Project 2 Research Paper

File Manager Application for Android

Ridvan B. Suleymanov

Developed at Georgia State University

April 25, 2023

Main Content:

Overview/Background Introduction:

My Simple File Manager Application is an Android application designed to make file management on mobile devices more efficient and effective for managing files and directories. The app is a necessary tool for anyone who frequently uses their mobile device for file management because it makes it easier and more efficient to navigate through files and folders. When creating a File Manager Application, there are several important aspects that must be considered to ensure that the app is functional, user-friendly, and efficient. One of the most important aspects of creating a File Manager Application is designing the user interface. The GUI of my Simple File Manager Application is designed to be compatible and easy to use, with a variety of features such as copying, moving, and deleting files. The app also allows users to select singular or multiple files at once, enabling them to perform batch operations on files and directories making user experience easier by making file management more efficient. Another essential aspect of creating a File Manager Application is the underlying code structure. The Simple File Manager Application is built using Java and the Android SDK, and employs a Model-View-Controller architecture to separate various aspects of the app. This helps in order to maintain a clear and organized codebase, making the app easier to maintain. There are also a variety of features such as runtime permissions and file input/output operations, ensuring that the user can effectively manage their files. The app grants users permission to access the files on their mobile device and the file input/output operations allow users to create, modify, and delete files and directories, making file management on mobile devices more efficient and effective. My Simple File Manager Application offers a user-friendly, efficient, and comprehensive way for users to manage their files on their Android device because it is centered around efficiently accessing, managing, and manipulating files and directories on the device's storage.

To provide a brief overview of my File Manager application, the app is designed to allow users to manage files and directories on their device's storage. It provides a user interface to navigate through files and directories, view their properties, and perform actions such as copying, moving, deleting, and renaming them. I was able to access and manipulate files and directories through the use of low-level functionalities of the operating system such as the java.io.File. The File Manager application code is comprised of a main activity that is responsible for handling the user interface and user interactions. To support features such as recycler view, permissions, and content providers, the app uses the AndroidX library. The main activity sets up the recycler view to display the files and folders present in the device's storage. To access the device's storage, the code checks whether the app has the necessary permissions, and if not, requests the user to grant permission. User actions such as deleting, copying, and moving files are handled by the code. When a user selects a file or folder, the code checks whether it is a directory or a file. If it is a directory, the FileUtil class is called to fetch the files and folders contained within it, and its contents are displayed. If it is a file, the code opens the file using an intent. The code also supports multi-selection of files and folders by enabling the user to select multiple files and folders at once. An Array List<FileManagerModel> is used to keep track of the selected files and folders. Overall, the code implements basic functionalities

such as browsing and managing files and folders, displaying files and folders in a recycler view, handling user actions, and requesting permissions to access the device's storage.

OS low-level functionality:

The app makes use of several OS low-level functionalities through the java.io.File class to access and manipulate files and directories. These functions include memory management, file system operations, process management, input/output operations, and network operations. Memory management involves allocating and freeing memory, managing memory pools, and other operations related to the use of system memory. File system operations include reading and writing files, managing file permissions, and other interactions with the file system. Process management includes creating and managing processes, managing inter-process communication, and other operations related to managing the execution of code on the system. These low-level functions are provided by the operating system and used by the app to interact with the system hardware and software. The java.io.File class provides various methods that allow the app to perform operations such as getting the absolute path of a file or directory, checking whether a File object represents a directory or not, returning an array of File objects representing the files and directories contained in a directory, deleting a file or directory, renaming a file or directory, and creating a directory and any necessary but nonexistent parent directories. These functionalities enable the app to perform basic file management tasks that users might need, such as deleting unwanted files, renaming files, and organizing their files and directories. The application uses several low-level functionalities offered by the operating system to perform file and directory actions. Functions like File.getAbsolutePath(), File.isDirectory(), and File.listFiles() are used to retrieve the absolute path of a file or directory, check if a File object is a directory or not and then return an array of File objects representing files and directories contained within a directory. Additionally, the app uses functions like File.delete() and File.renameTo() to delete or rename a file or directory, and File.mkdirs() to create a directory and any necessary parent direvtories that do not already exist. By using these functions that are given by the operating system through the java.io.File class, the application can access and manipulate files and directories on the device’s storage efficiently.

Design:

Now in order explain the design of my File Manager Application, we will look at all the elements that were used together to make a fully functioning app. It was important that the code was written in a way that is efficient, maintainable, and compatible to the user. This involves using appropriate data structures and algorithms as well as paying attention to established coding standards and practices. The app has several design aspects, including file system management, permissions management, process management, user interface management, intents and content providers, and resource management. In terms of file system management, the app utilizes the File class to work with files and directories and accesses external storage using the Environment class. To support features such as recycler view, permissions, and content providers, the code

uses the AndroidX library. The main activity sets up the recycler view to display files and folders on the device's storage. Before accessing the storage, the code checks if the app has the necessary permissions and requests the user to grant permission if not. The way it does this is by requesting runtime permissions for accessing external storage and using the PackageManager class to check whether a specific permission is granted. For process management, the app launches background tasks for file operations using the AsyncTask class and checks the SDK version to determine which intent to use for managing app permissions. The app's user interface is created using various Android UI components such as RecyclerView, PopupMenu, and Toast. Lastly, the app uses the LayoutInflater class to inflate the activity's layout file and the R class to access various resources such as strings and layouts.

To use the app, users must grant permission to access external storage on their device. After accepting the necessary permissions, the app loads the root directory of the device's storage and presents the contents in a RecyclerView. The app starts with the onCreate method, where the necessary view elements are set up and permissions to access device storage are requested. If permission is granted, the root files of the device are loaded using loadRootFiles. Otherwise, the app will close. The app uses a RecyclerView to display the files and folders on the device, and the initRecycler method sets up the RecyclerView and its adapter. Users can select a single file or directory by clicking on it within the RecyclerView. To enable multi-selection mode, users can click on the selection icon located in the top-right corner of the app. When in multi-selection mode, users can select multiple files and directories by clicking on them, and can carry out various actions like copying, moving, deleting, and renaming them. Clicking on a file or folder opens it, while long-clicking enters multi-selection mode, allowing the user to select multiple files or folders at once. When the user selects one of these menu options, a corresponding method is called. For example, the enableCopy method is called when the "Copy" button is pressed, while the doCopy method is called when the "Paste" button is pressed, and the "Copy" option is selected. These methods use the FileManagerModel class to manipulate files, folders, and make use of the FileUtil class to perform file operations in the background. Finally, a DialogUtil class is used to display progress and error dialogs during these operations. There are two important classes which are FileUtil and ActionExecuter, which help manage files and directories on external storage. The FileUtil class offers various methods for reading and writing files and directories, while the ActionExecuter class is responsible for performing file operations like copy, move, and delete. When the user performs a delete operation, the app utilizes the ActionExecuter class to delete selected files and directories. When the user copies or moves a file, the app enables the corresponding function and allows the user to choose the destination folder. Finally, the app executes the corresponding operation when the user pastes the selected files and directories. With these classes and methods, the app can effectively manage and manipulate files and directories stored on external storage.

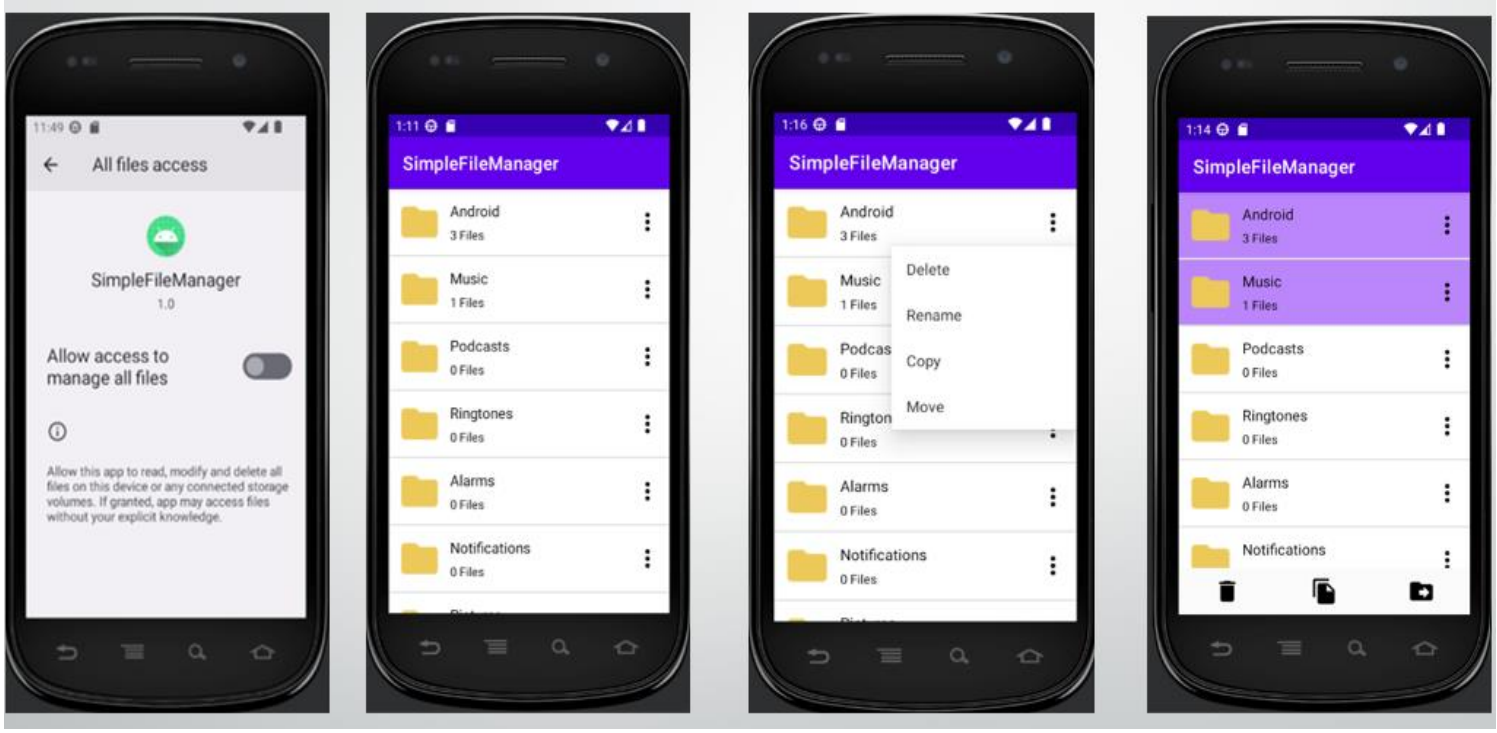
System calls are another important feature that is needed in operating systems because they provide a way for programs to request services from the operating system, such as accessing hardware devices or managing files. Without system calls, applications would have to perform these tasks themselves, resulting in a lot of duplicated effort and potential errors. System calls allow for efficient communication between programs and the operating system. There were many

system calls used to develop my application, Environment.getExternalStorageDirectory() method is used to retrieve the primary shared/external storage directory where the app can place its files.checkSelfPermission() is used to check whether the app has been granted permission to write to external storage, while ActivityCompat.requestPermissions() can be used to request permission if it hasn't been granted yet. The onCreate() method initializes various UI components and sets up listeners for button clicks and recycler view item clicks, while the setContentView() method sets the activity layout. FileUtil() performs file operations like listing files in a directory, copying, moving, and deleting files. The startActivityForResult() method starts a new activity and expects a result back from it. onResume() is called when the activity becomes visible to the user, onStop() is called when the activity is no longer visible to the user and has been stopped, and onDestroy() is called when the activity is destroyed by the system. To grant an app the ability to manage all files on a user's external storage, including the ability to read, write, delete, and modify any files stored there, the android.permission.WRITE_EXTERNAL_STORAGE, and android.permission.MANAGE_EXTERNAL_STORAGE permission is used.

Another important component about my app is how users use it and the efficiency of it. That's where the user interface aspect comes in because it's the way people interact with the app. To provide the user with the most efficient and compatible way to interact with my app, I had to ensure that my app is user-friendly and easily capable of enabling users to navigate and access its features with ease. This involves careful selection of colors, fonts, and icons, as well as consideration of the layout and placement of different elements. Some ways I intended on making my app easily useable was through the use of user interface elements such as buttons and dialogs to facilitate file management. It is implemented using the Android SDK and its built-in UI components, such as RecyclerView, buttons, and dialogs. The User Interface of this app is composed of several key components that work together to allow using the app to be an easier experience. First, the RecyclerView serves as the main view for displaying the files and folders in the current directory. This view is populated using an adapter that creates a ViewHolder for each file or folder item, allowing for easy scrolling and navigation. In addition to the RecyclerView, there are several buttons that provide user actions. These buttons allow users to delete, copy, move, paste, and cancel actions. Each button is connected to click listeners that trigger the appropriate actions, making it easy for users to navigate and manage their files and folders. The RecyclerAdaptor is responsible for creating and binding views for the list items. The buttons available to the user are: Delete button (binding.btnDelete): This button deletes the selected files from the file system. It is enabled only when one or more files are selected. Copy button (binding.btnCopy): This button enables the user to copy the selected files to a different location. It is enabled only when one or more files are selected. Move button (binding.btnMove): This button enables the user to move the selected files to a different location. It is enabled only when one or more files are selected. Cancel button (binding.btnCancel): This button cancels the selection mode and deselects all the selected files. It is enabled only when one or more files are selected. Paste button (binding.btnPaste): This button pastes the files that were copied or moved to the current folder and it is enabled only when the user has copied or moved one or more files.

Two dialogs are also used in the app to provide additional functionality. The first is a rename dialog that is displayed when the user wants to rename a file or folder. This dialog allows

users to enter a new name for the item and save the changes. The second dialog is a permission dialog that is displayed when the app needs to request permission to access files on the device. This is an important security feature that ensures the app only accesses files it is authorized to access. There is a also a menuLayout (binding.menuLayout) that is displayed when the user long-presses an item in the list. This popup menu provides additional actions such as renaming, copying, moving, and deleting files. By providing these additional options, users can quickly and easily manage their files and folders without needing to navigate through multiple screens or menus. The MainActivity class also maintains several boolean flags and lists to keep track of the current state of the UI, such as whether multi-selection mode is on, which files are currently selected, and which files have been copied or moved. These flags and lists are used to enable or disable the buttons and perform the appropriate actions when the user interacts with the UI. Overall, the combination of the RecyclerView, buttons, dialogs, and popup menu provide a complex User Interface that allows users to manage their files and folders with ease. For example, users have the option to confirm their actions before deleting or renaming files. The user interface is implemented using XML and is inflated in the onCreate() method of the MainActivity class. The user interface consists of a RecyclerView widget to display a list of files and folders, and a set of buttons to perform various actions on the selected files. The RecyclerView is initialized using a RecyclerAdaptor, which is a custom adapter class that extends RecyclerView.Adapter.



In conclusion, my File Manager Application provides users with the ability to browse through their device's storage and access files and directories and offers a variety of file management options such as the ability to copy, move, and delete single or multiple files at a time, rename, and open files as well. The app uses Java programming language and the Android operating system's APIs and libraries to access and manipulate files and folders on the device's storage. The app features a user interface created using the Android XML layout design, allowing users to interact with the app's features such as copying, deleting, and moving files. Additionally, the app uses RecyclerView and RecyclerView.Adapter to display and manage lists of files and folders, and it incorporates features such as multi-selection and long-click selection, which provide users an easier and convenient way to use the app. Overall, the app uses a combination of technologies to provide users with a streamlined and easy-to-use interface for managing files on their Android devices. The code utilizes the File class and various Android classes, including DialogUtil, RecyclerAdaptor, and FileManagerModel, to create the user interface and handle file actions. The MainActivity class implements the RecyclerItemInterface, which defines methods to handle item clicks and long clicks. The code also requests and checks for storage permissions before loading files. The application uses various Android UI elements, including a PopupMenu and a menu layout, to provide users with an intuitive interface for managing their files. To conclude, my File Manager Application utilizes a variety of components, including user interfaces, file system APIs, and storage permissions to provide users with an easy and efficient way to go about file management.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE" />

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:requestLegacyExternalStorage="true"
        android:supportsRtl="true"
        android:theme="@style/Theme.SimpleFileManager"
        tools:targetApi="31">
        <provider
            android:name="androidx.core.content.FileProvider"
            android:authorities="${applicationId}.provider"
            android:exported="false"
            android:grantUriPermissions="true">
            <meta-data
                android:name="android.support.FILE_PROVIDER_PATHS"
                android:resource="@xml/provider" />
        </provider>
        <activity
            android:name=".activities.MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
```

Code Snippets of Main Activity with OS low level functions

```java
import java.io.File;
import java.util.ArrayList;

public class MainActivity extends AppCompatActivity implements RecyclerItemInterface {
    private ActivityMainBinding binding;
    private RecyclerAdaptor mAdaptor;

    private boolean isMultiSelectionOn = false;
    private boolean isCopiedEnable = false;
    private boolean isMoveEnable = false;
    private ArrayList<FileManagerModel> selectedItemList;
    private ArrayList<String> filePathList;


    private DialogUtil mDialogUtil;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());
        selectedItemList = new ArrayList<>();
        mDialogUtil = new DialogUtil(this);
        initRecycler();
        if (isPermissionGranted()) {
            loadRootFiles();
        } else {
            requestForPermissions();
        }
        binding.btnDelete.setOnClickListener(view -> {
            dismissSelection();
            new ActionExecuter(ActionEnum.Delete, this, "Deleting file...", new ResultCallBack() {
                @Override
                public void onSuccess() {
                    Toast.makeText(MainActivity.this, "Files Delete successful", Toast.LENGTH_SHORT).show();
                    loadCurrentFolder();
```

```java
    private void loadRootFiles() {
        filePathList.add(Environment.getExternalStorageDirectory().getAbsolutePath());
        new FileUtil(this, mFiles -> mAdaptor.submitList(mFiles)).execute(new String[]{Environment.getExternalStorageDirect

    }

    private void requestForPermissions() {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
            try {
                Intent intent = new Intent(Settings.ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION);
                intent.addCategory("android.intent.category.DEFAULT");
                intent.setData(Uri.parse(String.format("package:%s", getApplicationContext().getPackageName())));
                startActivityForResult(intent, 222);
            } catch (Exception e) {
                Intent intent = new Intent();
                intent.setAction(Settings.ACTION_MANAGE_ALL_FILES_ACCESS_PERMISSION);
                startActivityForResult(intent, 222);
            }
        } else {
            ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE}, 111);
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode == 111) {
            if (isPermissionGranted()) {
                loadRootFiles();
            } else {
                finishAffinity();
            }
        }
    }
```

```java
private String getFileExtension(String fileName) {
    String extension = "";

    int i = fileName.lastIndexOf('.');
    if (i > 0) {
        extension = fileName.substring(i + 1);
    }
    return extension;
}

private void moveItem(FileManagerModel model, int position) {
    if (!selectedItemList.contains(model)) {
        selectedItemList.add(model);
    }
    enableMove();
    selectedItemList.add(model);
}

private void doMove() {
    new ActionExecuter(ActionEnum.Move, this, "Moving files...", new ResultCallBack() {
        @Override
        public void onSuccess() {
            Toast.makeText(MainActivity.this, "Moved successful", Toast.LENGTH_SHORT).show();
            loadCurrentFolder();
            disableActions();
        }

        @Override
        public void onFail() {
            Toast.makeText(MainActivity.this, "Fail to move", Toast.LENGTH_SHORT).show();
            disableActions();
        }
    }, filePathList.get(filePathList.size() - 1)).execute(new ArrayList[]{selectedItemList});
}
```

```java
310     private void doCopy() {
311         new ActionExecuter(ActionEnum.Copy, this, "Coping file...", new ResultCallBack() {
312             @Override
313             public void onSuccess() {
314                 Toast.makeText(MainActivity.this, "Copied successful", Toast.LENGTH_SHORT).show();
315                 loadCurrentFolder();
316                 disableActions();
317             }
318
319             @Override
320             public void onFail() {
321                 Toast.makeText(MainActivity.this, "Fail to copy", Toast.LENGTH_SHORT).show();
322                 disableActions();
323             }
324         }, filePathList.get(filePathList.size() - 1)).execute(new ArrayList[]{selectedItemList});
325     }
326
327     private void deleteItem(FileManagerModel model, int position) {
328         disableActions();
329         selectedItemList.add(model);
330         new ActionExecuter(ActionEnum.Delete, this, "Deleting file...", new ResultCallBack() {
331             @Override
332             public void onSuccess() {
333                 Toast.makeText(MainActivity.this, "Delete successful", Toast.LENGTH_SHORT).show();
334                 mAdaptor.mList.remove(position);
335                 mAdaptor.notifyItemRemoved(position);
336                 mAdaptor.notifyItemRangeChanged(position, mAdaptor.mList.size());
337                 disableActions();
338             }
339
340             @Override
341             public void onFail() {
342                 Toast.makeText(MainActivity.this, "Fail to delete", Toast.LENGTH_SHORT).show();
343                 disableActions();
344             }
345         }).execute(new ArrayList[]{selectedItemList});
346     }
```