Lesson 2.3: Implicit intents

Introduction

In a previous section you learned about explicit intents. In an explicit intent, you carry out an activity in your app, or in a different app, by sending an intent with the fully qualified class name of the activity. In this section you learn more about *implicit* intents and how to use them to carry out activities.

With an implicit intent, you initiate an activity without knowing which app or activity will handle the task. For example, if you want your app to take a photo, send email, or display a location on a map, you typically don't care which app or activity performs the task.

Conversely, your activity can declare one or more intent filters in the `AndroidManifest.xml` file to advertise that the activity can accept implicit intents, and to define the types of intents that the activity will accept.

To match your request with an app installed on the device, the Android system matches your implicit intent with an activity whose intent filters indicate that they can perform the action. If multiple apps match, the user is presented with an app chooser that lets them select which app they want to use to handle the intent.

In this practical you build an app that sends an implicit intent to perform each of the following tasks:

- Open a URL in a web browser.

- Open a location on a map.
- Share text.

Sharing—sending a piece of information to other people through email or social media—is a popular feature in many apps. For the sharing action you use the `ShareCompat.IntentBuilder` class, which makes it easy to build an implicit intent for sharing data.

Finally, you create a simple intent-receiver that accepts an implicit intent for a specific action.

What you should already know

You should be able to:

- Use the layout editor to modify a layout.
- Edit the XML code of a layout.
- Add a `Button` and a click handler.
- Create and use an `Activity`.
- Create and send an `Intent` between one `Activity` and another.

What you'll learn

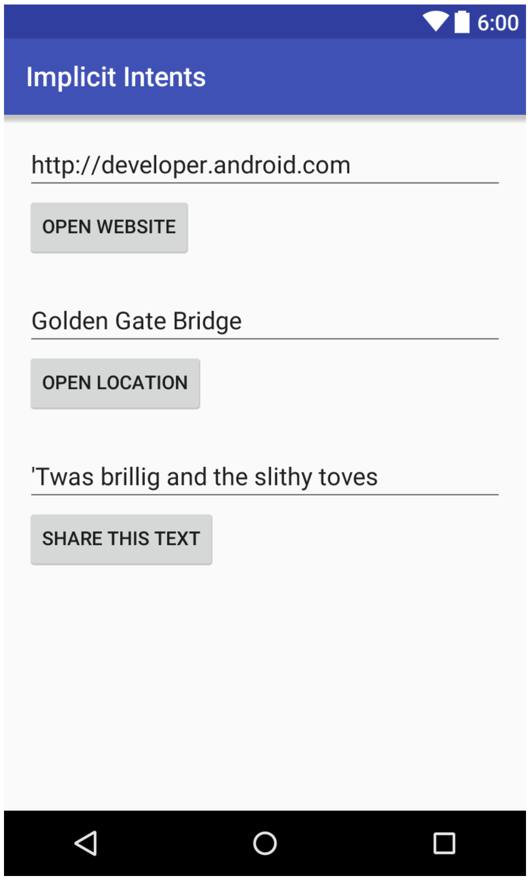
- How to create an implicit `Intent`, and use its actions and categories.
- How to use the `ShareCompat.IntentBuilder` helper class to create an implicit `Intent` for sharing data.
- How to advertise that your app can accept an implicit `Intent` by declaring `Intent` filters in the `AndroidManifest.xml` file.

What you'll do

- Create a new app to experiment with implicit `Intent`.
- Implement an implicit `Intent` that opens a web page, and another that opens a location on a map.
- Implement an action to share a snippet of text.
- Create a new app that can accept an implicit `Intent` for opening a web page.

App overview

In this section you create a new app with one `Activity` and three options for actions: open a web site, open a location on a map, and share a snippet of text. All of the text fields are editable (`EditText`), but contain default values.



Task 1: Create the project and layout

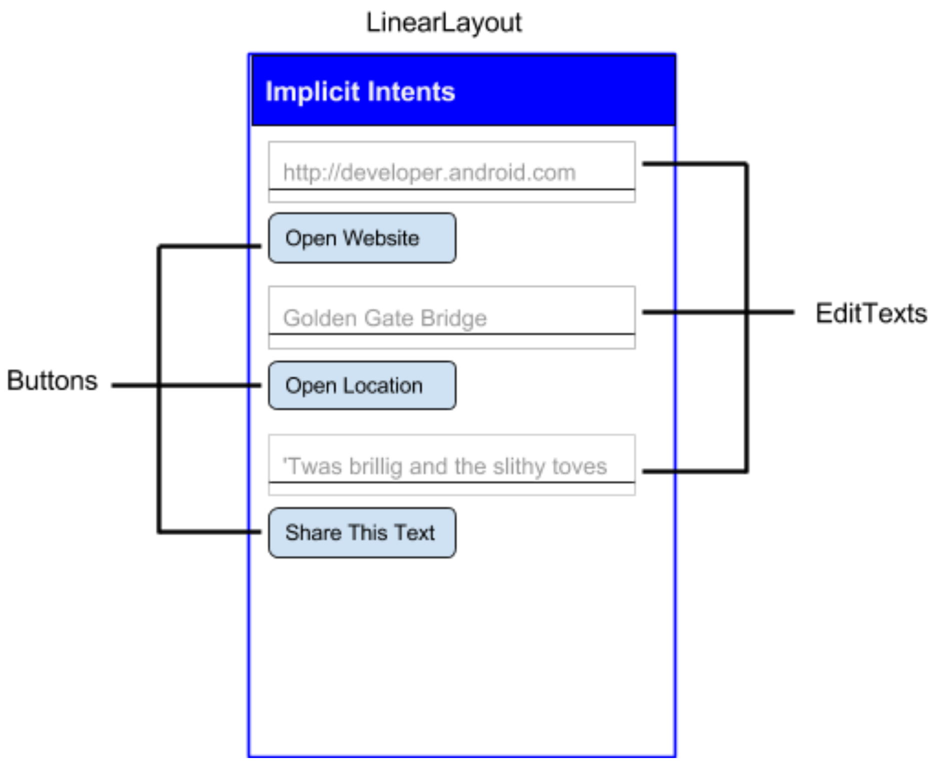
For this exercise, you create a new project and app called Implicit Intents, with a new layout.

1.1 Create the project

1. Start Android Studio and create a new Android Studio project. Name your app **Implicit Intents**.
2. Choose **Empty Activity** for the project template. Click **Next.**
3. Accept the default `Activity` name (`MainActivity`). Make sure the **Generate Layout file** box is checked. Click **Finish**.

1.2 Create the layout

In this task, create the layout for the app. Use a `LinearLayout`, three `Button` elements, and three `EditText` elements, like this:



1. Open **app > res > values > strings.xml** in the **Project > Android** pane, and add the following string resources:

```
<string name="edittext_uri">http://developer.android.com</string>
<string name="button_uri">Open Website</string>

<string name="edittext_loc">Golden Gate Bridge</string>
<string name="button_loc">Open Location</string>

<string name="edittext_share">\'Twas brillig and the slithy toves</string>
<string name="button_share">Share This Text</string>
```

2. Open **res > layout > activity_main.xml** in the **Project > Android** pane. Click the **Text** tab to switch to XML code.
3. Change `android.support.constraint.ConstraintLayout` to `LinearLayout`, as you learned in a previous practical.
4. Add the `android:orientation` attribute with the value `"vertical"`. Add the `android:padding` attribute with the value `"16dp"`.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context="com.example.android.implicitintents.MainActivity">
```

5. Remove the `TextView` that displays "Hello World".
6. Add a set of UI elements to the layout for the **Open Website** button. You need an `EditText` element and a `Button` element. Use these attribute values:

EditText attribute	**Value**
`android:id`	`"@+id/website_edittext"`
`android:layout_width`	`"match_parent"`
`android:layout_height`	`"wrap_content"`
`android:text`	`"@string/edittext_uri"`
Button attribute	**Value**
`android:id`	`"@+id/open_website_button"`
`android:layout_width`	`"wrap_content"`
`android:layout_height`	`"wrap_content"`
`android:layout_marginBottom`	`"24dp"`
`android:text`	`"@string/button_uri"`

`android:onClick`	`"openWebsite"`

The value for the `android:onClick` attribute will remain underlined in red until you define the callback method in a subsequent task.

7. Add a set of UI elements (`EditText` and `Button`) to the layout for the **Open Location** button. Use the same attributes as in the previous step, but modify them as shown below. (You can copy the values from the **Open Website** button and modify them.)

EditText attribute	**Value**
`android:id`	`"@+id/location_edittext"`
`android:text`	`"@string/edittext_loc"`
Button attribute	**Value**
`android:id`	`"@+id/open_location_button"`
`android:text`	`"@string/button_loc"`
`android:onClick`	`"openLocation"`

The value for the `android:onClick` attribute will remain underlined in red until you define the callback method in a subsequent task.

8. Add a set of UI elements (`EditText` and `Button`) to the layout for the **Share This** button. Use the attributes shown below. (You can copy the values from the **Open Website** button and modify them.)

EditText attribute	**Value**
`android:id`	`"@+id/share_edittext"`
`android:text`	`"@string/edittext_share"`
Button attribute	**Value**
`android:id`	`"@+id/share_text_button"`
`android:text`	`"@string/button_share"`
`android:onClick`	`"shareText"`

Depending on your version of Android Studio, your `activity_main.xml` code should look something like the following. The values for the `android:onClick` attributes will remain underlined in red until you define the callback methods in a subsequent task.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context="com.example.android.implicitintents.MainActivity">

    <EditText
        android:id="@+id/website_edittext"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
         android:text="@string/edittext_uri"/>

    <Button
        android:id="@+id/open_website_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
         android:layout_marginBottom="24dp"
        android:text="@string/button_uri"
        android:onClick="openWebsite"/>

    <EditText
        android:id="@+id/location_edittext"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/edittext_uri"/>

    <Button
        android:id="@+id/open_location_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="24dp"
        android:text="@string/button_loc"
        android:onClick="openLocation"/>

    <EditText
        android:id="@+id/share_edittext"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/edittext_share"/>

    <Button
        android:id="@+id/share_text_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="24dp"
        android:text="@string/button_share"
        android:onClick="shareText"/>

</LinearLayout>
```

Task 2: Implement the Open Website button

In this task you implement the on-click handler method for the first button in the layout, **Open Website**. This action uses an implicit `Intent` to send the given URI to an `Activity` that can handle that implicit `Intent` (such as a web browser).

2.1 Define openWebsite()

1. Click `"openWebsite"` in the `activity_main.xml` XML code.
2. Press `Alt+Enter` (`Option+Enter` on a Mac) and select **Create 'openWebsite(View)' in 'MainActivity.**

The `MainActivity` file opens, and Android Studio generates a skeleton method for the `openWebsite`() handler.

```
public void openWebsite(View view) {
}
```

3. In `MainActivity`, add a private variable at the top of the class to hold the `EditText` object for the web site URI.

```
private EditText mWebsiteEditText;
```

4. In the `onCreate()` method for `MainActivity`, use `findViewById()` to get a reference to the `EditText` instance and assign it to that private variable:

```
mWebsiteEditText = findViewById(R.id.website_edittext);
```

2.2 Add code to openWebsite()

1. Add a statement to the new `openWebsite()` method that gets the string value of the `EditText`:

```
String url = mWebsiteEditText.getText().toString();
```

2. Encode and parse that string into a Uri object:

```
Uri webpage = Uri.parse(url);
```

3. Create a new `Intent` with `Intent.ACTION_VIEW` as the action and the URI as the data:

```
Intent intent = new Intent(Intent.ACTION_VIEW, webpage);
```

This `Intent` constructor is different from the one you used to create an explicit `Intent`. In the previous constructor, you specified the current context and a specific component (`Activity` class) to send the `Intent`. In this constructor you specify an action and the data for that action. Actions are defined by the `Intent` class and can include `ACTION_VIEW` (to view the given data), `ACTION_EDIT` (to edit the given data), or `ACTION_DIAL` (to dial a phone number). In this case the action is `ACTION_VIEW` because you want to display the web page specified by the URI in the `webpage` variable.

4. Use the `resolveActivity()` method and the Android package manager to find an `Activity` that can handle your implicit `Intent`. Make sure that the request resolved successfully.

```
if (intent.resolveActivity(getPackageManager()) != null) {
}
```

This request that matches your `Intent` action and data with the `Intent` filters for installed apps on the device. You use it to make sure there is at least one `Activity` that can handle your requests.

5. Inside the `if` statement, call `startActivity()` to send the `Intent`.

```
startActivity(intent);
```

6. Add an `else` block to print a `Log` message if the `Intent` could not be resolved.

```
} else {
   Log.d("ImplicitIntents", "Can't handle this!");
}
```

The `openWebsite()` method should now look as follows. (Comments added for clarity.)

```
public void openWebsite(View view) {
   // Get the URL text.
   String url = mWebsiteEditText.getText().toString();

   // Parse the URI and create the intent.
   Uri webpage = Uri.parse(url);
   Intent intent = new Intent(Intent.ACTION_VIEW, webpage);

   // Find an activity to hand the intent and start that activity.
   if (intent.resolveActivity(getPackageManager()) != null) {
       startActivity(intent);
   } else {
       Log.d("ImplicitIntents", "Can't handle this intent!");
   }
}
```

Task 3: Implement the Open Location button

In this task you implement the on-click handler method for the second button in the UI, **Open Location**. This method is almost identical to the `openWebsite()` method. The difference is the use of a geo URI to indicate a map location. You can use a geo URI with latitude and longitude, or use a query string for a general location. In this example we've used the latter.

3.1 Define openLocation()

1. Click "`openLocation`" in the `activity_main.xml` XML code.
2. Press `Alt+Enter` (`Option+Enter` on a Mac) and select **Create 'openLocation(View)' in MainActivity.**

   Android Studio generates a skeleton method in `MainActivity` for the `openLocation`() handler.

```
public void openLocation(View view) {
}
```

3. Add a private variable at the top of `MainActivity` to hold the `EditText` object for the location URI.

```
private EditText mLocationEditText;
```

4. In the `onCreate()` method, use `findViewByID()` to get a reference to the `EditText` instance and assign it to that private variable:

```
mLocationEditText = findViewById(R.id.location_edittext);
```

3.2 Add code to openLocation()

1. In the new `openLocation()` method, add a statement to get the string value of the `mLocationEditText` `EditText`.

```
String loc = mLocationEditText.getText().toString();
```

2. Parse that string into a Uri object with a geo search query:

```
Uri addressUri = Uri.parse("geo:0,0?q=" + loc);
```

3. Create a new `Intent` with `Intent.ACTION_VIEW` as the action and loc as the data.

```
Intent intent = new Intent(Intent.ACTION_VIEW, addressUri);
```

4. Resolve the `Intent` and check to make sure that the `Intent` resolved successfully. If so, `startActivity()`, otherwise log an error message.

```
if (intent.resolveActivity(getPackageManager()) != null) {
   startActivity(intent);
} else {
   Log.d("ImplicitIntents", "Can't handle this intent!");
}
```

The `openLocation()` method should now look as follows (comments added for clarity):

```
public void openLocation(View view) {
   // Get the string indicating a location. Input is not validated; it is
   // passed to the location handler intact.
   String loc = mLocationEditText.getText().toString();

   // Parse the location and create the intent.
   Uri addressUri = Uri.parse("geo:0,0?q=" + loc);
   Intent intent = new Intent(Intent.ACTION_VIEW, addressUri);

   // Find an activity to handle the intent, and start that activity.
   if (intent.resolveActivity(getPackageManager()) != null) {
       startActivity(intent);
   } else {
       Log.d("ImplicitIntents", "Can't handle this intent!");
   }
}
```

Task 4: Implement the Share This Text button

A share action is an easy way for users to share items in your app with social networks and other apps. Although you could build a share action in your own app using an implicit `Intent`, Android provides the `ShareCompat.IntentBuilder` helper class to make implementing sharing easy. You can use `ShareCompat.IntentBuilder` to build an `Intent` and launch a chooser to let the user choose the destination app for sharing.

In this task you implement sharing a bit of text in a text edit, using the `ShareCompat.IntentBuilder` class.

4.1 Define shareText()

1. Click `"shareText"` in the `activity_main.xml` XML code.
2. Press `Alt+Enter` (`Option+Enter` on a Mac) and select **Create 'shareText(View)' in MainActivity.**

   Android Studio generates a skeleton method in `MainActivity` for the `shareText`() handler.

```
public void shareText(View view) {
}
```

3. Add a private variable at the top of `MainActivity` to hold the `EditText`.

```
private EditText mShareTextEditText;
```

4. In `onCreate()`, use `findViewById()` to get a reference to the `EditText` instance and assign it to that private variable:

```
mShareTextEditText = findViewById(R.id.share_edittext);
```

4.2 Add code to shareText()

1. In the new `shareText()` method, add a statement to get the string value of the `mShareTextEditText` `EditText`.

```
String txt = mShareTextEditText.getText().toString();
```

2. Define the mime type of the text to share:

```
String mimeType = "text/plain";
```

3. Call `ShareCompat.IntentBuilder` with these methods:

```
ShareCompat.IntentBuilder
            .from(this)
            .setType(mimeType)
            .setChooserTitle("Share this text with: ")
            .setText(txt)
            .startChooser();
```

4. Extract the value of `.setChoosterTitle` to a string resource.

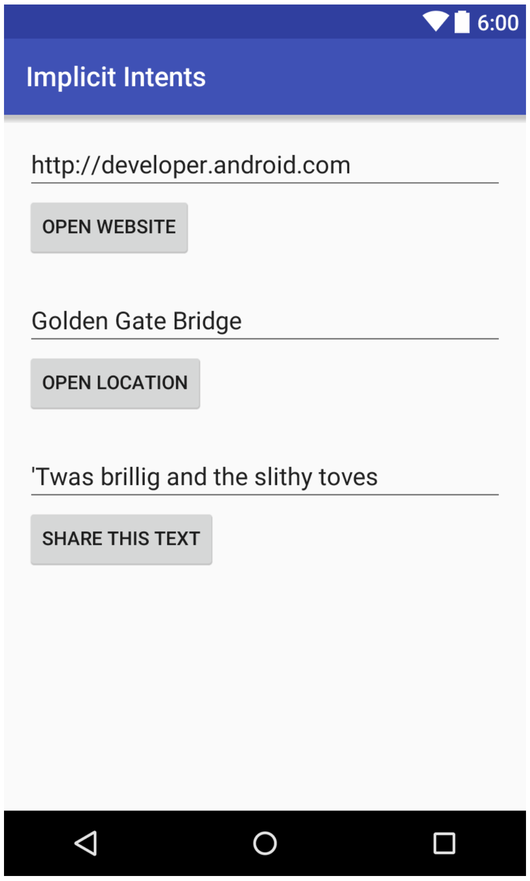
The call to `ShareCompat.IntentBuilder` uses these methods:

Method	**Description**
`from()`	The `Activity` that launches this share `Intent` (`this`).
`setType()`	The MIME type of the item to be shared.
`setChooserTitle()`	The title that appears on the system app chooser.
`setText()`	The actual text to be shared
`startChooser()`	Show the system app chooser and send the `Intent`.

This format, with all the builder's setter methods strung together in one statement, is an easy shorthand way to create and launch the `Intent`. You can add any of the additional methods to this list.

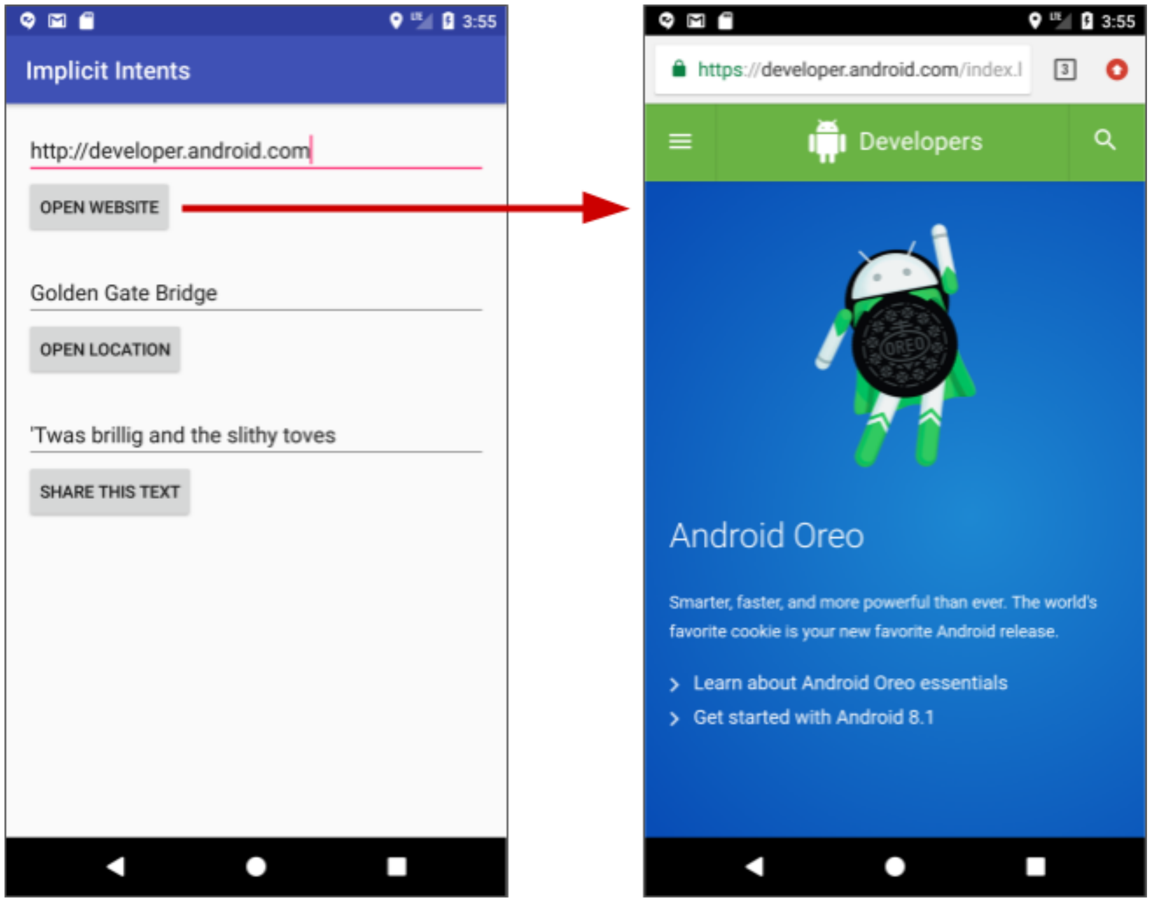
The `shareText()` method should now look as follows:

```
public void shareText(View view) {
      String txt = mShareTextEditText.getText().toString();
      String mimeType = "text/plain";
      ShareCompat.IntentBuilder
                  .from(this)
                  .setType(mimeType)
                  .setChooserTitle(R.string.share_text_with)
                  .setText(txt)
                  .startChooser();
}
```

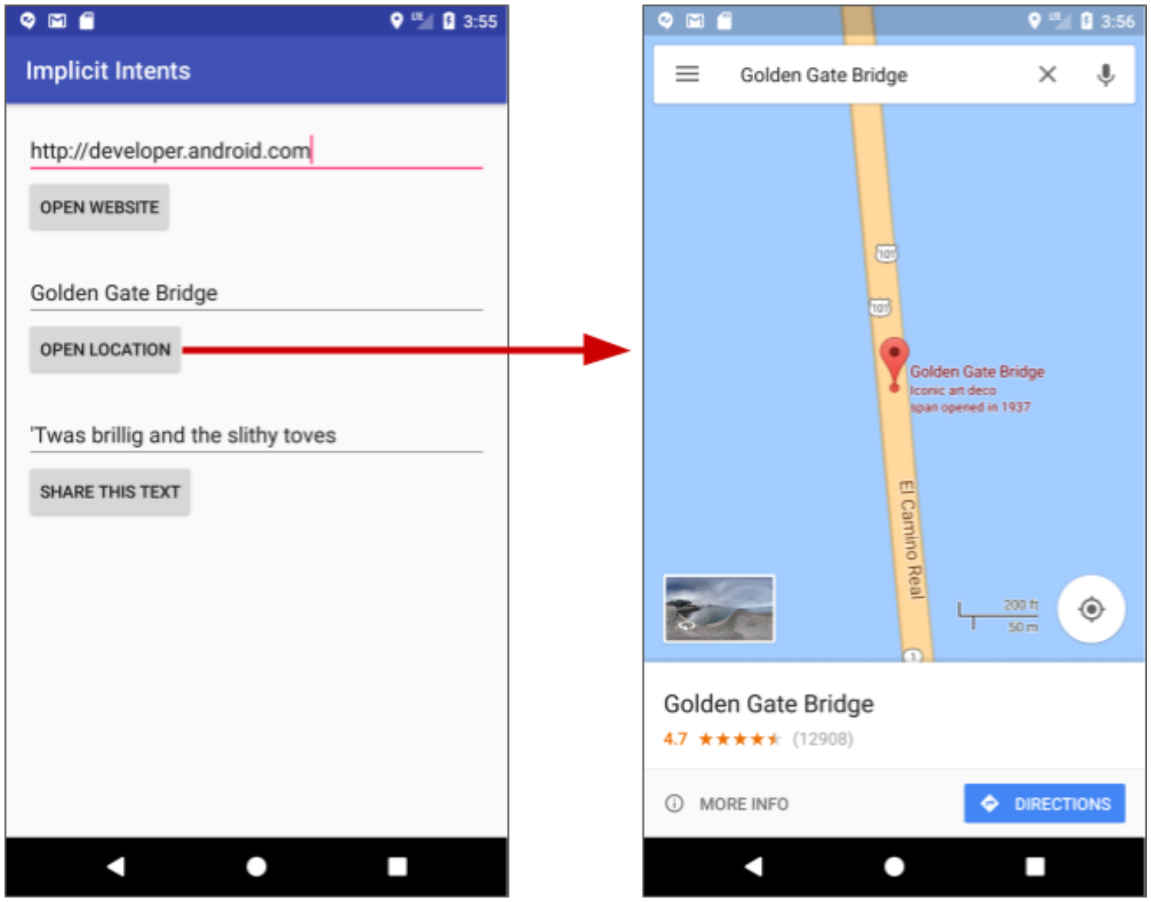


4.3 Run the app

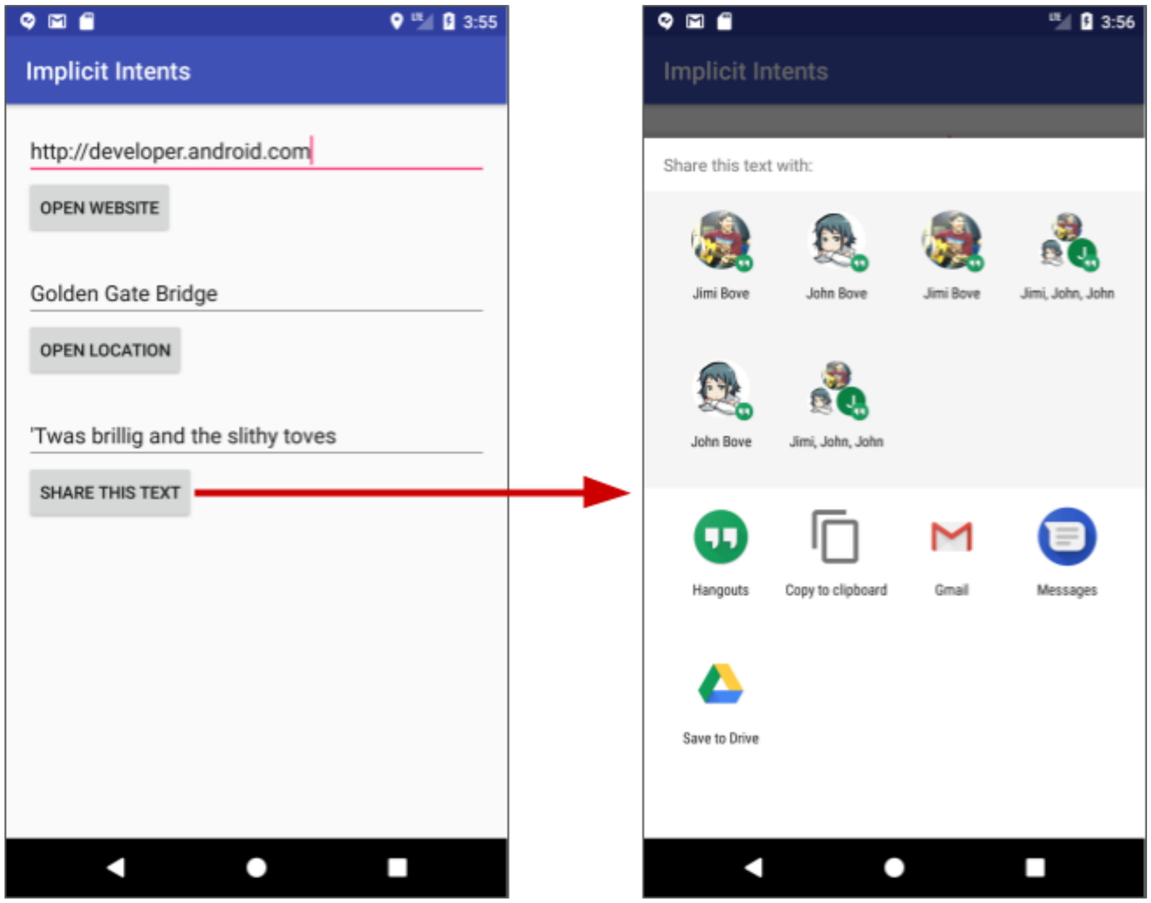
1. Run the app.
2. Click the **Open Website** button to launch a browser with the website URL in the `EditText` above the `Button`. The browser and website should appear as shown below.


Implicit Intents
http://developer.android.com
OPEN WEBSITE
Golden Gate Bridge
OPEN LOCATION
'Twas brillig and the slithy toves
SHARE THIS TEXT
https://developer.android.com/index.l
Developers
Android Oreo
Smarter, faster, and more powerful than ever. The world's favorite cookie is your new favorite Android release.
Learn about Android Oreo essentials
Get started with Android 8.1

3. Click the **Open Location** button to launch the map with the location in the `EditText` above the `Button`. The map with the location should appear as shown below.



4. Click the **Share This Text** button to launch a dialog with choices for sharing the text. The dialog with choices should appear as shown below.



Task 4 solution code

Android Studio project: ImplicitIntents

Task 5: Receive an implicit Intent

So far, you've created an app that uses an implicit `Intent` in order to launch some other app's `Activity`. In this task you look at the problem from the other way around: allowing an `Activity` in your app to respond to an implicit `Intent` sent from some other app.

An `Activity` in your app can always be activated from inside or outside your app with an explicit `Intent`. To allow an `Activity` to receive an implicit `Intent`, you define an `Intent` *filter* in your app's `AndroidManifest.xml` file to indicate which types of implicit `Intent` your `Activity` is interested in handling.

To match your request with a specific app installed on the device, the Android system matches your implicit `Intent` with an `Activity` whose `Intent` filters indicate that they can perform that action. If there are multiple apps installed that match, the user is presented with an app chooser that lets them select which app they want to use to handle that `Intent`.

When an app on the device sends an implicit `Intent`, the Android system matches the action and data of that `Intent` with any available `Activity` that includes the right `Intent` filters. When the `Intent` filters for an `Activity` match the `Intent`:

- If there is only one matching `Activity`, Android lets the `Activity` handle the `Intent` itself.
- If there are multiple matches, Android displays an app chooser to allow the user to pick which app they'd prefer to execute that action.

In this task you create a very simple app that receives an implicit `Intent` to open the URI for a web page. When activated by an implicit `Intent`, that app displays the requested URI as a string in a `TextView`.

5.1 Create the project and layout

1. Create a new Android Studio project with the app name **Implicit Intents Receiver** and choose **Empty Activity** for the project template.
2. Accept the default `Activity` name (`MainActivity`). Click **Next**.
3. Make sure the **Generate Layout file** box is checked. Click **Finish**.
4. Open **activity_main.xml**.
5. In the existing ("Hello World") `TextView` , delete the `android:text` attribute. There's no text in this `TextView` by default, but you'll add the URI from the `Intent` in `onCreate()`.
6. Leave the `layout_constraint` attributes alone, but add the following attributes:

Attribute	**Value**
`android:id`	`"@+id/text_uri_message"`
`android:textSize`	`"18sp"`
`android:textStyle`	`"bold"`

5.2 Modify AndroidManifest.xml to add an Intent filter

1. Open the `AndroidManifest.xml` file.
2. Note that `MainActivity` already has this `Intent` filter:

```
<intent-filter>
   <action android:name="android.intent.action.MAIN" />
   <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

This `Intent` filter, which is part of the default project manifest, indicates that this `Activity` is the main entry point for your app (it has an `Intent` action of `"android.intent.action.MAIN"`), and that this `Activity` should appear as a top-level item in the launcher (its category is `"android.intent.category.LAUNCHER"`).

3. Add a second `<intent-filter>` tag inside `<activity>`, and include these elements :

```
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
<data android:scheme="http" android:host="developer.android.com" />
```

These lines define an `Intent` filter for the `Activity`, that is, the kind of `Intent` that the `Activity` can handle. This `Intent` filter declares these elements:

Filter type	**Value**	**Matches**

action	`"android.intent.action.VIEW"`	Any `Intent` with view actions.
category	`"android.intent.category.DEFAULT"`	Any implicit `Intent`. This category must be included for your `Activity` to receive **any** implicit `Intent`.
category	`"android.intent.category.BROWSABLE"`	Requests for browsable links from web pages, email, or other sources.
data	`android:scheme="http"` `android:host="developer.android.com"`	URIs that contain a scheme of `http` *and* a host name of `developer.android.com`.

Note that the data filter has a restriction on both the kind of links it will accept and the hostname for those URIs. If you'd prefer your receiver to be able to accept any links, you can leave out the `<data>` element.

The `application` section of `AndroidManifest.xml` should now look as follows:

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>

        <intent-filter>
            <action android:name="android.intent.action.VIEW" />
            <category android:name="android.intent.category.DEFAULT" />
            <category android:name="android.intent.category.BROWSABLE" />
            <data android:scheme="http"
                                    android:host="developer.android.com" />
        </intent-filter>
    </activity>
</application>
```

5.3 Process the Intent

In the `onCreate()` method for your `Activity`, process the incoming `Intent` for any data or extras it includes. In this case, the incoming implicit `Intent` has the URI stored in the `Intent` data.

1. Open **MainActivity**.
2. In the `onCreate()` method, get the incoming `Intent` that was used to activate the `Activity`:

```
Intent intent = getIntent();
```

3. Get the `Intent` data. `Intent` data is always a URI object:

```
Uri uri = intent.getData();
```

4. Check to make sure that the `uri` variable is not `null`. If that check passes, create a string from that URI object:

```
if (uri != null) {
   String uri_string = "URI: " + uri.toString();
}
```

5. Extract the `"URI: "` portion of the above into a string resource (`uri_label`).
6. Inside that same `if` block, get the `TextView` for the message:

```
TextView textView = findViewById(R.id.text_uri_message);
```

7. Also inside the `if` block, set the text of that `TextView` to the URI:

```
textView.setText(uri_string);
```

The `onCreate()` method for `MainActivity` should now look like the following:

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

        Intent intent = getIntent();
        Uri uri = intent.getData();
        if (uri != null) {
            String uri_string = getString(R.string.uri_label)
                   + uri.toString();
            TextView textView = findViewById(R.id.text_uri_message);
            textView.setText(uri_string);
        }
}
```

5.4 Run both apps

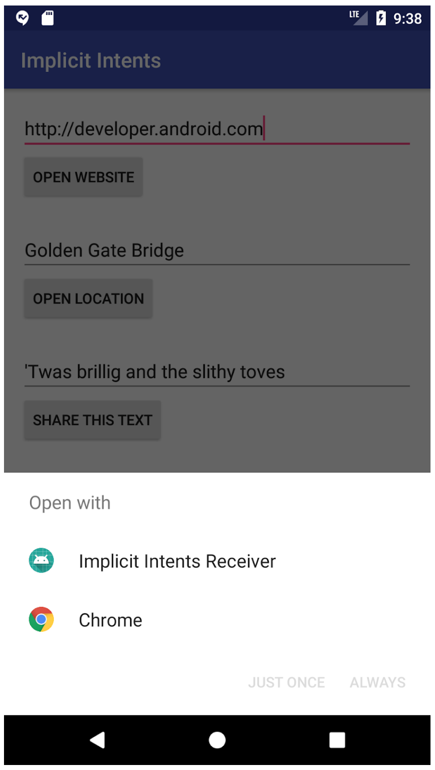
To show the result of receiving an implicit `Intent`, you will run both the Implicit Intents Receiver and Implicit Intents apps on the emulator or your device.

1. Run the Implicit Intents Receiver app.

   Running the app on its own shows a blank `Activity` with no text. This is because the `Activity` was activated from the system launcher, and not with an `Intent` from another app.

2. Run the Implicit Intents app, and click **Open Website** with the default URI.

   An app chooser appears asking if you want to use the default browser (Chrome in the figure below) or the Implicit Intents Receiver app. Select **Implicit Intents Receiver**, and click **Just Once**. The Implicit Intents Receiver app launches and the message shows the URI from the original request.



3. Tap the Back button and enter a different URI. Click **Open Website**.