care utilizatorul le poate filtra, și, aplicația îi va cere acestuia permisiunile necesare pentru a putea scana telefonul și pentru a putea detecta respectivele fișiere inutile. Următoarele opțiuni vor fi disponibile pentru utilizatori: posibilitatea de a șterge memoria cache, opțiunea de a verifica dacă există în prezent fișiere/foldere în coșul de gunoi și o opțiune de-al goli dacă utilizatorul dorește sau nu, opțiunea de a șterge datele rămase ale unei aplicați care a fost dezinstalată, opțiunea de a șterge folderele goale, fișierele de arhivă și fișierele APK și opțiunea de a verifica lista de aplicații ale utilizatorului și de a vedea ultima dată când acesta le-a accesat, cu posibilitatea de a accesa informațiile app și de a le șterge/dezactiva sau de a le lăsa în pace.

Astfel de opțiuni reprezintă părțile mai ascunse ale gestionări datelor, majoritatea utilizatorilor neștiind că spațiul lor de stocare este epuizat de astfel de fișiere, uitând de aplicațiile pe care nu le-au folosit de ceva vreme sau pur și simplu uitând să își golească dosarul de gunoi.

Cuvinte cheie: Android Studio, Kotlin, date, gestionare, cache, curățare, fișiere, ștergere, XML, memorie.

Contents

List of Figures

List of acronyms

app	Application	3
MB	Megabytes	9
IDE	Integrated Development Environment	19
NDK	Android Native Development Kit	19
p	pixels	9
API	Application programming interface	19
JSON	JavaScript Object Notation	20
AI	Artificial Inteligence	40
4K	3840 x 2160 resolution	9

Chapter 1

Introduction

DataManager is designed with the users' phones' data management in the foreground, focusing on a simple interface that doesn't require much from the user, with options for emptying their trash, cleaning unneeded data, and being able to see what applications they have not used regularly, each done with one press of a button only.

The app offers different menus for each of its use cases, each with an easy-to-use and easy-to-understand layout. For further convenience, the app also offers the ability to enter a general settings menu where the user can change the overall theme of the app, from a light theme to a dark one, and to change the language from English to Romanian.

The three main screens of the app consist of the cache cleaning screen, the unused applications screen, and the main menu screen. The main menu gives access to the other screens, the option to empty the trash folder, and the general settings screen, where the user can change certain preferences about the application. The other two screens are for data management, the cache screen allows the user to select filters for what he wants to delete, and the unused applications screen lists the applications of the user and the last time they were used, with the option to delete them if he wished to.

For accessing the storage, internet, and so on, the application will ask the user for the required permissions. All permissions are described in a special menu found in the main menu that tells the user exactly what are they used for.

1.1 Motivation

Owning a phone has become in the last decade a basic human need for all of us, regardless of age, culture, or background. The phone helps connect us with one another, not just by making phone calls or writing messages to one another, but modern phones also allow us to use the internet, share our stories, photos, life experiences, and so much more. Besides connecting with one another, it gives us countless possibilities, from entertaining us to educating us and so on.

Free up storage space by deleting unused data such as rarely used apps and unnecessary documents.

Figure 1.1: The common sight of an emptying storage

Owning to this fact, one of the most important factors of our phones, that we usually do not take into consideration or notice too late is the well-being of the phone, usually being represented by the neglect of our storage spaces. It is easy to lose track of our storage and find ourselves in the awkward situation of needing to take a picture, download a file for work, or just install the hottest new app, and being unable to do so because we are hit with the surprise of no more storage space left. Not only that, but one aspect of the phone that everybody seems to not pay attention to is the well-being of the phone, many forget to delete leftover data from uninstalled applications, empty the trash folder, and so on, little things that in time add up to a worse functioning phone and a smaller storage space.

Such unneeded files like caches, logs, APK files, and so on, are the type of files that the user will never think about, and especially not think to clean up to save on his or her phone memory.

1.2 Problem Statement

This problem becomes more and more prevalent with the size of newer applications, the increase in resolution for photos and videos, and the amount of data that remains from them. A simple 1080pixels (p) photo is around 6 Megabytes (MB), while a phone with 3840 x 2160 resolution (4K) capabilities is around 24 MB, which is an increase of 4 times. Things are even worse when looking at video memory, a 1080p 1-minute video has around 100 MB while a 4K one has around 450 MB.

With this in mind, every little amount of memory that the user can save up is usually more than welcomed, but unfortunately, phones do not come with built-in memory managers from the manufacturers.

As such, the problem of how to save our storage space for our phones usually remains in the hands of the consumer, usually by downloading a third-party application that gets the job done.

1.3 Thesis Scope and Objectives

Possible solutions to such problems include the very functions of the proposed application, with how much we tend to use our phones in everyday life, too little time is spent taking care of them and managing their storage in a healthy manner.

The improvements will have certain liberties that the user will have to give his consent to, such as the functionality to access the external storage and usage for the application, however, such improvements will prove fruitful to those who want an increase in storage space, without the need for searching through every application for signs of residual data.

1.4 Thesis Structure

The thesis will have a format consisting of 5 main chapters, each with various sections, each chapter looking at a different aspect of the thesis. The 5 chapters are as follows:

• The first chapter focuses on the what and the why of the project, sharing the motivation behind the project and a general overview behind it;

• The second chapter looks at the application architecture, its main use cases, its main flow, and the expected app requirements;

• Chapter 3 focuses on similar applications that are in the same genre as this thesis, looking at their varied utilities, and functions, and concluding at the end of the chapter with a comparison between them;

• Chapter 4 looks at the implementation that was used for creating the application, and the methods and classes that were used;

• The last chapter is the user manual for the application, guiding the user through each of the main screens, what they do, what permissions they require, and what setting options the user has access to;

After these chapters, the app will also have a chapter for conclusions and discussions based on the results, as well as a bibliography for users' reference.

Chapter 2

Related Work

2.1 Similar Applications

1. AVG Cleaner

Developer: AVG Mobile

Description: AVG is a mobile cleaning tool designed to delete junk files from the phones' storage.

Characteristics:

• **App analyzer**: The application will identify applications that could be causing mobile data to drain faster;

• **App remover**: The app can has options for app removal;

• **Junk cleaner**: The application can delete junk and leftover files;

Data security:

• This application may share data with third parties;

• This application may collect these types of data (Location, Personal info and 3 others);

• Data is encrypted in transit;

• Data can be deleted.

2. Phone Cleaner - Cache Clean

Developer: Games Tree

Description: Phone cleaner is a cleaning app for an Android device and has the following characteristics: file cleaning capabilities, it can cool the CPU, a built-in application manager, and battery-saving functions.

Characteristics:

• **Cache Cleaner**: The app can clean apps or residual data, such as cache memory;

• **Storage Cleaner**: This function scans the phone for data to delete and free up space so it can increase the phone's performance;

• **CPU Cooling**: The app can cool the CPUs temperature and improve its performance;

• **Battery saving actions**: Another function of the app is the disabling of power-consuming apps;

• **Notification cleaning functions**: The last function of the app is its notification cleaning capabilities;

Data security:

• No data was collected;

• Data will be encrypted during transit;

• The user can request the data to be deleted;

• No data is shared with third parties.

3. CCleaner – Phone Cleaner

Developer: Piriform

Description: CCleaner helps you remove junk messages from your Android phone. This Android cleaning app clears your browser history, app cache, and clipboard contents. This Android device cleaner also helps to boot faster and provide better performance.

Characteristics:

• The app can clean cache memory, browser memory, and much more;

• Storage cleaning: The app can also analyze storage space and delete unwanted and residual data to boost performance;

• Application hibernation function, which allows the app to keep apps closed until opened manually;

• Ram boosting functions;

• Increases performance and battery life;

• Analyzes the impact of applications;

• Optimizes the storage of photos;

• Monitors the system.

Data security:

• The application may send different types of data to third parties;

• The application may collect the following types of data (Location, Personal Information);

• Data will be encrypted during the transmission period;

• The data can be deleted at request.

4. 1TapCleaner

Developer: Sam Lu

Description: 1-Tab Cleaner has the ability to clear caches, search histories, and defaults.

Characteristics:

• The app can clean the cache memory;

• Another characteristic is the creation of a list of default apps and the clearing of their selected defaults;

• The application can also clear history data;

• The app can also notify the user if any app uses to much space with its cache memory.

Data security:

• This application doesn't share data with third parties;

• This application may collect these types of data (Location, app info);

• Data is encrypted in transit;

• Data can't be deleted.

5. Droid Optimizer

Developer: Ashampoo®

Description: Droid Optimizer is an application that aims to monitor the phone's battery, notify the user of the remaining battery time, optimize the phone and automatically perform cleaning tasks.

Characteristics:

• The application can perform cleaning tasks automatically;

• The application can also find and delete unwanted files;

• Clears system and application cache;

• The application employs different methods for saving energy and improving battery life;

• The application can disable WI-FI at predetermined times or every time the screen is turned off;

• Manages applications;

• The app has the option to expose apps that spy on their user and that have critical permissions.

Data security:

• No data is shared with third parties;

• This application may collect these types of data: Location, application activity, and application information and performance;

• Data is encrypted in transit;

• Data cannot be deleted.

2.2 Comparison

Each application presented has some specific characteristics, and although most of them have similar features, each one brings something different:

1. Droid Optimizer automatically turns off the user's mobile data and lets the user take a look at critical app permissions, exposing spy apps;

2. 1TapCleaner allows the user to delete search histories and defaults and can notify the user if any app uses too much space for its cache memory;

3. CCleaner optimizes photo storage and presents the Task Killer functionality;

4. Phone Cleaner presents the Notification Cleaner functionality that deactivates and deletes unwanted notifications, if necessary;

5. AVG Cleaner has the ability to clean applications that drain mobile data.

One drawback of these applications is the frequent use of free trials, ads, and such that either ruin the user experience or lock the user behind a paywall.

From the point of view of data security, Droid Optimizer, 1TapCleaner, and Phone Cleaner do not share data with third parties while CCleaner and AVG Cleaner share data, and from the point of view of data collection, Battery Guru and Phone Cleaner do not collect, while the rest of the applications collect data.

For the purpose of creating an application that offers a similar function as the ones listed above, the app should not be a one-to-one copy of another, as such, we need to look at these sorts of comparisons, to better understand what could be improved upon and added. For example, the proposed app of this paper looks at different types of files to delete, checks up on apps by their last time being used, and checks the trash folder for potential residual files. In the following page, we will look at a comparison table that looks at what each application offers, providing a clear view of each functionality listed above and which app either has such an option or not.

Comparison	**AVG Cleaner**	**Phone cleaner**	**CCleaner**	**1TapCleaner**	**Droid Optimizer**
Cache cleaning options	NO	YES	YES	YES	YES
Disable WI-FI	NO	NO	NO	NO	YES
Notify the user if any app has cache memory that consumes too much memory	NO	NO	NO	YES	NO
Analyze application impact	YES	NO	YES	NO	NO
RAM Boosting options	NO	NO	YES	NO	NO
System monitoring	NO	NO	YES	NO	NO
Clear Defaults options	NO	NO	NO	YES	NO
Clear history options	NO	YES	NO	YES	NO
CPU cooling options	NO	YES	NO	NO	NO
Application Manager	NO	NO	NO	NO	YES
Managing critical permissions	NO	NO	NO	NO	YES
Data collection	YES	NO	YES	NO	YES
Optimizing the storage of photos	NO	NO	YES	NO	NO
Memory space recovery	YES	YES	YES	YES	YES
Notification cleaner	NO	YES	NO	NO	NO
Sharing with third parties	YES	NO	YES	NO	NO
Encrypted data in transit	YES	YES	YES	YES	YES
Data may be deleted	NO	YES	YES	NO	NO

Chapter 3

App Arhitecture

3.1 Use Cases Diagram and Sequence Diagram

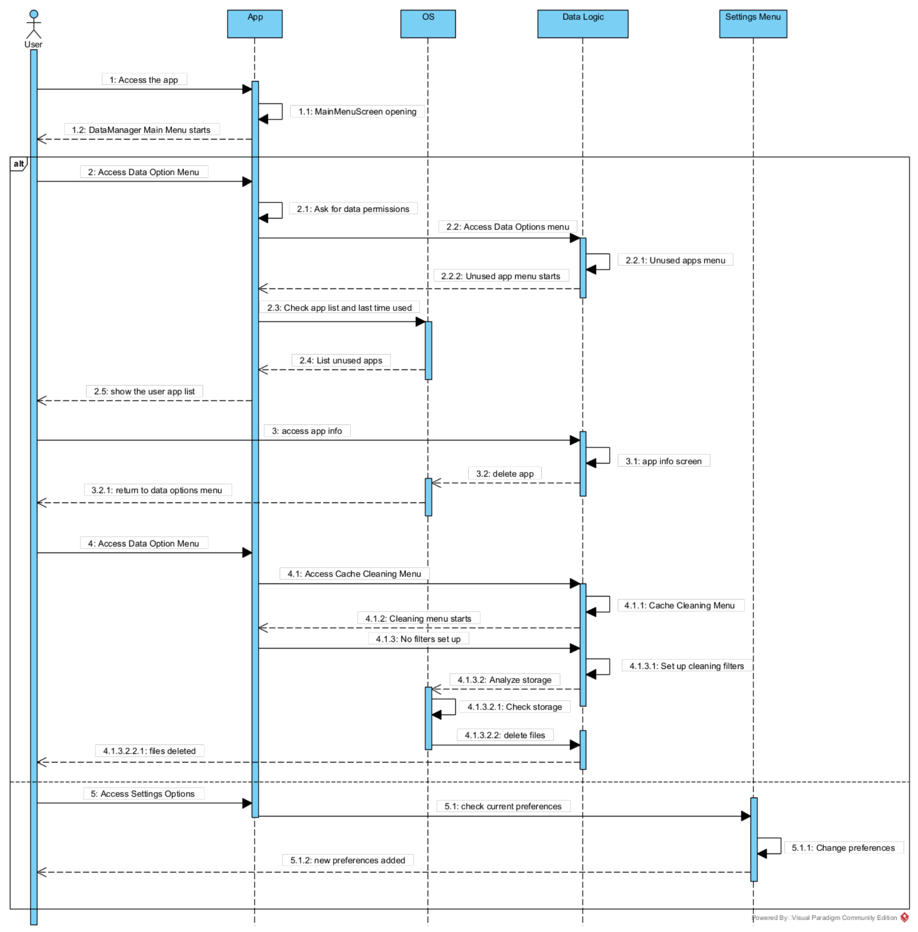

User
App
OS
Data Logic
Settings Menu
1: Access the app
1.1: MainMenuScreen opening
1.2: DataManager Main Menu starts
alt
2: Access Data Option Menu
2.1: Ask for data permissions
2.2: Access Data Options menu
2.2.1: Unused apps menu
2.2.2: Unused app menu starts
2.3: Check app list and last time used
2.4: List unused apps
2.5: show the user app list
3: access app info
3.1: app info screen
3.2: delete app
3.2.1: return to data options menu
4: Access Data Option Menu
4.1: Access Cache Cleaning Menu
4.1.1: Cache Cleaning Menu
4.1.2: Cleaning menu starts
4.1.3: No filters set up
4.1.3.1: Set up cleaning filters
4.1.3.2: Analyze storage
4.1.3.2.1: Check storage
4.1.3.2.2: delete files
4.1.3.2.2.1: files deleted
5: Access Settings Options
5.1: check current preferences
5.1.1: Change preferences
5.1.2: new preferences added
Powered By: Visual Paradigm Community Edition

Figure 3.1: Basic Flow

The main flow that the application will follow. It starts with the main menu screen, where the user can either access the data option menu or the settings menu.

The 2 screens have the following functionalities: the settings menu will offer the option to change the theme of the application, change the language of the text inside the application, and read, in a new menu, some information about it, while the data options menu has the cache cleaning scree, where the user can delete unneeded files by setting up filters for them, and the unused applications list, where the user can find what applications he doesn't usually use.

The diagram has one actor, the user, and 4 lifelines, the Operating System (OS), data options menu, settings menu, and the app itself.

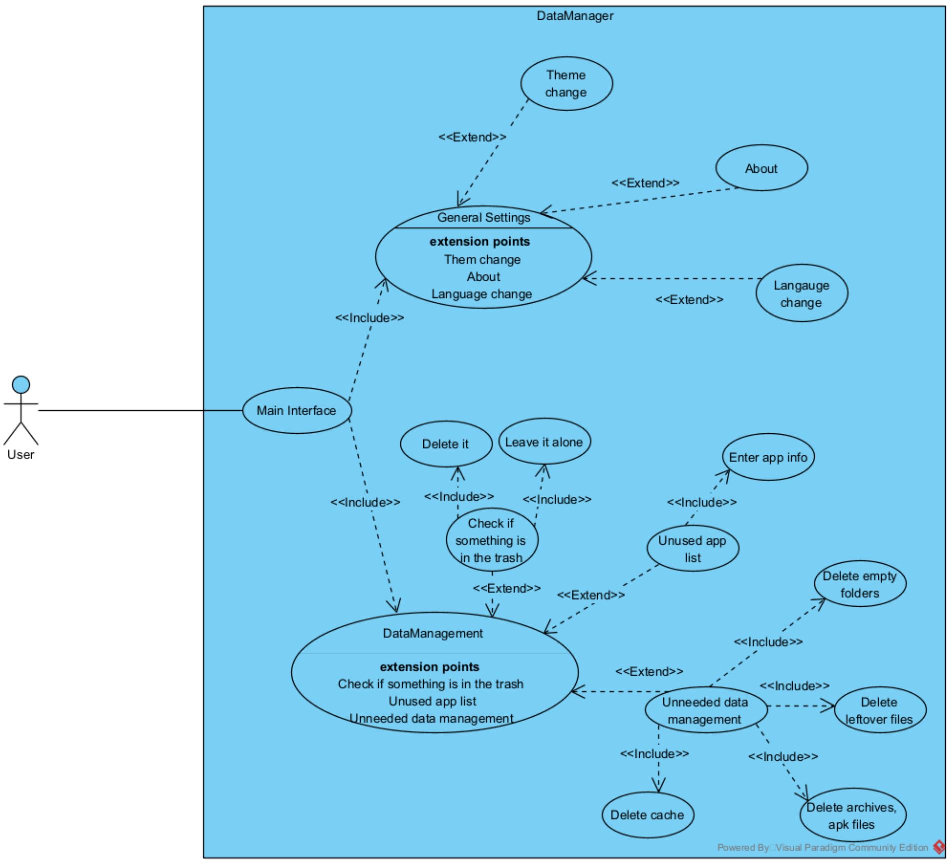


Figure 3.2: Use Cases Diagram

The above diagram shows the use cases of the application, what the user has access to, and the decision he can make with the options provided.

The 3 main functionalities that the app offers are as follows:

• The option to delete APK files, cache files, archived files, leftover files, and empty folders from the storage of the phone, each type of file having a filter slider that can be turned on or off;

• Check whether or not there are files in the trash can and the option for the automatic deletion of what's in it;

• A general setting area where the user can change the theme and language of the app, while also being able to access an about menu for information about the application;

• The application also has a percentage estimate for the remaining battery life and remaining storage space

3.2 App Architecture

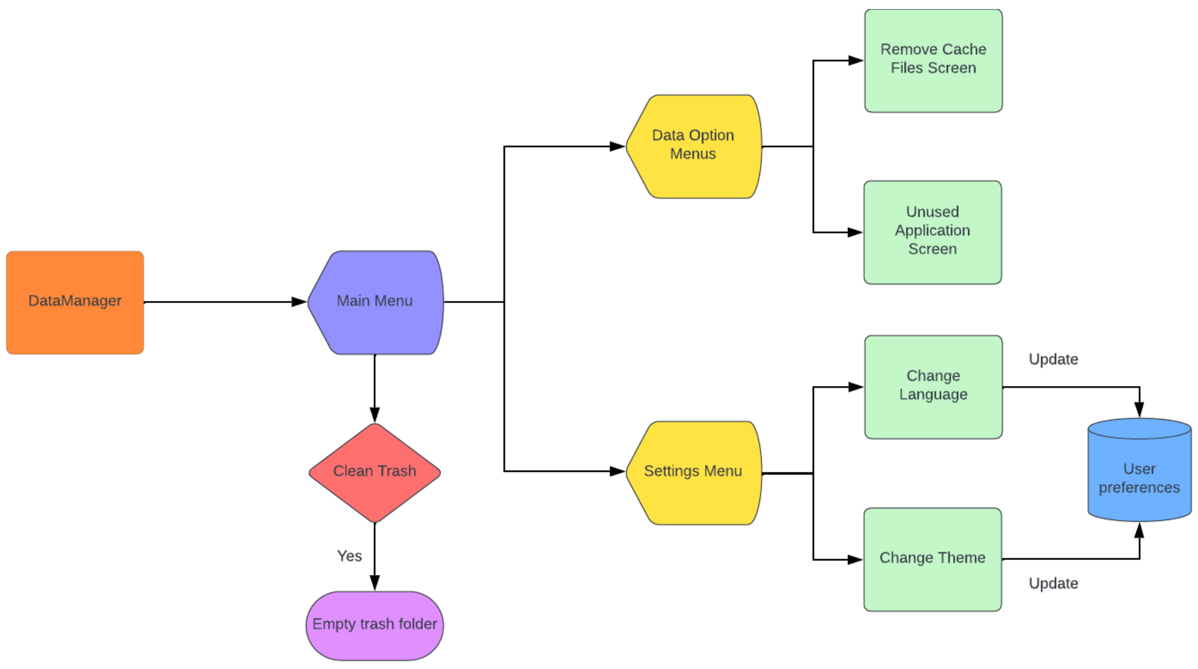


Figure 3.3: App Arhitecture

This figure presents how the app will function, when it starts, it will display a general User Interface (UI) interface where the user can go between the data options and settings menu. The data options menus can delete unneeded data and scan the phone for a list of all applications and the last time they were used.

The other functionalities that the app offers, are the settings menu where the user can set up a different or a different language, and look at an about menu to learn more about the app. These preferences will be saved in a database for the convenience of the user so as to not set them each time they exit and enter back on the application.

The last functionality is the one for emptying the trash folder, a function that when clicked, the user will be alerted if he still wants to go through with this option or not.

Chapter 5 will delve deeper into each screen and how should it be used, while Chapter 4 will focus on the implementation of the application.

Chapter 4

Implementation Logic

4.1 System Requirments & Technologies Used

For the implementation of the proposed app, the development medium chosen was Android Studio, while the programming language used was Kotlin. Details on all technologies used are as follows:

Integrated Development Environment (IDE) used:
Android Studio [1] is an official IDE for Android app development, based on the tools offered by the IntelliJ IDEA, which is designed with powerful features for increased productivity in app development. The features on offer by Android Studio are as follows: a Gradle-based build system, an emulator for testing and experimenting on the apps developed, offering for the user the ability to test his or her application on different Application programming interface (API) levels, and personalize their emulator settings for varied testing environments, the ability to develop applications for all Android devices, the ability to edit and update composables in real-time while the emulator or physical device is running. The IDE also offers different code templates, testing tools, and frameworks for an increase in productivity and development time, while also including a GitHub integration and support for C++ and Android Native Development Kit (NDK).

Programming Language used:
Kotlin [5] is a cross-platform, general-purpose programming language developed and released in 2011 by JetBrains, being built as a statically typed language, meaning that the variable type is going to be known at compile time. One key feature of Kotlin is in its design policy, the language being developed to be fully interoperable with Java, making the use of Java and Kotlin, in the same project, possible. With it being a concise, and safe programming language, it is usually recommended the use of Kotlin in applications development, being more often than not the best choice for such projects. The main reasons why Kotlin is a good choice for development stems from its many features that allow its users to write code in an easier manner, such as null safety, coroutines used for asynchronous programming, and extension functions.

Database technology used:

Firebase [2] is an app development platform that offers a variety of tools and services to help in the development of high-quality applications.

One of the key features of Firebase is the Firebase Realtime Database [3], which is a cloud-hosted database. The data is stored in a JavaScript Object Notation (JSON) file, letting the user store and sync data in real-time, for every connected client. Such features help when building cross-platform applications, with all clients sharing one Realtime Database instance and automatically receiving updates with the newest data.

Other technology used:

Other technologies used include the Jetpack Compose toolkit, used for the UI layout, and the use of XML files which were used for certain screen layouts, strings, color pallets, permission, component definition, various graphical elements, and so on.

Jetpack Compose [4] is a toolkit designed for building Android UI. Its main benefits consist of simplifying and accelerating UI development on Android by allowing by providing the user with a set of powerful tools, and intuitive Kotlin APIs, being fully declarative, which means that the UI elements can just be described and Compose will take care of the rest. Another big benefit of Compose is that the UI automatically updates.

XML files are used in Android Studio and app development for several purposes, such as:

• Layout: This type of XML file is used to define the general layout of the user interface, holding all elements such as text views, buttons, sliders, and others;
• Strings: This file is used as an alternative for hard-coded strings. They define strings that can be accessed throughout the entire app, being especially useful for changing the text of an app from one language to another;
• Colors: This XML file is used to define different colors that the user can use throughout the application, this file is especially useful when changing the theme of an app;
• Drawables: These are XML files that are used to create various graphic icons or images;
• Manifest: Lastly, these files are used to define the components of the application, including the packages, activities, receivers, and permissions that the application needs;

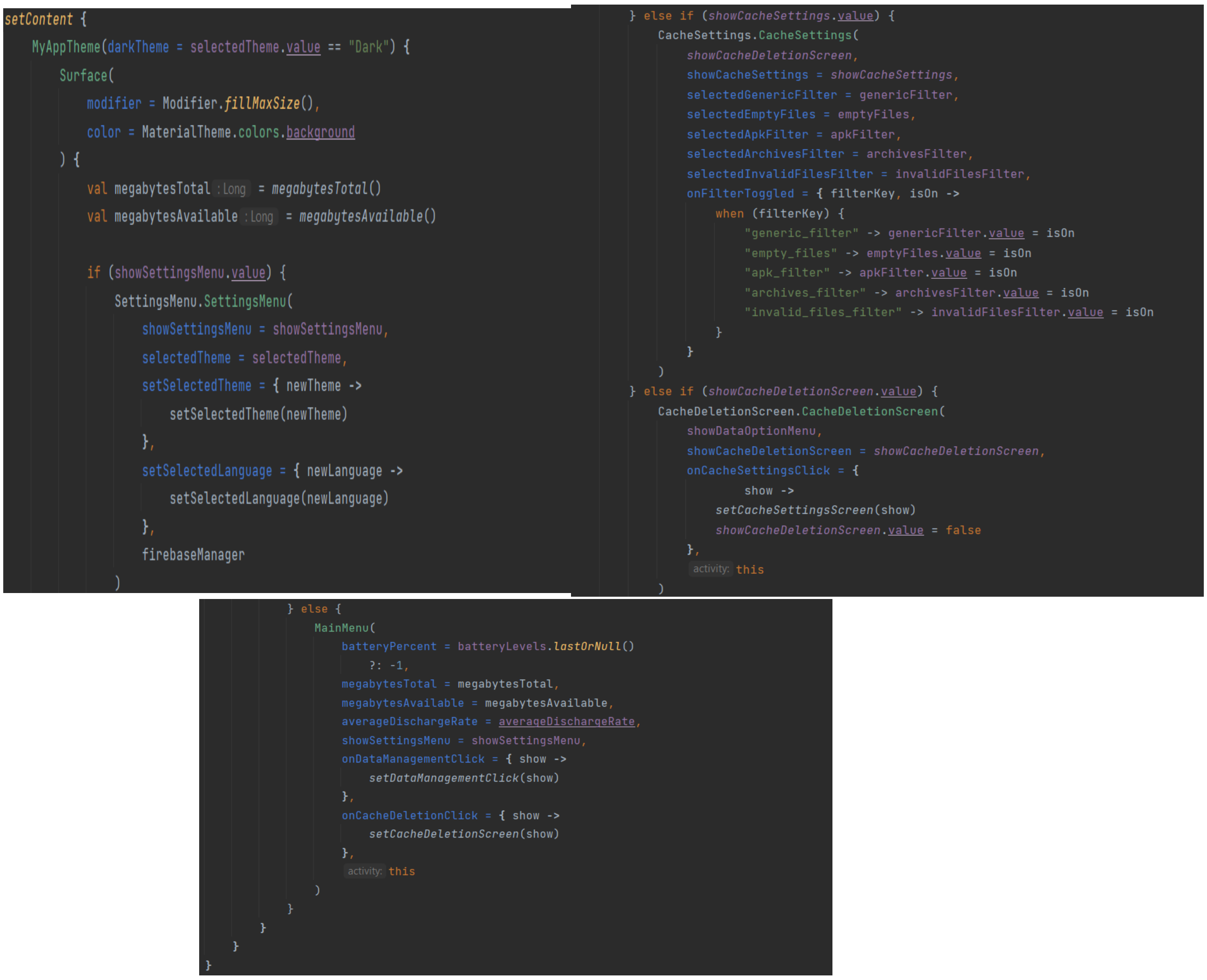
For testing, the app was experimented on the emulators provided by Android Studio, on an emulator with API 30 and 33, and the app was also tested on different physical devices, mainly a Samsung J6, and a Samsung A 54.

The basic system requirements for running the app are as follows:

• An Android phone with the Android Version 11+;
• Connection to the Internet;

4.2 Implementation

4.2.1 Main Menu Implementation



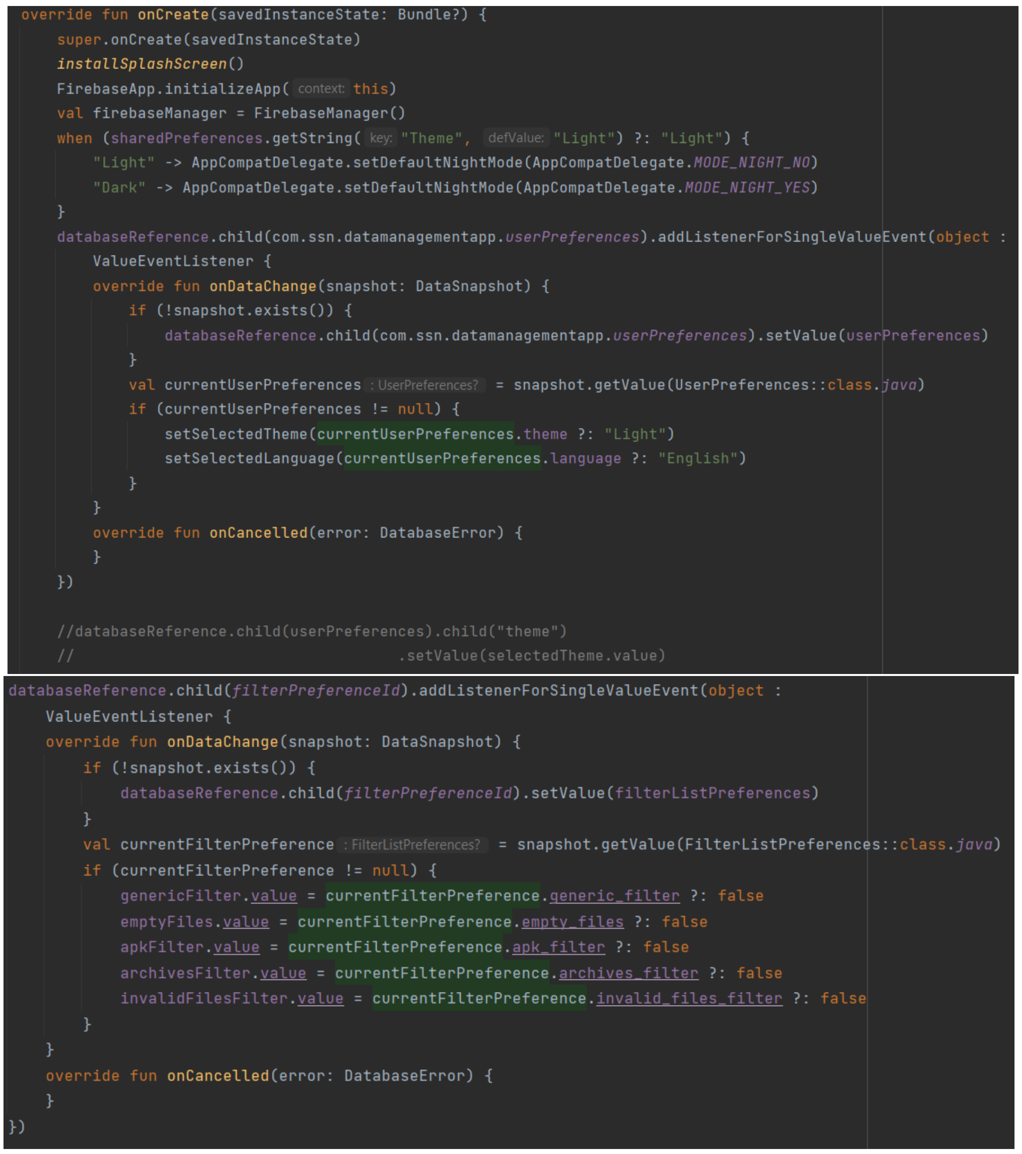
```
setContent {
    MyAppTheme(darkTheme = selectedTheme.value == "Dark") {
        Surface(
            modifier = Modifier.fillMaxSize(),
            color = MaterialTheme.colors.background
        ) {
            val megabytesTotal :Long = megabytesTotal()
            val megabytesAvailable :Long = megabytesAvailable()

            if (showSettingsMenu.value) {
                SettingsMenu.SettingsMenu(
                    showSettingsMenu = showSettingsMenu,
                    selectedTheme = selectedTheme,
                    setSelectedTheme = { newTheme ->
                        setSelectedTheme(newTheme)
                    },
                    setSelectedLanguage = { newLanguage ->
                        setSelectedLanguage(newLanguage)
                    },
                    firebaseManager
                )
            } else if (showCacheSettings.value) {
                CacheSettings.CacheSettings(
                    showCacheDeletionScreen,
                    showCacheSettings = showCacheSettings,
                    selectedGenericFilter = genericFilter,
                    selectedEmptyFiles = emptyFiles,
                    selectedApkFilter = apkFilter,
                    selectedArchivesFilter = archivesFilter,
                    selectedInvalidFilesFilter = invalidFilesFilter,
                    onFilterToggled = { filterKey, isOn ->
                        when (filterKey) {
                            "generic_filter" -> genericFilter.value = isOn
                            "empty_files" -> emptyFiles.value = isOn
                            "apk_filter" -> apkFilter.value = isOn
                            "archives_filter" -> archivesFilter.value = isOn
                            "invalid_files_filter" -> invalidFilesFilter.value = isOn
                        }
                    }
                )
            } else if (showCacheDeletionScreen.value) {
                CacheDeletionScreen.CacheDeletionScreen(
                    showDataOptionMenu,
                    showCacheDeletionScreen = showCacheDeletionScreen,
                    onCacheSettingsClick = {
                            show ->
                        setCacheSettingsScreen(show)
                        showCacheDeletionScreen.value = false
                    },
                    activity: this
                )
            } else {
                MainMenu(
                    batteryPercent = batteryLevels.lastOrNull()
                        ?: -1,
                    megabytesTotal = megabytesTotal,
                    megabytesAvailable = megabytesAvailable,
                    averageDischargeRate = averageDischargeRate,
                    showSettingsMenu = showSettingsMenu,
                    onDataManagementClick = { show ->
                        setDataManagementClick(show)
                    },
                    onCacheDeletionClick = { show ->
                        setCacheDeletionScreen(show)
                    },
                    activity: this
                )
            }
        }
    }
}
```

Figure 4.1: Set Content

Figure 4.1 shows the logic behind screen hopping, the app is actually composed of only two activities, the main activity and the permission activity, the content of the main activity being created through the setContent() method.

As such, the main activity sets up screens and values for whether these screens are shown or not. The main screen is the main menu, where the user accesses the other screens. Each screen is built with a composable, with the bulk of the UI elements being comprised of Material3 elements.



```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    installSplashScreen()
    FirebaseApp.initializeApp( context: this)
    val firebaseManager = FirebaseManager()
    when (sharedPreferences.getString( key: "Theme",  defValue: "Light") ?: "Light") {
        "Light" -> AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_NO)
        "Dark" -> AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_YES)
    }
    databaseReference.child(com.ssn.datamanagementapp.userPreferences).addListenerForSingleValueEvent(object :
        ValueEventListener {
        override fun onDataChange(snapshot: DataSnapshot) {
            if (!snapshot.exists()) {
                databaseReference.child(com.ssn.datamanagementapp.userPreferences).setValue(userPreferences)
            }
            val currentUserPreferences : UserPreferences?  = snapshot.getValue(UserPreferences::class.java)
            if (currentUserPreferences != null) {
                setSelectedTheme(currentUserPreferences.theme ?: "Light")
                setSelectedLanguage(currentUserPreferences.language ?: "English")
            }
        }
        override fun onCancelled(error: DatabaseError) {
        }
    })

    //databaseReference.child(userPreferences).child("theme")
    //                                      .setValue(selectedTheme.value)
```

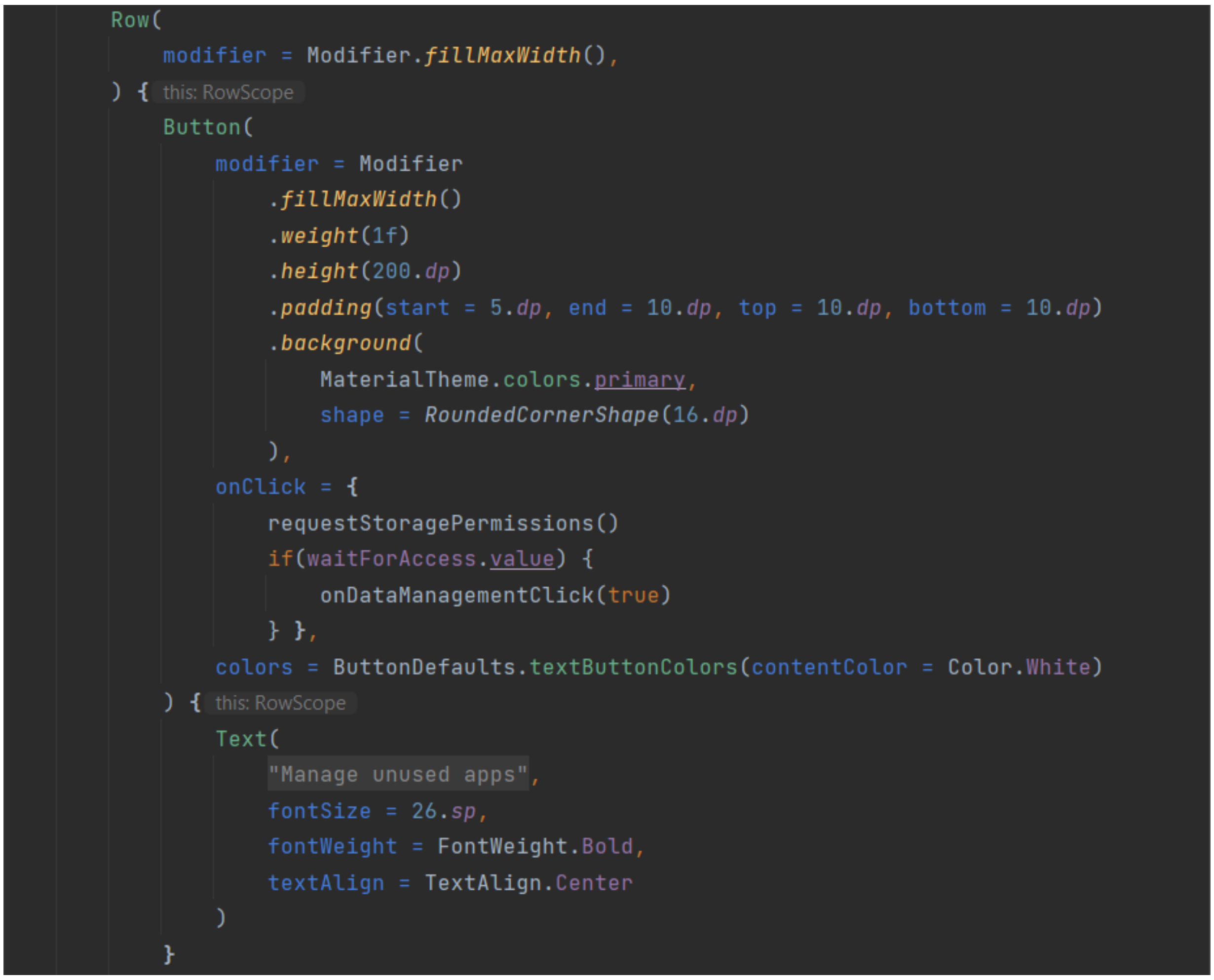
```
databaseReference.child(filterPreferenceId).addListenerForSingleValueEvent(object :
    ValueEventListener {
    override fun onDataChange(snapshot: DataSnapshot) {
        if (!snapshot.exists()) {
            databaseReference.child(filterPreferenceId).setValue(filterListPreferences)
        }
        val currentFilterPreference : FilterListPreferences?  = snapshot.getValue(FilterListPreferences::class.java)
        if (currentFilterPreference != null) {
            genericFilter.value = currentFilterPreference.generic_filter ?: false
            emptyFiles.value = currentFilterPreference.empty_files ?: false
            apkFilter.value = currentFilterPreference.apk_filter ?: false
            archivesFilter.value = currentFilterPreference.archives_filter ?: false
            invalidFilesFilter.value = currentFilterPreference.invalid_files_filter ?: false
        }
    }
    override fun onCancelled(error: DatabaseError) {
    }
})
```

Figure 4.2: Firebase fetch & update data

In the above figure, 4.2, the onCreate function is called, which is a function usually called when first entering the application. It is here the basic logic of the application takes place, like the Firebase setup and preference assignation.

After which, we can see how the FirebaseApp is initialized, how the app selects the theme when the application is started, and how the data for the user preferences is being assigned at the beginning of the app, the values being assigned from the databaseReference value. At the beginning of the onDataChange method, if the collection userPreferences doesn't exist, if so, the app creates it. Then, the app assigns the current value for each preference value and updates the UI accordingly.

Below it, the same operation is taking place for the preferences set by the user for the file filters that will be used for the cache deletion screen.



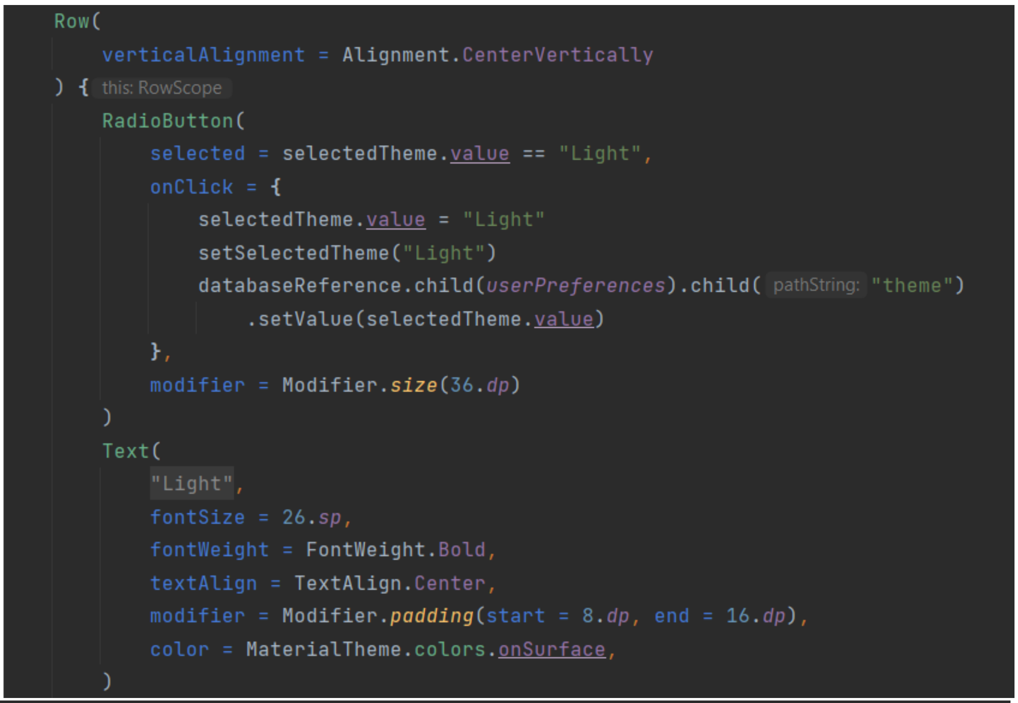
```
Row(
    modifier = Modifier.fillMaxWidth(),
) { this: RowScope
    Button(
        modifier = Modifier
            .fillMaxWidth()
            .weight(1f)
            .height(200.dp)
            .padding(start = 5.dp, end = 10.dp, top = 10.dp, bottom = 10.dp)
            .background(
                MaterialTheme.colors.primary,
                shape = RoundedCornerShape(16.dp)
            ),
        onClick = {
            requestStoragePermissions()
            if(waitForAccess.value) {
                onDataManagementClick(true)
            } },
        colors = ButtonDefaults.textButtonColors(contentColor = Color.White)
    ) { this: RowScope
        Text(
            "Manage unused apps",
            fontSize = 26.sp,
            fontWeight = FontWeight.Bold,
            textAlign = TextAlign.Center
        )
    }
```

Figure 4.3: Button Implementation Example

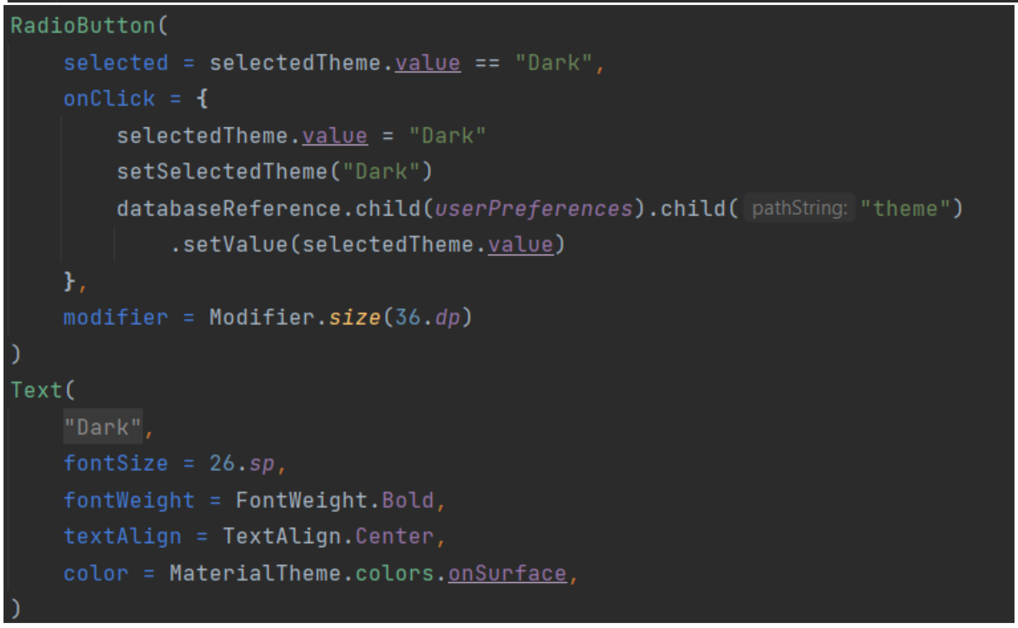
Here, figure 4.3, shows an example of the implementation of a button for menu hopping between the screens of the application. Most buttons throughout the app have a similar structure, utilizing Material3 elements, and the logic set up in the Main Activity for going through different menus.

For better alignment, a row composable is added, with a modifier to fill the maximum width of the screen. The button is then created, utilizing a button compose of the following variables: modifier, which is used for the shape of the button and the background color of it, onClick, which is used for managing the action that takes place when the button is clicked, and colors, which is used to color the text composable inside the button. The text composable has set up custom text alignment, custom font size, and weight, and for the text itself, the app uses the following method: context.getString(R.string.textdatamanagement), context is used as an instance for the current activity that is running, while R.string is used by the app to communicate with the file strings.xml, which holds various strings for ease of use and ease of potentially changing the application's language.

4.2.2 Settings Menu Implementation



```
Row(
    verticalAlignment = Alignment.CenterVertically
) { this: RowScope
    RadioButton(
        selected = selectedTheme.value == "Light",
        onClick = {
            selectedTheme.value = "Light"
            setSelectedTheme("Light")
            databaseReference.child(userPreferences).child( pathString: "theme")
                .setValue(selectedTheme.value)
        },
        modifier = Modifier.size(36.dp)
    )
    Text(
        "Light",
        fontSize = 26.sp,
        fontWeight = FontWeight.Bold,
        textAlign = TextAlign.Center,
        modifier = Modifier.padding(start = 8.dp, end = 16.dp),
        color = MaterialTheme.colors.onSurface,
    )
```



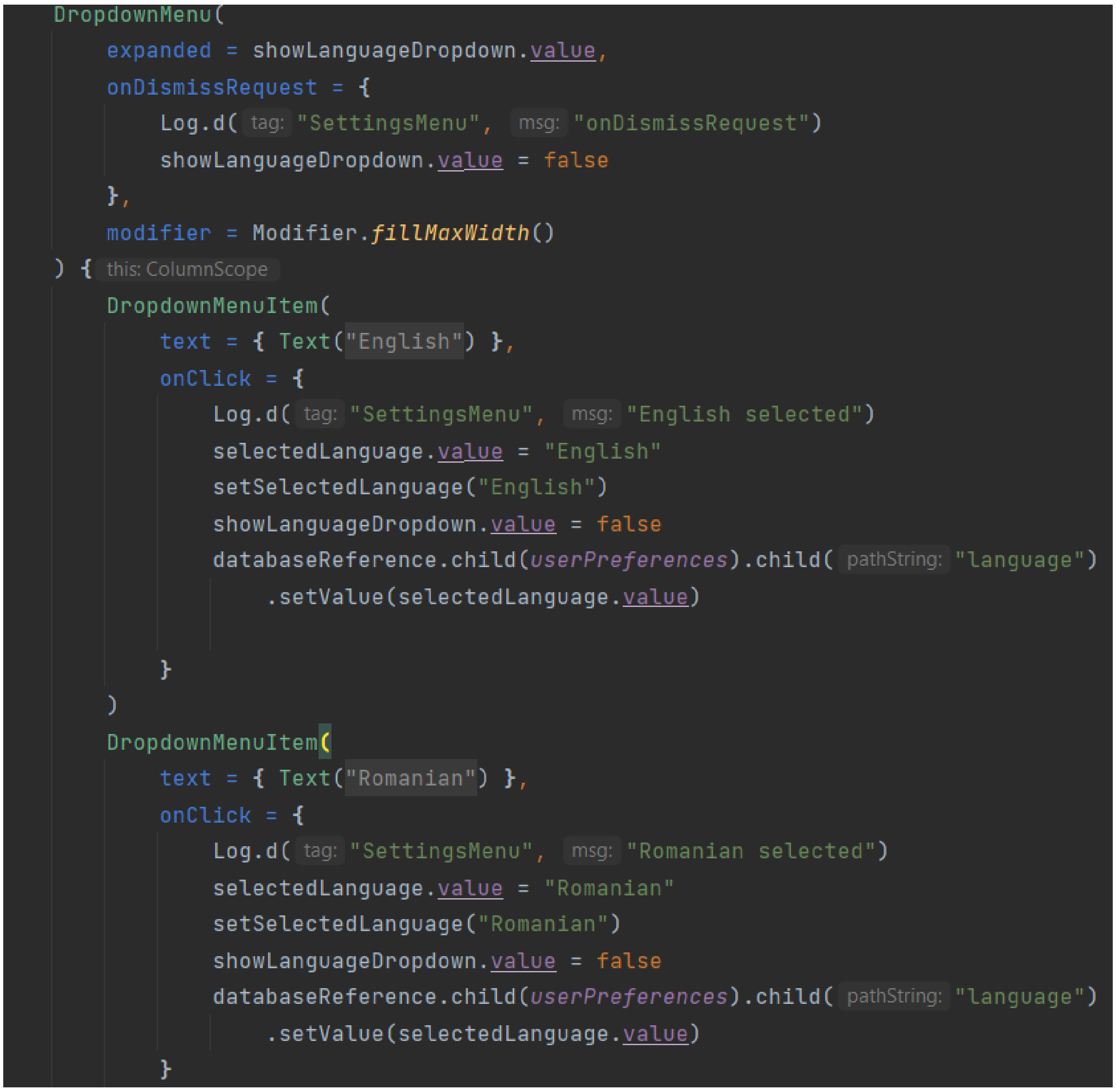
```
RadioButton(
    selected = selectedTheme.value == "Dark",
    onClick = {
        selectedTheme.value = "Dark"
        setSelectedTheme("Dark")
        databaseReference.child(userPreferences).child( pathString: "theme")
            .setValue(selectedTheme.value)
    },
    modifier = Modifier.size(36.dp)
)
Text(
    "Dark",
    fontSize = 26.sp,
    fontWeight = FontWeight.Bold,
    textAlign = TextAlign.Center,
    color = MaterialTheme.colors.onSurface,
)
```

Figure 4.4: Theme Options

Figure 4.4 showcases the implementation of theme options. For that, a set of radio buttons was preferred, the first radio button being used for the light theme, while the second being used for the dark theme. After clicking on one of the radio buttons, the other button is closed, and the changes are updated as well in the Firebase Database, where the theme value is updated accordingly.

SelectedTheme is used as a way to check which radio button should be checked, while setSelectedTheme is used to set for the whole app the new theme is chosen.

As mentioned in the explanation of Figure 4.3, the way these elements are created consists of composable functions, each having the benefit of being easy to use.



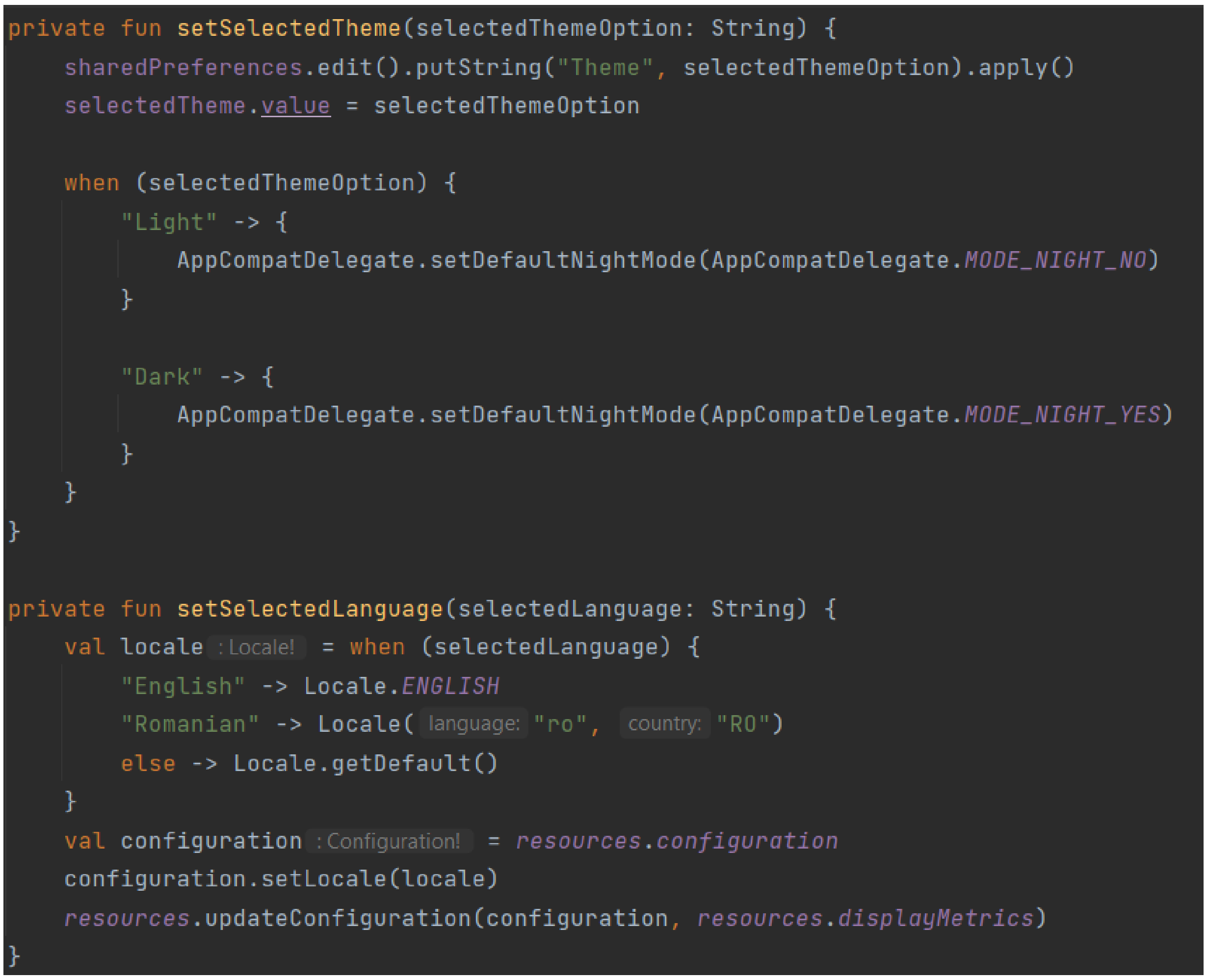
```
DropdownMenu(
    expanded = showLanguageDropdown.value,
    onDismissRequest = {
        Log.d( tag: "SettingsMenu",  msg: "onDismissRequest")
        showLanguageDropdown.value = false
    },
    modifier = Modifier.fillMaxWidth()
) { this: ColumnScope
    DropdownMenuItem(
        text = { Text("English") },
        onClick = {
            Log.d( tag: "SettingsMenu",  msg: "English selected")
            selectedLanguage.value = "English"
            setSelectedLanguage("English")
            showLanguageDropdown.value = false
            databaseReference.child(userPreferences).child( pathString: "language")
                .setValue(selectedLanguage.value)

        }
    )
    DropdownMenuItem(
        text = { Text("Romanian") },
        onClick = {
            Log.d( tag: "SettingsMenu",  msg: "Romanian selected")
            selectedLanguage.value = "Romanian"
            setSelectedLanguage("Romanian")
            showLanguageDropdown.value = false
            databaseReference.child(userPreferences).child( pathString: "language")
                .setValue(selectedLanguage.value)
        }
```

Figure 4.5: Language Dropdown Menu

Here, in Figure 4.5 can be seen the implementation for the language dropdown menu, where the user can choose either Romanian or English as the preferred language for the application.

A DropdownMenu composable was used to achieve this, the menu showing up only when the user clicks on an icon with a text called Language, and closing after the user chooses a language. The menu is populated by items made with the composable DropdownMenuItem which hold text and a clickable button that selects the new language and calls a function inside the main activity that changes the text to the desired language.



```
private fun setSelectedTheme(selectedThemeOption: String) {
    sharedPreferences.edit().putString("Theme", selectedThemeOption).apply()
    selectedTheme.value = selectedThemeOption

    when (selectedThemeOption) {
        "Light" -> {
            AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_NO)
        }

        "Dark" -> {
            AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_YES)
        }
    }
}

private fun setSelectedLanguage(selectedLanguage: String) {
    val locale :Locale! = when (selectedLanguage) {
        "English" -> Locale.ENGLISH
        "Romanian" -> Locale( language: "ro", country: "RO")
        else -> Locale.getDefault()
    }
    val configuration :Configuration! = resources.configuration
    configuration.setLocale(locale)
    resources.updateConfiguration(configuration, resources.displayMetrics)
}
```

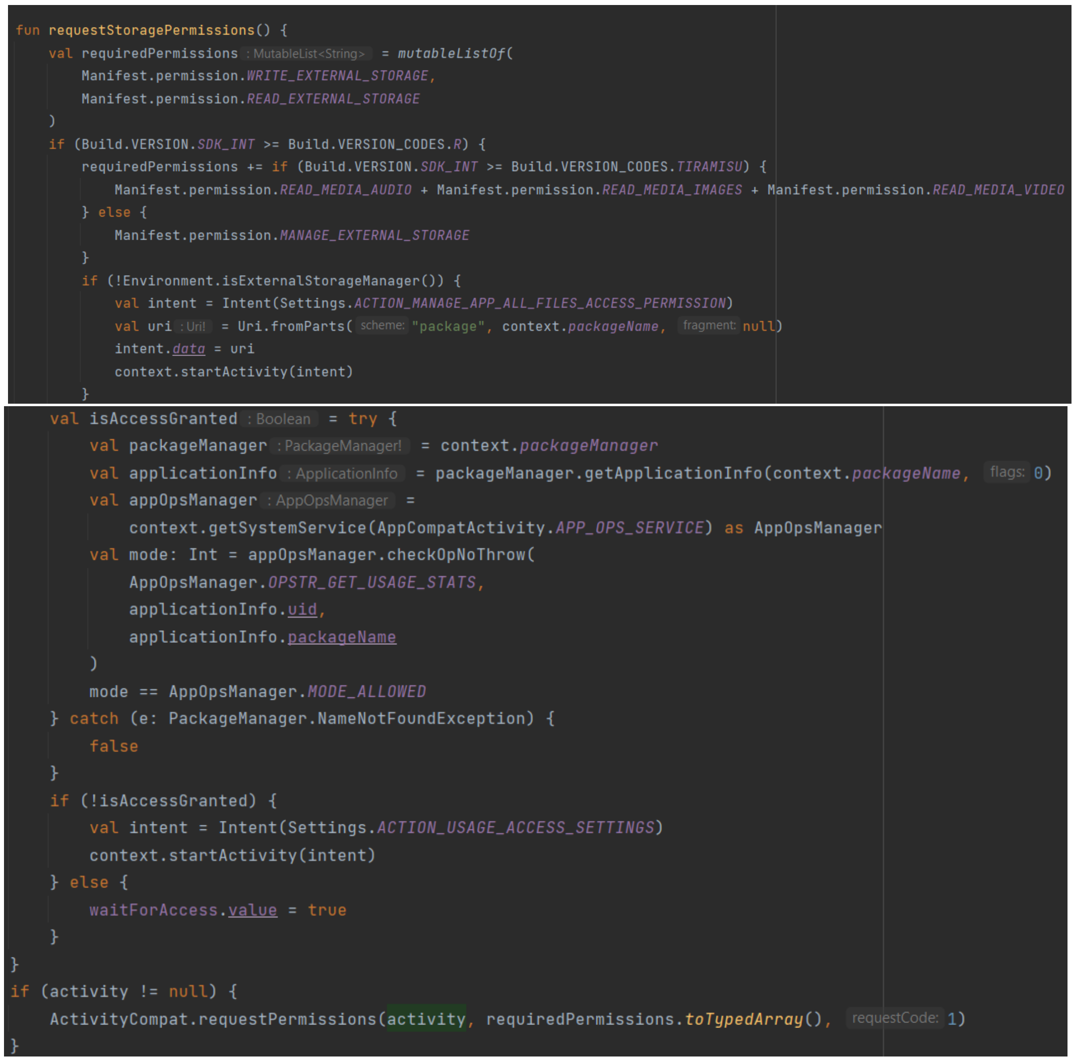
Figure 4.6: Theme Options

Figure 4.6 showcases how the language and theme are being applied, the code being found in the main activity class.

For the theme, the main activity looks at the sharedPreferences and edits them by applying the new selected theme. If the theme is selected as light, the app turns off night mode, while if the theme selected is the dark one, the night mode is turned on.

The language is applied by selecting the Locale as either English or Romanian, this object is used to define what language, symbols, and so on should be used.

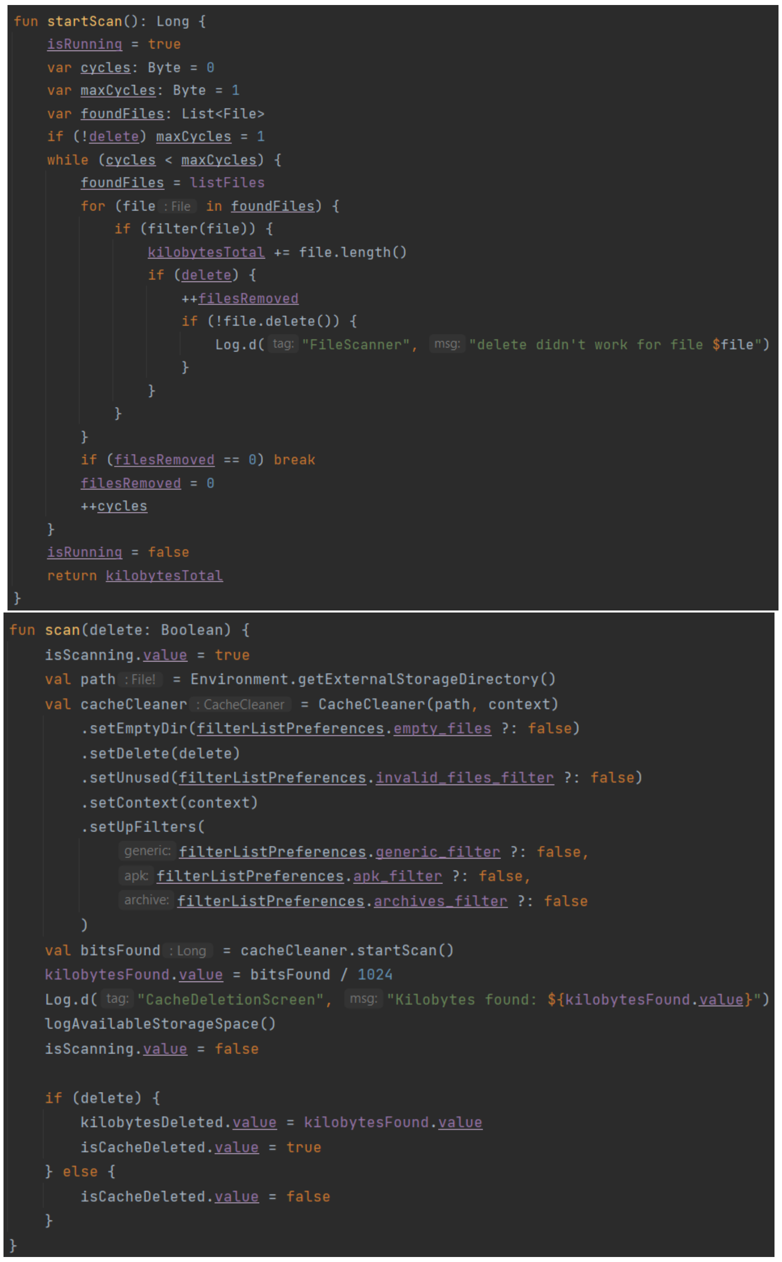
4.2.3 Cache Deletion Screen Implementation



```
fun requestStoragePermissions() {
    val requiredPermissions : MutableList<String> = mutableListOf(
        Manifest.permission.WRITE_EXTERNAL_STORAGE,
        Manifest.permission.READ_EXTERNAL_STORAGE
    )
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
        requiredPermissions += if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
            Manifest.permission.READ_MEDIA_AUDIO + Manifest.permission.READ_MEDIA_IMAGES + Manifest.permission.READ_MEDIA_VIDEO
        } else {
            Manifest.permission.MANAGE_EXTERNAL_STORAGE
        }
        if (!Environment.isExternalStorageManager()) {
            val intent = Intent(Settings.ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION)
            val uri : Uri! = Uri.fromParts( scheme: "package", context.packageName, fragment: null)
            intent.data = uri
            context.startActivity(intent)
        }
    val isAccessGranted : Boolean = try {
        val packageManager : PackageManager! = context.packageManager
        val applicationInfo : ApplicationInfo = packageManager.getApplicationInfo(context.packageName, flags: 0)
        val appOpsManager : AppOpsManager =
            context.getSystemService(AppCompatActivity.APP_OPS_SERVICE) as AppOpsManager
        val mode: Int = appOpsManager.checkOpNoThrow(
            AppOpsManager.OPSTR_GET_USAGE_STATS,
            applicationInfo.uid,
            applicationInfo.packageName
        )
        mode == AppOpsManager.MODE_ALLOWED
    } catch (e: PackageManager.NameNotFoundException) {
        false
    }
    if (!isAccessGranted) {
        val intent = Intent(Settings.ACTION_USAGE_ACCESS_SETTINGS)
        context.startActivity(intent)
    } else {
        waitForAccess.value = true
    }
}
if (activity != null) {
    ActivityCompat.requestPermissions(activity, requiredPermissions.toTypedArray(), requestCode: 1)
}
```

Figure 4.7: Permission Activity

Firstly, the application requires some permissions from the user. As such, the above figure shows the logic for asking for such permissions. Firstly, it checks the Android version and adds the required permissions to a list. It then checks if the usage access is granted and starts an activity to request permission if it is not granted. Lastly, the function uses ActivityCompat.requestPermissions to request storage permissions from the user.



```
fun startScan(): Long {
    isRunning = true
    var cycles: Byte = 0
    var maxCycles: Byte = 1
    var foundFiles: List<File>
    if (!delete) maxCycles = 1
    while (cycles < maxCycles) {
        foundFiles = listFiles
        for (file : File in foundFiles) {
            if (filter(file)) {
                kilobytesTotal += file.length()
                if (delete) {
                    ++filesRemoved
                    if (!file.delete()) {
                        Log.d( tag: "FileScanner",  msg: "delete didn't work for file $file")
                    }
                }
            }
        }
        if (filesRemoved == 0) break
        filesRemoved = 0
        ++cycles
    }
    isRunning = false
    return kilobytesTotal
}
```

```
fun scan(delete: Boolean) {
    isScanning.value = true
    val path : File! = Environment.getExternalStorageDirectory()
    val cacheCleaner : CacheCleaner = CacheCleaner(path, context)
        .setEmptyDir(filterListPreferences.empty_files ?: false)
        .setDelete(delete)
        .setUnused(filterListPreferences.invalid_files_filter ?: false)
        .setContext(context)
        .setUpFilters(
            generic: filterListPreferences.generic_filter ?: false,
            apk: filterListPreferences.apk_filter ?: false,
            archive: filterListPreferences.archives_filter ?: false
        )
    val bitsFound : Long = cacheCleaner.startScan()
    kilobytesFound.value = bitsFound / 1024
    Log.d( tag: "CacheDeletionScreen",  msg: "Kilobytes found: ${kilobytesFound.value}")
    logAvailableStorageSpace()
    isScanning.value = false

    if (delete) {
        kilobytesDeleted.value = kilobytesFound.value
        isCacheDeleted.value = true
    } else {
        isCacheDeleted.value = false
    }
}
```

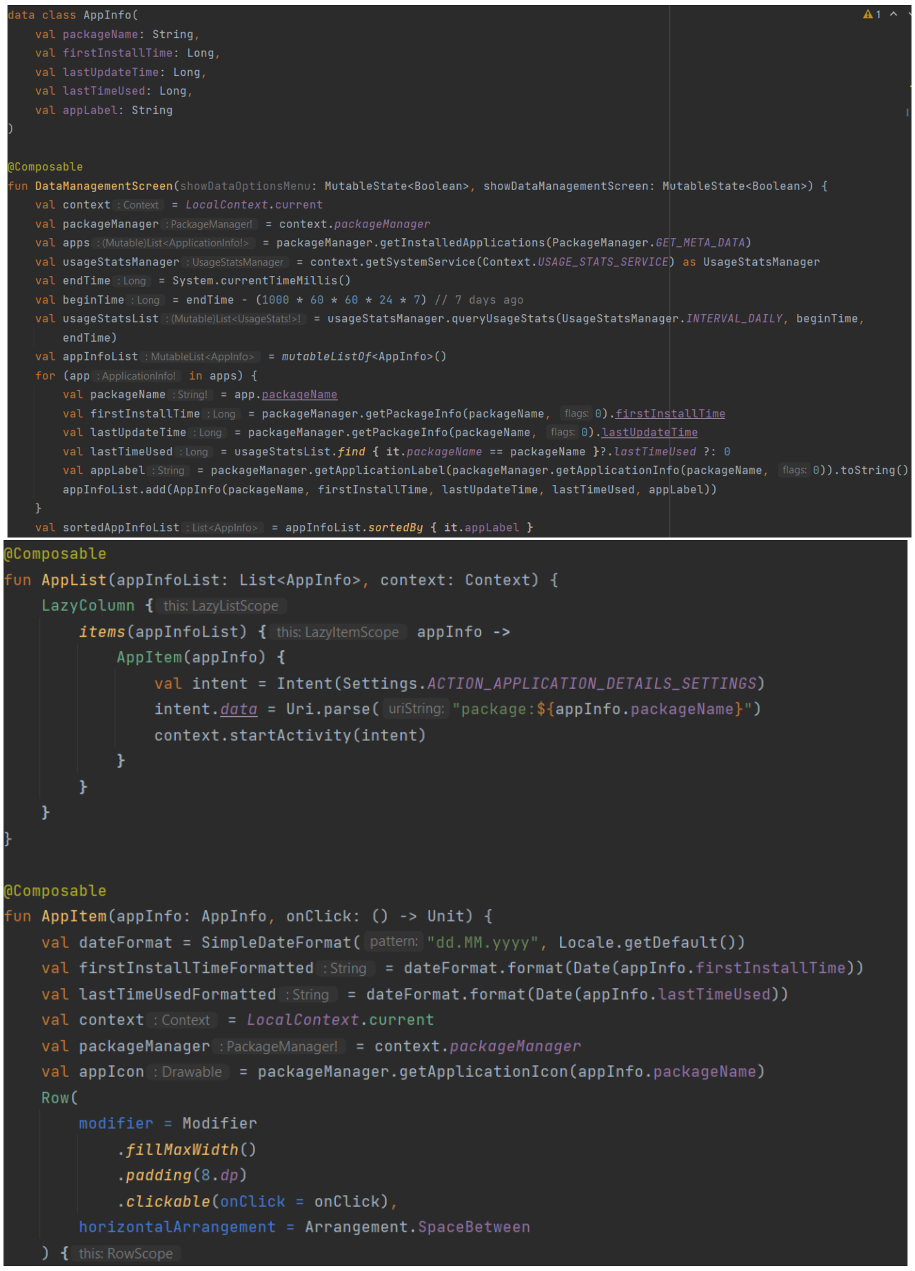
Figure 4.8: File Scanner

The Cache Screen consists of two main buttons, analyze and delete. Both of these buttons are created using compose functions like text, buttons, and so on. However, the main point of this app is the scanning feature. As such, when the user presses one of the buttons, the app utilizes the function defined in Figure 4.8.

Scan() gets the general path of the external storage and creates an instance of the CacheCleaner class that sets up various filters for the type of files the user wants to delete and passes a value for delete, whether the user wants to remove or not the files scanned. To compute the number of kilobytes found, the method uses the CacheCleaner instance to start another function for the proper scanning.

The function startScan() first creates a list of all the files in the storage and then begins going through each one and selecting only those that have their respective filter set on. If the delete value is set to true the removing process can begin and all selected files are deleted from the system. Finally, the app returns the amount of memory it has freed up.

4.2.4 Unused App Screen Implementation



```kotlin
data class AppInfo(
    val packageName: String,
    val firstInstallTime: Long,
    val lastUpdateTime: Long,
    val lastTimeUsed: Long,
    val appLabel: String
)

@Composable
fun DataManagementScreen(showDataOptionsMenu: MutableState<Boolean>, showDataManagementScreen: MutableState<Boolean>) {
    val context : Context = LocalContext.current
    val packageManager : PackageManager! = context.packageManager
    val apps : (Mutable)List<ApplicationInfo!> = packageManager.getInstalledApplications(PackageManager.GET_META_DATA)
    val usageStatsManager : UsageStatsManager = context.getSystemService(Context.USAGE_STATS_SERVICE) as UsageStatsManager
    val endTime : Long = System.currentTimeMillis()
    val beginTime : Long = endTime - (1000 * 60 * 60 * 24 * 7) // 7 days ago
    val usageStatsList : (Mutable)List<UsageStats!>! = usageStatsManager.queryUsageStats(UsageStatsManager.INTERVAL_DAILY, beginTime,
        endTime)
    val appInfoList : MutableList<AppInfo> = mutableListOf<AppInfo>()
    for (app : ApplicationInfo! in apps) {
        val packageName : String! = app.packageName
        val firstInstallTime : Long = packageManager.getPackageInfo(packageName, flags: 0).firstInstallTime
        val lastUpdateTime : Long = packageManager.getPackageInfo(packageName, flags: 0).lastUpdateTime
        val lastTimeUsed : Long = usageStatsList.find { it.packageName == packageName }?.lastTimeUsed ?: 0
        val appLabel : String = packageManager.getApplicationLabel(packageManager.getApplicationInfo(packageName, flags: 0)).toString()
        appInfoList.add(AppInfo(packageName, firstInstallTime, lastUpdateTime, lastTimeUsed, appLabel))
    }
    val sortedAppInfoList : List<AppInfo> = appInfoList.sortedBy { it.appLabel }
```

```kotlin
@Composable
fun AppList(appInfoList: List<AppInfo>, context: Context) {
    LazyColumn { this: LazyListScope
        items(appInfoList) { this: LazyItemScope appInfo ->
            AppItem(appInfo) {
                val intent = Intent(Settings.ACTION_APPLICATION_DETAILS_SETTINGS)
                intent.data = Uri.parse( uriString: "package:${appInfo.packageName}")
                context.startActivity(intent)
            }
        }
    }
}

@Composable
fun AppItem(appInfo: AppInfo, onClick: () -> Unit) {
    val dateFormat = SimpleDateFormat( pattern: "dd.MM.yyyy", Locale.getDefault())
    val firstInstallTimeFormatted : String = dateFormat.format(Date(appInfo.firstInstallTime))
    val lastTimeUsedFormatted : String = dateFormat.format(Date(appInfo.lastTimeUsed))
    val context : Context = LocalContext.current
    val packageManager : PackageManager! = context.packageManager
    val appIcon : Drawable = packageManager.getApplicationIcon(appInfo.packageName)
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .padding(8.dp)
            .clickable(onClick = onClick),
        horizontalArrangement = Arrangement.SpaceBetween
    ) { this: RowScope
```

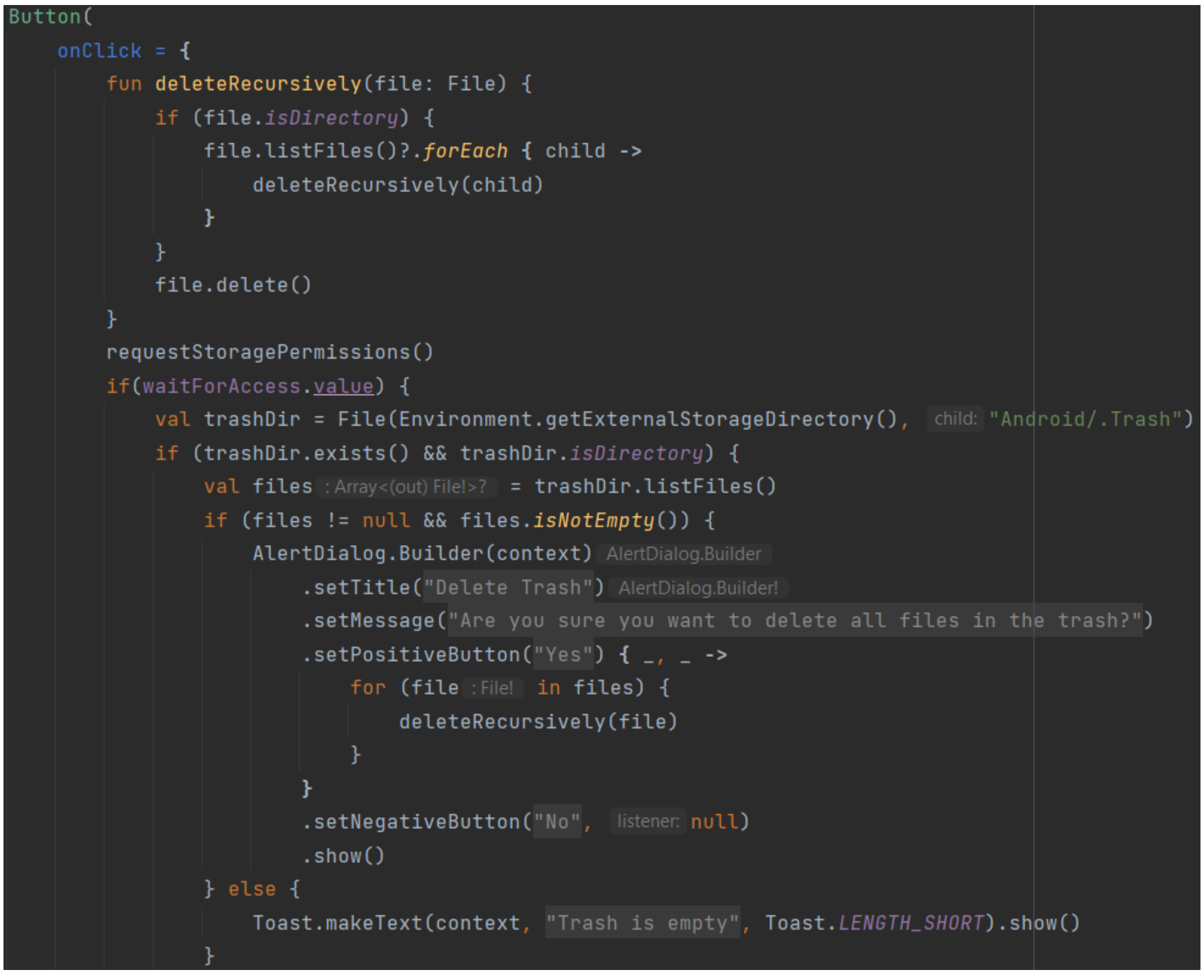
Figure 4.9: File Scanner

The following Figure, 4.9, shows the logic behind the unused app list. The app list is given through a package manager which retrieves all applications which are then sorted by alphabetical order. The usage stats manager retrieves all the information on when the application was used, when was it installed and when was it last updated.

The composable function AppList assigns to each app their respective app info screen so that when one of the applications in the list is clicked, the app info activity and screen are opened.

AppItem is used to create the visual representation for each application in the list.

4.2.5 Trash Emptying Implementation



```
Button(
    onClick = {
        fun deleteRecursively(file: File) {
            if (file.isDirectory) {
                file.listFiles()?.forEach { child ->
                    deleteRecursively(child)
                }
            }
            file.delete()
        }
        requestStoragePermissions()
        if(waitForAccess.value) {
            val trashDir = File(Environment.getExternalStorageDirectory(), child: "Android/.Trash")
            if (trashDir.exists() && trashDir.isDirectory) {
                val files : Array<(out) File!>? = trashDir.listFiles()
                if (files != null && files.isNotEmpty()) {
                    AlertDialog.Builder(context) AlertDialog.Builder
                        .setTitle("Delete Trash") AlertDialog.Builder!
                        .setMessage("Are you sure you want to delete all files in the trash?")
                        .setPositiveButton("Yes") { _, _ ->
                            for (file : File! in files) {
                                deleteRecursively(file)
                            }
                        }
                        .setNegativeButton("No", listener: null)
                        .show()
                } else {
                    Toast.makeText(context, "Trash is empty", Toast.LENGTH_SHORT).show()
                }
```

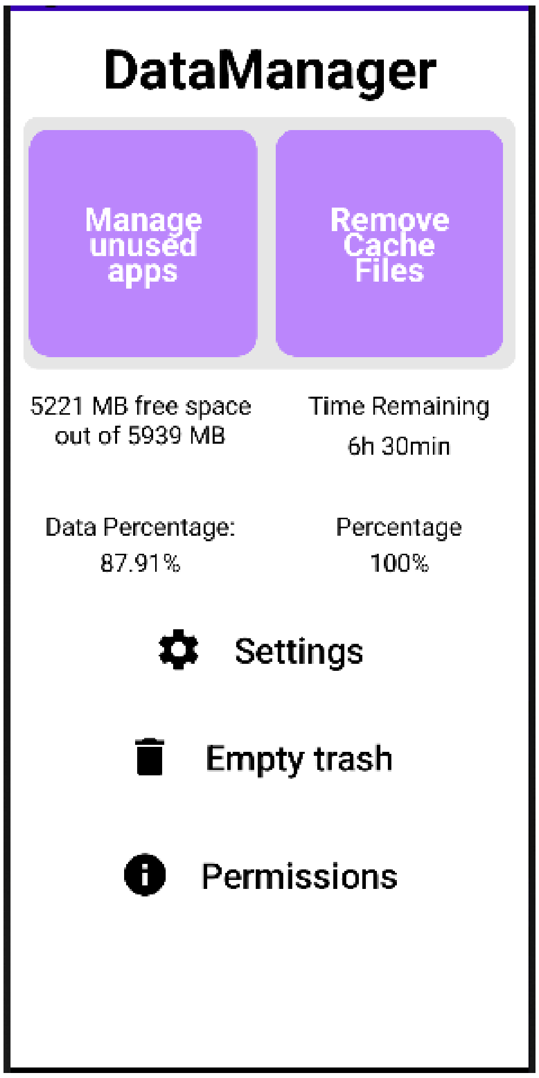
Figure 4.10: App Info

The last Figure, 4.10, shows the logic for emptying the trash folder, it first requires the needed permissions, and if they are granted, the app will check for the existence of the trash directory. If it exists, the app lists the files inside it and starts recursively deleting them after creating an alert menu. If no apps are found, the application creates a toast composable, a pop-up message, saying that the trash is empty.

Chapter 5

User Manual

5.1 Main Menu



5221 MB free space out of 5939 MB

Time Remaining 6h 30min

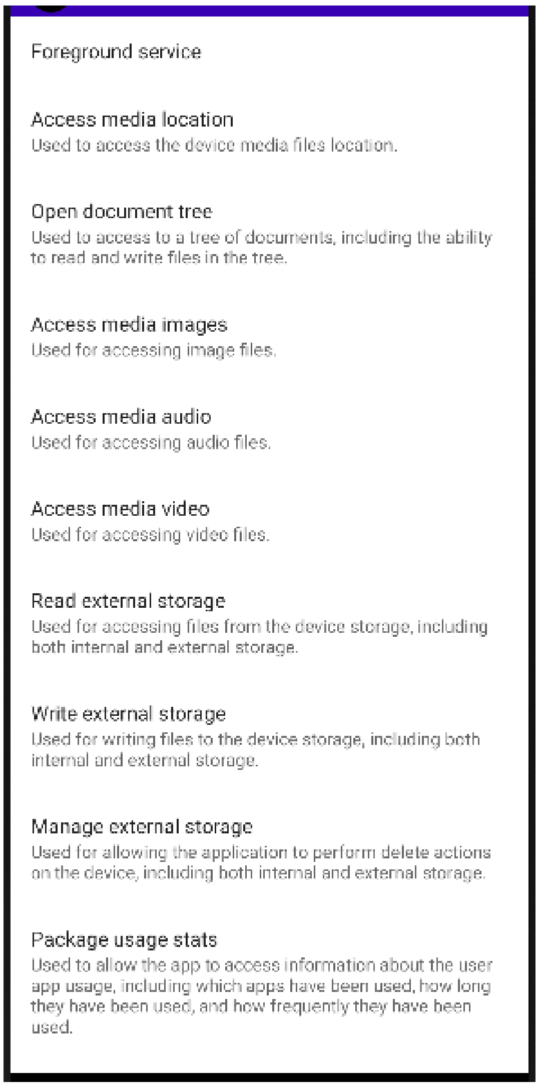
Data Percentage: 87.91%

Percentage 100%

Figure 5.1: Main Menu Screen

In Figure 5.1 we can see the main menu screen of the application, the user is greeted with a menu consisting of 5 buttons, each with its own unique functionality. The first time the user opens the application, some default settings will be placed and stored in the database, with the theme being a light one, the language being English, and no permissions being given by default, the app waiting for the user to give them when he either enters the cache screen and tries to analyze for residual files when he tries to enter the manage unused apps, or when he tries to press on the empty trash button. Each time the user tries one of these actions, the application will see if the permissions were granted, if so, the application can continue.

5.2 Permissions Menu



Foreground service

Access media location
Used to access the device media files location.

Open document tree
Used to access to a tree of documents, including the ability to read and write files in the tree.

Access media images
Used for accessing image files.

Access media audio
Used for accessing audio files.

Access media video
Used for accessing video files.

Read external storage
Used for accessing files from the device storage, including both internal and external storage.

Write external storage
Used for writing files to the device storage, including both internal and external storage.

Manage external storage
Used for allowing the application to perform delete actions on the device, including both internal and external storage.

Package usage stats
Used to allow the app to access information about the user app usage, including which apps have been used, how long they have been used, and how frequently they have been used.

Figure 5.2: Permissions Menu Screen

Figure 5.2 shows the screen that pops up when the user presses the permissions button in the main menu. The screen is used for telling the user what is he about to give his consent to, and what each of the required permissions is about. The permissions are as follows:

- Access media location;
- Access foreground services;

- Access image, video, and audio files;
- Read both internal and external storage;
- Write on both internal and external storage;
- Manage both internal and external storage;
- Package Usage stats, which allows the app to access information about the app usage;
- Query all packages, which allows the app to gather information on all installed packages on the device;

5.3 Settings Menu

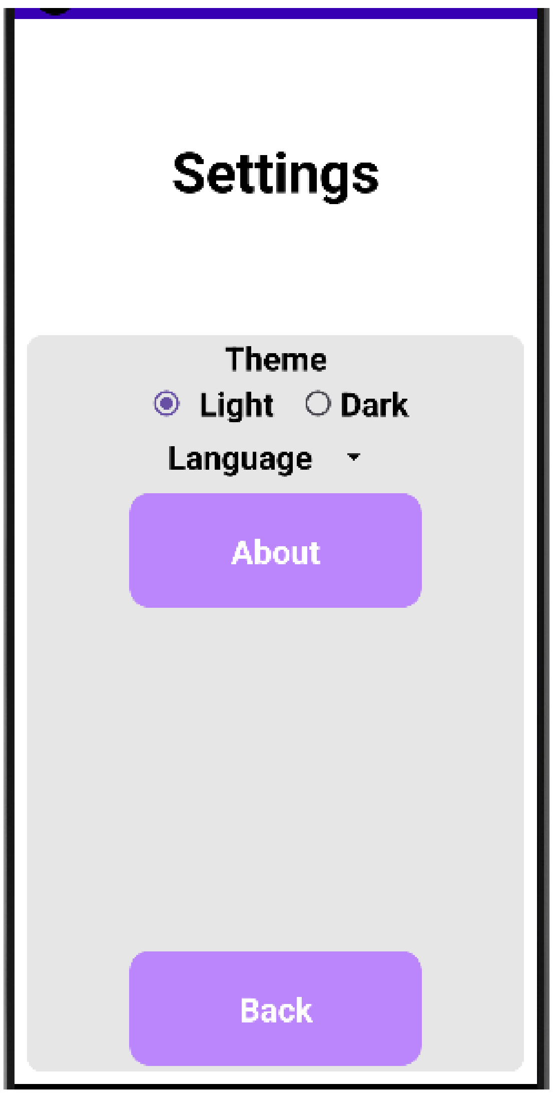


Figure 5.3: Light Settings Menu

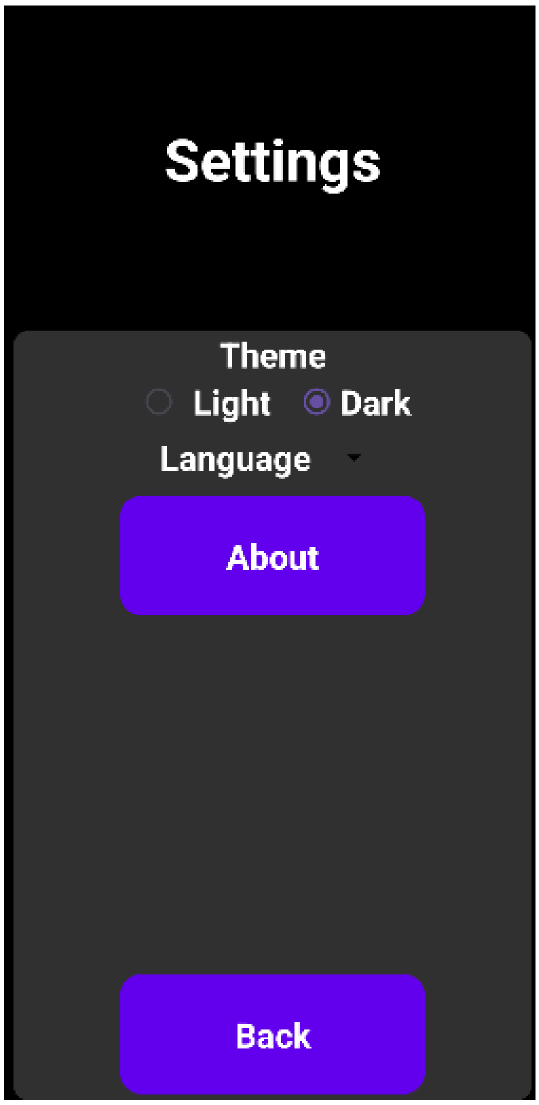


Figure 5.4: Dark Settings Menu

Figures 5.3 and 5.4 showcase the settings menu and its buttons, those being the 2 radio buttons for the theme of the app, consisting of a Light and Dark mode, the Language dropdown menu used for changing the language of the application, the About button that opens the about menu which offers some additional information on what the app is about and the back button for going back to the main menu screen. The preferences set by the user are saved in the Firebase Realtime Database.

The following 2 figures, Figures 5.5 and 5.6, showcase the changes in the layout of the application when the user chooses to change the language from English to Romanian, each element of text being changed to the other language seamlessly.

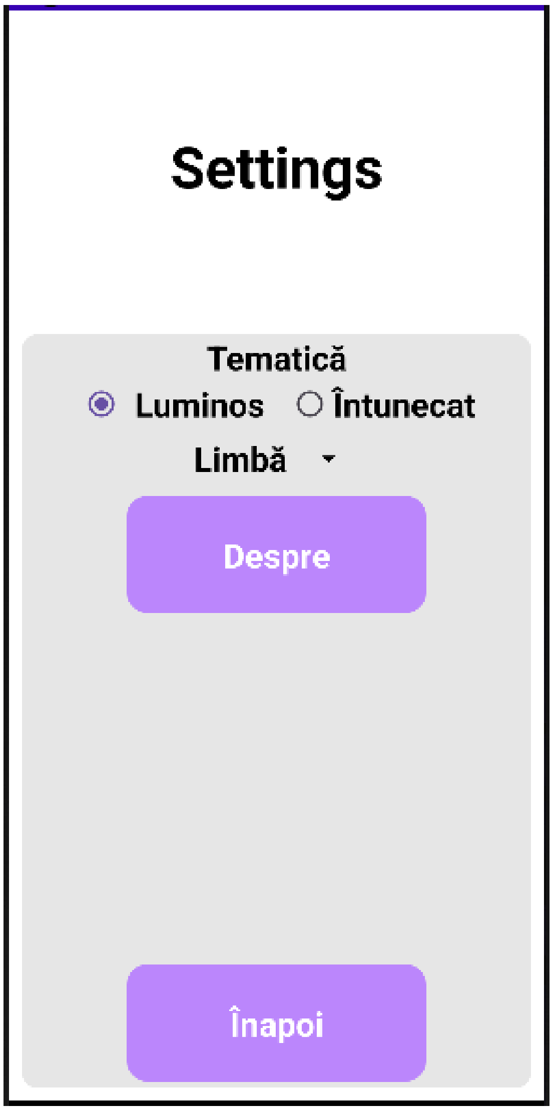
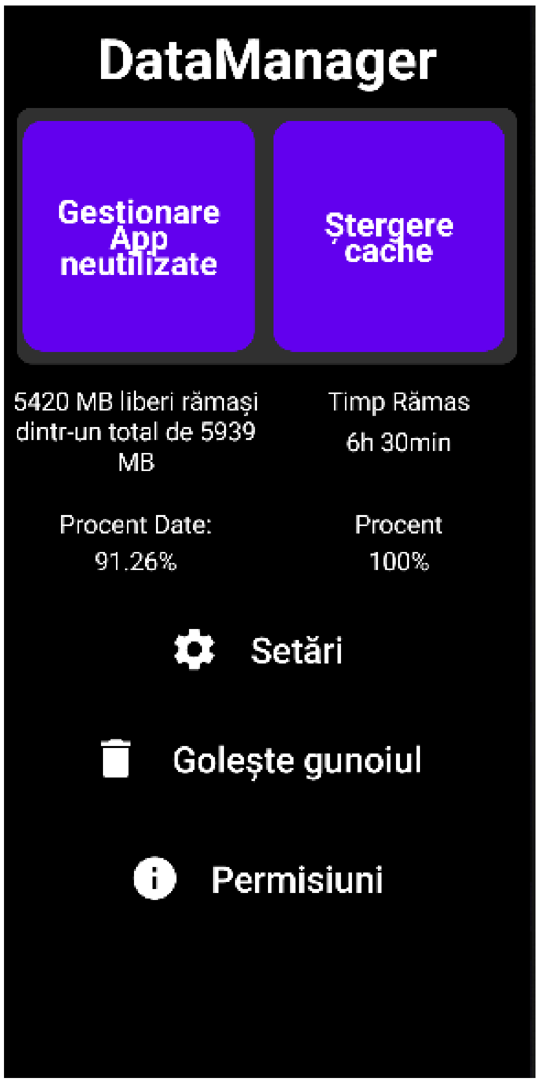


Figure 5.5: Settings Menu with Romanian text



5420 MB liberi rămași dintr-un total de 5939 MB

Timp Rămas 6h 30min

Procent Date: 91.26%

Procent 100%

Figure 5.6: Main Menu with Romanian text

5.4 Cache Screen

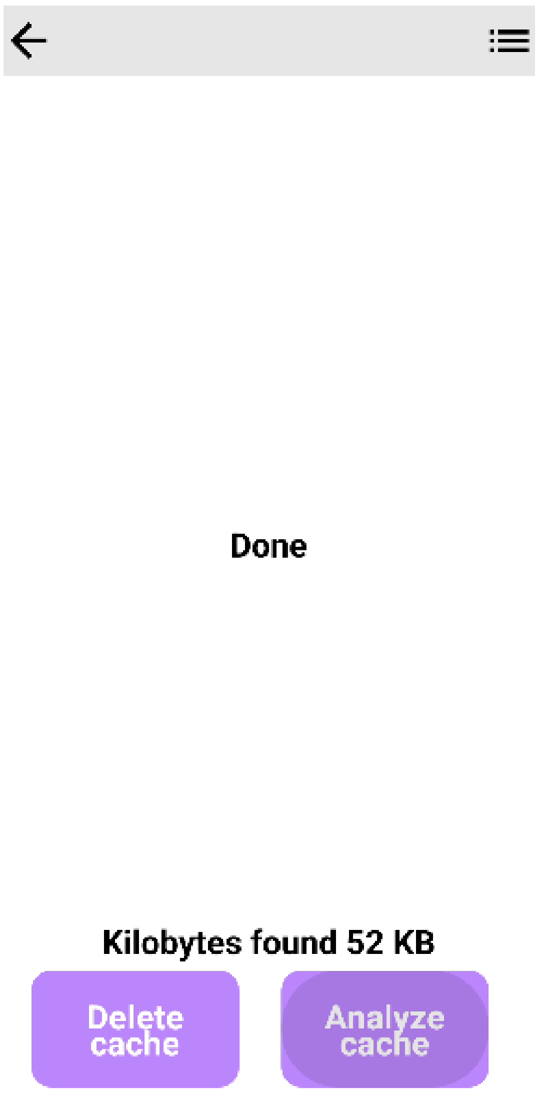
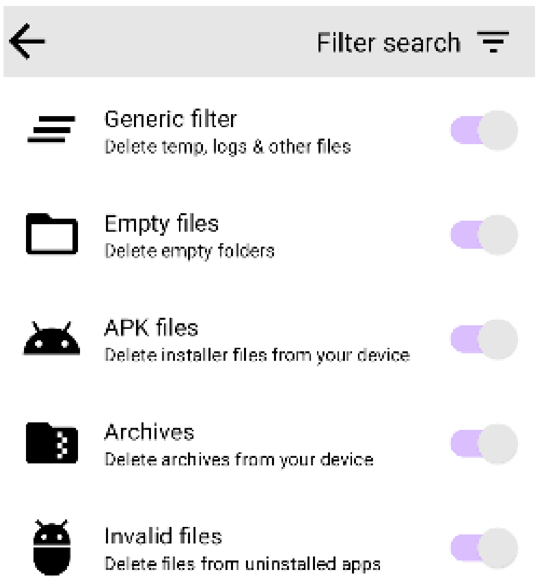


Figure 5.7: Cache Remover Screen

Figure 5.7 showcases the Cache Deletion Screen. Here the user has several options, first, up top, is the back button on the top left, marked by a left-pointed arrow, and the second, top right, the filter settings, represented by the three lines, which, when pressed, takes the user to the filter settings of the app.

On the other hand, at the bottom, the other two buttons, analyze and delete, do pretty much what the name suggests, starting a general scan of the storage of the phone for signs of the type of files the user has selected, while the delete button removes said files. When clicking the buttons, a progress bar will appear showcasing that the search is still ongoing, and, after the search is done, 2 new texts appear, one saying "Done", and the other showcasing the amount of memory found/deleted.



Filter search

Generic filter
Delete temp, logs & other files

Empty files
Delete empty folders

APK files
Delete installer files from your device

Archives
Delete archives from your device

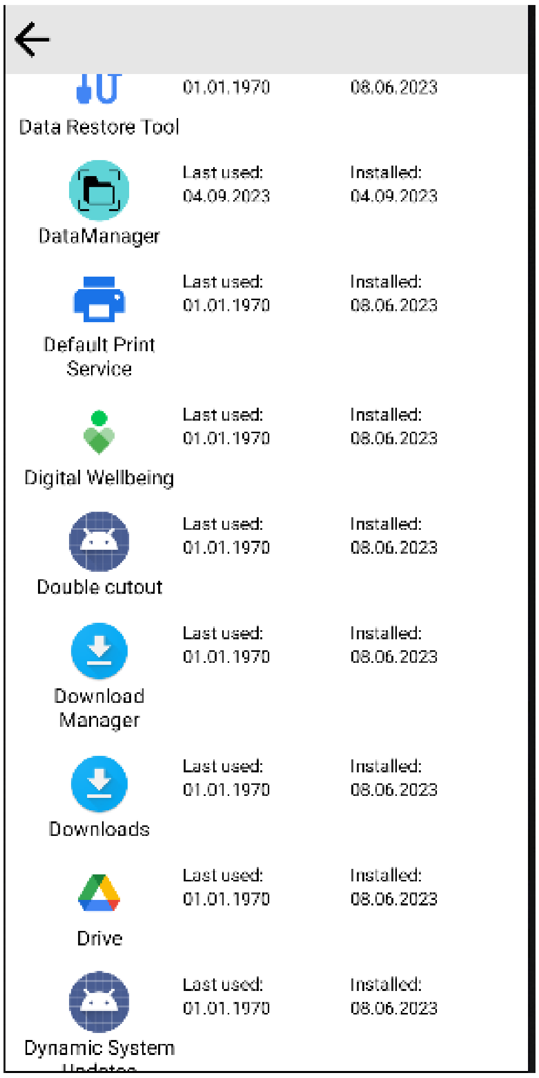
Invalid files
Delete files from uninstalled apps

Figure 5.8: File Filter Menu

The above figure, 5.8, presents the filter search settings menu that the app includes in the cache deleting screen. Here the user can switch on and off what type of files he or she wants to delete, those types ranging from temp, and logs to empty folders and leftover files, the screen shows for each file type a short description of what exactly the app will look for.

Each option is saved in a separate collection from the user preferences one, in Firebase Realtime Database, so that the user doesn't have to re-select what type of file he wants to get rid of each time they re-enters the application.

5.5 Unused App Screen

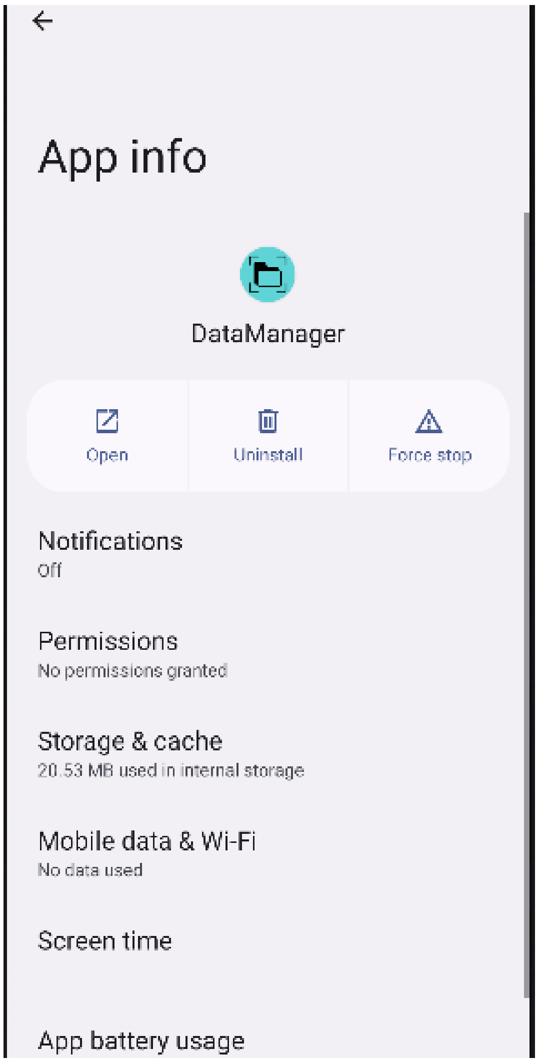


Data Restore Tool	01.01.1970	08.06.2023
DataManager	Last used: 04.09.2023	Installed: 04.09.2023
Default Print Service	Last used: 01.01.1970	Installed: 08.06.2023
Digital Wellbeing	Last used: 01.01.1970	Installed: 08.06.2023
Double cutout	Last used: 01.01.1970	Installed: 08.06.2023
Download Manager	Last used: 01.01.1970	Installed: 08.06.2023
Downloads	Last used: 01.01.1970	Installed: 08.06.2023
Drive	Last used: 01.01.1970	Installed: 08.06.2023
Dynamic System	Last used: 01.01.1970	Installed: 08.06.2023

Figure 5.9: Unused App Menu

The Unused App Screen, shown in the 5.9 figure, lists all applications present in the users' application list, being ordered in an alphabetical manner.

The list of applications showcases in three columns the name and icon, the last time they were used, and the date that the applications were installed. For applications that were never used or have been installed by default, usually, a generic date will be displayed, usually the Unix epoch, more specifically 1.01.1970, while apps that were actually used and installed by the user will showcase the correct dates.


App info

DataManager

Notifications
Off

Permissions
No permissions granted

Storage & cache
20.53 MB used in internal storage

Mobile data & Wi-Fi
No data used

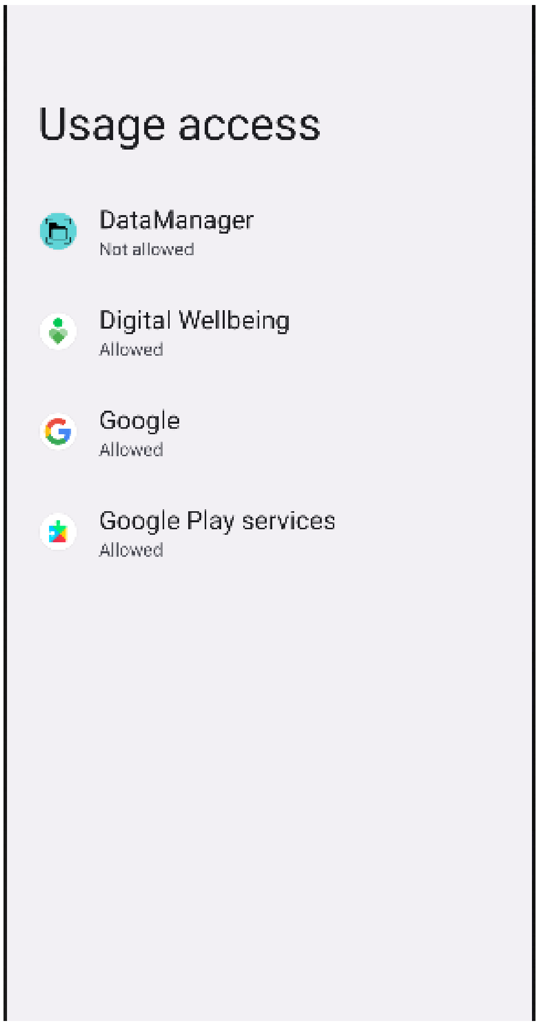
Screen time

App battery usage

Figure 5.10: App Info

The user has the option in the Unused App Screen to click on any of the listed applications and be transported to the general App Info screen, as seen in the 5.10 figure. Here the user has the liberty to do whatever they want. It's up to them if they want to delete, force stop, set new preferences, and so on for the app they clicked on, or they can just as well leave it alone.

When done with the app, the user can go back to the DataManager app by simply clicking on the above arrow, the user being transported back to where they were on the app list.


Usage access

DataManager
Not allowed

Digital Wellbeing
Allowed

Google
Allowed

Google Play services
Allowed

Figure 5.11: Usage Access

The last figure, 5.11, represents the usage access screen that asks the user for the above-mentioned permissions, found in Figure 5.2. What the user needs to do here is simple, he needs to click on the DataManagerApp, and switch on the permission for usage access.

After closing the usage access screen, another one will pop up that asks for access to all files in the system. Again, the user just has to click on the app, in the list, and check on the switch for storage permissions. After switching on the two options, the user will be able to use the functionalities of the application.

Chapter 6

Conclusions & Future work

6.1 Conclusion

In conclusion, the development of such applications represents an important step in the everyday user's ability to control and maintain their phones' storage and in keeping the phone clean of any unwanted data. Without such applications, most of us would be unaware of the reasons why our phones' storage capacity is being drained without our input. As such, the disadvantage of not being able to use our memory space however we want is diminished, leaving us with more freedom for how we use our phones.

This process of data cleaning can have lasting effects on the phone's life, most people not being aware of the fact that they can improve the lifespan of their phone by such methods. Too often one hears that people need to change their phone because it started to work slower, or that they need to buy one because they have too little memory left in it. With this in mind, making sure that our phone is well kept and once in a while cleaned is the same as cleaning our own homes, no one wants to live in a filthy house, so our phones should be treated with the same manner of respect.

6.2 Future work

Further improvements to applications such as this one are possible, ranging from data analysis over a set period of time to media optimization, such as the removal of duplicate image, audio, or video files, the management of notifications, application analysis, and so on.

Other areas that can be looked on are:

- Automation: We could look to improve the automation of data cleaning processes. This involves the development of new algorithms and techniques for automatically detecting errors in data. With the current Artificial Inteligence (AI) revolution, such automation processes could be the future of data-cleaning applications;

- The handling of complex data types: With data becoming more complex each passing day, there might come a growing need for data cleaning methods that can handle these types of complex data types and structures.

Bibliography

[1] Android studio. https://developer.android.com/studio.

[2] Firebase. https://firebase.google.com/.

[3] Firebase realtime database. https://firebase.google.com/docs/database.

[4] Jetpack compose. https://developer.android.com/jetpack/compose.

[5] Kotlin. https://kotlinlang.org/docs/home.html.