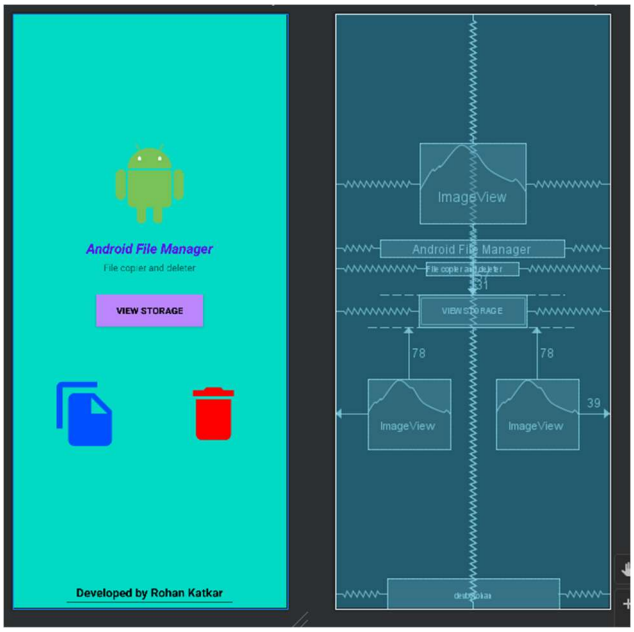
Rohan Katkar

CSC 4320

Project 2 Report

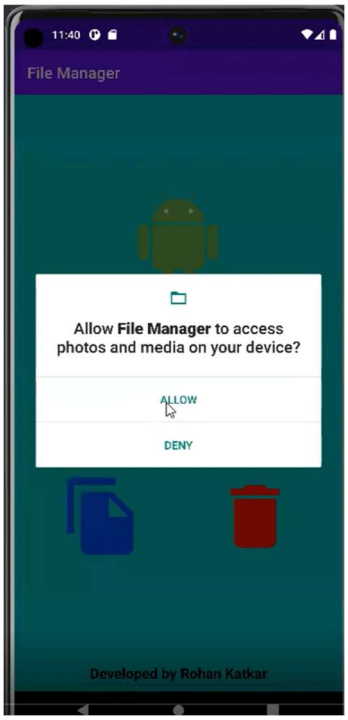
For project 2, I created a file manager using Android Studios. The two languages I used in this project are XML and Java. I used XML for the design layout of each screen, and I used Java to code the back-end of each functionalities. The two functions I focused on in this project are the copy and delete functions. My aim with this app is to make accessing files and using the functionalities as simple as possible. I used a total of 4 XML files and 3 Java files. The first XML file is for the home page, which has the following: file manager name, “view storage” button, my name, and images/color properties to create a comfortable layout for the best user experience. After clicking the “view storage” button, it will prompt you to the second XML file. This file is the layout of all the folders. This XML file contains icons for the folders and files. Once you click on the folder, it will lead you to the appropriate files associated with the folder. If there are no files, it will show a text in the center of the page that will say “No Files Found!”. I used a RecyclerView for the file layout. I used this instead of a linear layout as there are more advanced functionalities for each recycler view that you can customize by each folder or file. This is included in the recyclerview XML file, which is the third XML file used. The fourth XML file is the AndroidManifest file, which contains the prompt for user permissions. Later in this report I will explain each XML file’s functionalities in detail. The first Java file I used is the MainActivity file, which contains the functionalities for the view storage button, which is in the home page. This also contains the methods that deal with the user’s permissions. The second Java file I use is the File List file, which contains the backend for the list of files and calls the adapter file. The third Java file is the code for the adapter file, which holds all the main functionalities for this file manager.

The code in the activity main XML file represents a simple Android file manager app, which allows users to view their storage, copy and delete files. The app interface is built using a Relative Layout, which is a flexible layout that allows views to be positioned relative to each other or to the parent container. The main functionalities of the app are represented by three main components, being TextView which is a header text that displays the name of the app ("Android File Manager"). It is positioned at the top of the screen using constraints, and styled using various attributes such as font size, style, and color. The other component is the button function, which is a button that allows users to view their storage. The third component is ImageView. There are three ImageViews in this. The first ImageView displays the Android logo, while the other two display icons for file copying and deletion. The ImageView elements are positioned using constraints, and their content is set using the app:srcCompat attribute. Additionally, the code contains an EditText element that displays my name ("Developed by Rohan Katkar") and is positioned at the bottom of the screen.



In this image above, you will see the design and where each element is placed.

Once the button is clicked, it will prompt the user with a pop up that asks the user for permission to external file storage, as shown in this image:



This is where the main activity java file comes in. The MainActivity class contains the code for the main activity of the application, which displays a button for the user to view their device's storage. The checkPermission() method checks whether the application has permission to access the device's external storage, and returns true if it does. If the application does not have permission, it returns false. The requestPermission() method is called if the application does not

have permission to access the device's external storage. It requests permission from the user using the ActivityCompat.requestPermissions() method. This interacts with the operating system of the device to request and manage permissions for accessing external storage and provides a user interface for viewing and managing files. The code uses the Environment.getExternalStorageDirectory().getPath() function to retrieve the path to the external storage directory of the device. This is an operating system function that allows the app to access the external storage of the device. The PackageManager.PERMISSION_GRANTED` and Manifest.permission.WRITE_EXTERNAL_STORAGE are also related to the operating system permissions in Android. The code checks if the app has the necessary permission to access the external storage and requests permission if it is not granted.

Here is the source code within the class that uses the operating system related functions:

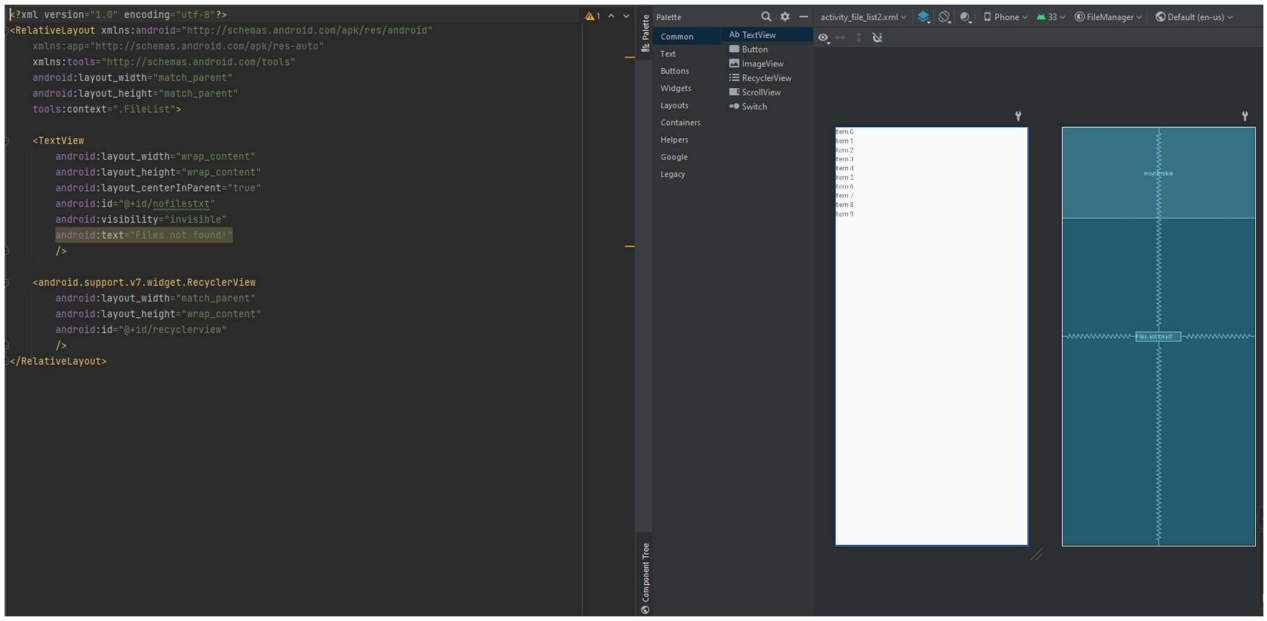
```
private boolean checkPermission() {
        int result = ContextCompat.checkSelfPermission(MainActivity.this,
Manifest.permission.WRITE_EXTERNAL_STORAGE);
        if(result == PackageManager.PERMISSION_GRANTED){
            return true;
    }else
        return false;


}
    //method for requesting permission

    private void requestPermission(){

if(ActivityCompat.shouldShowRequestPermissionRationale(MainActivity.this,
Manifest.permission.WRITE_EXTERNAL_STORAGE)){
            Toast.makeText(MainActivity.this,"You need permission! Please
enable in settings.",Toast.LENGTH_SHORT).show();
        } else
        ActivityCompat.requestPermissions(MainActivity.this,new String[]
{Manifest.permission.WRITE_EXTERNAL_STORAGE},111);


    }
```

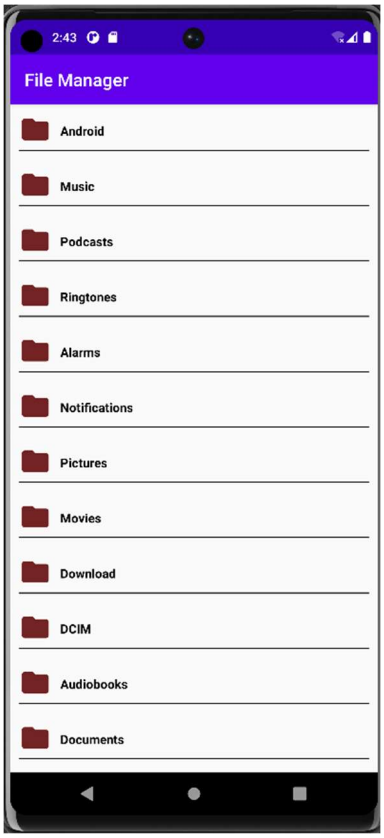
The Manifest XML file specifies the permissions required by the application to access the device's external storage. It also declares the activities in the application and sets the theme for the user interface. The "requestLegacyExternalStorage" attribute is set to true to indicate that the application uses the legacy external storage model for backwards compatibility with older devices. Additionally, the manifest includes two custom XML files for data extraction rules and backup rules. The "tools:targetApi" attribute specifies the target API level for the application. The manifest file describes the application's components and permissions, which is needed to run on an Android device.

After the user enables access to external files on the device, the user will then be sent to the activity file list XML file. This is an XML file that defines the layout for the FileList activity in the app. The layout is defined using a RelativeLayout. The layout contains a TextView and a RecyclerView, both of which are given unique IDs for referencing in the code. The TextView is

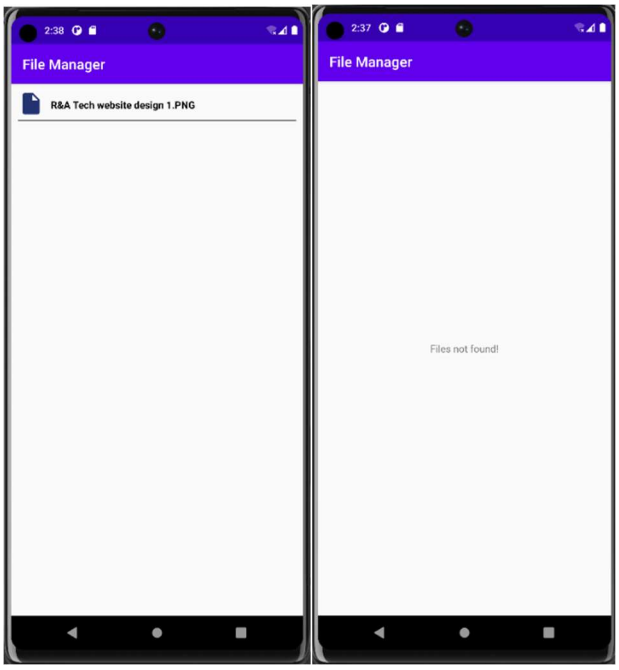
initially set to be invisible, and displays a message that reads "Files not found!" if there are no files to display in RecyclerView. The RecyclerView is a UI component that displays a list of items, and in this case, is used to display a list of files in the app. In terms of the java aspect, the FileList class extends the AppCompatActivity class and provides the logic for displaying a list of files and folders in the specified directory. The onCreate() method is called when the activity is created and inflates the layout file named activity_file_list2.xml using the setContentView() method. It then initializes a RecyclerView and TextView object using findViewById() method to get the corresponding view object based on the ID in the layout file. The method retrieves the path of the directory to be displayed from the intent that started the activity using the getIntent() method and extracting the path using the getStringExtra() method. Next, it uses the path to get a File object that represents the directory using the File class. It then uses the listFiles() method of the File class to get an array of files and folders in the directory. The method checks whether the filesandFolders array is null or empty, and if so, it sets the visibility of the TextView object to visible and returns, since no files or folders are available to display. Otherwise, it sets the TextView's visibility to invisible. Finally, the method sets the layout manager for the RecyclerView to a new instance of LinearLayoutManager and sets the adapter to a new instance of the adapter class, passing in the filesandFolders array as an argument to the constructor. The adapter class handles the logic for displaying each file or folder in RecyclerView. The code uses the File class and its methods to interact with the operating system's file system and retrieve the list of files and folders in a specified directory. It also uses the RecyclerView class to display the files and folders in a list, and the adapter class to provide the views for each item in the list.

You will see the file list design along with its code here:





Above is what the user will see on the app once the viewstorage button has been clicked.



You will notice in the first image above what the user will see if there are files in the folder. In the second image, you will see what the user will see if there are no files in the opened folder.

Below you'll see the source code for the operating system related functions:

```
public class FileList extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_file_list2);

        RecyclerView recyclerView = findViewById(R.id.recyclerview);
        TextView nofilestxt = findViewById(R.id.nofilestxt);

        String path = getIntent().getStringExtra("path");
        //provides all files and folders in the root
        File root = new File(path);
        File[] filesandFolders = root.listFiles();

        //if else statement, files not found text will be visible if no files
are found
        if(filesandFolders==null || filesandFolders.length==0){
            nofilestxt.setVisibility(View.VISIBLE);
            return;
        }

        nofilestxt.setVisibility(View.INVISIBLE);

        //calls adapter file for recyclerview

        recyclerView.setLayoutManager(new LinearLayoutManager(this));
        recyclerView.setAdapter(new
adapter(getApplicationContext(),filesandFolders));

    }
}
```

Before I get into the adapter class, the recycler view XML file creates a relative layout that contains an ImageView, a TextView, and a LinearLayout for the displayed files. The ImageView displays the icon for the files. The TextView displays the name of a file or folder. The LinearLayout is used to create a horizontal line beneath the filename text. Overall, this layout is designed to display a list of files or folders, with each item consisting of an icon and a name.

The adapter file is an implementation that displays the list of files and folders in a directory, allows users to navigate through folders, and perform actions like copying and deleting files. The app uses various built-in functions and APIs designed for the Android operating system. Some of the important functions used in the code are RecyclerView, File class, Intent, PopupMenu, Toast, Environment, FileInputStream, and FileOutputStream. The RecyclerView function is a UI component that displays a list of items.

```
public class adapter extends RecyclerView.Adapter<adapter.Viewholder>{

    Context context;
    File[] filesandFolders;
```

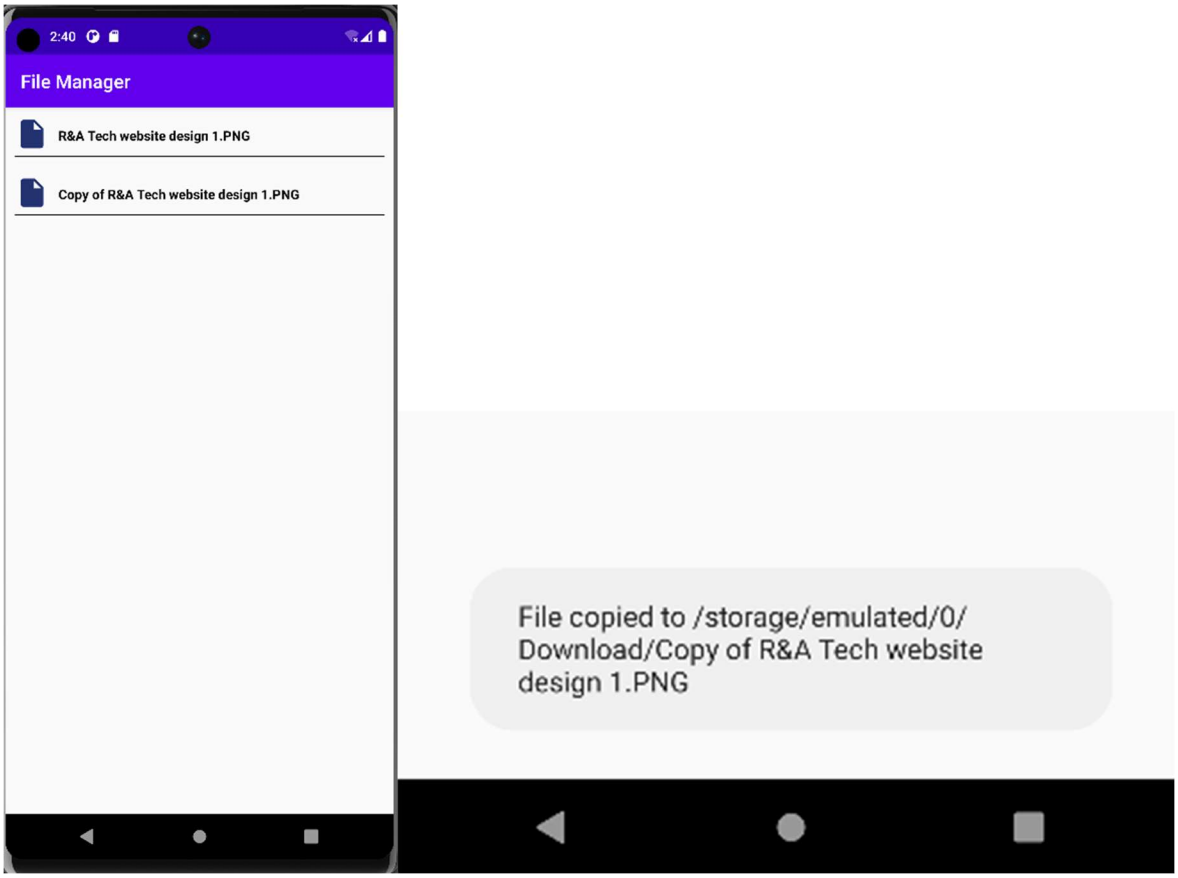
The File function is a Java class that represents a file or directory in the file system. It is used to manipulate files and folders in the app using a constructor and defined methods. I defined a method “onCreateViewHolder” which lets the recycler know of the item. I also defined a method “onBindViewHolder” which defines which file to take based on the position “j” which contains an if else statement that states that if the folder is in the directory, it will set the folder icon, and if it is a file, it will set the driver file icon. I also defined another method within this method as “onClick”, which has an if else statement that opens the file if it is a file. Within the onClick function I use the Intent function, which is a messaging object that enables communication between components in the app or between the app and other apps. It is used to open files in the app and launch activities. I also set a long click listener on a view holder, which creates a popupmenu.

```
viewholder.itemView.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if(selectedFile.isDirectory()){
            Intent intent = new Intent(context, FileList.class);
            String path = selectedFile.getAbsolutePath();
            intent.putExtra("path", path);
            intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            context.startActivity(intent);
        }else {
            //if it is file, this opens the file
            try {
                Intent intent = new Intent();
                intent.setAction(android.content.Intent.ACTION_VIEW);
                String type = "image/*";

intent.setDataAndType(Uri.parse(selectedFile.getAbsolutePath()), type);
                intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                context.startActivity(intent);
            }catch (Exception e){
                Toast.makeText(context.getApplicationContext(), "Unable to
open file!", Toast.LENGTH_SHORT).show();
            }
        }
    }
});
```

The PopupMenu function is a UI component that displays a list of options in a popup view. It creates a floating menu once long clicked, which takes two parameters, being a context and a view which is used to determine the location of the menu. It is used to show options like copy and delete when a user long clicks on a file or folder. These two options are the main functions of this file manager.

Above you will see an image of the popupmenu once the item is long-clicked.

I use the getMenu function the return the menu object associated with the popup menu. I use the add function to add the copy and delete options to the menu. I use the setOnMentItemClickListener function to set a listener to be called when the user selects a menu item, which takes a PopupMenu.OnMenuItemClickListener object as a parameter. Within this function I set an onMenuItemClick method which I call to check which item was selected. This is where I begin the copy function, in which I create a new file in the same folder with a new name using selectedFile.getParentFile to create a new file with “Copy of” before it. To copy the contents of the selected file into a new file, I use the FileInputStream function which reads the data from a file as a stream of bytes. I then use the FileOutputStream function to write data to a file as a stream of bytes. I use “byte[] buffer” as an array that holds the data read from the input stream, and the integer “length” to hold the length of the data in a while loop that reads the data from the input stream and writes it to the output stream until there is no more data to read. After that, I use the close() function to close the input and output streams. Finally, I use Toast.maketext to display a short message to the user stating where the file has been copied to.



In the images above, you will see the output of the copy function, in which a copy of the file is created as shown on the first image and a short message is shown on the second image.

In my delete function, I create an if else statement with a Boolean function “deleted”. I use selectedFile.delete() to delete the file. If the file is deleted, I use Toast.makeText to display a message saying “Deleted” to the user, and set the visibility to GONE.

As shown in the image above, once the file has been successfully deleted, a short message is shown at the bottom of the screen saying “Deleted” along with the file (in this case the copy of my png) being gone.

```
viewholder.itemView.setOnLongClickListener(new View.OnLongClickListener() {
    @Override
    public boolean onLongClick(View view) {

        PopupMenu popupMenu = new PopupMenu(context, view);
        popupMenu.getMenu().add("Copy");
        popupMenu.getMenu().add("Delete");
        popupMenu.setOnMenuItemClickListener(new
PopupMenu.OnMenuItemClickListener() {
                                                @Override
                                                public boolean
onMenuItemClick(MenuItem menuItem) {
                                                    if
(menuItem.getTitle().equals("Copy")) {
                                                        //copies menu item

                                                        // create a new file
in the same folder with a new name
                                                        File newFile = new
File(selectedFile.getParentFile(), "Copy of " + selectedFile.getName());
                                                        try {
                                                            // copy the
contents of the selected file to the new file
                                                            InputStream in =
new FileInputStream(selectedFile);
                                                            OutputStream out
= new FileOutputStream(newFile);
```

```
                                                        byte[] buffer =
new byte[1024];
                                                        int length;
                                                        while ((length =
in.read(buffer)) > 0) {

out.write(buffer, 0, length);
                                                        }
                                                        in.close();
                                                        out.close();

Toast.makeText(context.getApplicationContext(), "File copied to " +
newFile.getAbsolutePath(), Toast.LENGTH_SHORT).show();
                                                    } catch (IOException
e) {

Toast.makeText(context.getApplicationContext(), "Unable to copy file!",
Toast.LENGTH_SHORT).show();
                                                    }
                                                }



                                                if
(menuItem.getTitle().equals("Delete")) {
                                                    //deletes menu item
                                                    boolean deleted =
selectedFile.delete();
                                                    if (deleted) {

Toast.makeText(context.getApplicationContext(), "Deleted",
Toast.LENGTH_SHORT).show();

view.setVisibility(view.GONE);
                                                    }
                                                }

                                                return true;
                                            }
                                        });

                    popupMenu.show();
                    return true;

    }
});
```

The Toast function is used in the main activity java file and adapter file, specifically within the onClick method and onLongClick method in the adapter class. This is a UI component that displays a short message to the user. It is used to show status messages like file copied or deleted in the long click method. It is also used to display messages to the user on whether they have permissions or not, along with whether a file is able to open or not. The Environment class is used to get information about the device's storage environment in the adapter class, which is a necessary operating system function. In conclusion, the adapter class acts as a controller that manages the data and updates the view.

In summary, I used a combination of XML and Java to create an android file manager app that allows users to copy and delete their files. The app interface is built using a RelativeLayout, and the Manifest XML file specifies the permissions required by the application to access the device's external storage. The app uses built-in file system APIs to read and manipulate files and folders. It also uses the Android permission system to request permission to access the file system.