Java Essentials for Android Course

Lab 7: App Rater

Intro

For this lab you will be developing a new Application named AppRater that suggests other Applications for users to download and try. The purpose of the application is to share fun and interesting applications with other users. The users can then rate the applications.

Objectives

At the end of this lab you will be expected to know:

- *How to create and use Services using ServiceIntent.*
  - *How to start and stop Services.*
  - *How to perform repeated procedures using Timer and TimerTask.*
  - *How to broadcast Intents.*
- *How to respond to Intent broadcasts using a BroadcastReceiver.*
- *How to post Notifications in the Notification area using NotificationManager and PendingIntent.*
  - *How to create Notifications using a Builder.*
- *How to use and record changes in an interactive RatingBar.*
- *How to check for app install status on an Android device using the PackageManager.*

Activities

For this lab you will be working with a brand new application, completely independent of the previous labs. Once you have a general understanding of the components of the application you will begin implementing it. You will be given more general instructions in some sections. You should be comfortable enough with Android by now that you can figure more out on your own.

IMPORTANT:
You will be given a Skeleton Project to work with. This project contains all of the java and resource files you will need to complete the lab. Some method stubs, member variables, and resource values and ids have been added as well.

1. Setting Up

1.1 Creating the Project

To begin, you will need to download and extract the skeleton project for the AppRater application.

Extract the project, making sure to preserve the directory structure.

Take note of the path to the root folder of the skeleton project.

Next you will need to setup an Android project for this app. Since the skeleton project was created in Eclipse, the easiest thing is to import this project into Eclipse.

- Select **File -> Import**.
- In the Import Wizard, expand **General** and select **Existing Projects into Workspace**. Click **Next**.
- In the **Import Project** wizard, click **select root directory** and click **Browse**. Select the root directory of the skeleton project that you extracted. Click **Open** and then **Finish**.
- Click on the project name in the Package Explorer. Select **File -> Rename** and change the name of your project to lab7<userid> where <userid> is your user id (e.g. jsmith).

Make sure to set up your project for using ActionBarSherlock. See Labs 3, 4 and 6 for examples of this.

Make sure the Manifest SDK targets the latest version of Android (17 at the time of updating this lab) and that the minimum API target is set to 10 (Gingerbread).

1.2 Familiarize Yourself with the Project

The project contains a sizable number of Java files and several XML files. This app has no contextual action menu, since everything is handled from within the application views or various Intents.

AppRater.java will contain the definition for the main AppRater Activity class. This is the class that will display the list of applications the user is supposed to test and rate. The AppRater class makes use of a very simple XML layout file called **app_list.xml** which has been completed for you. It is composed of a ListView that displays a list of applications that a user is supposed to try out.

Applications that users are supposed to try out and rate are represented by the App model class, defined in **App.java**. This is a simple class that encapsulates the name of the application, a rating for it, the Market-URI from which the app can be downloaded (*we'll discuss this later, but think of it as a link to installing it from the Market application for now*), a boolean used to flag the application as installed, and a unique float ID.

The **AppView** class is a custom class that is used for visualizing the state of an App object. Similar to how JokeView and Joke worked in Lab 4, the AppView class has an App member variable on which it bases the state of its display. Its layout is defined in the **app_view.xml** XML layout file, which you will will out shortly.

An AppView has three different "states":

- If the AppView's corresponding App has not been installed yet, then its background color is red.
- If the AppView's corresponding App *has* been installed but *hasn't* been rated, then its background color is yellow.
- If the AppView's corresponding App has been installed *and* rated, then its background color is green.

The AppRater Activity gets the Apps it will display from the **AppContentProvider** class as a Cursor of Apps. This is quite similar to how the JokeList Activity in Lab 4 got a Cursor of Jokes from the JokeContentProvider. The AppRater Activity's ListView then uses the **AppCursorAdapter** class to bind AppViews to the cursor of Apps. If you are not familiar with Content Providers, it is strongly recommended that you complete Lab 4 before continuing this lab.

The AppRater Activity starts the **AppDownloadService** in order to add new Apps to the underlying database. The new Apps are retrieved from a website through this Service. Using a Service is like launching a new Activity except that there is no User Interface for the Service. The AppDownloadService then downloads and inserts new Apps into the database if they do not already exist, and occasionally checks for new Apps to add as long as it is running. In order to insert new Apps into the database the AppDownloadService must use the AppContentProvider's **insert(...)** method via accessing a Content Resolver. This is similar to the download functionality covered in Lab 5.

Once the AppDownloadService has successfully added a new App, it broadcasts a special Intent with a certain action associated with it. The AppRater Activity will use its internal **DownloadCompleteReceiver** class to listen for and match that special Intent. When it sees the Intent it knows that it has to update the list of apps in AppRater with a new Cursor.

Once an App is downloaded into the list, it may then be installed by clicking on it to launch the Google Market/Play Store that shows the information about the App as it appears in the store. The user can then test the app freely outside of the AppRater application. The Apps that are installed are marked as such in the ListView via a non-interactive CheckBox as well as the background color mentioned above.

Lastly, a user can give each App a rating. The user can apply a rating to an App via a RatingBar. The AppRater Activity then saves the changes to the App through the AppContentProvider using a custom **OnAppChangeListener** (similar to the OnJokeChangeListener from Lab 4).

The following classes/files have been fully filled in for you:

- App.java
- AppContentProvider.java
- AppCursorAdapter.java
- AppDatabaseHelper.java
- AppTable.java
- app_list.xml

You will fill in the remainder of these classes/files:

- AppDownloadService.java
- AppRater.java
    - DownloadCompleteReceiver
- AppView.java
- app_view.xml

Study the classes and files that have been fully filled in for you and make sure you understand what's going on inside them before moving on. In particular, the Content Provider classes should be familiar from Lab 4: AppRater will store all App data inside of a database wrapped by a content provider, which provides an interface to the main application for interacting with the database in the form of a table.

2. AppView

Similar to how Lab 4 had JokeView, AppRater has an **AppView** class to implement. This is the View representation of an App in the list of Apps, and contains data from an internally stored App object.

In this section you will implement the XML layout for the AppView class, and then fill out the AppView class itself. There will be some minor modifications to make to other classes along the way as well.

2.1 AppView XML Layout

Each AppView will have three main components: A TextView containing the name of the app, a *non-interactive* CheckBox (meaning the user cannot change its checked state) and a RatingBar that lets the user dynamically change their rating of the app.

When you have finished implementing the XML layout file, it will look something like this (click image for full view):

Remember that this is just one instance of the AppView. It will scrunch to fit minimally inside the ListView.

Before you begin, let's talk RatingBars and why it gets its own row in the above image.

2.1.1 About Android's RatingBar

A RatingBar is something you've seen in Android before. When you install an application from the Google Play Store, you see a "Rate and Review" area on an app's Play Store informational/install page (if you need a refresher, there is an image in section 3.1.2 below that demonstrates this) and it's got a RatingBar for rating the application.

RatingBar is a great app component in theory--it acts as a subclass of a progress bar, so the user can touch a star to give the RatingBar a rating ***or*** slide their finger along the RatingBar until they release it, and then the rating is set. You can change the size of the default RatingBar in two ways, modify the step size (e.g. to enable half-star ratings such as 3.5 or 4.5 out of 5

stars, set the step size as 0.5), the number of stars and set the current rating at any time. Furthermore, you have the option of styling the RatingBar to add custom rating "star" images, sizes and progressive drawing (meaning, Android has a built-in mechanism to automatically fill in an image in the bar based on step size, so you only need to provide a full and empty "star" image!). Finally, it has its own Listener for detecting a rating change.

Unfortunately, there are just as many problems with this component as there are perks. For starters, read the Class Overview section in the RatingBar documentation. You'll notice some immediate restrictions:

- You can change the size of the RatingBar to two other defaults. If you want a smaller RatingBar, for example, all you have to do is change the RatingBar's style to `ratingBarStyleSmall`. Unfortunately, by doing so ***you lose the ability to interact with the RatingBar entirely***. If you want to place an interactive RatingBar in a layout, it's got to be the default one.

- If you want an interactive RatingBar, Google discourages placing any components to its left or right. This restricts your layout options significantly.

- If you want to change the number of stars in the RatingBar, you must declare its layout_width to be **wrap_content**. Anything else will produce unpredictable rendering results.

You can choose to style your own RatingBar to change its size and other properties while still preserving interactivity, but then you must worry about supporting multiple screen densities and sizes, which means you are stuck having to create a series of icons for each density or be forced to limit the size of your stylized RatingBar to the size of the custom icons. Even then, you may see a stretched 'bleed' effect on custom rating bar styles.

Feel free to experiment with stylizing the RatingBar in your application, but don't make it your life's mission. RatingBar is very restrictive and unpredictable.

2.1.2 app_view.xml

Now that you know how unruly the default RatingBar component can be (and that you're stuck with it unless you want to take a crack at stylizing it) you can plan your implementation of the XML file around this outstanding issue.

Open up **app_view.xml**. If you preview it as-is in the Graphical Layout tab, everything is in disarray because the TextView text is not set and the default ViewGroup is a RelativeLayout without any organizational properties set.

You are tasked with filling in the remaining component attributes to make the view look like the above image. Hints follow:

- You can change the ViewGroup root to something other than RelativeLayout if you'd like. If you do, you must change the type of the variable **m_vwContainer** in **AppView.java** as well as its extended class to reflect this.

- The RelativeLayout guide could prove useful if you choose to stick with it.

- The RatingBar should be interactive, have only 5 stars and only allow for whole integer ratings (i.e., ratings such as 1.0 and 4.0 are allowed, but not 1.5).

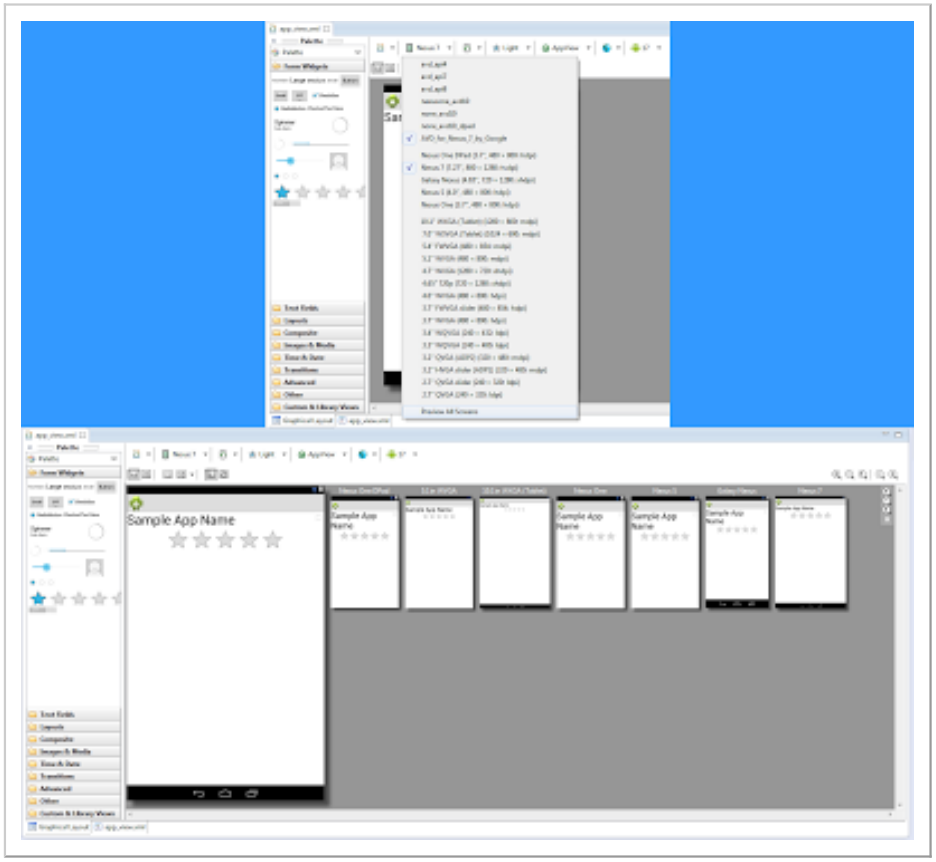
- Check the **dimens.xml** and **strings.xml** files for values for the TextView's text size and sample app name text, respectively. Feel free to change them as you please.

    - If you haven't already, you should get used to putting all of your values in Values files instead of declaring them directly inside of a component's property.

- Remember that the CheckBox must be non-interactive. This means you should modify a property so it can't be clicked.

- Remember that focus problem regarding the ListView's OnItemClickListener? You don't need to set each focusable component inside the root ViewGroup to non-focusable. Just set the root ViewGroup's `descendantFocusability` property appropriately.

- Curious to see how the layout looks on multiple devices across different resolutions? There is a Preview All Screens option at the top of the Graphical Layout tab that will show previews for all AVD devices and densities/resolutions. You can then click on any of the screens to blow it up to full size, or zoom in as you please. See image below for an example (click image for full size):



This might help you catch problems that happen on some devices (but not all) more easily.

2.2 AppView.java

Now that we have our static layout, we can implement the Java class. Open **AppView.java** and examine the class again before continuing on to do the implementation.

2.2.1 Layout and App Initialization

Start by filling in the constructor for AppView.

- Initialize the Context class variable.
- Inflate the **app_view.xml** layout.

- Initialize all View components (**m_vw***) appropriately.
- Make a call to **setApp(...)**.
- Set the onAppChangeListener to **null**.

Fill in the **setApp(...)** method.

- Set **m_app** appropriately.
- Check to see if the app in question is installed on the Android device. Change **m_app** and the CheckBox's checked status appropriately based on the app's install status.
    - The PackageManager class will be helpful here.
    - You need the package name of the app, but you have the Market information/installation URI in **m_app**. See section 3.1.2 below for a table that shows the format of this URI. You'll have to extract the package from the URI somehow so the PackageManager can check with the correct package String.
    - Don't forget to call **notifyOnAppChangeListener()** whenever **m_app**'s data changes (e.g. when the app goes from uninstalled to installed).
- Set the app name text appropriately.
- Change the AppView's background color depending on installation and rating status.
    - The three color resource IDs you need are in **colors.xml**.
    - Change **m_vwContainer**'s background color depending on whether the app is:
        - Not installed (red)
        - Installed but unrated (yellow)

        - Installed and rated (green)

        - *You do not have to notify the listener when this is set, since the color of the AppView in the list is not stored in the database.*

- Set the RatingBar's rating appropriately.

2.2.2 OnRatingBarChangeListener

Similar to how the RadioButtons in Lab 4 had an OnCheckedChangeListener associated with them, the RatingBar class has its own listener: **OnRatingBarChangeListener**. You will now implement this in AppView.

Make AppView implement **OnRatingBarChangeListener**.

- Override **onRatingChanged(...)**.

    - Change **m_app**'s rating to the value in the rating parameter.

        - *This means that **m_app**'s data has changed. You have one more thing to do when this happens.*

Now AppView is ready for action. Unfortunately, the rest of the pieces aren't quite ready yet, so you cannot test the application right now. After the next section that will change.

3. Services

In this section you will implement a Service called **AppDownloadService** that downloads a list of applications from a web-hosted file and populates AppRater's ListView with AppViews. This of course means that you will need to implement AppView, but we will first focus on the class where Apps are conceived in the application scope.

The flow of data in the application is similar to that of Lab 4, in particular the content provider data cycle, but contains the following extra behaviorisms:

-Download list of app data to populate ListView with AppViews
-Broadcast an intent when a new app is added
-Receive broadcast and launch Notification in Notification Bar that new app is ready
-Launch Intent to install app on user's device when an app in the list is touched

What are Services?

Simply put, a ***Service*** is like an Activity without a user interface that runs in the background. An application can start a Service, let it do its thing in the background, and still provide app functionality without any disruption to its user. Any time you need to perform long and/or routine tasks in your code, consider enlisting Services to help accomplish this. However, there are some key points to Services that must be understood before using them:

1. Services are not separate processes. When launched, they run in the same process as the application they were started in. This means you should not use them for work-heavy processes like playing MP3s. See the documentation on Processes and Threads for more information on how to run CPU-intensive asynchronous tasks.

2. Services are not threads. They are not a method for working off of the main running thread. However, IntentServices (which will be discussed shortly) do have their own thread.

3. Services can be accessed by other applications than the one that started them. This creates a sort of client-server interface, but requires much more code setup than running a service local to just one application. We will not be using this functionality, but for more information on this read the Bound Services documentation.

4. Services are not AsyncTasks, as similar as they may be. The latter is designed to run very short operations that do one thing and quickly exit. The former is designed to run for longer events and even when your application Activity is not open. Sometimes it's better to use one over the other. For our lab we want to run recurring code on a timer over more than just a few seconds, so we will not be using AsyncTask. For more information on AsyncTask read its documentation.

For more information and a guide on Services, look here.

3.1 AppDownloadService

The first step in this lab is to implement an IntentService called **AppDownloadService**. An ***IntentService*** is a Service started using an Intent. You already have experience starting

something with an intent: Launching a device's Camera using an Intent (Lab 6). An IntentService is launched exactly the same way, but this time we are going to implement our own Intent instead of rely on Android's built-in one.

For more information and an example of an IntentService, look here.

3.1.1 Extend IntentService

Open up AppDownloadService.java and read the documentation for the methods you will implement. However, before you implement them you will first make AppDownloadService extend IntentService:

- Make AppDownloadService extend **android.app.IntentService**.
    - Fill in the constructor with the appropriate **super(...)** call, passing in the name of your service.
    - Override **onCreate()**.
        - Initialize **m_updateTimer** to a new Timer object.
        - Initialize **m_updateTask** to a new TimerTask anonymous inner class, overriding its **run()** method that will simply call **getAppsFromServer()**.
        - Call **super.onCreate()**.
    - Override **onDestroy()**.
        - Cancel **m_updateTimer**.
        - Call **super.onDestroy()**.
    - You have one abstract method to implement: **onHandleIntent(...)**.

- *Note that **onCreate()** is called before **onHandleIntent(...)**, therefore initialization in **onCreate()** was ideal.*

- Make a call to **m_updateTimer**'s **scheduleAtFixedRate(...)** method.

    - Fill in the parameters using the appropriate class member variables given to you, passing in **0** for the delay.

    - *This will cause the TimerTask, which calls **getAppsFromServer()**, to be run every so often.*

3.1.2 Get Apps From Server

As you can probably guess from the above subsection title, you will now fill in **getAppsFromServer()**. This method will fetch app data from a file hosted on a server and then translate that into a new App object that will be added to the ListView in the next subsection.

All app data will be on a single line. All data for an app is comma-delimited, with apps separated by a semicolon. The format of a single app's data you will be reading in is as follows:

```
AppName,market://details?id=app.package.name;
```

A formatted example of data in this file for three apps is given below:

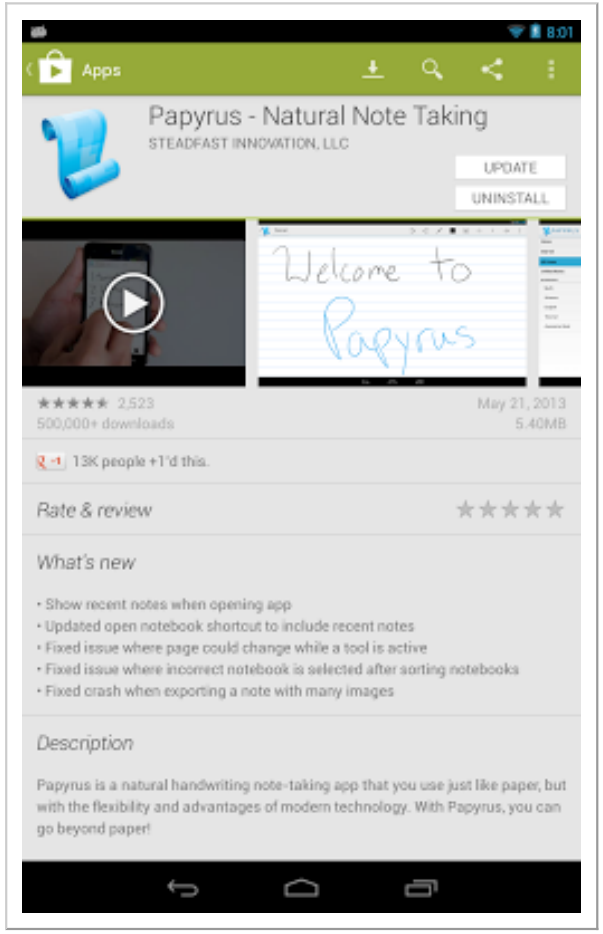
```
Papyrus,market://details?id=com.steadfastinnovation.android.projectpapyrus;Flow Live
Wallpaper,market://details?id=pong.flow.lite;Go Make It Rain,market://details?
id=com.fireswing.makeitrain;
```

This corresponds to the following information about the apps:

App Name	App Market Information URI	App Package Name
Papyrus	market://details?id=com.steadfastinnovation.android.projectpapyrus	com.steadfastinnovation.android.projectpapyrus
Flow Live	market://details?id=pong.flow.lite	pong.flow.lite

Wallpaper		
Go Make It Rain	market://details?id=com.fireswing.makeitrain	com.fireswing.makeitrain

The App Market Information URI is the URI corresponding to the app's information/installation page if you search for and touch the app in the Google Play Store. For example, if you went to the Google Play Store, searched for the Papyrus app and then clicked on it in the search results for more information, it would take you to the page that corresponds to its App Market Information URI. See below image for clarification (click image for full view):



This page in the Play Store corresponds to the URI ***market://details?id=com.steadfastinnovation.android.projectpapyrus***. URIs are everywhere!

Anyway, let's get to downloading!

Fill in the **getAppsFromServer()** method.

- Create a new URL object and initialize it with the URL String provided in the class. This is **http://www.simexusa.com/aac/getAll.php**.
    - Feel free to look at the website that URL points to in your web browser to see what the contents look like.
- For each app you read in call **addNewApp(...)**, passing in the app name as the first parameter and the app information/install URI as the second.
    - You are free to do this as you please. However, consider using Java's InputStreamReader class.
- You will have to use a try-catch block to catch possible Exceptions. If one happens, just post an Error-level Logcat message from your application, with the package as the tag and the exception's **getMessage()** as the message.
- *Note: This method is a great place to place break points in Eclipse if you are trying to debug issues with adding apps to the list.*

3.1.3 Add New App

To no one's surprise, this subsection has you filling in the **addNewApp(...)** method. This is where the content provider app insertion occurs.

Fill in the **addNewApp(...)** method.

- *Note: This method acts very similar to the process of inserting a new Joke into the joke database table in Lab 4. If you get stuck with this method, reference the insertion process in your Lab 4 project.*

- Fetch the ContentResolver for AppRater.
    - *The ContentResolver obtained here allows access to the operations in the AppContentProvider. For example, calling **query(...)** from the ContentResolver object will invoke **AppContentProvider.query(...)**. There's no need to set up a Cursor managing system to perform operations on the database or anything like that. So easy, right? ContentResolver is the real hero here.*
- Create a Uri using **Uri.parse()** and perform a query on the ContentResolver for an app in the database with the passed-in App's name.
    - Look at **AppContentProvider.query(...)** if you want to see which Uri format it expects for querying for a single database row.
- If the Cursor returned from the query contains 0 rows, this means that the app does not yet exist in the database and therefore you should add it in.
    - Declare and initialize a new ContentValues object, putting in the app's data (name, rating, installURI, installed) one at a time.
        - There is no Boolean data type in SQLite. Instead, you will use 0 to represent false in the database and 1 to represent true. See the SQLite Datatypes documentation for more information.
    - Parse a new Uri for insertion into the database.
        - Look at **AppContentProvider.insert(...)** if you want to see which Uri format it expects for inserting a single database row.
    - Perform the insertion into the database.
        - Retrieve the ID from the Uri returned from the result of the insertion and

assign it as the app's ID variable.

    - Call **announceNewApp()**.

- Close the Cursor object you retrieved from the earlier query. Because you are not using CursorLoader in this class, you are responsible for closing any Cursors you open.

3.1.4 Announce New App

Finally, you will fill in the **announceNewApp()** method. This is where Intent broadcasting happens.

Fill in the **announceNewApp()** method.

- Create a new Intent and set its Action to the String **ACTION_NEW_APP_TO_REVIEW**.
    - This String can be found in the **DownloadCompleteReceiver** subclass inside of the **AppRater** class.
- Set the Intent's category to **Intent.CATEGORY_DEFAULT**.
    - *This is necessary for matching the soon-to-be broadcast Intent with the broadcast receiver. Don't forget this!*
- Invoke **sendBroadcast(...)**.
    - *Note: If you know there will only be local broadcasts of Intents within your application's process, there is an optional LocalBroadcastManager you can use. We leave it up to you to implement this feature in place of **sendBroadcast(...)** if you wish to use it.*

That does it for implementing the IntentService. Now we have to actually include it in our

application.

- In the Manifest file, add the following line between the **<application>** open and close tags:

```
<service android:enabled="true"
android:name="edu.calpoly.android.apprater.AppDownloadService"/>
```

- Since we are downloading data from a file hosted online, we obviously need to include a permission for accessing the internet. Add the following line above the **<application>** open tag:

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

3.1.5 Starting and Stopping Services

Now that we have our IntentService, we have to incorporate it into AppRater. To use a Service, you start it using **startService(...)**. To stop a Service, you stop it using **stopService(...)**. Simple enough, right? You will implement this start/stop functionality in AppRater's menu items.

Open up **AppRater.java** and look at the overridden **onCreateOptionsMenu(...)** method. Based on the inflation going on, there has already been a menu layout file created for you, **mainmenu.xml**. Take a look at that file and see what the menu will contain: three menu options. These options will allow users of the application to start or stop the service that downloads the apps from the server to the list and rechecks the server periodically for more, or remove all apps from the list.

In **AppRater.onOptionsItemSelected(...)**:

- If the start download menu item is pressed, start the AppDownloadService. You will need to pass in an Intent with the proper context and service class.

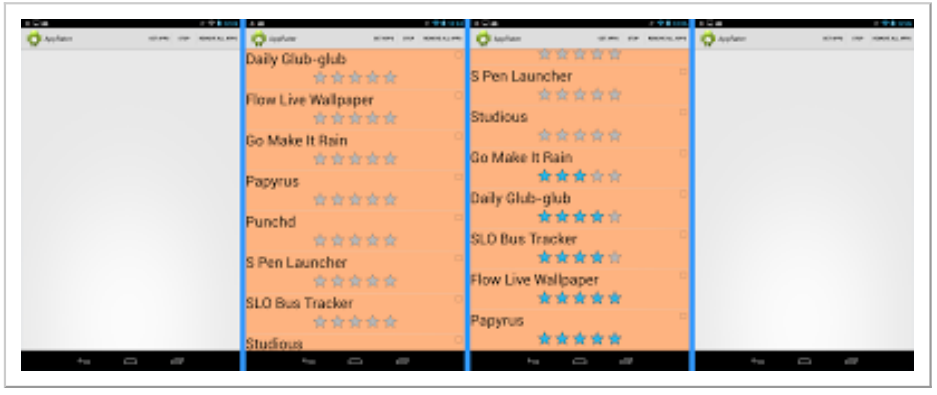
- If the stop download menu item is pressed, stop the AppDownloadService. You will need to pass in an Intent with the proper context and service class.

- If the remove all apps menu item is pressed, remove all apps from the list.
    - Construct a Uri to remove all apps from the list. Look at **AppContentProvider.delete(...)** to format the correct Uri properly.
    - Retrieve a ContentResolver instance for the application and invoke its **delete(...)** operation method, passing in the Uri.
    - *It is reasonable to keep the service running when remove is pressed. The user will have to press Stop to stop the service manually.*

In **AppRater.fillData()**:

- Instead of reassigning the adapter to the ListView after restarting the CursorLoader like you did in Lab 4, call the adapter's **notifyDataSetChanged()** method.
    - *This prevents the ListView from always scrolling back up to the top of the list on every change, but preserves its refresh.*

Hooray, now we can finally run the application! If you run it you should see something similar to the following (click image for full size):



You can assign ratings to apps that aren't installed by default. You can change this behavior later on if you'd like.

The above image contains screenshots from a continuous run of the application up until this

step. Each screenshot shows what happens when the following actions are taken:
1: Starting the app up fresh
2: Pressing the "Get Apps" button
3: Rating several apps
4: Pressing the "Remove All Apps" button

The application should also persist ratings when you move away from the application by pressing Back or Home, and also when orientation changes. Just one of the many benefits of using a database for persistence!

Note that you **can** set an application's rating back to 0/5 stars. This is perfectly acceptable.

Now we have covered the following data flow segments:

-Download list of app data to populate ListView with AppViews
-Broadcast an intent when a new app is added

Now we need to actually catch the broadcast we sent. We'll do this and more in the next section.

4. The Broadcast Receiver

Great, so we've added apps to the database and told our application that there's a new app to review. Our application needs to catch wind of that broadcast somehow, and it will be done using a Broadcast Receiver. In this section, you will implement a custom BroadcastReceiver for catching Intents broadcast from AppDownloadService. You will then have this BroadcastReceiver (**DownloadCompleteReceiver**) show a Notification in the Notification Bar on your device every time there is a new app added to the list.

4.1 DownloadCompleteReceiver

A BroadcastReceiver is essentially a type of listener that watches for specific Intents launched via **sendBroadcast(...)**. Our AppDownloadService already invokes this call, but so far we have not acted to catch it. By implementing a BroadcastReceiver, we not only gain complete control over what to do when a specific broadcast is received, but we can handle these Intents from potentially *any* application as long as we register the BroadcastReceiver properly! (We won't be doing this, but it's still nice.)

In this subsection, you will implement the **DownloadCompleteReceiver** class which is nested inside of the AppRater class. In DownloadCompleteReceiver, if a broadcast Intent with the action **ACTION_NEW_APP_TO_REVIEW** is received, we want to update the list of apps and post a Notification to the Notification Bar. You will implement the former and set up the latter in this subsection.

Fill in the **onReceive(...)** method in **DownloadCompleteReceiver**.

- If the received Intent's action matches **ACTION_NEW_APP_TO_REVIEW**:
    - You may optionally make a Toast here. If you do, use **R.string.newAppToast** as the Toast text.
    - Make a call to **fillData()**.
        - *This may or may not be required depending on how you handle new app arrivals.*
    - Make a call to **showNotification(...)**.
        - *You will implement this in section 4.2.*

4.1.1 Registering and IntentFilters

You have to register a BroadcastReceiver with the application in order to properly listen for Intents. You will do this by calling **registerReceiver(...)** and **unregisterReceiver(...)**, respectively.

Remember IntentFilters? You might not, considering they were pretty out of your way for the majority of these labs, but they've been right under your nose this whole time. You used them in every lab so far to indicate which Activity was the main Activity in your Manifest file, and now they're back for another round. When you register a BroadcastReceiver, you must pass it an IntentFilter so that it knows to search for Intents with appropriate categories. Earlier, you set the broadcast Intent's category to **CATEGORY_DEFAULT**, so you will add this as a category to an Intent Filter that gets paired up with your DownloadCompleteReceiver.

For more information about Intents and Intent Filters, see the guide here.

Properly register and unregister your DownloadCompleteReceiver.

- In **AppRater.onResume()**, before the **super()** call:
    - Create and initialize a new IntentFilter with the proper Action and Category.
    - Initialize **m_receiver** and call **registerReceiver(...)**.
- In **AppRater.onPause()**, before the **super()** call:
    - Call **unregisterReceiver(...)**.

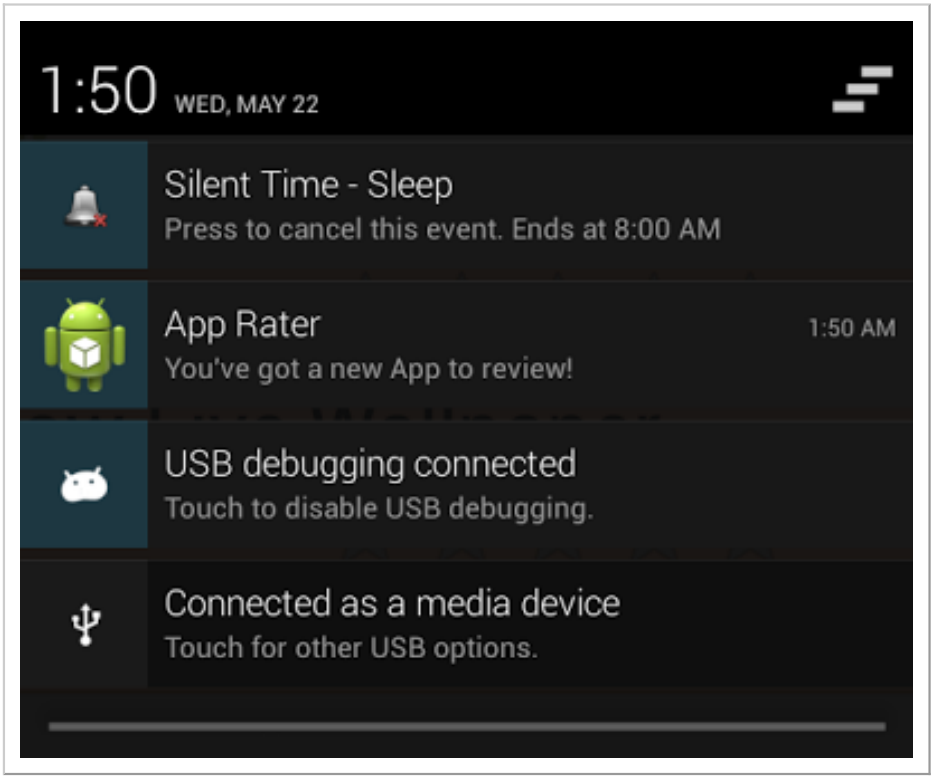
4.2 Notifications

"You've got a new App to review!"

Probably the coolest feature introduced in this lab, Notifications are messages that show up in an Android device's notification area (sometimes referred to as the Notification Bar or Notification Drawer). They have two display types: Normal and Big. We will be creating Normal view Notifications in this step.

If you use an Android device on a regular basis, you are likely more than familiar with Notifications: They're those small notices you get from various apps and services. If you get an email in a Gmail account that's hooked up to your Android device, you get a notification. On newer devices, if you take a screenshot then you'll get a notification in the area too. These are just a few examples of many.

Below is a screenshot of the Notification Drawer in Android 4.2 (Nexus 7) with some sample Notifications (click image for full size):



You will obviously be implementing one of these.

Something you may not be familiar with is that Notifications have a very large pool of options on the programming side. Because of this, Google has added a Notification.Builder class in API 11 (Android 3.0, Honeycomb) to allow for easy Notification customization and creation. However, since we want our backwards-compatibility (and Google does too) we will use the NotificationCompat.Builder class that comes with the v4 Support Library. Take a look at that Builder class and see just how much we can do with Notifications!

Before you go completely nuts with Notifications though, be warned that Notification backwards-compatibility is not 100% functional across different versions of Android. For Notification guidelines, visit the Notifications Patterns page.

Once built, Notifications cannot post themselves, the NotificationManager has to be made aware of them and post them itself. To do this, Notifications use PendingIntents. A PendingIntent is basically a regular Intent--it contains some action to perform. But unlike regular Intents, PendingIntents grant the right to perform the action or operation in them to whatever they are given to to be executed. In this case, a PendingIntent allows the NotificationManager to post a Notification to the Notification area. The NotificationManager can be considered a 'foreign'

application, and it will use your application's permissions instead of its own.

In DownloadCompleteReceiver, fill in the **showNotification(...)** method.

- Obtain an instance of this application's Resources from the context.
- Create a new PendingIntent using the static **getActivity(...)** method in the PendingIntent class.
    - The Intent parameter should be an Intent that points to the AppRater class.
    - If you are unsure of what to pass in for certain parameters, **0** is acceptable.
- Create a local NotificationCompat.Builder variable and instantiate it with the context parameter.
- Looking at the Normal view Notification guide and the above image, set the appropriate properties of your soon-to-be Notification in the Builder.
    - See if you can figure out which resources to use for which properties.
        - While you are using the NotificationCompat.Builder class, the documentation for Notification.Builder is sometimes more substantial for certain methods.
        - One property you also need to include is the Ticker text. This is the text that appears briefly in the minimized Notification area when the Notification is first added to it before disappearing after several seconds.
    - Call **setContentIntent(...)** on the Builder.
    - Call **setAutoCancel(...)** with true on the Builder. This will make the Notification disappear from the Notification area when it is pressed.

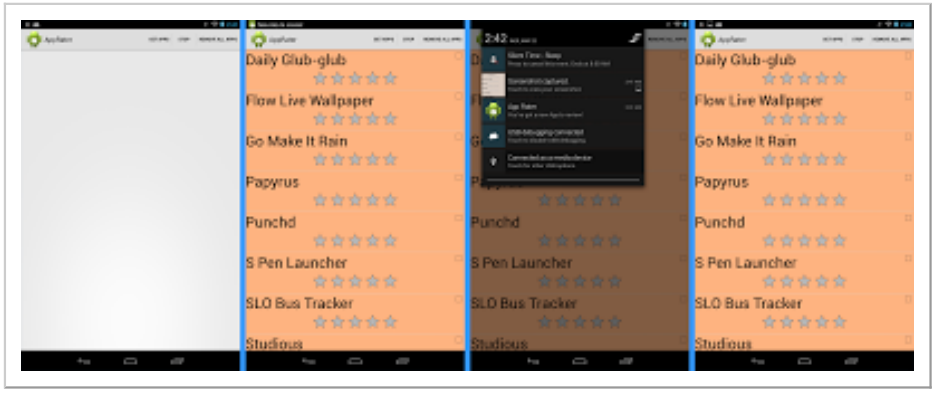
- *You can call **setDefaults(...)** on the Builder with **DEFAULT_ALL** as the parameter if you don't want anything special added to your Notification when it's built.*

- *Feel free to also play around with other properties such as the lights or anything else you are interested in exploring.*

- Obtain a reference to the almighty **NotificationManager**.

  - Call its **notify(...)** method, passing in the appropriate Notification ID and the call to your Builder object's **build()** method.

Run your application and make sure that the Notification ticker text pops up in the minimized notification area, then that the Notification is in the maximized Notification area (click image for full size):



Screen capture Notification included as a bonus.

The above image contains screenshots from a continuous run of the application up until this step. Each screenshot shows what happens when the following actions are taken:
1: Starting the app up fresh and pressing the "Get Apps" button
2: Observing Ticker text and Icon appear in minimized Notification area (Icon is not properly sized, this is expected if you choose too large of an icon unless you want to add your own that is the proper size)
3: Expanding Notification area to show App Rater Notification with properly-sized Icon
4: Touching the App Rater Notification minimizes Notification area and removes App Rater Notification

We have now finished implementing the third piece of new data flow in this application:

-Receive broadcast and launch Notification in Notification Bar that new app is ready

Now there's only one more thing to do!

5. App Installation & Play Store Intent

Your final task is to implement the final piece of new data flow in this application:

-Launch Intent to install app on user's device when an app in the list is touched

Right now, the application is a sea of red (or orange) because we have not been able to install any of these apps yet. That is about to change! You may have noticed that touching or pressing one of the AppViews has shown some level of interaction: certain components appear to be selected while holding down the touch or press on the AppView until you let go.

- If this is ***not*** the case, then you have a focus issue and should resolve that by removing focus from focusable components in **app_view.xml** (it was hinted at in a previous step).
- If this ***is*** the case, then you are all set to incorporate the final piece of the puzzle, which is filling in the **onItemClick(...)** method in AppRater to launch the Google Play Store/Market Intent using an App's installURI.

As you have probably anticipated from the beginning, you are left to implement this final task yourself. However, hints follow:

- Look at **MARKET_REQUEST_CODE**. It may be of use to you.
- The Intent page has a certain String that you ***must*** use to launch the Market/Google Play Store Intent properly.
- Make sure app status refreshes immediately upon returning to your application. For

example, if the Papyrus app is not installed, but you install it after launching the Google Play Store Intent and then return to your application, it should change to green or yellow (depending on rating) and not red.

- Some devices may not be compatible with certain applications. If this is the case, ignore them or, better yet, be adventurous and devise a method for injecting your own set of apps into the list (this requires knowing package names of applications already in the Play Store, though).

- *It is up to you to decide if you want to allow the Intent to be launched if the app is already installed on the Device.*

Run your application and make sure that you can now launch Play Store/Market Intents that show the application and allow you to install the application, and that an app changes appropriately when you install, rate or uninstall an application. A sample storyboard is presented below (click image for full size; warning, it's large!):

Certain screens may vary depending on device. Above is taken on a Nexus 7 running Android 4.2.

The above image contains screenshots from a continuous run of the application up until this step. Each screenshot shows what happens when the following actions are taken:

1: Fresh start of application after clearing Notification
2: Clicking S Pen Launcher app (can't install on Nexus 7, sad day)
3: Going back and clicking Papyrus app
4: Going through motions of installing Papyrus app
5: Installation of Papyrus app in progress
6: Installation of Papyrus app complete
7: Press back; Papyrus now yellow
8: Papyrus rated 5 stars; now green and ordered accordingly
9: Papyrus rated 0 stars; now yellow
10: Studious app rated 4 stars pre-installation
11: Going through motions of installing Studious
12: Fast-forward to where Studious is now installed; Studious now immediately green

13: App flipped to landscape orientation
14: About to uninstall Studious app (many ways to do this depending on device)
15: Uninstall Studious
16: Uninstall of Studious complete
17: Go back to App Rater
18: Studious now uninstalled; Studious still rated 4 stars and now red

In conclusion...

This lab was a brief introduction to Services, BroadcastReceivers and Notifications, with some other components thrown in for good measure. We hope you learned how to implement some really cool features, and that you can use this lab as a springboard for further Android development!

6. Deliverables

To complete this lab you will be required to:

1. Put your entire project directory into a .zip or .tar file, similar to the stub you were given. Submit the archive to PolyLearn. This effectively provides time-stamped evidence that you submitted the lab on time should there be any discrepancy later on in the quarter. The name of your archive should be **lab7<userid>** with a **.zip** or **.tar** extension. So if your username is **jsmith** and you created a zip file, then your file would be named **lab7jsmith.zip**.
2. Load your app on a mobile device and bring it to class on the due date to demo for full credit.
3. Complete the survey for Lab 7**:** https://www.surveymonkey.com/s/436F13Lab7

Primary Authors: James Reed and Kennedy Owen
Adviser: Dr. David Janzen

Comments

Commenting disabled due to a network error. Please reload the page.

You do not have permission to add comments.