Mobile Application Development (COMP2008)

Lecture 5: App Interaction

Updated: 4th October, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

App-to-App Interaction

- Software is rarely “standalone”. It almost always interacts with other software (even aside from the OS).
- Android provides high-level API calls for doing this.
  - We won't be using `shm_open()` or `mmap()`!
- There are good reasons for high-level app-to-app interaction.
- **1. Reuse.**
  - Why re-invent the wheel?
  - Just as you would reuse methods and classes, so you can reuse whole apps.
  - Less code means less development time (and money).
  - Less code also means (most likely) fewer defects.

App-to-App Interaction

2. **User experience.**
   - Even if you want to re-invent the wheel, your users may prefer you didn't.
   - Users have existing apps for taking photos, sending emails, selecting contacts, etc.
     - Let them use what they're familiar with.
     - Let them choose dedicated apps for those specific tasks.
   - Apps with a single purpose tend to be very good at it.
     - Taking photos probably isn't your app's main task.
     - You probably can't do it better than a dedicated camera app.
     - So just use the existing app!

App-to-App Interaction

3. Security by compartmentalisation.

- Reduces the amount you need to trust things.
- Android has a system of permissions:
  - Apps need permission to do certain things.
  - The more permissions any one app has, the more you have to trust it.
  - An attacker who compromises an app has the same permissions as the app.
- When apps interact (in a well-designed manner) then:
  - Each one needs fewer permissions (good!)
  - Less overall trust required (good!)
- For instance, an app can:
  - Request a contact without having access to the whole contact database.
  - Request a photo from the ordinary camera app, *without* being able to secretly turn on the camera and spy on you!

Implicit Intents

- You've already used "*explicit*" intents to start other activities in your app:

```
Intent intent = new Intent(this, NewActivity.class);
intent.putExtra(...);
...
startActivityForResult(intent);
// Or, startActivity(intent);
```

  - We have to specify which activity we're starting.
- Implicit intents are used to start activities in *other* apps.
  - You don't directly specify the activity to start.
  - Instead, you tell the OS what functionality you need.
  - The OS examines which other apps provide that functionality.
  - The OS asks the user, if there's more than one possibility.

Using Implicit Intents

- (Unfortunately?), each app-to-app interaction has its own specific way of using intent objects.
  - To be expected.
  - Sending emails, taking photos, picking contacts have very little in common.
  - Trying to do them all "in the same way" is a little superficial.
- However, there are some common concepts.

Intent Structure

Intent objects are made up of these parts:

Component –	the name of the activity to launch. If not specified, you have an implicit intent.
Action –	a string label specifying (approximately) what you want to do.
Data –	a URL/URI indicating some relevant source of data.
Type –	a “MIME” type, which identifies the type of data.
Categories –	a set of string labels that help refine what you want to do.
Extras –	(as you know) a set of name-value pairs that can be used to pass whatever other parameters may be needed.

Example: Dialing a Phone Number

If your app deals with phone numbers, why not have a call button?

```
private Intent callIntent;
...
callIntent = new Intent(Intent.ACTION_DIAL,
                        Uri.parse("tel:0123456789"));

button.setOnClickListener(new View.OnClickListener()
{
    @Override
    public void onClick(View v)
    {
        startActivity(callIntent);
    }
});
```

Example: Showing Map Locations

If your app deals with geographic coordinates, why not provide a button to show them on a map?

```
private Intent mapIntent;
...
mapIntent = new Intent(Intent.ACTION_VIEW,
                        Uri.parse("geo:-32.0062,115.8944"));

button.setOnClickListener(...);
```

Actions

- Always needed for implicit intents.
- Basically one of several pre-defined string constants; e.g.:

  `Intent.ACTION_VIEW` – views a document, image, etc.
  `Intent.ACTION_DIAL` – enters a phone number into the phone-call app.
  `Intent.ACTION_SEND` – sends text or data via email, text message, etc.[1]
  `Intent.ACTION_PICK` – asks the user to choose something a contact, file, etc., to be returned to your app.
  `MediaStore.ACTION_IMAGE_CAPTURE` – invokes the camera app to take a photo, to be returned to your app.
- Use `startActivity()` or `startActivityForResult()` as appropriate.

[1]https://developer.android.com/training/sharing/send

Actions (2)

- You can provide an action to the Intent constructor:

```
Intent intent = new Intent(Intent.ACTION_SEND);
```

```
Intent intent = new Intent(Intent.ACTION_SEND, ...);
```

OR mutator:

```
Intent intent = new Intent();
intent.setAction(Intent.ACTION_SEND);
```

Data

- This is a URI (Uniform Resource Indicator).
- *Usually* needed for implicit intents (but not always).
  - If the action is “what” to do, then. . .
  - The data is “which” resource to do it with.
- Could be an internet-based resource.
  - e.g. “`http://www.curtin.edu.au/`”.
- Could be a local file on the device.
  - e.g. “`file:///data/data/edu.curtin.myapp/myphoto.jpg`”.
- Can also be a specialised way of encoding a small amount of information:
  - Geographic coordinates: “`geo:-32.0062,115.8944`”.
  - Phone number: “`tel:0892665212`”.

Uri Objects

- Android has a "`Uri`" class.
  - Be careful! Standard Java also has both `URI` and `URL` classes.
  - They *all* work differently!
- In simple cases, you can create a `Uri` object like this:

```
int phoneNumber = ...;
Uri phoneUri = Uri.parse("tel:" + phoneNumber);
```

  Or, using `String.format()` from standard Java:

```
double latitude = ...;
double longitude = ...;
Uri mapUri = Uri.parse(String.format("geo:%f,%f",
                                     latitude,
                                     longitude));
```

- Sometimes you must get the `Uri` from somewhere else.

Intents and Uris

- You can provide a Uri to the Intent constructor, right after the action:

```
Uri uri = ...;
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
```

  OR via a mutator:

```
Uri uri = ...;
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(uri);
```

Type

- Indicates the type of the data:

`text/plain`	– plain text!
`message/rfc822`	– email message.
`application/pdf`	– PDF.
`image/jpg`	– JPEG image.

- Not always needed.
  - Sometimes the data could only really be one thing anyway.
- Set alongside data via a combined mutator:

```
Uri uri = ...;
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setDataAndType(uri, "text/plain");
```

Beware:

  - `setData()` will clear the type.
  - `setType()` will clear the data.
  - If you want both, you *must* use `setDataAndType()`.

Categories

- A set of additional string labels (tags).
- Rarely needed (at least on the calling side), and entirely situation-dependent.
- If the data is "what" and the type is "which", then categories are "how", "when", etc.
- Added as follows:

```
Intent intent = new Intent();
intent.addCategory(Intent.CATEGORY_APP_MESSAGING);
```

Verifying Capabilities

- So what if the OS *cannot* find a suitable external app?
- `startActivity()` (or `startActivityForResult()`) throws `ActivityNotFoundException`.
    - Ordinarily this will cause your app to crash.
- One option is to catch it, and show an error message.
- Another option is to *preempt* it – check beforehand whether there is an existing activity to receive your intent.
    - Recommended approach.
    - Uses a standard Android class called `PackageManager`.
    - You can “grey-out” (disable) whatever button would have started the external activity, if needed.

PackageManager and Implicit Intents

```java
private Button runAppButton = ...;      // Inside a fragment
private Intent runAppIntent = ...;

public View onCreateView(...)
{
    ...
    PackageManager pm = getActivity().getPackageManager();
    if(pm.resolveActivity(runAppIntent,
        PackageManager.MATCH_DEFAULT_ONLY) == null)
    {
        runAppButton.setEnabled(false);
    }
}
```

- We set up our intent on start-up, and ask PackageManager to find a suitable external app.
- If null, disable the button that would have launched the app.

Returning Data: Thumbnail Photos

- You may expect a result from the app you're calling.
- This will invoke the camera to take a (small) photo:

```
private static final int REQUEST_THUMBNAIL = 1;
private Intent thumbnailPhotoIntent;
...
thumbnailPhotoIntent = new Intent(
    MediaStore.ACTION_IMAGE_CAPTURE);
...
startActivityForResult(thumbnailPhotoIntent,
                       REQUEST_THUMBNAIL);
```

- Obviously we now need to *receive* the photo somehow. . .

Returning Data: Thumbnail Photos

```java
private ImageView iv;
...
@Override
public void onActivityResult(int requestCode, int resultCode,
                             Intent ret)
{
    if( resultCode == Activity.RESULT_OK &&
        requestCode == REQUEST_THUMBNAIL)
    {
        // The return intent from the camera contains the
        // small ("thumbnail") photo inside an extra called
        // "data".
        Bitmap thumbnail =
            (Bitmap) ret.getExtras().get("data");

        iv.setImageBitmap(thumbnail); // Display photo
    }
}
```

Returning Data: Contact Information

- If our app requires personal/contact data, we can invoke the contacts app.
- The user can pick a contact, and return it to our app.
- Beware! This is the start of a rabbit hole!

```
private static final REQUEST_CONTACT = 2;
private Intent contactIntent;
...
contactIntent = new Intent(
    Intent.ACTION_PICK,
    ContactsContract.Contacts.CONTENT_URI);
...
startActivityForResult(contactIntent, REQUEST_CONTACT);
```

- Notice the URI is defined in a constant that looks suspiciously like a database schema class.
- This is no coincidence.

Content Providers

- Apps (and the base Android system) can come with “content providers”.
- These allow other apps to query their databases.
  - Indirectly! But still very similar to what we’ve seen before.
- Contact information is made available this way.
  - The `ContactsContract` class is essentially a gigantic schema.
  - Contact information is not just one database table, but several.
    - Each person has one name, but a potentially indefinite number of physical addresses, phone numbers, email addresses, etc.
- We use `Cursors` (again) to access the results from a query.

Contact Selection Results

```
@Override
public void onActivityResult(int requestCode, int resultCode,
                             Intent ret)
{
    if( resultCode == RESULT_OK &&
        requestCode == REQUEST_CONTACT)
    {
        Uri contactUri = resultIntent.getData();

        String[] queryFields = new String[] {
            ContactsContract.Contacts._ID,
            ContactsContract.Contacts.DISPLAY_NAME
        };

        Cursor c = getActivity().getContentResolver().query(
            contactUri, queryFields, null, null, null);
        );
        ...
```

What Just Happened?

- The code on the previous slide runs *after* the user has selected a contact.
- The contacts app has just sent back a result intent.
- The result intent has its own data `Uri`.
  - Basically a “pointer” to a specific contact entry.
  - Behind the scenes, the contact app has *granted permission* for us to access this specific entry.
  - (Otherwise we’d get a security exception.)
- We use another object called a `ContentResolver` to actually obtain the data.
  - It has a query method, like `SQLiteDatabase`, but not quite as fully-featured.
  - We give it the URI to look at, and the fields (table columns) to obtain.
  - It returns a cursor, which you already know about.

Contact Selection Results – The Cursor

```
Cursor c = ...;
try
{
    if(c.getCount() > 0) // Verify at least one result.
    {
        c.moveToFirst();
        int contactId = c.getInt(0);          // ID first
        String contactName = c.getString(1); // Name second
        ... // Do something with the data
    }
}
finally
{
    c.close();
}
```

But. . . what about the *contact* information?!

- So far, we've only *picked* a contact. That's all the contact app allowed.
- We can only really get the contact's name and (internal-to-the-device) unique ID this way.
  - Perhaps you just want a name!
- How do we get email addresses and phone numbers?
- We need to make another content provider query!
  - Note: we *don't* send another intent. The user has already told us which contact they want.
  - We just need to ask the system for more information.

Querying Email Addresses (1)

- `ContactsContract.CommonDataKinds.Email` contains schema information for what we want.
  - (There's also `ContactsContract.CommonDataKinds.Phone`.)
- We'll reduce the noise by importing the nested class directly:

```
import android.provider.ContactsContract.CommonDataKinds.Email;
```

- Our new query looks like this:

```
int id = ...; // Contact ID (from previous query)

Cursor c1 = getActivity().getContentResolver().query(
    Email.CONTENT_URI,             // URI of the provider.
    new String[] {Email.ADDRESS}, // Info we want to get.
    Email.CONTACT_ID + " = ?",    // "Where" clause.
    new String[] {String.valueOf(id)}, // "Where" args.
    null, null);
```

Querying Email Addresses (2)

And now there's just another cursor to deal with:

```
Cursor c1 = ...;
try
{
    if(c1.getCount() > 0)
    {
        c1.moveToFirst();
        String email = c1.getString(0);
        ... // Do something with the email address
    }
}
finally
{
    c1.close();
}
```

BUT, We Need PERMISSION!

- If you try to execute this query, we'll get `SecurityException`.
- We don't have permission to access the contacts database in this way.
  - Finding contact *names* is fine, but email addresses and phone numbers are not.
- Our app must *ask* for permission in `app/src/main/AndroidManifest.xml`:

```
<manifest ...>
  <uses-permission
    android:name="android.permission.READ_CONTACTS" />

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

- But it's still up to the user to *grant* this permission.
- Consider catching `SecurityException` in case they don't.

Cameras and `FileProviders`

- We can already take photos, but (so far) they're only thumbnails.
- Android doesn't like encoding large data directly inside `Intent` objects.
- So what's the alternative?
- Define a `FileProvider`, a specific kind of `ContentProvider`.
  - This can define a particular file location (a not-yet-existing filename).
  - You give this, *as a Uri*, to the camera app.
  - You give the camera app permission to write to it.
- Note that the situation is reversed here!
  - This is *your* provider, and *some other* app will access it.

FileProviders: XML

- A lot of the setting up is done in XML files.
- First, in app/src/main/AndroidManifest.xml:

```
<manifest ...>
  <application ...>
    <activity ...>...</activity>
    <activity ...>...</activity>
    <provider
      android:name="android.support.v4.content.FileProvider"
      android:authorities="edu.curtin.myapp.fileprovider"
      android:exported="false"
      android:grantUriPermissions="true" >
      <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/files" />
    </provider>
  </application>
</manifest>
```

FileProviders: XML (2)

- There's a *lot* of indirect referencing going on here!
  - (The price for flexibility.)
- `FileProvider` needs something to provide, and this is what `<meta-data .../>` is for.
- Inside *that*, notice we wrote:

```
android:resource="@xml/files"
```

- This is a reference to `app/src/main/res/xml/files.xml`.
- You may need to create this from scratch. It should contain:

```
<paths>
    <files-path name="loc-name" path="loc-path" />
</paths>
```

  - "`loc-name`" is a publicly-visible name of the location.
  - "`loc-path`" is an actual directory (possibly "."), within your app's own internal storage area.

FileProviders: Meanwhile, In Java. . .

There's some setting up to do to launch the camera app.

- First, we need to define the photo filename.
- We need this in order to make a Uri, to provide to the camera app.
- And we'd also like to access the photo afterwards too!

```
private File photoFile;
...
// Location where the actual photo will be.
photoFile = new File(getActivity().getFilesDir(),
                     "photo.jpg");
```

- getFilesDir() returns a File object representing the app's local storage directory.
- We then use this as the base directory for "photo.jpg".

FileProviders: The Implicit Intent

```
private File photoFile;
private Intent photoIntent;
...
photoIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
Uri cameraUri = FileProvider.getUriForFile(
    getActivity(),
    "edu.curtin.myapp.fileprovider", // "Authority"
    photoFile);
photoIntent.putExtra(MediaStore.EXTRA_OUTPUT, uri);
...
```

- The camera app expects a `Uri` *as an extra*.
  - It's not in the "data" field, because there's no actual data (yet).
- `FileProvider.getUriForFile()` builds the required Uri.

Authorities

- Providers are identified by their "authority".
- In AndroidManifest.XML, we wrote:

```
android:authorities="edu.curtin.myapp.fileprovider"
```

- And we need to provide this when constructing the Uri:

```
Uri cameraUri = FileProvider.getUriForFile(
    getActivity(),
    "edu.curtin.myapp.fileprovider", // "Authority"
    photoFile);
```

- Every provider (across all apps on the device) must have a unique authority.
- The authority forms the main part of the Uri for accessing the provider. e.g.

```
content://edu.curtin.myapp.fileprovider/loc/photo.jpg
```

  - You don't normally deal with this directly though.

Granting File Write Permissions

- Remember PackageManager? We need again:

```
photoIntent = ...;
uri = ...;
PackageManager pm = getPackageManager();

for(ResolveInfo a : pm.queryIntentActivities(
                        photoIntent,
                        PackageManager.MATCH_DEFAULT_ONLY))
{
    getActivity().grantUriPermission(
        a.activityInfo.packageName,
        uri,
        Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
}
```

- Iterating over all the activities that *could* take a photo.
- Give them all permission to write to our photo file (Uri).

Taking photos – Summary!

- Set up the file provider in AndroidManifest.xml.
    - Create `app/src/main/res/xml/files.xml` as needed.
- Define the `File` representing the photo location.
- Build a `Uri`, based on the `File` and the file provider's authority.
- Grant write permission to the file, to all camera apps.
- Send the intent!

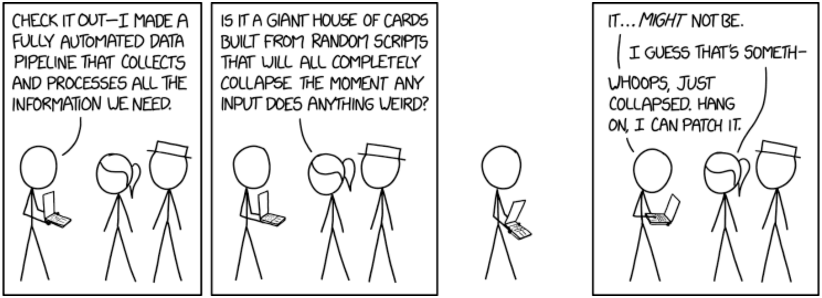
```
private static final int REQUEST_PHOTO = 3;
private File photoFile;
private Intent photoIntent;
...
startActivityForResult(photoIntent, REQUEST_PHOTO);
```

- In `onActivityResult()`, read the photo file...

Obtaining the full photo

```
@Override
public void onActivityResult(int requestCode, int resultCode,
                             Intent ret)
{
    if( resultCode == RESULT_OK &&
        requestCode == REQUEST_PHOTO)
    {
        Bitmap photo = BitmapFactory.decodeFile(
            photoFile.toString());
        ...
    }
}
```

- `BitmapFactory.decodeFile()` gives us an “bitmap”.
- WARNING! A bitmap is an *uncompressed* image. For a full-sized photo, this likely to take up a lot of memory.


CHECK IT OUT—I MADE A FULLY AUTOMATED DATA PIPELINE THAT COLLECTS AND PROCESSES ALL THE INFORMATION WE NEED.
IS IT A GIANT HOUSE OF CARDS BUILT FROM RANDOM SCRIPTS THAT WILL ALL COMPLETELY COLLAPSE THE MOMENT ANY INPUT DOES ANYTHING WEIRD?
IT... *MIGHT* NOT BE.
I GUESS THAT'S SOMETH–
WHOOPS, JUST COLLAPSED. HANG ON, I CAN PATCH IT.

(https://xkcd.com/2054/)