

TransFarmers

Solving Transportation issues for farmers.

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology.

in

INFORMATION TECHNOLOGY

under the Supervision of

Ms. Reema Aswani

By

Mr.Yash Chopra(111421)

to



Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled “TransFarmers.”, submitted by Yash Chopra in partial fulfillment for the award of degree of Bachelor of Technology in Information Technology to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:

Supervisor’s Name :

Ms. Reema Aswani

Designation:

AssistantProfessor(Grade 1)

Acknowledgement

It is my privilege to express my sincere regards to my project coordinator, Ms Reema Aswani, for her valuable inputs, able guidance, encouragement, whole-hearted cooperation and constructive criticism throughout the duration of my project.

I deeply express my sincere thanks to the Head of Department **Prof. Dr. RMK Sinha** for encouraging and allowing me to present the project on the topic “TransFarmers“at our department premises for the partial fulfillment of the requirements leading to the award of B.Tech degree.

I take this opportunity to thank all my lecturers who have directly or indirectly helped in my project. I pay my respect and love to my parents and all other family members for their love and encouragement throughout my career. Last but not the least I express my thanks to my friends for their cooperation and support.

Date:

Name of the student

Yash chopra

INDEX

CONTENT	PAGE NO.
1.ABSTRACT	6
2.MOTIVATION	7
3.CHAPTER 1	
INTRODUCTION TO ANDROID	
DEVELOPMENT	8
4.CHAPTER 2	
TransFarmers	48
5.CHAPTER 3	
SCREENSHOTS	62
6.CHAPTER 4	
HARDWARE AND SOFTWARE	
REQUIREMENTS	67
7.CHAPTER 5	
CONCLUSION	68
8.REFERENCES	69

INDEX OF FIGURES

FIGURES	PAGE NO.
1.THE ACTIVITY LIFECYCLE	24
2.WAYS IN WHICH ACTIVITY RETURNS TO USER	29
3.ILLUSTRATION OF HOW INTENT IS DELIVERED TO START ANOTHER ACTIVITY	32
4.A CHOOSER DIALOG	38
5.BLOCK DIAGRAM	52
6.WELCOME SCREEN	62
7.CATEGORY SCREEN	63
8.LOGIN SCREEN	64
9.LIST OF DETAILS BUS SERVICE	65
10.TRANSPORTER DETAILS	66

INDEX OF TABLES

TABLE	PAGE NO.
1. LIFECYCLE CALLBACK METHODS	25

ABSTRACT

The main aim of the application is to convert private transport into public transport.

This application revolve around the farmers and local people who lack a medium of transport or are finding people to share ride and divide the cost of transport.

Now this is an android application in which farmers and transport company representatives will sign up and create their accounts. After they signup, transport company can post their route on application which will be broadcasted to all the users of the application. The farmers will view the route and if the route falls on their way, the both contact each other and decide the meeting point.

After deciding the meeting point and time, farmer and the transport company representative meet and farmer loads the good he want to send to the city.

In this way, profit cut which was previously kept by the mediocre is now kept by farmers themselves.

This application works similarly as the Dijkstra's Algorithm in which the nodes are the farmers and the transport company representatives and vertices are the routes.

MOTIVATION

In the current scenario, our farmers have very less exposure to outside world and lack very basic needs of life out of which one is the means of transport.

They mostly rely on the other transport services which collect their yield from them and supply it to the market.

Chapter 1

INTRODUCTION TO ANDROID DEVELOPMENT

Android provides a rich application framework that allows you to build innovative apps and games for mobile devices in a Java language environment. The documents listed in the left navigation provide details about how to build apps using Android's various APIs.

If you're new to Android development, it's important that you understand the following fundamental concepts about the Android app framework.

1.1 Applications provide multiple entry points

Android apps are built as a combination of distinct components that can be invoked individually. For instance, an individual activity provides a single screen for a user interface, and a service independently performs work in the background.

From one component you can start another component using an intent. You can even start a component in a different app, such an activity in a maps app to show an address. This model provides multiple entry points for a single app and allows any app to behave as a user's "default" for an action that other apps may invoke.

1.2 Applications adapt to different devices

Android provides an adaptive app framework that allows you to provide unique resources for different device configurations. For example, you can create different XML layout files for different screen sizes and the system determines which layout to apply based on the current device's screen size.

You can query the availability of device features at runtime if any app features require specific hardware such as a camera. If necessary, you can also declare features your app requires so app markets such as Google Play Store do not allow installation on devices that do not support that feature.

1.3 Application Fundamentals

Android apps are written in the Java programming language. The Android SDK tools compile your code—along with any data and resource files—into an APK: an Android package, which is an archive file with an .apk suffix. One APK file contains all the contents of an Android app and is the file that Android-powered devices use to install the app.

Once installed on a device, each Android app lives in its own security sandbox:

The Android operating system is a multi-user Linux system in which each app is a different user.

By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app). The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them.

Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps.

By default, every app runs in its own Linux process. Android starts the process when any of the app's components need to be executed, then shuts down the process when it's no longer needed or when the system must recover memory for other apps.

In this way, the Android system implements the principle of least privilege. That is, each app, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an app cannot access parts of the system for which it is not given permission.

However, there are ways for an app to share data with other apps and for an app to access system services:

It's possible to arrange for two apps to share the same Linux user ID, in which case they are able to access each other's files. To conserve system resources, apps with the same user ID can also arrange to run in the same Linux process and share the same VM (the apps must also be signed with the same certificate).

An app can request permission to access device data such as the user's contacts, SMS messages, the mountable storage (SD card), camera, Bluetooth, and more. All app permissions must be granted by the user at install time.

That covers the basics regarding how an Android app exists within the system. The rest of this document introduces you to:

The core framework components that define your app.

The manifest file in which you declare components and required device features for your app.

Resources that are separate from the app code and allow your app to gracefully optimize its behavior for a variety of device configurations.

1.4 Application Components

App components are the essential building blocks of an Android app. Each component is a different point through which the system can enter your app. Not all components are actual entry points for the user and some depend on each other, but each one exists as its own entity and plays a specific role—each one is a unique building block that helps define your app's overall behavior.

There are four different types of app components. Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed.

Here are the four types of app components:

1.4.1 Activities

An activity represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities (if the email app allows it). For example, a camera app can start the activity in the email app that composes new mail, in order for the user to share a picture.

An activity is implemented as a subclass of `Activity` and you can learn more about it in the `ActivitiesDeveloper` guide.

1.4.2 Services

A service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it.

A service is implemented as a subclass of `Service` and you can learn more about it in the `ServicesDeveloper` guide.

1.4.3 Content providers

A content provider manages a shared set of app data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your app can access. Through the content provider, other apps can query or even modify the data (if the content provider allows it). For example, the Android system provides a content provider that manages the user's contact information. As such, any app with the proper permissions can query part of the content provider (such as `ContactsContractData`) to read and write information about a particular person.

Content providers are also useful for reading and writing data that is private to your app and not shared. For example, the Note Pad sample app uses a content provider to save notes.

A content provider is implemented as a subclass of `ContentProvider` and must implement a standard set of APIs that enable other apps to perform transactions. For more information, see the [Content Providers developer guide](#).

1.4.4 Broadcast receivers

A broadcast receiver is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Apps can also initiate broadcasts—for example, to let other apps know that some data has been downloaded to the device and is available for them to use. Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event.

A broadcast receiver is implemented as a subclass of `BroadcastReceiver` and each broadcast is delivered as an `Intent` object. For more information, see the `BroadcastReceiver` class.

A unique aspect of the Android system design is that any app can start another app's component. For example, if you want the user to capture a photo with the device camera, there's probably another app that does that and your app can use it, instead of developing an activity to capture a photo yourself. You don't need to incorporate or even link to the code from the camera app. Instead, you can simply start the activity in the camera app that captures a photo. When complete, the photo is even returned to your app so you can use it. To the user, it seems as if the camera is actually a part of your app.

When the system starts a component, it starts the process for that app (if it's not already running) and instantiates the classes needed for the component. For example, if your app starts the activity in the camera app that captures a photo, that activity runs in the process that belongs to the camera app, not in your app's process. Therefore, unlike apps on most other systems, Android apps don't have a single entry point (there's no `main()` function, for example).

Because the system runs each app in a separate process with file permissions that restrict access to other apps, your app cannot directly activate a component from another app. The Android system, however, can. So, to activate a component in another app, you must deliver a message to the system that specifies your intent to start a particular component. The system then activates the component for you.

1.5 The Manifest File

Before the Android system can start an app component, the system must know that the component exists by reading the app's `AndroidManifest.xml` file (the "manifest" file). Your app must declare all its components in this file, which must be at the root of the app project directory.

The manifest does a number of things in addition to declaring the app's components, such as:

- Identify any user permissions the app requires, such as Internet access or read-access to the user's contacts.
- Declare the minimum API Level required by the app, based on which APIs the app uses.
- Declare hardware and software features used or required by the app, such as a camera, Bluetooth services, or a multi-touch screen.
- API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library.
- And more

1.5.1 Declaring components

The primary task of the manifest is to inform the system about the app's components. For example, a manifest file can declare an activity as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <applicationandroid:icon="@drawable/app_icon.png" ... >
    <activityandroid:name="com.example.project.ExampleActivity"
      android:label="@string/example_label" ... >
    </activity>
    ...
  </application>
</manifest>
```

In the `<application>` element, the `android:icon` attribute points to resources for an icon that identifies the app.

In the `<activity>` element, the `android:name` attribute specifies the fully qualified class name of the Activity subclass and the `android:label` attribute specifies a string to use as the user-visible label for the activity.

You must declare all app components this way:

`<activity>` elements for activities

`<service>` elements for services

`<receiver>` elements for broadcast receivers

`<provider>` elements for content providers

Activities, services, and content providers that you include in your source but do not declare in the manifest are not visible to the system and, consequently, can never run. However, broadcast receivers can be either declared in the manifest or created dynamically in code (as `BroadcastReceiver` objects) and registered with the system by calling `registerReceiver()`.

For more about how to structure the manifest file for your app, see [The AndroidManifest.xml File documentation](#).

1.5.2 Declaring component capabilities

As discussed above, in [Activating Components](#), you can use an `Intent` to start activities, services, and broadcast receivers. You can do so by explicitly naming the target component (using the component class name) in the intent. However, the real power of intents lies in the concept of implicit intents. An implicit intent simply describes the type of action to perform (and, optionally, the data upon which you'd like to perform the action) and allows the system to find a component on the device that can perform the action and start it. If there are multiple components that can perform the action described by the intent, then the user selects which one to use.

The way the system identifies the components that can respond to an intent is by comparing the intent received to the intent filters provided in the manifest file of other apps on the device.

When you declare an activity in your app's manifest, you can optionally include intent filters that declare the capabilities of the activity so it can respond to intents from other apps. You can declare an intent filter for your component by adding an `<intent-filter>` element as a child of the component's declaration element.

For example, if you've built an email app with an activity for composing a new email, you can declare an intent filter to respond to "send" intents (in order to send a new email) like this:

```
<manifest ... >
...
<application ... >
```

```
<activityandroid:name="com.example.project.ComposeEmailActivity">
  <intent-filter>
    <actionandroid:name="android.intent.action.SEND"/>
    <dataandroid:type="*/*/>
    <categoryandroid:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
</application>
</manifest>
```

Then, if another app creates an intent with the `ACTION_SEND` action and pass it to `startActivity()`, the system may start your activity so the user can draft and send an email.

For more about creating intent filters, see the [Intents and Intent Filters](#) document.

1.5.3 Declaring app requirements

There are a variety of devices powered by Android and not all of them provide the same features and capabilities. In order to prevent your app from being installed on devices that lack features needed by your app, it's important that you clearly define a profile for the types of devices your app supports by declaring device and software requirements in your manifest file. Most of these declarations are informational only and the system does not read them, but external services such as Google Play do read them in order to provide filtering for users when they search for apps from their device.

For example, if your app requires a camera and uses APIs introduced in Android 2.1 (API Level 7), you should declare these as requirements in your manifest file like this:

```
<manifest ... >
  <uses-featureandroid:name="android.hardware.camera.any"
    android:required="true"/>
  <uses-sdkandroid:minSdkVersion="7"android:targetSdkVersion="19"/>
  ...
</manifest>
```

Now, devices that do NOT have a camera and have an Android version LOWER than 2.1 cannot install your app from Google Play.

However, you can also declare that your app uses the camera, but does not REQUIRE it. In that case, your app must set the required attribute to "false" and check at runtime whether the device has a camera and disable any camera features as appropriate.

1.6 Activities

An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. The back stack abides to the basic "last in, first out" stack mechanism, so, when the user is done with the current activity and presses the Back button, it is popped from the stack (and destroyed) and the previous activity resumes. (The back stack is discussed more in the Tasks and Back Stack document.)

When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides you the opportunity to perform specific work that's appropriate to that state change. For instance, when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted. These state transitions are all part of the activity lifecycle.

The rest of this document discusses the basics of how to build and use an activity, including a complete discussion of how the activity lifecycle works, so you can properly manage the transition between various activity states.

Creating an Activity

To create an activity, you must create a subclass of Activity (or an existing subclass of it). In your subclass, you need to implement callback methods that the system calls when the activity transitions

between various states of its lifecycle, such as when the activity is being created, stopped, resumed, or destroyed. The two most important callback methods are:

`onCreate()` - You must implement this method. The system calls this when creating your activity. Within your implementation, you should initialize the essential components of your activity. Most importantly, this is where you must call `setContentView()` to define the layout for the activity's user interface.

`onPause()` - The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back).

There are several other lifecycle callback methods that you should use in order to provide a fluid user experience between activities and handle unexpected interruptions that cause your activity to be stopped and even destroyed. All of the lifecycle callback methods are discussed later, in the section about Managing the Activity Lifecycle.

Implementing a user interface

The user interface for an activity is provided by a hierarchy of views—objects derived from the `View` class. Each view controls a particular rectangular space within the activity's window and can respond to user interaction. For example, a view might be a button that initiates an action when the user touches it.

Android provides a number of ready-made views that you can use to design and organize your layout. "Widgets" are views that provide a visual (and interactive) elements for the screen, such as a button, text field, checkbox, or just an image. "Layouts" are views derived from `ViewGroup` that provide a unique layout model for its child views, such as a linear layout, a grid layout, or relative layout. You can also subclass the `View` and `ViewGroup` classes (or existing subclasses) to create your own widgets and layouts and apply them to your activity layout.

The most common way to define a layout using views is with an XML layout file saved in your application resources. This way, you can maintain the design of your user interface separately from the source code that defines the activity's behavior. You can set the layout as the UI for your activity with `setContentView()`, passing the resource ID for the layout. However, you can also create new Views in your activity code and build a view hierarchy by inserting new Views into a `ViewGroup`, then use that layout by passing the root `ViewGroup` to `setContentView()`.

For information about creating a user interface, see the User Interface documentation.

Declaring the activity in the manifest

You must declare your activity in the manifest file in order for it to be accessible to the system. To declare your activity, open your manifest file and add an `<activity>` element as a child of the `<application>` element. For example:

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
  ...
</manifest >
```

There are several other attributes that you can include in this element, to define properties such as the label for the activity, an icon for the activity, or a theme to style the activity's UI. The `android:name` attribute is the only required attribute—it specifies the class name of the activity. Once you publish your application, you should not change this name, because if you do, you might break some functionality, such as application shortcuts (read the blog post, [Things That Cannot Change](#)).

See the `<activity>` element reference for more information about declaring your activity in the manifest.

Using intent filters

An `<activity>` element can also specify various intent filters—using the `<intent-filter>` element—in order to declare how other application components may activate it.

When you create a new application using the Android SDK tools, the stub activity that's created for you automatically includes an intent filter that declares the activity responds to the "main" action and should be placed in the "launcher" category. The intent filter looks like this:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

The `<action>` element specifies that this is the "main" entry point to the application. The `<category>` element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).

If you intend for your application to be self-contained and not allow other applications to activate its activities, then you don't need any other intent filters. Only one activity should have the "main" action and "launcher" category, as in the previous example. Activities that you don't want to make available to other applications should have no intent filters and you can start them yourself using explicit intents (as discussed in the following section).

However, if you want your activity to respond to implicit intents that are delivered from other applications (and your own), then you must define additional intent filters for your activity. For each type of intent to which you want to respond, you must include an `<intent-filter>` that includes an `<action>` element and, optionally, a `<category>` element and/or a `<data>` element. These elements specify the type of intent to which your activity can respond.

For more information about how your activities can respond to intents, see the [Intents and Intent Filters](#) document.

Starting an Activity

You can start another activity by calling `startActivity()`, passing it an `Intent` that describes the activity you want to start. The intent specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application). An intent can also carry small amounts of data to be used by the activity that is started.

When working within your own application, you'll often need to simply launch a known activity. You can do so by creating an intent that explicitly defines the activity you want to start, using the class name. For example, here's how one activity starts another activity named `SignInActivity`:

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

However, your application might also want to perform some action, such as send an email, text message, or status update, using data from your activity. In this case, your application might not have its own activities to perform such actions, so you can instead leverage the activities provided by other applications on the device, which can perform the actions for you. This is where intents are really valuable—you can create an intent that describes an action you want to perform and the system launches the appropriate activity from another application. If there are multiple activities that can handle the intent, then the user can select which one to use. For example, if you want to allow the user to send an email message, you can create the following intent:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

The `EXTRA_EMAIL` extra added to the intent is a string array of email addresses to which the email should be sent. When an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form. In this situation, the email application's activity starts and when the user is done, your activity resumes.

Starting an activity for a result

Sometimes, you might want to receive a result from the activity that you start. In that case, start the activity by calling `startActivityForResult()` (instead of `startActivity()`). To then receive the result from the subsequent activity, implement the `onActivityResult()` callback method. When the subsequent activity is done, it returns a result in an `Intent` to your `onActivityResult()` method.

For example, perhaps you want the user to pick one of their contacts, so your activity can do something with the information in that contact. Here's how you can create such an intent and handle the result:

```
private void pickContact() {
    // Create an intent to "pick" a contact, as defined by the content provider URI
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST);
}
```

```

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode == PICK_CONTACT_REQUEST) {
        // Perform a query to the contact's content provider for the contact's name
        Cursor cursor = getContentResolver().query(data.getData(),
            new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not empty
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Do something with the selected contact's name...
        }
    }
}

```

This example shows the basic logic you should use in your `onActivityResult()` method in order to handle an activity result. The first condition checks whether the request was successful—if it was, then the `resultCode` will be `RESULT_OK`—and whether the request to which this result is responding is known—in this case, the `requestCode` matches the second parameter sent with `startActivityForResult()`. From there, the code handles the activity result by querying the data returned in an `Intent` (the `data` parameter).

What happens is, a `ContentResolver` performs a query against a content provider, which returns a `Cursor` that allows the queried data to be read. For more information, see the [Content Providers](#) document.

For more information about using intents, see the [Intents and Intent Filters](#) document.

Shutting Down an Activity

You can shut down an activity by calling its `finish()` method. You can also shut down a separate activity that you previously started by calling `finishActivity()`.

Note: In most cases, you should not explicitly finish an activity using these methods. As discussed in the following section about the activity lifecycle, the Android system manages the life of an activity for you, so you do not need to finish your own activities. Calling these methods could adversely affect the expected user experience and should only be used when you absolutely do not want the user to return to this instance of the activity.

Managing the Activity Lifecycle

Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application. The lifecycle of an activity is directly affected by its association with other activities, its task and back stack.

An activity can exist in essentially three states:

- Resumed - The activity is in the foreground of the screen and has user focus. (This state is also sometimes referred to as "running".)
- Paused - Another activity is in the foreground and has focus, but this one is still visible. That is, another activity is visible on top of this one and that activity is partially transparent or doesn't cover the entire screen. A paused activity is completely alive (the Activity object is retained in memory, it maintains all state and member information, and remains attached to the window manager), but can be killed by the system in extremely low memory situations.
- Stopped - The activity is completely obscured by another activity (the activity is now in the "background"). A stopped activity is also still alive (the Activity object is retained in memory, it maintains all state and member information, but is not attached to the window manager). However, it is no longer visible to the user and it can be killed by the system when memory is needed elsewhere.

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its `finish()` method), or simply killing its process. When the activity is opened again (after being finished or killed), it must be created all over.

Implementing the lifecycle callbacks

When an activity transitions into and out of the different states described above, it is notified through various callback methods. All of the callback methods are hooks that you can override to do appropriate work when the state of your activity changes. The following skeleton activity includes each of the fundamental lifecycle methods:

```

public class ExampleActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // The activity is being created.
    }
    @Override
    protected void onStart() {
        super.onStart();
        // The activity is about to become visible.
    }
    @Override
    protected void onResume() {
        super.onResume();
        // The activity has become visible (it is now "resumed").
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Another activity is taking focus (this activity is about to be "paused").
    }
    @Override
    protected void onStop() {
        super.onStop();
        // The activity is no longer visible (it is now "stopped")
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // The activity is about to be destroyed.
    }
}

```

Note: Your implementation of these lifecycle methods must always call the superclass implementation before doing any work, as shown in the examples above.

Taken together, these methods define the entire lifecycle of an activity. By implementing these methods, you can monitor three nested loops in the activity lifecycle:

The entire lifetime of an activity happens between the call to `onCreate()` and the call to `onDestroy()`. Your activity should perform setup of "global" state (such as defining layout) in `onCreate()`, and release all remaining resources in `onDestroy()`. For example, if your activity has a thread running in the background to download data from the network, it might create that thread in `onCreate()` and then stop the thread in `onDestroy()`.

The visible lifetime of an activity happens between the call to `onStart()` and the call to `onStop()`. During this time, the user can see the activity on-screen and interact with it. For example, `onStop()` is called when a new activity starts and this one is no longer visible. Between these two methods, you can maintain resources that are needed to show the activity to the user. For example, you can register a `BroadcastReceiver` in `onStart()` to monitor changes that impact your UI, and unregister it in `onStop()` when the user can no longer see what you are displaying. The system might call `onStart()` and `onStop()` multiple times during the entire lifetime of the activity, as the activity alternates between being visible and hidden to the user.

The foreground lifetime of an activity happens between the call to `onResume()` and the call to `onPause()`. During this time, the activity is in front of all other activities on screen and has user input focus. An activity can frequently transition in and out of the foreground—for example, `onPause()` is called when the device goes to sleep or when a dialog appears. Because this state can transition often, the code in these two methods should be fairly lightweight in order to avoid slow transitions that make the user wait.

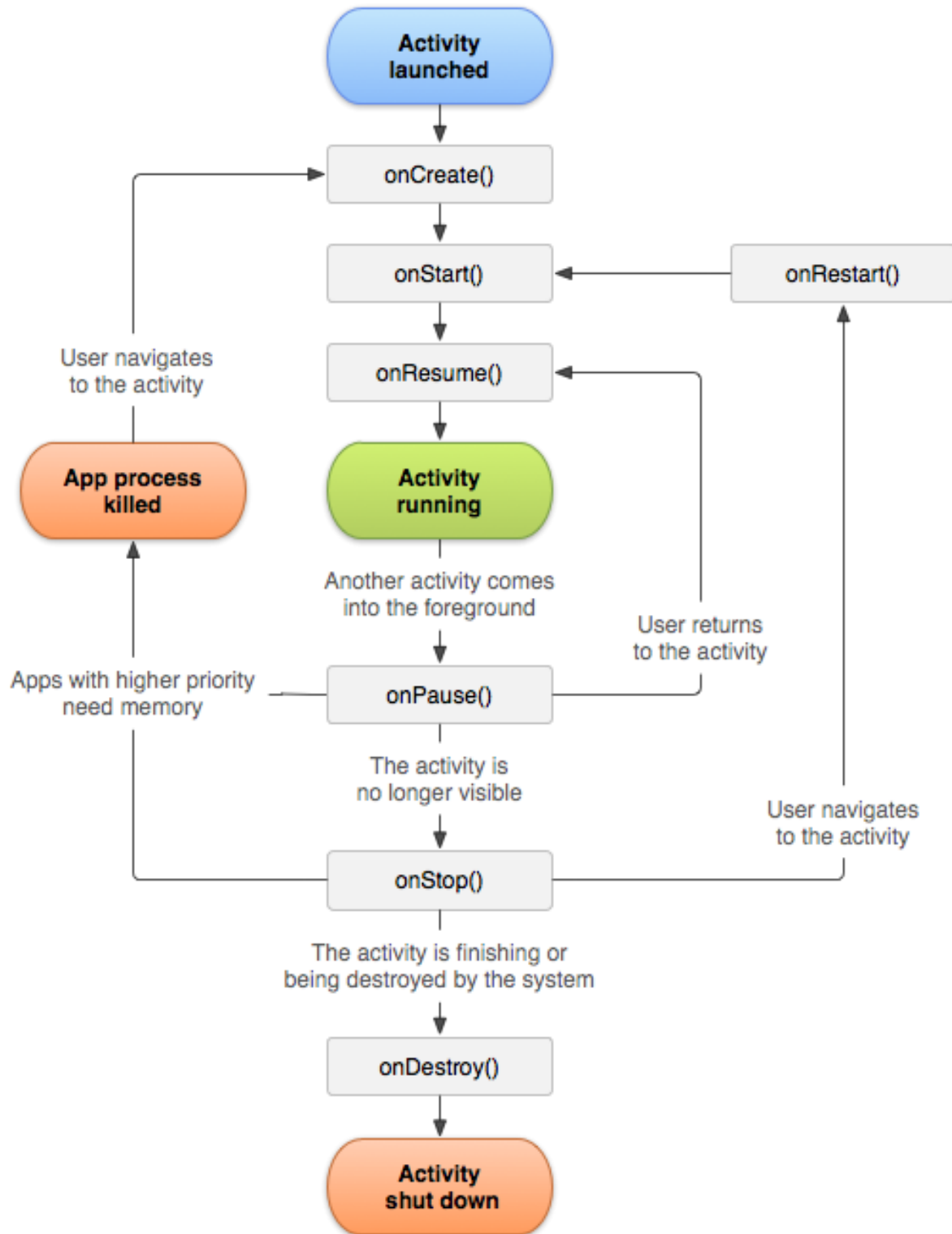


Fig 1. The activity lifecycle

Figure 1 illustrates these loops and the paths an activity might take between states. The rectangles represent the callback methods you can implement to perform operations when the activity transitions between states.

The same lifecycle callback methods are listed in table 1, which describes each of the callback methods in more detail and locates each one within the activity's overall lifecycle, including whether the system can kill the activity after the callback method completes.

Method	Description	Killable after?	Next
<u>onCreate()</u>	Called when the activity is first created. This is where you should do all of your normal static set up — create views, bind data to lists, and so on. This method is passed a Bundle object containing the activity's previous state, if that state was captured (see Saving Activity State, later). Always followed by onStart().	No	onStart()
<u>onRestart()</u>	Called after the activity has been stopped, just prior to it being started again. Always followed by onStart()	No	onStart()
<u>onStart()</u>	Called just before the activity becomes visible to the user. Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.	No	onResume() or onStop()
<u>onResume()</u>	Called just before the activity starts interacting with the user. At this point the activity is at the top of the	No	onPause()

	<p>activity stack, with user input going to it.</p> <p>Always followed by onPause().</p>		
<u>onPause()</u>	<p>Called when the system is about to start resuming another activity. This method is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.</p> <p>Followed either by onResume() if the activity returns back to the front, or by onStop() if it becomes invisible to the user.</p>	Yes	onResume() or onStop()
<u>onStop()</u>	<p>Called when the activity is no longer visible to the user. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it.</p> <p>Followed either by onRestart() if the activity is coming back to interact with the user, or by onDestroy() if this activity is going away.</p>	Yes	onRestart() or onDestroy()

<u>onDestroy()</u>	Called before the activity is destroyed. This is the final call that the activity will receive. It could be called either because the activity is finishing (someone called finish() on it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the isFinishing() method.	Yes	nothing
---------------------------	--	------------	----------------

Table 1. A summary of the activity lifecycle's callback methods.

The column labeled "Killable after?" indicates whether or not the system can kill the process hosting the activity at any time after the method returns, without executing another line of the activity's code. Three methods are marked "yes": (onPause(), onStop(), and onDestroy()). Because onPause() is the first of the three, once the activity is created, onPause() is the last method that's guaranteed to be called before the process can be killed—if the system must recover memory in an emergency, then onStop() and onDestroy() might not be called. Therefore, you should use onPause() to write crucial persistent data (such as user edits) to storage. However, you should be selective about what information must be retained during onPause(), because any blocking procedures in this method block the transition to the next activity and slow the user experience.

Methods that are marked "No" in the Killable column protect the process hosting the activity from being killed from the moment they are called. Thus, an activity is killable from the time `onPause()` returns to the time `onResume()` is called. It will not again be killable until `onPause()` is again called and returns.

Note: An activity that's not technically "killable" by this definition in table 1 might still be killed by the system—but that would happen only in extreme circumstances when there is no other recourse. When an activity might be killed is discussed more in the Processes and Threading document.

Saving activity state

The introduction to *Managing the Activity Lifecycle* briefly mentions that when an activity is paused or stopped, the state of the activity is retained. This is true because the Activity object is still held in memory when it is paused or stopped—all information about its members and current state is still alive. Thus, any changes the user made within the activity are retained so that when the activity returns to the foreground (when it "resumes"), those changes are still there.

However, when the system destroys an activity in order to recover memory, the Activity object is destroyed, so the system cannot simply resume it with its state intact. Instead, the system must recreate the Activity object if the user navigates back to it. Yet, the user is unaware that the system destroyed the activity and recreated it and, thus, probably expects the activity to be exactly as it was. In this situation, you can ensure that important information about the activity state is preserved by implementing an additional callback method that allows you to save information about the state of your activity: `onSaveInstanceState()`.

The system calls `onSaveInstanceState()` before making the activity vulnerable to destruction. The system passes this method a Bundle in which you can save state information about the activity as name-value pairs, using methods such as `putString()` and `putInt()`. Then, if the system kills your application process and the user navigates back to your activity, the system recreates the activity and passes the Bundle to both `onCreate()` and `onRestoreInstanceState()`. Using either of these methods, you can extract your saved state from the Bundle and restore the activity state. If there is no state information to restore, then the Bundle passed to you is null (which is the case when the activity is created for the first time).

Note: There's no guarantee that `onSaveInstanceState()` will be called before your activity is destroyed, because there are cases in which it won't be necessary to save the state (such as when the user leaves your activity using the Back button, because the user is explicitly closing the activity). If the system calls `onSaveInstanceState()`, it does so before `onStop()` and possibly before `onPause()`.

However, even if you do nothing and do not implement `onSaveInstanceState()`, some of the activity state is restored by the Activity class's default implementation of `onSaveInstanceState()`. Specifically, the default implementation calls the corresponding `onSaveInstanceState()` method for every View in the layout, which allows each view to provide information about itself that should be saved. Almost every widget in the Android framework implements this method as appropriate, such that any visible changes to the UI are automatically saved and restored when your activity is recreated. For example, the EditText widget saves any text entered by the user and the CheckBox widget saves whether it's checked or not. The only work required by you is to provide a unique ID (with the `android:id` attribute) for each widget you want to save its state. If a widget does not have an ID, then the system cannot save its state.

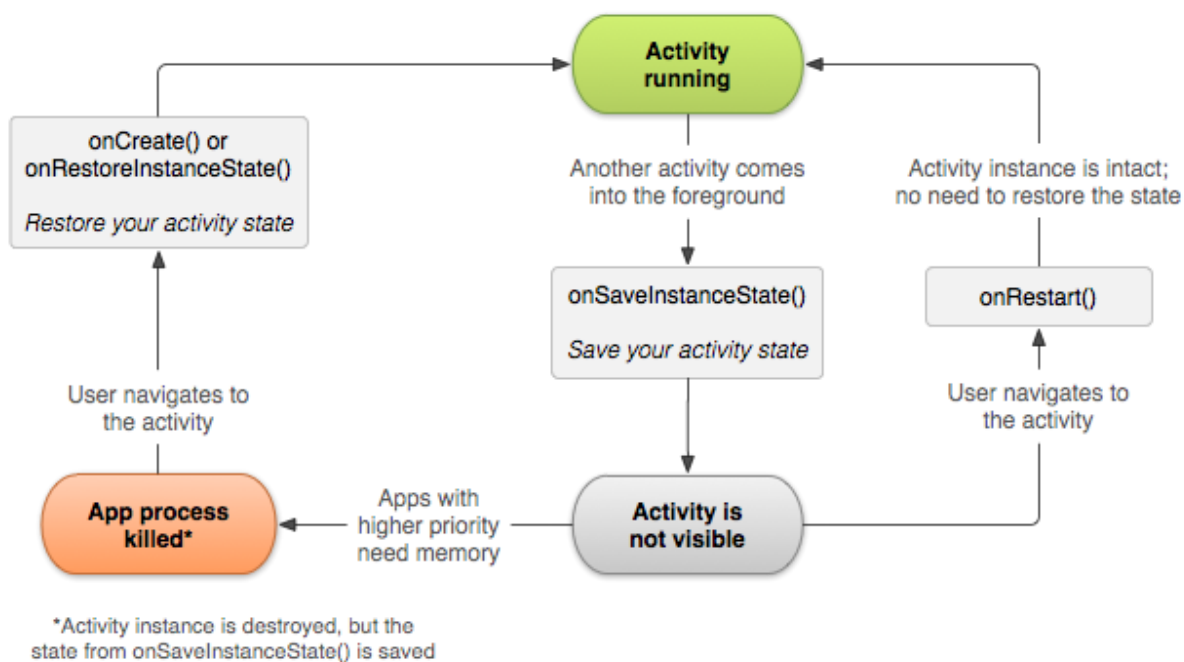


Figure 2. The two ways in which an activity returns to user focus with its state intact: either the activity is destroyed, then recreated and the activity must restore the previously saved state, or the activity is stopped, then resumed and the activity state remains intact.

Although the default implementation of `onSaveInstanceState()` saves useful information about your activity's UI, you still might need to override it to save additional information. For example, you might need to save member values that changed during the activity's life (which might correlate to values restored in the UI, but the members that hold those UI values are not restored, by default).

Because the default implementation of `onSaveInstanceState()` helps save the state of the UI, if you override the method in order to save additional state information, you should always call the superclass implementation of `onSaveInstanceState()` before doing any work. Likewise, you should also call the

superclass implementation of `onRestoreInstanceState()` if you override it, so the default implementation can restore view states.

Note: Because `onSaveInstanceState()` is not guaranteed to be called, you should use it only to record the transient state of the activity (the state of the UI)—you should never use it to store persistent data. Instead, you should use `onPause()` to store persistent data (such as data that should be saved to a database) when the user leaves the activity.

A good way to test your application's ability to restore its state is to simply rotate the device so that the screen orientation changes. When the screen orientation changes, the system destroys and recreates the activity in order to apply alternative resources that might be available for the new screen configuration. For this reason alone, it's very important that your activity completely restores its state when it is recreated, because users regularly rotate the screen while using applications.

Handling configuration changes

Some device configurations can change during runtime (such as screen orientation, keyboard availability, and language). When such a change occurs, Android recreates the running activity (the system calls `onDestroy()`, then immediately calls `onCreate()`). This behavior is designed to help your application adapt to new configurations by automatically reloading your application with alternative resources that you've provided (such as different layouts for different screen orientations and sizes).

If you properly design your activity to handle a restart due to a screen orientation change and restore the activity state as described above, your application will be more resilient to other unexpected events in the activity lifecycle.

The best way to handle such a restart is to save and restore the state of your activity using `onSaveInstanceState()` and `onRestoreInstanceState()` (or `onCreate()`), as discussed in the previous section.

For more information about configuration changes that happen at runtime and how you can handle them, read the guide to [Handling Runtime Changes](#).

Coordinating activities

When one activity starts another, they both experience lifecycle transitions. The first activity pauses and stops (though, it won't stop if it's still visible in the background), while the other activity is created. In case these activities share data saved to disc or elsewhere, it's important to understand that the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.

- The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process and one is starting the other. Here's the order of operations that occur when Activity A starts Activity B:
- Activity A's `onPause()` method executes.
- Activity B's `onCreate()`, `onStart()`, and `onResume()` methods execute in sequence. (Activity B now has user focus.)

Then, if Activity A is no longer visible on screen, its `onStop()` method executes.

This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another. For example, if you must write to a database when the first activity stops so that the following activity can read it, then you should write to the database during `onPause()` instead of during `onStop()`.

1.6 Intents and Intent Filters

An Intent is a messaging object you can use to request an action from another app component. Although intents facilitate communication between components in several ways, there are three fundamental use-cases:

To start an activity:

An Activity represents a single screen in an app. You can start a new instance of an Activity by passing an Intent to `startActivity()`. The Intent describes the activity to start and carries any necessary data.

If you want to receive a result from the activity when it finishes, call `startActivityForResult()`. Your activity receives the result as a separate Intent object in your activity's `onActivityResult()` callback. For more information, see the Activities guide.

To start a service:

A Service is a component that performs operations in the background without a user interface. You can start a service to perform a one-time operation (such as download a file) by passing an Intent to `startService()`. The Intent describes the service to start and carries any necessary data.

If the service is designed with a client-server interface, you can bind to the service from another component by passing an Intent to `bindService()`. For more information, see the Services guide.

To deliver a broadcast:

A broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. You can deliver a broadcast to other apps by passing an Intent to `sendBroadcast()`, `sendOrderedBroadcast()`, or `sendStickyBroadcast()`.

Intent Types

There are two types of intents:

- Explicit intents specify the component to start by name (the fully-qualified class name). You'll typically use an explicit intent to start a component in your own app, because you know the class name of the activity or service you want to start. For example, start a new activity in response to a user action or start a service to download a file in the background.
- Implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if you want to show the user a location on a map, you can use an implicit intent to request that another capable app show a specified location on a map.

When you create an explicit intent to start an activity or service, the system immediately starts the app component specified in the Intent object.

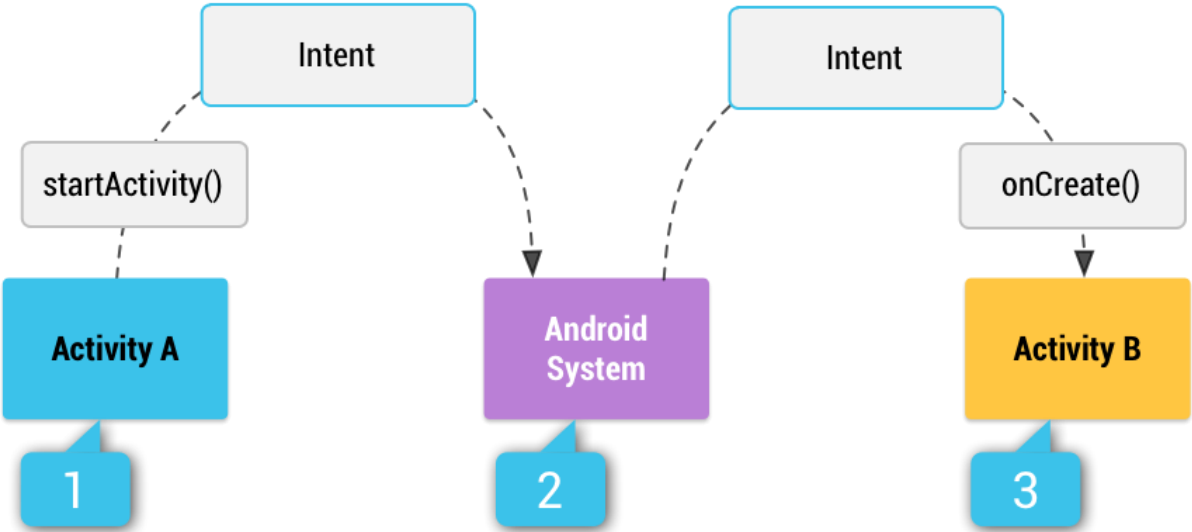


Figure 1. Illustration of how an implicit intent is delivered through the system to start another activity: (1) Activity A creates an Intent with an action description and passes it to startActivity(). (2) The Android System searches all apps for an intent filter that matches the intent. When a match is found, (3) the system starts the matching activity (Activity B) by invoking its onCreate() method and passing it the Intent.

When you create an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the intent filters declared in the manifest file of other apps on the device. If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

An intent filter is an expression in an app's manifest file that specifies the type of intents that the component would like to receive. For instance, by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with a certain kind of intent. Likewise, if you do not declare any intent filters for an activity, then it can be started only with an explicit intent.

Caution: To ensure your app is secure, always use an explicit intent when starting a Service and do not declare intent filters for your services. Using an implicit intent to start a service is a security hazard because you cannot be certain what service will respond to the intent, and the user cannot see which service starts. Beginning with Android 5.0 (API level 21), the system throws an exception if you call bindService() with an implicit intent.

Building an Intent

An Intent object carries information that the Android system uses to determine which component to start (such as the exact component name or component category that should receive the intent), plus information that the recipient component uses in order to properly perform the action (such as the action to take and the data to act upon).

The primary information contained in an Intent is the following:

Component name

The name of the component to start.

This is optional, but it's the critical piece of information that makes an intent explicit, meaning that the intent should be delivered only to the app component defined by the component name. Without a component name, the intent is implicit and the system decides which component should receive the

intent based on the other intent information (such as the action, data, and category—described below). So if you need to start a specific component in your app, you should specify the component name.

Note: When starting a Service, you should always specify the component name. Otherwise, you cannot be certain what service will respond to the intent, and the user cannot see which service starts.

This field of the Intent is a ComponentName object, which you can specify using a fully qualified class name of the target component, including the package name of the app. For example, `com.example.ExampleActivity`. You can set the component name with `setComponent()`, `setClass()`, `setClassName()`, or with the Intent constructor.

Action

A string that specifies the generic action to perform (such as view or pick).

In the case of a broadcast intent, this is the action that took place and is being reported. The action largely determines how the rest of the intent is structured—particularly what is contained in the data and extras.

You can specify your own actions for use by intents within your app (or for use by other apps to invoke components in your app), but you should usually use action constants defined by the Intent class or other framework classes. Here are some common actions for starting an activity:

- **ACTION_VIEW**

Use this action in an intent with `startActivity()` when you have some information that an activity can show to the user, such as a photo to view in a gallery app, or an address to view in a map app.

- **ACTION_SEND**

Also known as the "share" intent, you should use this in an intent with `startActivity()` when you have some data that the user can share through another app, such as an email app or social sharing app.

See the Intent class reference for more constants that define generic actions. Other actions are defined elsewhere in the Android framework, such as in Settings for actions that open specific screens in the system's Settings app.

You can specify the action for an intent with `setAction()` or with an `Intent` constructor.

If you define your own actions, be sure to include your app's package name as a prefix. For example:

```
static final String ACTION_TIMETRAVEL = "com.example.action.TIMETRAVEL";
```

Data

The URI (a `Uri` object) that references the data to be acted on and/or the MIME type of that data. The type of data supplied is generally dictated by the intent's action. For example, if the action is `ACTION_EDIT`, the data should contain the URI of the document to edit.

When creating an intent, it's often important to specify the type of data (its MIME type) in addition to its URI. For example, an activity that's able to display images probably won't be able to play an audio file, even though the URI formats could be similar. So specifying the MIME type of your data helps the Android system find the best component to receive your intent. However, the MIME type can sometimes be inferred from the URI—particularly when the data is a content: URI, which indicates the data is located on the device and controlled by a `ContentProvider`, which makes the data MIME type visible to the system.

To set only the data URI, call `setData()`. To set only the MIME type, call `setType()`. If necessary, you can set both explicitly with `setDataAndType()`.

Caution: If you want to set both the URI and MIME type, do not call `setData()` and `setType()` because they each nullify the value of the other. Always use `setDataAndType()` to set both URI and MIME type.

Category

A string containing additional information about the kind of component that should handle the intent. Any number of category descriptions can be placed in an intent, but most intents do not require a category. Here are some common categories:

- **CATEGORY_BROWSABLE**

The target activity allows itself to be started by a web browser to display data referenced by a link—such as an image or an e-mail message.

- **CATEGORY_LAUNCHER**

The activity is the initial activity of a task and is listed in the system's application launcher.

See the Intent class description for the full list of categories.

You can specify a category with `addCategory()`.

These properties listed above (component name, action, data, and category) represent the defining characteristics of an intent. By reading these properties, the Android system is able to resolve which app component it should start.

However, an intent can carry additional information that does not affect how it is resolved to an app component. An intent can also supply:

Extras

Key-value pairs that carry additional information required to accomplish the requested action. Just as some actions use particular kinds of data URIs, some actions also use particular extras.

You can add extra data with various `putExtra()` methods, each accepting two parameters: the key name and the value. You can also create a `Bundle` object with all the extra data, then insert the `Bundle` in the Intent with `putExtras()`.

For example, when creating an intent to send an email with `ACTION_SEND`, you can specify the "to" recipient with the `EXTRA_EMAIL` key, and specify the "subject" with the `EXTRA_SUBJECT` key.

The Intent class specifies many `EXTRA_*` constants for standardized data types. If you need to declare your own extra keys (for intents that your app receives), be sure to include your app's package name as a prefix. For example:

```
static final String EXTRA_GIGAWATTS = "com.example.EXTRA_GIGAWATTS";
```

Flags

Flags defined in the Intent class that function as metadata for the intent. The flags may instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's launched (for example, whether it belongs in the list of recent activities).

For more information, see the `setFlags()` method.

Example explicit intent

An explicit intent is one that you use to launch a specific app component, such as a particular activity or service in your app. To create an explicit intent, define the component name for the Intent object—all other intent properties are optional.

For example, if you built a service in your app, named `DownloadService`, designed to download a file from the web, you can start it with the following code:

```
// Executed in an Activity, so 'this' is the Context

// The fileUrl is a string URL, such as "http://www.example.com/image.png"
Intent downloadIntent = new Intent(this, DownloadService.class);
downloadIntent.setData(Uri.parse(fileUrl));
startService(downloadIntent);
```

The `Intent(Context, Class)` constructor supplies the app Context and the component a Class object. As such, this intent explicitly starts the `DownloadService` class in the app.

For more information about building and starting a service, see the [Services guide](#).

Example implicit intent

An implicit intent specifies an action that can invoke any app on the device able to perform the action. Using an implicit intent is useful when your app cannot perform the action, but other apps probably can and you'd like the user to pick which app to use.

For example, if you have content you want the user to share with other people, create an intent with the `ACTION_SEND` action and add extras that specify the content to share. When you call `startActivity()` with that intent, the user can pick an app through which to share the content.

Caution: It's possible that a user won't have any apps that handle the implicit intent you send to `startActivity()`. If that happens, the call will fail and your app will crash. To verify that an activity will receive the intent, call `resolveActivity()` on your `Intent` object. If the result is non-null, then there is at least one app that can handle the intent and it's safe to call `startActivity()`. If the result is null, you should not use the intent and, if possible, you should disable the feature that issues the intent.

```
// Create the text message with a string
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType(HTTP.PLAIN_TEXT_TYPE); // "text/plain" MIME type

// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(sendIntent);
}
```

Note: In this case, a URI is not used, but the intent's data type is declared to specify the content carried by the extras.

When `startActivity()` is called, the system examines all of the installed apps to determine which ones can handle this kind of intent (an intent with the `ACTION_SEND` action and that carries "text/plain" data). If there's only one app that can handle it, that app opens immediately and is given the intent. If multiple activities accept the intent, the system displays a dialog so the user can pick which app to use..

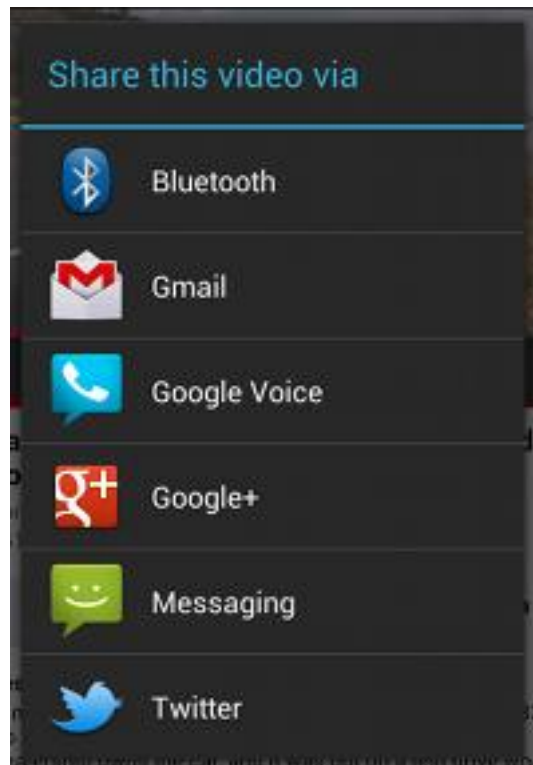


Figure 2. A chooser dialog.

Forcing an app chooser

When there is more than one app that responds to your implicit intent, the user can select which app to use and make that app the default choice for the action. This is nice when performing an action for which the user probably wants to use the same app from now on, such as when opening a web page (users often prefer just one web browser) .

However, if multiple apps can respond to the intent and the user might want to use a different app each time, you should explicitly show a chooser dialog. The chooser dialog asks the user to select which app to use for the action every time (the user cannot select a default app for the action). For example, when your app performs "share" with the `ACTION_SEND` action, users may want to share using a

different app depending on their current situation, so you should always use the chooser dialog, as shown in figure 2.

To show the chooser, create an `Intent` using `createChooser()` and pass it to `startActivity()`. For example:

```
Intent sendIntent = new Intent(Intent.ACTION_SEND);
...
// Always use string resources for UI text.
// This says something like "Share this photo with"
String title = getResources().getString(R.string.chooser_title);
// Create intent to show the chooser dialog
Intent chooser = Intent.createChooser(sendIntent, title);
// Verify the original intent will resolve to at least one activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

This displays a dialog with a list of apps that respond to the intent passed to the `createChooser()` method and uses the supplied text as the dialog title.

Receiving an Implicit Intent

To advertise which implicit intents your app can receive, declare one or more intent filters for each of your app components with an `<intent-filter>` element in your manifest file. Each intent filter specifies the type of intents it accepts based on the intent's action, data, and category. The system will deliver an implicit intent to your app component only if the intent can pass through one of your intent filters.

Note: An explicit intent is always delivered to its target, regardless of any intent filters the component declares.

An app component should declare separate filters for each unique job it can do. For example, one activity in an image gallery app may have two filters: one filter to view an image, and another filter to edit an image. When the activity starts, it inspects the `Intent` and decides how to behave based on the information in the `Intent` (such as to show the editor controls or not).

Each intent filter is defined by an `<intent-filter>` element in the app's manifest file, nested in the corresponding app component (such as an `<activity>` element). Inside the `<intent-filter>`, you can specify the type of intents to accept using one or more of these three elements.

<action>

Declares the intent action accepted, in the name attribute. The value must be the literal string value of an action, not the class constant.

<data>

Declares the type of data accepted, using one or more attributes that specify various aspects of the data URI (scheme, host, port, path, etc.) and MIME type.

<category>

Declares the intent category accepted, in the name attribute. The value must be the literal string value of an action, not the class constant.

Note: In order to receive implicit intents, you must include the `CATEGORY_DEFAULT` category in the intent filter. The methods `startActivity()` and `startActivityForResult()` treat all intents as if they declared the `CATEGORY_DEFAULT` category. If you do not declare this category in your intent filter, no implicit intents will resolve to your activity.

For example, here's an activity declaration with an intent filter to receive an `ACTION_SEND` intent when the data type is text:

```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```

It's okay to create a filter that includes more than one instance of `<action>`, `<data>`, or `<category>`. If you do, you simply need to be certain that the component can handle any and all combinations of those filter elements.

When you want to handle multiple kinds of intents, but only in specific combinations of action, data, and category type, then you need to create multiple intent filters.

Restricting access to components

Using an intent filter is not a secure way to prevent other apps from starting your components. Although intent filters restrict a component to respond to only certain kinds of implicit intents, another app can potentially start your app component by using an explicit intent if the developer determines your component names. If it's important that only your own app is able to start one of your components, set the exported attribute to "false" for that component.

An implicit intent is tested against a filter by comparing the intent to each of the three elements. To be delivered to the component, the intent must pass all three tests. If it fails to match even one of them, the Android system won't deliver the intent to the component. However, because a component may have multiple intent filters, an intent that does not pass through one of a component's filters might make it through on another filter. More information about how the system resolves intents is provided in the section below about Intent Resolution.

Caution: To avoid inadvertently running a different app's Service, always use an explicit intent to start your own service and do not declare intent filters for your service.

Note: For all activities, you must declare your intent filters in the manifest file. However, filters for broadcast receivers can be registered dynamically by calling `registerReceiver()`. You can then unregister the receiver with `unregisterReceiver()`. Doing so allows your app to listen for specific broadcasts during only a specified period of time while your app is running.

Example filters

To better understand some of the intent filter behaviors, look at the following snippet from the manifest file of a social-sharing app.

```
<activity android:name="MainActivity">
  <!-- This activity is the main entry, should appear in app launcher -->
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
<activity android:name="ShareActivity">
  <!-- This activity handles "SEND" actions with text data -->
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
  <!-- This activity also handles "SEND" and "SEND_MULTIPLE" with media data -->
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <action android:name="android.intent.action.SEND_MULTIPLE"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="application/vnd.google.panorama360+jpg"/>
    <data android:mimeType="image/*"/>
    <data android:mimeType="video/*"/>
  </intent-filter>
</activity>
```

The first activity, MainActivity, is the app's main entry point—the activity that opens when the user initially launches the app with the launcher icon:

The `ACTION_MAIN` action indicates this is the main entry point and does not expect any intent data. The `CATEGORY_LAUNCHER` category indicates that this activity's icon should be placed in the system's app launcher. If the `<activity>` element does not specify an icon with `icon`, then the system uses the icon from the `<application>` element.

These two must be paired together in order for the activity to appear in the app launcher.

The second activity, `ShareActivity`, is intended to facilitate sharing text and media content. Although users might enter this activity by navigating to it from `MainActivity`, they can also enter `ShareActivity` directly from another app that issues an implicit intent matching one of the two intent filters.

Note: The MIME type, `application/vnd.google.panorama360+jpg`, is a special data type that specifies panoramic photos, which you can handle with the Google panorama APIs.

Using a Pending Intent

A `PendingIntent` object is a wrapper around an `Intent` object. The primary purpose of a `PendingIntent` is to grant permission to a foreign application to use the contained `Intent` as if it were executed from your app's own process.

Major use cases for a pending intent include:

Declare an intent to be executed when the user performs an action with your Notification (the Android system's `NotificationManager` executes the `Intent`).

Declare an intent to be executed when the user performs an action with your App Widget (the Home screen app executes the `Intent`).

Declare an intent to be executed at a specified time in the future (the Android system's `AlarmManager` executes the `Intent`).

Because each `Intent` object is designed to be handled by a specific type of app component (either an `Activity`, a `Service`, or a `BroadcastReceiver`), so too must a `PendingIntent` be created with the same consideration. When using a pending intent, your app will not execute the intent with a call such as `startActivity()`. You must instead declare the intended component type when you create the `PendingIntent` by calling the respective creator method:

`PendingIntent.getActivity()` for an `Intent` that starts an `Activity`.

`PendingIntent.getService()` for an `Intent` that starts a `Service`.

`PendingIntent.getBroadcast()` for a `Intent` that starts an `BroadcastReceiver`.

Unless your app is receiving pending intents from other apps, the above methods to create a `PendingIntent` are the only `PendingIntent` methods you'll probably ever need.

Each method takes the current app `Context`, the `Intent` you want to wrap, and one or more flags that specify how the intent should be used (such as whether the intent can be used more than once).

More information about using pending intents is provided with the documentation for each of the respective use cases, such as in the Notifications and App Widgets API guides.

Intent Resolution

When the system receives an implicit intent to start an activity, it searches for the best activity for the intent by comparing the intent to intent filters based on three aspects:

- The intent action
- The intent data (both URI and data type)
- The intent category

The following sections describe how an intents are matched to the appropriate component(s) in terms of how the intent filter is declared in an app's manifest file.

Action test

To specify accepted intent actions, an intent filter can declare zero or more `<action>` elements. For example:

```
<intent-filter>
  <action android:name="android.intent.action.EDIT" />
  <action android:name="android.intent.action.VIEW" />
  ...
</intent-filter>
```

To get through this filter, the action specified in the Intent must match one of the actions listed in the filter.

If the filter does not list any actions, there is nothing for an intent to match, so all intents fail the test. However, if an Intent does not specify an action, it will pass the test (as long as the filter contains at least one action).

Category test

To specify accepted intent categories, an intent filter can declare zero or more `<category>` elements. For example:

```
<intent-filter>
```

```
<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
...
</intent-filter>
```

For an intent to pass the category test, every category in the Intent must match a category in the filter. The reverse is not necessary—the intent filter may declare more categories than are specified in the Intent and the Intent will still pass. Therefore, an intent with no categories should always pass this test, regardless of what categories are declared in the filter.

Note: Android automatically applies the the CATEGORY_DEFAULT category to all implicit intents passed to `startActivity()` and `startActivityForResult()`. So if you want your activity to receive implicit intents, it must include a category for "android.intent.category.DEFAULT" in its intent filters (as shown in the previous `<intent-filter>` example).

Data test

To specify accepted intent data, an intent filter can declare zero or more `<data>` elements. For example:

```
<intent-filter>
  <data android:mimeType="video/mpeg" android:scheme="http" ... />
  <data android:mimeType="audio/mpeg" android:scheme="http" ... />
  ...
</intent-filter>
```

Each `<data>` element can specify a URI structure and a data type (MIME media type). There are separate attributes — `scheme`, `host`, `port`, and `path` — for each part of the URI:

```
<scheme>://<host>:<port>/<path>
```

For example:

```
content://com.example.project:200/folder/subfolder/etc
```

In this URI, the scheme is `content`, the host is `com.example.project`, the port is `200`, and the path is `folder/subfolder/etc`.

Each of these attributes is optional in a <data> element, but there are linear dependencies:

- If a scheme is not specified, the host is ignored.
- If a host is not specified, the port is ignored.
- If both the scheme and host are not specified, the path is ignored.
- When the URI in an intent is compared to a URI specification in a filter, it's compared only to the parts of the URI included in the filter. For example:
- If a filter specifies only a scheme, all URIs with that scheme match the filter.
- If a filter specifies a scheme and an authority but no path, all URIs with the same scheme and authority pass the filter, regardless of their paths.
- If a filter specifies a scheme, an authority, and a path, only URIs with the same scheme, authority, and path pass the filter.

Note: A path specification can contain a wildcard asterisk (*) to require only a partial match of the path name.

The data test compares both the URI and the MIME type in the intent to a URI and MIME type specified in the filter. The rules are as follows:

- An intent that contains neither a URI nor a MIME type passes the test only if the filter does not specify any URIs or MIME types.
- An intent that contains a URI but no MIME type (neither explicit nor inferable from the URI) passes the test only if its URI matches the filter's URI format and the filter likewise does not specify a MIME type.
- An intent that contains a MIME type but not a URI passes the test only if the filter lists the same MIME type and does not specify a URI format.
- An intent that contains both a URI and a MIME type (either explicit or inferable from the URI) passes the MIME type part of the test only if that type matches a type listed in the filter. It passes the URI part of the test either if its URI matches a URI in the filter or if it has a content: or file: URI and the filter does not specify a URI. In other words, a component is presumed to support content: and file: data if its filter lists only a MIME type.

This last rule, rule (d), reflects the expectation that components are able to get local data from a file or content provider. Therefore, their filters can list just a data type and do not need to explicitly name the content: and file: schemes. This is a typical case. A <data> element like the following, for example, tells Android that the component can get image data from a content provider and display it:

<intent-filter>

<data android:mimeType="image/*" />

```
...  
</intent-filter>
```

Because most available data is dispensed by content providers, filters that specify a data type but not a URI are perhaps the most common.

Another common configuration is filters with a scheme and a data type. For example, a `<data>` element like the following tells Android that the component can retrieve video data from the network in order to perform the action:

```
<intent-filter>  
  <data android:scheme="http" android:type="video/*" />  
  ...  
</intent-filter>
```

Intent matching

Intents are matched against intent filters not only to discover a target component to activate, but also to discover something about the set of components on the device. For example, the Home app populates the app launcher by finding all the activities with intent filters that specify the `ACTION_MAIN` action and `CATEGORY_LAUNCHER` category.

Your application can use intent matching in a similar way. The `PackageManager` has a set of `query...()` methods that return all components that can accept a particular intent, and a similar series of `resolve...()` methods that determine the best component to respond to an intent. For example, `queryIntentActivities()` returns a list of all activities that can perform the intent passed as an argument, and `queryIntentServices()` returns a similar list of services. Neither method activates the components; they just list the ones that can respond. There's a similar method, `queryBroadcastReceivers()`, for broadcast receivers.

Chapter 2

TransFarmers

What farmers are lacking today is need of transportation as they can't afford to buy a vehicle to carry their yield to cities or other places. Due to this problem, they send their yield through a mediocre who keeps the maximum profit cut and farmer gets nothing.

Another issue is, when trucks are carrying goods from one place to another, after the work is done, the truck returns empty without carrying anything, thus wastage of fuel.

So to overcome this problem, this application is designed, called **TransFarmers**.

Project “TransFarmer ” is aimed at developing an Android application to help in terms of transportation and resource usability of an individual farmer or transporter. It is android based application that can be accessed by all authorized user farmer and transporter. This app claims to add and maintain vehicle records anywhere and anytime. It will provide an online way of accessing the vehicle . It provides online procedure to add new transporter vehicle and make them share to other farmer.

It will act as a complete resource for user to gain access of travelling vehicle at low cost and its source destination cost of transporting and date , details list of all available records .

TransFarmers is an android application which is designed to help farmers in terms of transportation. This application works the similar way as other applications like CAB SHARING SERVICE.

In this application, user can be a farmer and a transport company representative.

Both the user and transport company representative can sign up and create their accounts. Now, what happens is, if there's a truck going from place A to place B, he will post his route and time of journey on the application. Now this detail will be broadcasted to all the other users of the application. After receiving the message containing the route and time of truck, if the farmer falls on the same route, he will be able to send his yield with the help of truck.

Farmer will contact the transport company representative and they will mutually decide a meeting point if the farmer falls in between that route A to B, say at C.

After deciding that meeting point, they both meet and farmer loads the good.

This is the way the application works and it is very similar to CAB SHARING SERVICE.

CAB SHARING SERVICE

City transportation is an increasing problem. Public transportation is cost effective, but do not provide door to door transportation; This makes the far more expensive cabs attractive and scarce.

The paper proposes a location-based Cab-Sharing Service (CSS), which reduces cab fare costs and effectively utilizes available cabs.

HOW CAB SHARING SERVICE WORKS?

- Before using the CSS, a user signs up with the CSS and creates a prepaid user account to pay for the service.
- A registered user sends a cab request in the form of a source–destination pair and an optional validity period of the request. In case no validity period is specified, the cab request is assumed to be valid from the time it is received until some default time limit has been reached.
- The requests are submitted via a mobile device by sending two address lines to a premium SMS service, called Premium Cab–Sharing Service.
- Once received, the Premium Cab–Sharing Service sends the two address lines to a Geocoding Service, which validates them and returns the exact coordinates for them.
- Then these source and destination coordinates, along with a user identifier are sent to a Cab–Sharing Engine.
- The Cab–Sharing Engine then automatically groups “close by” requests to minimize the total transportation cost, thereby providing significant savings to the users of the service.
- Once a cab–share is constructed, the cost of the share is deducted from the account of every participant of the cab–share.
- Then, after receiving the information about the cab–share, the CSS forwards the information to the Cab–Scheduling / Cab–Routing Engine. Finally, the CSS sends an SMS to the user about the cab fare, such as cost and schedule of the fare.

WORK FLOW

All the farmers and the transport company representatives can sign-up on the application.

Then the transport company can post their timing and the route of travel on the application.

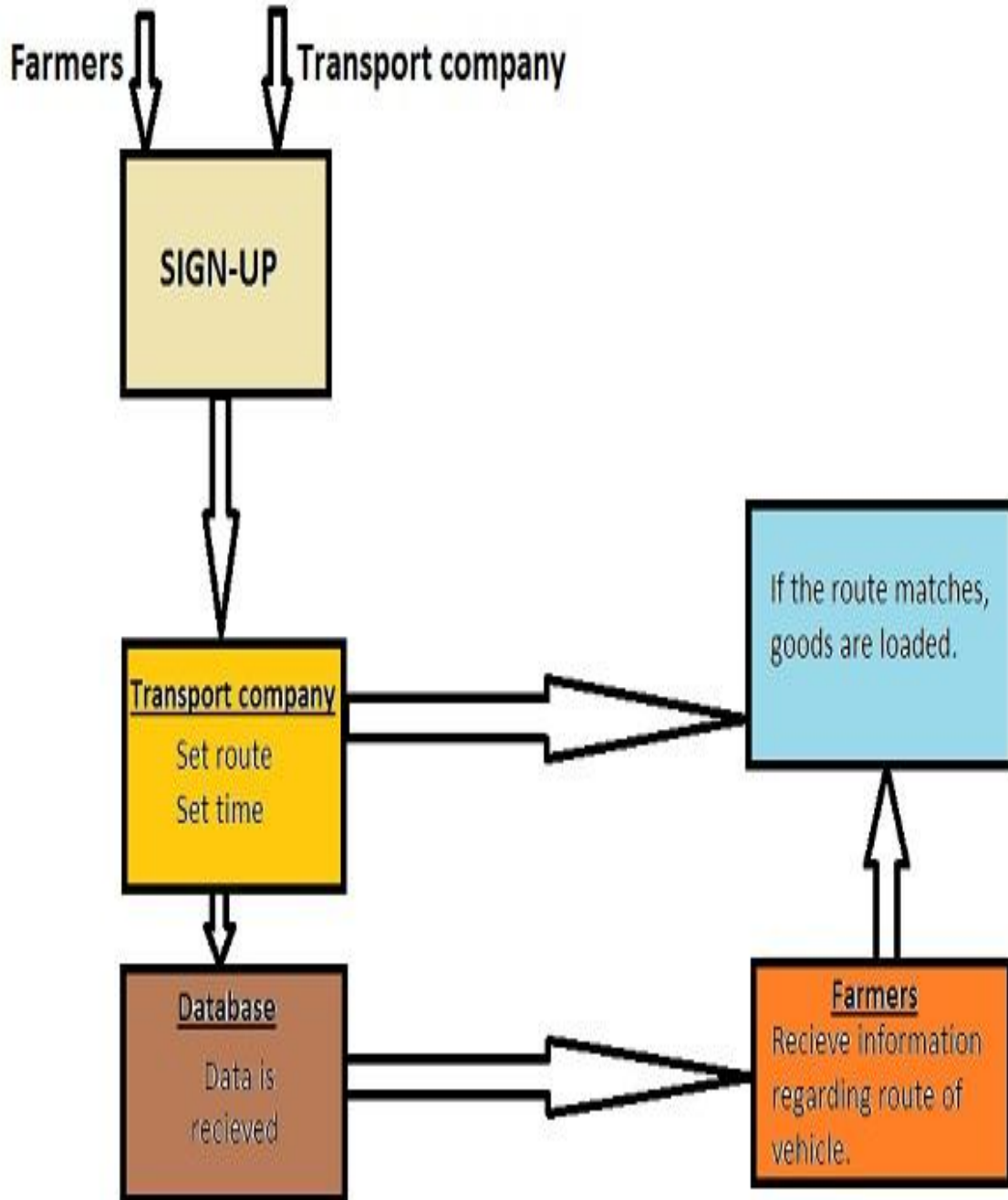
Security is given priority in this application. Every user who signs-up for application has to scan the QR-code on their aadhar cards. In this way no one can sign-up twice or with fake ID.

The farmer that falls on that route can see the timing of all the trucks and click on any particular truck to select it. Then the transport company will be informed about the farmer and the size of the goods he will sending. They will be informed about the timing of the truck and will mutually decide a pick up point where they will load all the goods.

This is the main work flow:

- 1) Register by giving your details.
- 2) Search for existing trucks.
- 3) Select an existing truck.
- 4) Create your own truck post.
- 5) View on Map

BLOCK DIAGRAM



IMPLEMENTATION

1. MainActivity(Code)

```
package com.example.gcmservice;

import static com.example.gcmservice.CommonUtilities.DISPLAY_MESSAGE_ACTION;
import static com.example.gcmservice.CommonUtilities.EXTRA_MESSAGE;
import static com.example.gcmservice.CommonUtilities.SENDER_ID;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;
import android.widget.Toast;

import com.google.android.gms.GCMRegistrar;

public class MainActivity extends Activity {

    // label to display gcm messages
    TextView lblMessage;
```

```

// Asynctask
AsyncTask<Void, Void, Void> mRegisterTask;

// Alert dialog manager
AlertDialogManager alert = new AlertDialogManager();

// Connection detector
ConnectionDetector cd;

public static String fname;
public static String lname;
public static String phone;
public static String add;
public static String qrcode;
public static String email;

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    cd = new ConnectionDetector(getApplicationContext());

    // Check if Internet present
    if (!cd.isConnectingToInternet()) {
        // Internet Connection is not present
        alert.showAlertDialog(MainActivity.this, "Internet Connection Error",
            "Please connect to working Internet connection", false);
        // stop executing code by return
        return;
    }

    // Getting name, email from intent
    Intent i = getIntent();

```

```

fname = i.getStringExtra("fname");
lname = i.getStringExtra("lname");
phone = i.getStringExtra("phone");
add = i.getStringExtra("add");
qrcode = i.getStringExtra("qrcode");
email = i.getStringExtra("email");

// Make sure the device has the proper dependencies.
GCMRegistrar.checkDevice(this);

// Make sure the manifest was properly set - comment out this line
// while developing the app, then uncomment it when it's ready.
GCMRegistrar.checkManifest(this);

lblMessage = (TextView) findViewById(R.id.lblMessage);

registerReceiver(mHandleMessageReceiver, new IntentFilter(
    DISPLAY_MESSAGE_ACTION));

// Get GCM registration id
final String regId = GCMRegistrar.getRegistrationId(this);

// Check if regid already presents
if (regId.equals("")) {
    // Registration is not present, register now with GCM
    GCMRegistrar.register(this, SENDER_ID);
} else {
    // Device is already registered on GCM
    if (GCMRegistrar.isRegisteredOnServer(this)) {
        // Skips registration.
        Toast.makeText(getApplicationContext(), "Already registered with GCM",
Toast.LENGTH_LONG).show();
    } else {

```



```

// Try to register again, but not in the UI thread.
// It's also necessary to cancel the thread onDestroy(),
// hence the use of AsyncTask instead of a raw thread.
final Context context = this;
mRegisterTask = new AsyncTask<Void, Void, Void>() {

    protected Void doInBackground(Void... params) {
        // Register on our server
        // On server creates a new user
        ServerUtilities.register(context, fname, lname, phone, add, qrcode, email, regId);
        return null;
    }

    protected void onPostExecute(Void result) {
        mRegisterTask = null;
    }

};
mRegisterTask.execute(null, null, null);
}
}
}

/**
 * Receiving push messages
 * */
private final BroadcastReceiver mHandleMessageReceiver = new BroadcastReceiver() {

    public void onReceive(Context context, Intent intent) {
        String newMessage = intent.getExtras().getString(EXTRA_MESSAGE);
        // Waking up mobile if it is sleeping
        WakeLocker.acquire(getApplicationContext());
    }
}
}
}

```

```

/**
 * Take appropriate action on this message
 * depending upon your app requirement
 * For now i am just displaying it on the screen
 */

// Showing received message
lblMessage.append(newMessage + "\n");
Toast.makeText(getApplicationContext(), "New Message: " + newMessage,
Toast.LENGTH_LONG).show();

// Releasing wake lock
WakeLocker.release();
}
};

protected void onDestroy() {
    if (mRegisterTask != null) {
        mRegisterTask.cancel(true);
    }
    try {
        unregisterReceiver(mHandleMessageReceiver);
        GCMRegistrar.onDestroy(this);
    } catch (Exception e) {
        Log.e("UnRegister Receiver Error", "> " + e.getMessage());
    }
    super.onDestroy();
}
}
}

```

2. RegisterActivity(Code)

```

package com.example.gcmserver;

import static com.example.gcmserver.CommonUtilities.SENDER_ID;
import static com.example.gcmserver.CommonUtilities.SERVER_URL;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class RegisterActivity extends Activity {
    // alert dialog manager
    AlertDialogManager alert = new AlertDialogManager();

    // Internet detector
    ConnectionDetector cd;

    // UI elements
    EditText efname;
    EditText elname;
    EditText eadd;
    EditText ephone;
    EditText email;

    // Register button
    Button btnRegister;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_register);
    }
}

```

```

cd = new ConnectionDetector(getApplicationContext());

// Check if Internet present
if (!cd.isConnectingToInternet()) {
    // Internet Connection is not present
    alert.showAlertDialog(RegisterActivity.this,
        "Internet Connection Error",
        "Please connect to working Internet connection", false);
    // stop executing code by return
    return;
}

// Check if GCM configuration is set
if (SERVER_URL == null || SENDER_ID == null || SERVER_URL.length() == 0
    || SENDER_ID.length() == 0) {
    // GCM sernder id / server url is missing
    alert.showAlertDialog(RegisterActivity.this, "Configuration Error!",
        "Please set your Server URL and GCM Sender ID", false);
    // stop executing code by return
    return;
}

fname = (EditText) findViewById(R.id.fname);
lname = (EditText) findViewById(R.id.lname);
email = (EditText) findViewById(R.id.email);
phone = (EditText) findViewById(R.id.phone);
btnRegister = (Button) findViewById(R.id.submit);

/*
 * Click event on Register button
 */
btnRegister.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View arg0) {

```

```

// Read EditText dat
String fname = efname.getText().toString();
String lname = elname.getText().toString();
String phone = ephone.getText().toString();

String email = email.getText().toString();

// Check if user filled the form
if(fname.trim().length() > 0 && email.trim().length() > 0){
    // Launch Main Activity
    Intent i = new Intent(getApplicationContext(), MainActivity.class);

    // Registering user on our server
    // Sending registraiton details to MainActivity
    i.putExtra("fname", fname);
    i.putExtra("lname", lname);
    i.putExtra("phone", phone);
    i.putExtra("add", "IITH");
    i.putExtra("qrcode", "1 1");
    i.putExtra("email", email);

    startActivity(i);
    finish();
}else{
    // user doesn't filled that data
    // ask him to fill the form
    alert.showAlertDialog(RegisterActivity.this, "Registration Error!", "Please enter your
details", false);
    }
    }
});
}
}
}
}

```

In RegisterActivity code, user is asked to fill the form and Register event is invoked. All the entries made by user are saved and an account is created.

Chapter 3

Screenshots

Image1-Welcome screen



Image 2- Select your category

Transporter or Farmer

Select Your Category

Farmer

Transporter

A Person Who Is Outstanding in his Field



Image 3-Login Screen

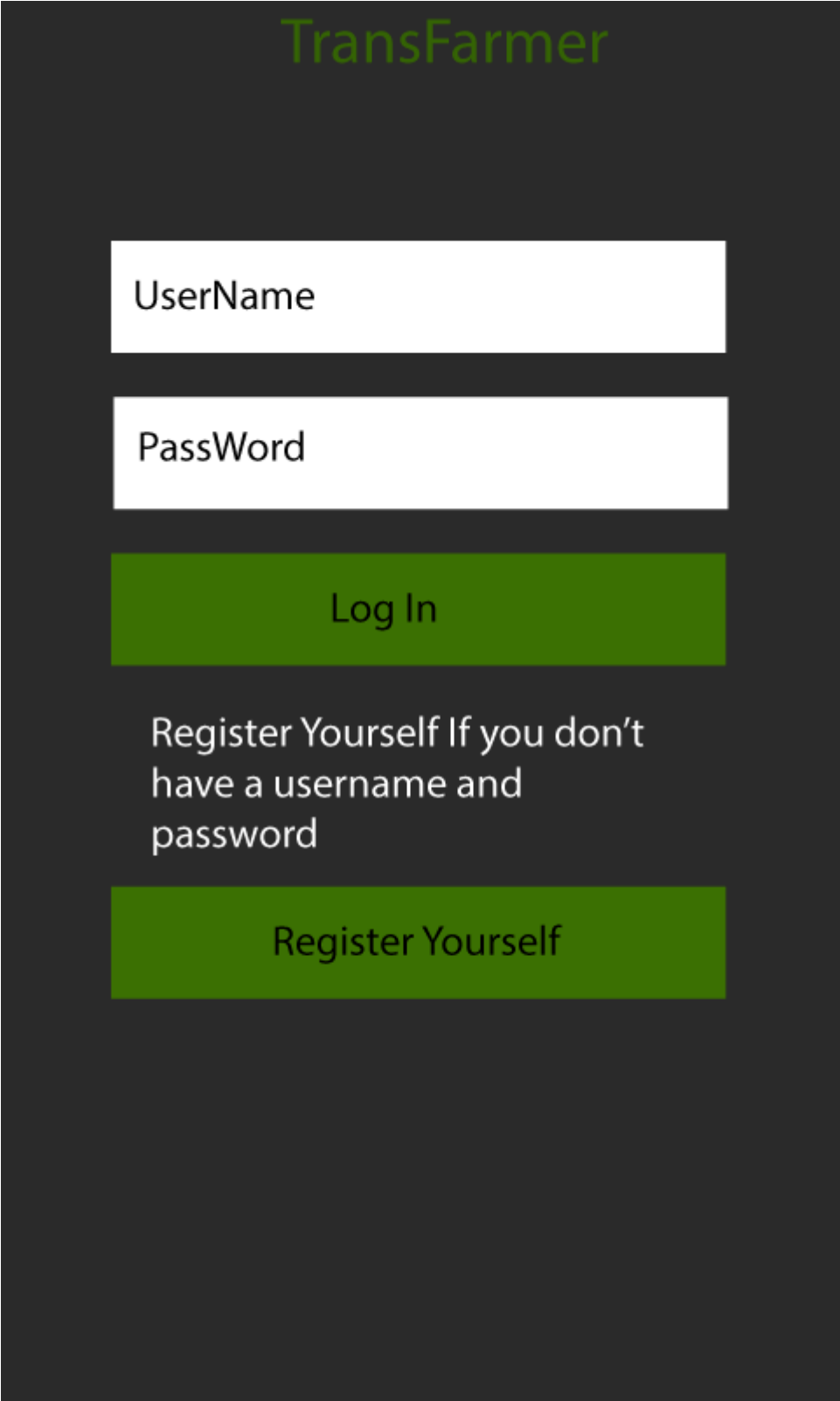
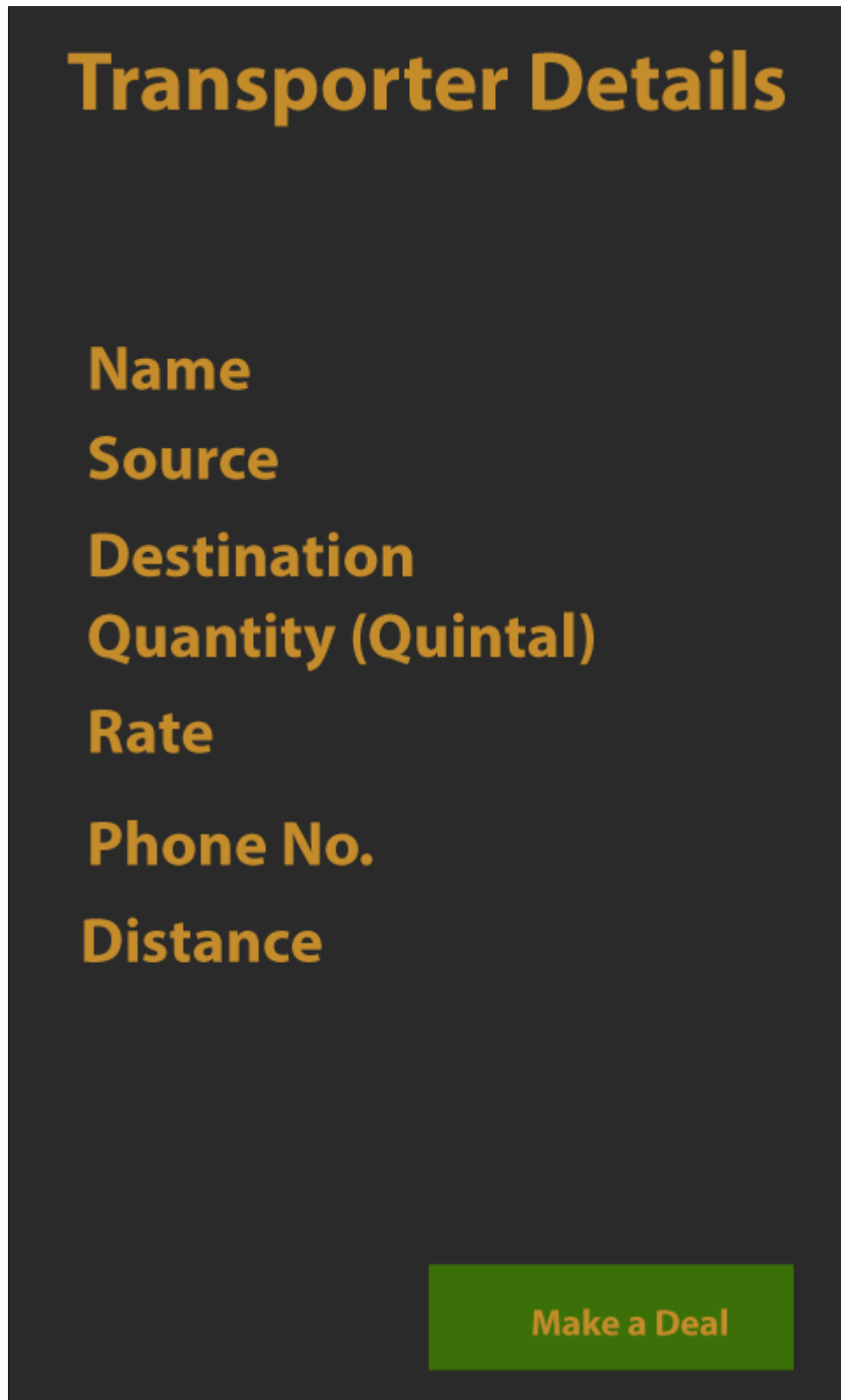


Image 4-List of details bus service

The image shows a mobile application interface with a dark background. At the top left, there is a green button labeled "Back". To its right, the title "List of Details Bus Service" is displayed in a large, bold, green font. Below the title, a table is outlined with a thin orange border. The table has four columns, each with a green header: "Source", "Destination", "Amount", and "Distance". The table body is currently empty.

Source	Destination	Amount	Distance
--------	-------------	--------	----------

Image 5-Transporter details

A dark grey rectangular card with a green button at the bottom right. The card contains the following text in a bold, orange font:

Transporter Details

Name

Source

Destination

Quantity (Quintal)

Rate

Phone No.

Distance

Make a Deal

Chapter 4

Hardware and Software Requirements

1. Hardware Requirements:

- Android Phone
 - CPU : 768MHz
 - RAM : 512 MB
 - Display : Any

2. Software Requirements:

- OS : Android
- Software : Eclipse

Chapter 5

CONCLUSION

This application, if properly used by farmers will surely help them and it will also benefit the transportation company as their trip doesn't get wasted i.e.

When they go from one place to another carrying goods, they come back with their trucks empty, hence there is wastage of fuel. So if they carry these farmer goods in their trip or in roundtrip, farmers get benefited and transport company as well.

Technologies Used:

- Android SDK(for android development)
- Google cloud messaging(for push notification to all farmers that truck is coming)
- Google maps api(for location of trucks and farmers)
- Apache server(for maintaining database)
- mySql(for querying database)

REFERENCES

[1] G. Gidófalvi and T.B. Pedersen. “Cab-sharing: An effective, door-to-door, on-demand transportation service,” in *Proc. of ITS*, 2007.

[2] <http://developer.android.com/training/basics/firstapp/index.html>

[3] <http://www.tutorialspoint.com/android/>