50 Android Hacks

Carlos Sessa

FOREWORD BY Jake Wharton

MANNING

SAMPLE CHAPTER

50 Android Hacks

by Carlos Sessa

Chapter 5

brief contents

5 Patterns

In this chapter, you'll read about different development patterns you can use inside Android.

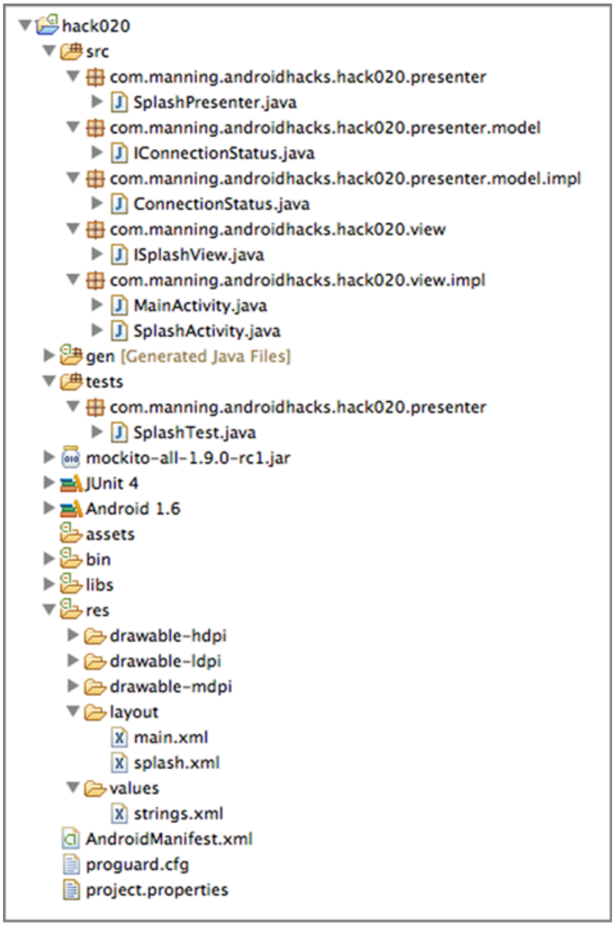
Hack 20 *The Model-View-Presenter pattern*

Android v1.6+

You've most likely heard of the MVC (Model-View-Controller) pattern, and you've probably used it in different frameworks. When I was trying to find a better way to test my Android code, I learned about the *MVP (Model-View-Presenter)* pattern. The basic difference between MVP and MVC is that in MVP, the presenter contains the UI business logic for the view and communicates with it through an interface.

In this hack, I'll show you how to use MVP inside Android and how it improves the testability of the code. To see how it works, we'll build a splash screen. A splash screen is a common place to put initialization code and verifications, before the application starts running. In this case, inside the splash screen we'll provide a progress bar while we're checking whether or not we have internet access. If we do, we continue to another activity, but if we don't, we'll show the user an error message to prevent them from moving forward.

To create the splash screen, we'll have a presenter that will take care of the communication between the model and the view. In this particular case, the presenter



```
▼ hack020
  ▼ src
    ▼ com.manning.androidhacks.hack020.presenter
      ▶ SplashPresenter.java
    ▼ com.manning.androidhacks.hack020.presenter.model
      ▶ IConnectionStatus.java
    ▼ com.manning.androidhacks.hack020.presenter.model.impl
      ▶ ConnectionStatus.java
    ▼ com.manning.androidhacks.hack020.view
      ▶ ISplashView.java
    ▼ com.manning.androidhacks.hack020.view.impl
      ▶ MainActivity.java
      ▶ SplashActivity.java
  ▶ gen [Generated Java Files]
  ▼ tests
    ▼ com.manning.androidhacks.hack020.presenter
      ▶ SplashTest.java
  ▶ mockito-all-1.9.0-rc1.jar
  ▶ JUnit 4
  ▶ Android 1.6
    assets
  ▶ bin
  ▶ libs
  ▼ res
    ▶ drawable-hdpi
    ▶ drawable-ldpi
    ▶ drawable-mdpi
    ▼ layout
        main.xml
        splash.xml
    ▼ values
        strings.xml
    AndroidManifest.xml
    proguard.cfg
    project.properties
```

Figure 20.1 MVP project structure

will have two functions: one that knows when we're online and another to take care of controlling the view. You can see the project structure in figure 20.1.

The presenter will use a model class called `ConnectionStatus` that will implement the `IConnectionStatus` interface. This interface will answer whether we have internet access with a single method:

```
public interface IConnectionStatus {
  boolean isOnline();
}
```

As you might be thinking, the code in charge of controlling the view will be an `Activity` that implements the `ISplashView` interface. The interface will be used by the presenter to control the flow of the application. Let's look at the code for the `ISplashView` interface:

```
public interface ISplashView {
  void showProgress();
  void hideProgress();
  void showNoInetErrorMsg();
  void moveToMainView();
}
```

Because we're coding in Android, the view will be the first to be created and afterward we'll give the control to the presenter. Let's see how we do that:

```
public class SplashActivity extends Activity implements ISplashView {
  private SplashPresenter mPresenter;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    ...

    mPresenter = new SplashPresenter();
    mPresenter.setView(this);
  }

  @Override
  protected void onResume() {
    super.onResume();
    mPresenter.didFinishLoading();
   }

}
```

❶ **Activity initialization code**

❷ **Instantiate presenter for this Activity**

❸ **Start presenter code when we reach onResume() method**

We'll first need to initialize the `Activity` ❶. Afterward, we create the presenter ❷ that will take care of getting everything done and we set the `Activity` instance to the presenter. We can override the `onResume()` method ❸ to let the presenter know the view is ready to give control to it.

The presenter code is simple. Following is the presenter's `didFinishLoading()` method:

```
public void didFinishLoading() {
  ISplashView view = getView();

  if (mConnectionStatus.isOnline()) {
    view.moveToMainView();
  } else {
    view.hideProgress();
    view.showNoInetErrorMsg();
  }
}
```

❶ **Getting view, in this case the Activity**

❷ **Logic to decide if we can move on**

We'll get a reference to the `ISplashView` implementation using a presenter's getter ❶. We'll use the model's `IConnectionStatus` implementation to verify whether we're online ❷. Depending on that, we'll do different things with the view. As you can see, the view is used through an interface without knowing it's implemented by an Android `Activity`. This will end up in a view that's easy to mock in a unit test.

20.1 The bottom line

Using the MVP pattern will make your code more organized and easier to test. In the sample code, you'll notice a test folder. The test needs to instantiate the presenter and mock the interfaces. Because you're not using any Android-specific code in the presenter, you don't need to run in an Android-powered device and instead can run it in the JVM. In this case, you've used Mockito to mock the interfaces.

Because you've been working with Android, you'll notice that a lot of code ends up in the `Activity`. Unfortunately, testing activities is painful. Using the MVP pattern will help you create tests and apply TDD (test-driven development) in an easy way.

20.2 External links

http://en.wikipedia.org/wiki/Model_View_Presenter

Hack 21 *BroadcastReceiver following Activity's lifecycle*

Android v1.6+

Android uses different kinds of messages to notify applications when something happens. For example, if you want to know whether or not a device has connected to the internet, you have to listen to an `Intent` whose action is `android.net.conn.CONNECTIVITY_CHANGE`. This `Intent` can be heard using a `BroadcastReceiver`.

Although using a `BroadcastReceiver` to listen to different notifications from the OS works well, you can't access an `Activity` from the receiver.

Imagine trying to update the UI depending on the connectivity status. How would you do it? What would you do if you wanted to get the receiver's information inside one of your activities? In this hack, I'll show you how to use a `BroadcastReceiver` as an `Activity`'s inner class to get broadcast `Intents`.

Setting up a `BroadcastReceiver` as an `Activity`'s inner class lets us do two important things:

- Call the `Activity`'s methods from inside the receiver
- Enable and disable the receiver depending on the `Activity`'s status

For this hack, we'll create a `Service` that, when activated, waits for 5 seconds and then broadcasts a message. For this toy application, the message we'll send is a string with a date. The implementation of the service isn't that important, but you should know that it'll broadcast an `Intent` with an action—`com.manning.androidhacks.hack021.SERVICE_MSG`—and the date travels as an extra.

Because we want to use the date information the `service` sends in order to update the UI, we'll want to listen to this message only when the `Activity`'s screen is shown. Let's see how to achieve that using the following code:

```
public class MainActivity extends Activity {
  private ProgressDialog mProgressDialog;
  private TextView mTextView;
  private BroadcastReceiver mReceiver;
  private IntentFilter mIntentFilter;

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

    mReceiver = new MyServiceReceiver();
    mIntentFilter = new IntentFilter(MyService.ACTION);

    startService(new Intent(this, MyService.class));
  }

  @Override
  protected void onResume() {
    super.onResume();
    registerReceiver(mReceiver, mIntentFilter);
   }

  @Override
  public void onPause() {
    super.onPause();
    unregisterReceiver(mReceiver);
   }

  private void update(String msg) {
    /* Do something with the msg */
  }

  class MyServiceReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
      update(intent.getExtras().getString(MyService.MSG_KEY));
    }
  }
}
```

❶ Creates new instance of BroadcastReceiver

❷ Creates and defines which type of Intent the receiver gets

❸ Registers receiver in onResume() method

❹ Unregisters receiver inside onPause() method

❺ Invokes Activity's update() method

We'll create a new instance of the `BroadcastReceiver` ❶ and create an `IntentFilter` ❷ that we'll use to define which type of `Intent` the receiver should get. Because the receiver is only used inside the `Activity`, we'll need to register it in the `onResume()` method ❸ and unregister it inside the `onPause()` method ❹. When the receiver is called ❺, it'll invoke the `Activity`'s `update()` method with the `Intent`'s extra information as a parameter.

That's it—we now have a receiver that only updates the UI when the `Activity` is shown.

21.1 The bottom line

The whole Android ecosystem uses `Intent`s to communicate. You'll need to use them sooner or later. By placing a receiver as an inner class in your `Activity`, you can give visual feedback using the information inside an `Intent`. Unregistering the receiver is a good way to avoid unnecessary calls to modify the UI when it's not needed.

21.2 External links

http://developer.android.com/reference/android/content/Intent.html
http://developer.android.com/reference/android/content/BroadcastReceiver.html

Hack 22 *Architecture pattern using Android libraries*

Android v1.6+

Before Android library projects were released, sharing code between Android projects was hard or even impossible. You could use a JAR to share Java code, but you couldn't share code that needed resources. Sharing an `Activity` or a custom view was impossible because you can't add resources to JARs and use them later in an Android application. Android library projects were created as a way to share Android code. In this hack, we'll look at a way to use them.

As an example, we'll create a small application with a login screen. The application is divided into three layers:

- Back-end logic and model (JAR file)
- Android library
- Android application

22.1 Back-end logic and model

This layer is a simple JAR file that can hold logic and doesn't involve or use Android-specific code. It's here that we place the server calls and business objects and logic. In our example, we'll have a project that creates a JAR file to handle login-specific functionality.

As you can see in figure 22.1, `Login` doesn't need to have Android as a dependency. The output of this project will be a JAR file to be included in our Android application. Having the business logic in a Java project means we can test everything with JUnit without setting up an Android test, which is painful. Also, separating code allows developers with different skills to work on the appropriate layer.

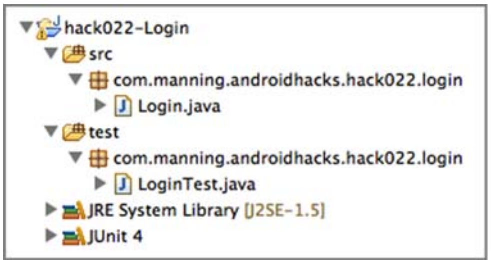
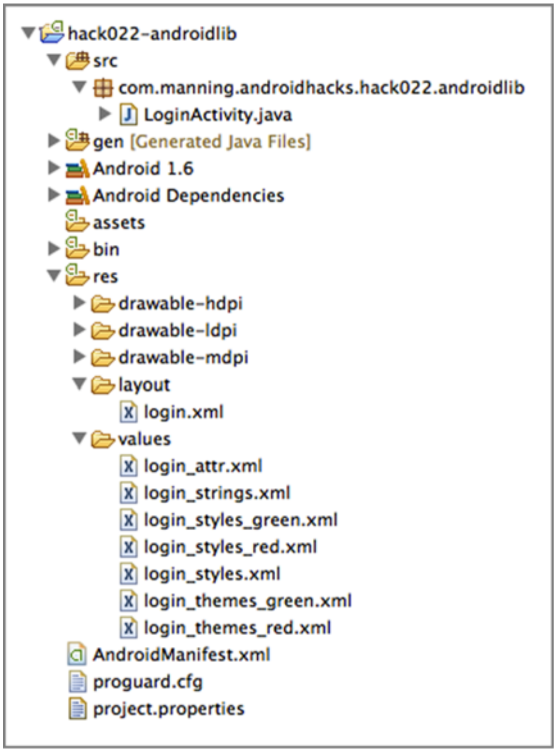

hack022-Login
src
com.manning.androidhacks.hack022.login
Login.java
test
com.manning.androidhacks.hack022.login
LoginTest.java
JRE System Library [J2SE-1.5]
JUnit 4

Figure 22.1 Login project loaded in Eclipse



```
▼ hack022-androidlib
  ▼ src
    ▼ com.manning.androidhacks.hack022.androidlib
      ▶ LoginActivity.java
  ▶ gen [Generated Java Files]
  ▶ Android 1.6
  ▶ Android Dependencies
    assets
  ▶ bin
  ▼ res
    ▶ drawable-hdpi
    ▶ drawable-ldpi
    ▶ drawable-mdpi
    ▼ layout
        login.xml
    ▼ values
        login_attr.xml
        login_strings.xml
        login_styles_green.xml
        login_styles_red.xml
        login_styles.xml
        login_themes_green.xml
        login_themes_red.xml
    AndroidManifest.xml
    proguard.cfg
    project.properties
```

Figure 22.2 The Android library loaded in Eclipse

22.2 Android library

As I mentioned earlier, an Android library is like a JAR file but with the possibility of using Android resources. When we add an Android library as a dependency of our application, we get a second `R` class with the library's IDs and we'll be able to use the library's resources from our code. This layer will have Android-specific activities, a custom view, or services that Android applications will be able to reuse.

In figure 22.2, you can see the Android library `androidlib`. Here you can see Android as a dependency, which means that you can use every Android class and resource. Every Android library will have its own `R` class.

Note that this library can use the JAR mentioned earlier as a dependency. In this example, we placed the JAR as a dependency for the Android library. This way, we have a modular and maintainable library to use in any Android project.

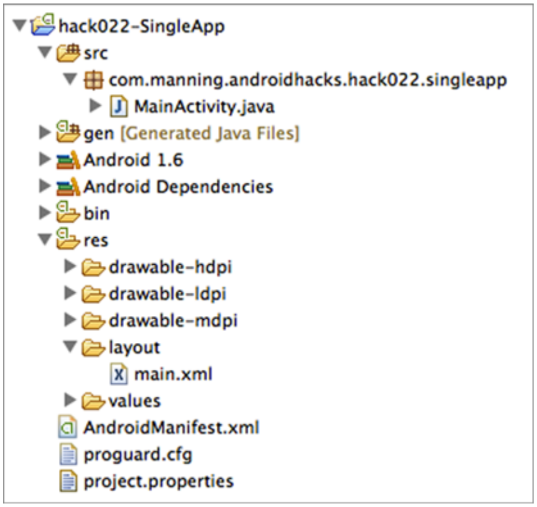
22.3 Android application

The resulting Android application depends on the back-end JAR to handle business logic and the Android library to handle Android-related stuff. You can see in figure 22.3 how the Android library is included in the project.

In this layer, we'll be able to use code from the JAR and from the Android library. We now can start developing our application, taking care of the distribution of code between layers.

22.4 The bottom line

This was a short introduction to a possible architecture design using Android libraries. Reusable code and maintainability is hard to achieve, but now that you have Android libraries, it's possible.



```
▼ hack022-SingleApp
  ▼ src
    ▼ com.manning.androidhacks.hack022.singleapp
      ▶ MainActivity.java
  ▶ gen [Generated Java Files]
  ▶ Android 1.6
  ▶ Android Dependencies
  ▶ bin
  ▼ res
    ▶ drawable-hdpi
    ▶ drawable-ldpi
    ▶ drawable-mdpi
    ▼ layout
        main.xml
    ▶ values
    AndroidManifest.xml
    proguard.cfg
    project.properties
```

Figure 22.3 Android application folder structure

22.5 External links

http://developer.android.com/tools/projects/index.html#LibraryProjects

http://developer.android.com/tools/projects/projects-eclipse.html#SettingUpLibraryProject

Hack 23 The SyncAdapter pattern
Android v2.2+

Almost every Android application uses the internet to fetch information or to sync data. If you've already created a couple of applications, you'll be able to describe many different ways to create a connection and show a progress animation while fetching results.

23.1 Common approaches

I've been working as a contractor for different companies, and in my experiences I've seen developers handle data fetching in a variety of ways. Most of the code I've seen falls into one of the approaches that I'll cover next.

23.1.1 Using the AsyncTask class

`AsyncTask` is an Android class that handles threads for you, making it easy to move logic to another thread. If you've used it in previous projects, the following story might ring a bell.

Some time ago, you started developing for Android. You learned that you shouldn't place background logic in the main thread. You searched the web for an explanation of how to do it and you found a nice Android developer's article entitled "Painless Threading." Near the end of the article (see section 23.4), it states this:

> Always remember these two rules about the single thread model. Do not block the UI thread, and make sure that you access the Android UI toolkit only on the UI thread.

AsyncTask just makes it easier to do both of these things.

So you learned how to use the AsyncTask class and you started using it everywhere. No matter how complex your UI was, or how long it took to parse those big chunks of data, the AsyncTask was always there for you. You left work early pointing and laughing at the iOS developers from your company, saying "Android is easier than iOS; I finished earlier than you. Enjoy your night coding, Apple fan boys!"

Unfortunately, this didn't last long. You noticed that if you rotated the device while an AsyncTask was running, your application crashed. It was hard to fix, but an ugly hack did the trick. Later you noticed that your application also crashed after some time due to a limitation in the amount of concurrent tasks the AsyncTask supported. When you tried to fix this second issue, you noticed that your Activity's code was polluted with a lot of inner classes extending AsyncTask. After a long day, you started questioning where you went wrong.

If you're planning to use an AsyncTask, think it over. The only reason to use it is when the background task is simple or you don't depend on the result. Let's look at another approach.

23.1.2 Using a Service

The second approach is to use a Service. Using a Service solves a lot of issues but comes with some difficulties. Following is a list of concerns that always caused me to wonder whether or not I was making the correct choice:

- Communicating with an Activity
- Deciding when and how to start the Service
- Detecting connectivity status while working
- Persisting data

The issue with this approach is the system's flexibility. For example, you have many ways to communicate with an Activity. Should the Activity bind to the Service? Should it use a Handler? Should it communicate via Intents? Should it communicate through a database? Many possibilities exist and the answer to the question of which you should use is always "it depends."

The question I started asking myself was, how does the Gmail application work? How does it sync and work offline without an issue? Google uses something called SyncAdapter. Unfortunately, this is one of Android's best but least documented features. If you ask Android developers if they know what it is, they'll say yes, but they've never used it.

In this hack, we'll see how to use a `SyncAdapter` to organize an internet-dependent application, making our development life easier.

23.2 What we'll create

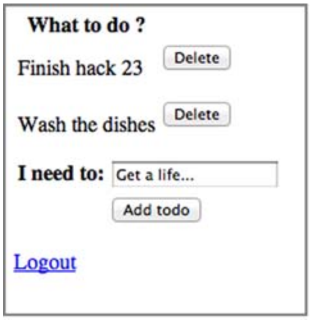
For this example, we'll create a TODO list. We'll use a server that will have a front end to add items from the browser. You can see how it looks in figure 23.1. The server will also have an API so we can have the same functionality in an Android device. The running Android application can be seen in figure 23.2.

23.2.1 What's a SyncAdapter?

A `SyncAdapter` is an Android `Service` that's started by the platform. There we'll place all of our sync logic. Before you get lost, go watch Virgil Dobjanschi's Google I/O 2010 Android REST (see section 23.4)client application presentation. This is without a doubt the best Google I/O presentation ever and the only good documentation on `SyncAdapters`.

The benefits of using `SyncAdapters` include

- Automatically syncs in the background (even when our application isn't open)
- Handles authentication against the server
- Handles retries
- Respects user's preferences regarding background syncs



What to do ?

Finish hack 23 Delete

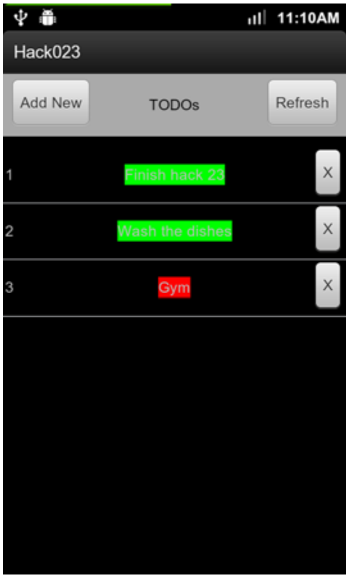
Wash the dishes Delete

I need to: Get a life...

Add todo

Logout

Figure 23.1 **Server's front end**



11:10AM

Hack023

Add New TODOs Refresh

1 Finish hack 23 X

2 Wash the dishes X

3 Gym X

Figure 23.2 **Android application's front end**

23.2.2 Hitting a database instead of the server

The first thing to do is to forget about syncing. We'll create the application to only work locally and save information inside a database. To do this, we'll need a DatabaseHelper, a TodoContentProvider, and a TodoDAO. Let's first understand the DatabaseHelper:

```
public class DatabaseHelper extends SQLiteOpenHelper {
  public static final String DATABASE_NAME = "todo.db";
  private static final int DATABASE_VERSION = 1;

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }

  @Override
  public void onCreate(SQLiteDatabase db) {
     db.execSQL("CREATE TABLE "
        + TodoContentProvider.TODO_TABLE_NAME + " ("
        + TodoContentProvider.COLUMN_ID
        + " INTEGER PRIMARY KEY AUTOINCREMENT,"
        + TodoContentProvider.COLUMN_SERVER_ID + " INTEGER,"
        + TodoContentProvider.COLUMN_TITLE + " LONGTEXT,"
        + TodoContentProvider.COLUMN_STATUS_FLAG + " INTEGER"
        + ");");
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
        int newVersion) {
     db.execSQL("DROP TABLE IF EXISTS " +
        TodoContentProvider.TODO_TABLE_NAME);
    onCreate(db);
  }
}
```

❶ Extends SQLiteOpenHelper

❷ Specifies database name and version

❸ Decides if tables need to be created

❹ Upgrades from an old schema

The DatabaseHelper extends SQLiteOpenHelper ❶. When the class is created, we specify the database name and its version ❷. The SQLiteOpenHelper will use that to decide whether some tables need to be created ❸ or upgraded from an old schema ❹. Don't worry about the schema for now. You'll understand all its rows in short order.

Now that we have the DatabaseHelper in place, we'll need to set up our ContentProvider. Note that if you've never used a ContentProvider, you should try doing a fast web search before you continue reading. The TodoContentProvider class for this hack has nothing out of the ordinary. Let's look at how the query method is created:

```
public class TodoContentProvider extends ContentProvider {
  public static final String TODO_TABLE_NAME = "todos";
  public static final String AUTHORITY = TodoContentProvider.class
      .getCanonicalName();

  public static final String COLUMN_ID = "_id";
  public static final String COLUMN_SERVER_ID = "server_id";
  public static final String COLUMN_TITLE = "title";
  public static final String COLUMN_STATUS_FLAG = "status_flag";

  private static final int TODO = 1;
```

❶ Extends ContentProvider

```
private static final int TODO_ID = 2;

private static HashMap<String, String> projectionMap;
private static final UriMatcher sUriMatcher;

public static final String CONTENT_TYPE =
  "vnd.android.cursor.dir/vnd.androidhacks.todo";
public static final String CONTENT_TYPE_ID =
  "vnd.android.cursor.item/vnd.androidhacks.todo";

public static final Uri CONTENT_URI = Uri.parse("content://"
    + AUTHORITY + "/" + TODO_TABLE_NAME);

private DatabaseHelper dbHelper;

static {
  sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
  sUriMatcher.addURI(AUTHORITY, TODO_TABLE_NAME, TODO);
  sUriMatcher.addURI(AUTHORITY, TODO_TABLE_NAME + "/#", TODO_ID);

  projectionMap = new HashMap<String, String>();
  projectionMap.put(COLUMN_ID, COLUMN_ID);
  projectionMap.put(COLUMN_SERVER_ID, COLUMN_SERVER_ID);
  projectionMap.put(COLUMN_TITLE, COLUMN_TITLE);
  projectionMap.put(COLUMN_STATUS_FLAG, COLUMN_STATUS_FLAG);
}

@Override
public boolean onCreate() {
  dbHelper = new DatabaseHelper(getContext());
  return true;
}

@Override
public Cursor query(Uri uri, String[] projection, String selection,
    String[] selectionArgs, String sortOrder) {

  SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
  switch (sUriMatcher.match(uri)) {
  case TODO:
    qb.setTables(TODO_TABLE_NAME);
    qb.setProjectionMap(projectionMap);
    break;
  case TODO_ID:
    qb.setTables(TODO_TABLE_NAME);
    qb.setProjectionMap(projectionMap);
    qb.appendWhere(COLUMN_ID + "=" + uri.getPathSegments().get(1));
    break;
  default:
    throw new RuntimeException("Unknown URI");
  }

  SQLiteDatabase db = dbHelper.getReadableDatabase();
  Cursor c = qb.query(db, projection, selection,
    selectionArgs, null, null, sortOrder);

  c.setNotificationUri(getContext().getContentResolver(),
     uri);
```

2 Decides which action to take for an incoming content URI

3 Changes match

4 Creates ContentProvider

5 Switches over a URI and sets query builder

6 Gets a Cursor from the database

7 Sets notification URI; Cursor watches for URI content changes

```
    return c;
  }
...
}
```

The `TodoContentProvider` extends `ContentProvider` ❶. Inside it we define a `UriMatcher` that will help us decide which action to take for an incoming content URI ❷. In this case, the content values to use with the `ContentProvider` have a one-to-one match with the database columns. If we want to change that, we can use a projection map ❸. When the `ContentProvider` is created ❹, we get an instance of the `DatabaseHelper`, which will be useful for querying the database. For the sake of brevity I only show the `query()` method. The rest of the `ContentProvider` methods look alike. Inside the `query()` method, we can see how to switch over a URI and set the query builder correctly ❺. After that we use the query builder to get a `Cursor` from the database that will be returned to the user ❻. Pay attention to the last line ❼. Before returning the `Cursor`, we set the notification URI. This will make the `Cursor` watch for URI content changes. This means that every time something gets modified, the `Cursor` will update automagically.

Finally, the `TodoDAO` will be in charge of calling the `ContentProvider` through a `ContentResolver`. This is the layer where conversions from Java objects to database values and from database values to Java objects occur, as follows:

```
public class TodoDAO {
  private static final TodoDAO instance = new TodoDAO();

  private TodoDAO() {}

  public static TodoDAO getInstance() {                  ❶ Implements singleton
     return instance;
  }

  public void addNewTodo(ContentResolver contentResolver,    ❷ Places calls
    Todo list, int flag) {
    ContentValues contentValue = getTodoContentValues(list, flag);
    contentResolver.insert(TodoContentProvider.CONTENT_URI,
        contentValue);
  }

  private ContentValues getTodoContentValues(Todo todo,     ❸ Converts to content values
    int flag) {
    ContentValues cv = new ContentValues();
    cv.put(TodoContentProvider.COLUMN_SERVER_ID, todo.getId());
    cv.put(TodoContentProvider.COLUMN_TITLE, todo.getTitle());
    cv.put(TodoContentProvider.COLUMN_STATUS_FLAG, flag);

    return cv;
  }
...
}
```

As you can see, the TodoDAO is implemented with a singleton ❶. There, we placed calls such as addNewTodo() ❷ which, after a proper conversion to content values ❸, will end in a database insert.

23.2.3 Populating the database

In this section, you'll see how to deal with the database from the application. We'll use two activities:

- MainActivity—Will show the list of TODOs
- AddNewActivity—Will present a form to add a new TODO

Both activities function in a similar way. When they need to modify some data, they'll do it through the TodoDAO. Let's take a look at the code for the MainActivity:

```
public class MainActivity extends Activity {

  private ListView mListView;
  private TodoAdapter mAdapter;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    mListView = (ListView) findViewById(R.id.main_activity_listview);

    mAdapter = new TodoAdapter(this);                      Creates
    mListView.setAdapter(mAdapter);                      ❶ ListView
  }
                                                                ❷ Starts
  public void addNew(View v) {                                    AddNewActivity
    startActivity(new Intent(this, AddNewActivity.class));        activity
   }
```

Nothing out of the ordinary here. We created a ListView that will use a TodoAdapter ❶, and every time the user clicks on the Add New button, we'll start the AddNewActivity activity ❷.

The TodoAdapter holds more interesting code. Let's see how it's done:

```
public class TodoAdapter extends CursorAdapter {

...

  private static final String[] PROJECTION_IDS_TITLE_AND_STATUS =
   new String[] {
      TodoContentProvider.COLUMN_ID,
      TodoContentProvider.COLUMN_TITLE,
      TodoContentProvider.COLUMN_STATUS_FLAG };

  public TodoAdapter(Activity activity) {
    super(activity, getManagedCursor(activity), true);
    mActivity = activity;
    ...
  }                                                                  ❶ Gets a
                                                                       Cursor
  private static Cursor getManagedCursor(Activity activity) {
     return activity.managedQuery(TodoContentProvider.CONTENT_URI,
        PROJECTION_IDS_TITLE_AND_STATUS,
        TodoContentProvider.COLUMN_STATUS_FLAG + " != "
             + StatusFlag.DELETE, null,
        TodoContentProvider.DEFAULT_SORT_ORDER);
  }

  @Override
  public void bindView(View view, Context context, Cursor c) {
    final ViewHolder holder = (ViewHolder) view.getTag();
    holder.id.setText(c.getString(mInternalIdIndex));
    holder.title.setText(c.getString(mTitleIndex));

    final int status = c.getInt(mInternalStatusIndex);
    if (StatusFlag.CLEAN != status) {
       holder.title.setBackgroundColor(Color.RED);
    } else {
      holder.title.setBackgroundColor(Color.GREEN);
    }

    final Long id = Long.valueOf(holder.id.getText().toString());
    holder.deleteButton.setOnClickListener(new OnClickListener() {

      @Override
      public void onClick(View v) {
        TodoDAO.getInstance().deleteTodo(
             mActivity.getContentResolver(), id);
      }
    });
  }

  ...
}
```

❷ Checks use of TodoContentProvider's URI and a projection

❸ Changes background of text

❹ Removes TODO from the list

When the `TodoAdapter` is created, we get a `Cursor` ❶ using `Activity`'s `managedQuery()` method. Check how we used the `TodoContentProvider`'s URI and a projection ❷. Finally, we have the `bindView()` method. With it we change the background of the text depending on the status flag (I'll discuss that later) ❸ and set a click listener for the Delete button. Inside the listener, we use the `TodoDAO` to remove the TODO from the list ❹.

Where's the `notifyDataSetChanged()`? There's no need for it. Do you remember the `setNotificationUri()` call we used inside the `TodoContentProvider`? The `Cursor` returned by the `TodoContentProvider` will get updated when changes are made to the database through the `ContentProvider`.

Up to this point, we have a working application that saves data to a database. Now we need to take the authentication step and sync with the server.

23.2.4 Adding login functionality

Before adding the `SyncAdapter` to our code, let's first see how to deal with the authentication with the server. Instead of saving the login details inside a database or a shared preference, we'll save them in an Android `Account`. To handle accounts, we'll use an Android class called `AccountManager`. The `AccountManager` is in charge of managing user credentials inside `Accounts`. The basic idea is that users enter their credentials once, and they're saved inside an Account. All of the applications that have the USE_CREDENTIALS permission can query the manager to obtain an account where an authentication token or whatever is necessary to authenticate against a server is saved.

Before coding this part, you need to understand that the login functionality will be used in these situations:

- When the application starts and no account has been created
- When the user goes to Accounts & Sync and clicks on New Account
- When the SyncAdapter tries to sync and the authentication fails

Let's look at the first two situations in this section and the last one after we have the SyncAdapter working. For the first one, we'll create a BootstrapActivity:

```
public class BootstrapActivity extends Activity {
  private static final int NEW_ACCOUNT = 0;
  private static final int EXISTING_ACCOUNT = 1;
  private AccountManager mAccountManager;

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

    mAccountManager = AccountManager.get(this);
    Account[] accounts = mAccountManager
         .getAccountsByType(AuthenticatorActivity.PARAM_ACCOUNT_TYPE);

    if (accounts.length == 0) {
      final Intent i = new Intent(this, AuthenticatorActivity.class);
      i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET);
      startActivityForResult(i, NEW_ACCOUNT);
    } else {
      String password = mAccountManager.getPassword(accounts[0]);
      if (password == null) {
        final Intent i = new Intent(this, AuthenticatorActivity.class);
        i.putExtra(AuthenticatorActivity.PARAM_USER, accounts[0].name);
        startActivityForResult(i, EXISTING_ACCOUNT);
      } else {
        startActivity(new Intent(this, MainActivity.class));
        finish();
      }
    }
  }

...

}
```

❶ Gets list of accounts of our type

❷ Creates a new account

❸ Asks user for password

❹ Continues to MainActivity

Inside the onCreate() method, we get a list of accounts of our type ❶. If we have no account, we launch the AuthenticatorActivity to help create a new account ❷. If the account exists but the AccountManager doesn't have a password for it, we need to ask the user for the password ❸. This can happen when the password gets invalidated. The last case is when everything is in place, so we can continue to the MainActivity ❹.

The second situation is more complicated but will leave everything in place for the last situation. To create a new account through the Accounts & Sync settings, we'll need to extend AbstractAccountAuthenticator.

The AbstractAccountAuthenticator is a base class for creating account authenticators. In order to provide an authenticator, we must extend this class, provide implementations for the abstract methods, and write a service that returns the result of getIBinder() in the service's onBind(android.content.Intent) method when invoked with an Intent with action AccountManager.ACTION_AUTHENTICATOR_INTENT.

We'll extend the AbstractAccountAuthenticator with a class called Authenticator. It's OK to return null values from the methods we're not going to use. The important ones are addAcount() and getAuthToken(). The code follows:

```
public class Authenticator extends AbstractAccountAuthenticator {
  private final Context mContext;

  public Authenticator(Context context) {
    super(context);
    mContext = context;
  }

  @Override
  public Bundle addAccount(AccountAuthenticatorResponse response,
      String accountType, String authTokenType,
      String[] requiredFeatures, Bundle options)
      throws NetworkErrorException {

    final Intent intent = new Intent(mContext,
         AuthenticatorActivity.class);
    intent.putExtra(AuthenticatorActivity.PARAM_AUTHTOKEN_TYPE,
        authTokenType);
    intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,
        response);
    final Bundle bundle = new Bundle();
    bundle.putParcelable(AccountManager.KEY_INTENT, intent);

    return bundle;
  }

..

  @Override
  public Bundle getAuthToken(AccountAuthenticatorResponse response,
      Account account, String authTokenType, Bundle options)
      throws NetworkErrorException {

    if (!authTokenType
        .equals(AuthenticatorActivity.PARAM_AUTHTOKEN_TYPE)) {

      final Bundle result = new Bundle();
      result.putString(AccountManager.KEY_ERROR_MESSAGE,
          "invalid authTokenType");

      return result;
    }

    final AccountManager am = AccountManager.get(mContext);
    final String password = am.getPassword(account);
```

1 Checks if required token is the same

```
  if (password != null) {
    boolean verified = false;
    String loginResponse = null;

    try {
      loginResponse = LoginServiceImpl.sendCredentials(
        account.name, password);
      verified = LoginServiceImpl.hasLoggedIn(loginResponse);
    } catch (AndroidHacksException e) {
      verified = false;
    }

    if (verified) {
      final Bundle result = new Bundle();
      result.putString(AccountManager.KEY_ACCOUNT_NAME, account.name);
      result.putString(AccountManager.KEY_ACCOUNT_TYPE,
          AuthenticatorActivity.PARAM_ACCOUNT_TYPE);

      return result;
    }
  }

  final Intent intent = new Intent(mContext,
       AuthenticatorActivity.class);
  intent.putExtra(AuthenticatorActivity.PARAM_USER, account.name);
  intent.putExtra(AuthenticatorActivity.PARAM_AUTHTOKEN_TYPE,
      authTokenType);
  intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,
      response);
  final Bundle bundle = new Bundle();
  bundle.putParcelable(AccountManager.KEY_INTENT, intent);
  return bundle;
}
```

❷ Gets a password

❸ Returns the result

❹ Lets caller know which activity to call for user to sign in

The addAccount() method is straightforward. There we prepare the Intent that the AccountManager will use to create a new account. Let's now investigate the getAuthToken() method. This method will be called when we need to log in to the server using the credentials inside the Account. We'll first check if the required token is the same as the one we handle ❶. Afterward, we use the AccountManager to get a password. If there's a password stored ❷, we sign in against the server, and if it's OK ❸, we return the result. If we can't sign in, we'll return an Intent to let the caller know which activity to call to let the user sign in ❹. This happens when the password changes or the credentials were revoked.

The next class to create is AuthenticatorActivity. This activity will be used to show the login form. You can see how it looks in figure 23.3.

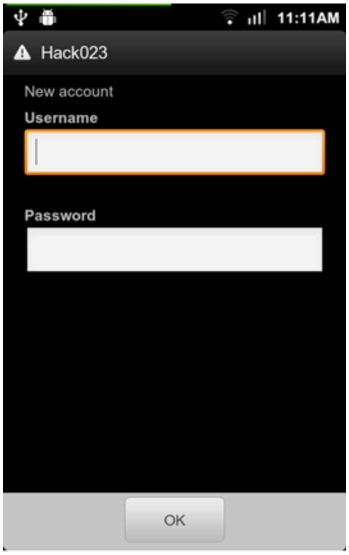


Figure 23.3 Login form from AuthenticatorActivity

The code is the following:

```
public class AuthenticatorActivity extends
    AccountAuthenticatorActivity {
  public static final String PARAM_ACCOUNT_TYPE =
    "com.manning.androidhacks.hack023";
  public static final String PARAM_AUTHTOKEN_TYPE = "authtokenType";
  public static final String PARAM_USER = "user";
  public static final String PARAM_CONFIRMCREDENTIALS =
    "confirmCredentials";
  private AccountManager mAccountManager;
  private Thread mAuthThread;
  private String mAuthToken;
  private String mAuthTokenType;
  private Boolean mConfirmCredentials = false;
  private final Handler mHandler = new Handler();
  protected boolean mRequestNewAccount = false;
  private String mUser;

...

  private void handleLogin(View view) {
    if (mRequestNewAccount) {
      mUsername = mUsernameEdit.getText().toString();
    }
    mPassword = mPasswordEdit.getText().toString();

    if (TextUtils.isEmpty(mUsername) || TextUtils.isEmpty(mPassword)) {
      mMessage.setText(getMessage());
    }

    showProgress();
    mAuthThread = NetworkUtilities.attemptAuth(mUsername,
       mPassword, mHandler, AuthenticatorActivity.this);
  }

  public void onAuthenticationResult(Boolean result) {
     hideProgress();

    if (result) {
      if (!mConfirmCredentials) {
        finishLogin();
      }
    } else {
      mMessage.setText("User and/or password are incorrect");
    }
  }

  private void finishLogin() {
     final Account account = new Account(mUsername, PARAM_ACCOUNT_TYPE);

    if (mRequestNewAccount) {
    mAccountManager.addAccountExplicitly(account, mPassword, null);
    } else {
      mAccountManager.setPassword(account, mPassword);
    }

    final Intent intent = new Intent();
    intent.putExtra(AccountManager.KEY_ACCOUNT_NAME, mUsername);
```

1 Launches thread that will hit server

2 Returns result to AuthenticatiorActivity

3 Calls finishLogin()

4 Sets a new password

```
    intent.putExtra(AccountManager.KEY_ACCOUNT_TYPE,
      PARAM_ACCOUNT_TYPE);

    if (mAuthTokenType != null
        && mAuthTokenType.equals(PARAM_AUTHTOKEN_TYPE)) {
      intent.putExtra(AccountManager.KEY_AUTHTOKEN, mAuthToken);
    }

    setAccountAuthenticatorResult(intent.getExtras());
    setResult(RESULT_OK, intent);
    finish();
  }

...

}
```

❺ **Sets the result**

When the user enters the login details and clicks OK, `handleLogin()` gets executed. There we launch a thread that will hit the server ❶ and return the result to the `AuthenticatorActivity` in the `onAuthenticationResult()` method ❷. If the service can authenticate correctly, we'll call `finishLogin()` ❸, and if not we'll show an error and let the user try again. Inside `finishLogin()`, if the Request New Account flag is set, we use the `AccountManager` to create an account. If the account exists, we'll set a new password ❹. Finally, we set the result that's to be sent as the result of the request that caused this activity to be launched ❺.

The last step is modifying the AndroidManifest.xml to register the `Service`. We do that by adding the following:

```
<service android:name=".authenticator.AuthenticationService"
  android:exported="true">

  <intent-filter>
     <action android:name="android.accounts.AccountAuthenticator" />
  </intent-filter>

  <meta-data android:name="android.accounts.AccountAuthenticator"
    android:resource="@xml/authenticator" />
 </service>
```

❶ **Returns an Authenticator**

❷ **Additional information**

The `android.accounts.AccountAuthenticator Intent` filter will make the system notice that this particular `Service` returns an `Authenticator` ❶. We'll also need to give additional information using a separate XML file ❷. In this example, the `authenticator` XML contains the following:

```
<account-authenticator
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="com.manning.androidhacks.hack023"
    android:icon="@drawable/ic_launcher"
    android:smallIcon="@drawable/ic_launcher"
    android:label="@string/app_name"/>
```

The most important piece of information is the `android:accountType`. That means that the `Service` will return an `Authenticator` to authenticate only accounts of type com.manning.androidhacks.hack023. The rest of the information we can place there determines how the Accounts & Sync row will look.

23.2.5 Adding the SyncAdapter

The last step is to add a SyncAdapter. After so many pages, we still don't know what it's for, so let's try to understand how the SyncAdapter will add a happy ending to everything we wrote so far.

The SyncAdapter is a Service handled by Android that will use an Account to authenticate to the server and a ContentProvider to sync data. When we finish coding it, the application will sync with the server without us telling it anything. The OS will register it with every other SyncAdapter inside the device. The SyncAdapters run one at a time to avoid making our internet connection choke. Isn't it the best Android feature you've used so far? Let's learn how to code it.

We first need to declare it in the AndroidManifest.xml:

```
<service android:name=".service.TodoSyncService"
    android:exported="true">
    <intent-filter>
        <action android:name="android.content.SyncAdapter" />
    </intent-filter>
    <meta-data android:name="android.content.SyncAdapter"
        android:resource="@xml/todo_sync_adapter" />
 </service>
```

❶ Defines the android.content .SyncAdapter

❷ Additional XML

Similar to the AuthenticationService, we define the android.content.SyncAdapter action to let Android know that TodoSyncService is a SyncAdapter ❶. It also has some additional XML ❷ with the following information:

```
<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"
    android:contentAuthority=
    "com.manning.androidhacks.hack023.provider.TodoContentProvider"
    android:accountType=
    "com.manning.androidhacks.hack023" />
```

This means that the TodoSyncService will use the TodoContentProvider's authority and will need a com.manning.androidhacks.hack023 account type.

The next step is to extend AbstractThreadedSyncAdapter. Following is the code:

```
public class TodoSyncAdapter extends AbstractThreadedSyncAdapter {
  private final ContentResolver mContentResolver;
  private AccountManager mAccountManager;
  private final static TodoDAO mTodoDAO = TodoDAO.getInstance();

  @Override
  public void onPerformSync(Account account, Bundle extras,
      String authority, ContentProviderClient provider,
      SyncResult syncResult) {

    try {
      List<Todo> data = fetchData();
      syncRemoteDeleted(data);
```

❶ Gets every TODO from the server

❷ Removes the TODOs from the local database

Calls syncFromServerToLocalStorage ❸

```
     syncFromServerToLocalStorage(data);
     syncDirtyToServer(
        mTodoDAO.getDirtyList(mContentResolver));
```

❹ Gets every TODO from database; either push a new TODO to server and update or delete

```
    } catch (Exception e) {
      handleException(e, syncResult);
    }

  }

...

  private void handleException(Exception e,
    SyncResult syncResult) {
```

❺ How exceptions are handled

```
    if (e instanceof AuthenticatorException) {
      syncResult.stats.numParseExceptions++;
    } else if (e instanceof IOException) {
      syncResult.stats.numIoExceptions++;
...

    }
  }
```

When the onPerformSync() method gets called, we're already in a background thread. Here's where we add the logic to sync with the server. In the next few lines, I'll explain a sync approach that works for me; it doesn't mean you're obliged to do it this way.

Do you remember what a row in the TODO table looked like? The TODO table has the following columns:

- *_id*—Local ID.
- *server_id*—After syncing, every row will get the server's ID.
- *status_flag*—The status can be CLEAN, MOD, ADD, DELETE.
- *title*—The text of the TODO.

When the sync starts, we first get every TODO from the server ❶. Note that if we have lots of TODOs, we might need to use some sort of pagination. The next step is removing from the local database TODOs that are no longer in the server ❷. We do this by getting a list of TODOs from our local database with the CLEAN flag set, and checking whether a TODO is in the server's list. If it's not there, we can delete it from our local database. After that, syncFromServerToLocalStorage is called ❸. There we'll iterate over the server's TODOs. We can use the server_id to check whether it exists locally. If it exists, we update it with the information from the server. If not, we create a new one. The last step is syncDirtyToServer() ❹. In this case, we get every TODO from the local database that's dirty (not *clean*). There, depending on the status flag, we push a new TODO to the server and update or delete.

Note how the exceptions are handled ❺. Depending on the exception, we modify the syncResult object. We do this to help the SyncManager decide when to call the SyncAdapter again.

The final step is to wrap the SyncAdapter inside the TodoSyncService, which we can do using the following code:

```
public class TodoSyncService extends Service {
    private static final Object sSyncAdapterLock = new Object();
    private static TodoSyncAdapter sSyncAdapter = null;

    @Override
    public void onCreate() {
        synchronized (sSyncAdapterLock) {
            if (sSyncAdapter == null) {
                sSyncAdapter = new TodoSyncAdapter(
                  getApplicationContext(), true);
            }
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return sSyncAdapter.getSyncAdapterBinder();
    }

}
```

23.3 The bottom line

You might be thinking that using a SyncAdapter is a lot of work, but note how after creating the model and the ContentProvider, everything got easier. Users can use the application offline or online; they won't notice the difference.

Note that I didn't explain anything about the server. For this example, I've coded a small Python server using web.py. If you're giving SyncAdapters a try, I recommend you use something like StackMob. You'll avoid wasting time coding the back end.

23.4 External links

http://developer.android.com/reference/android/os/AsyncTask.html

http://www.youtube.com/watch?feature=player_embedded&v=xHXn3Kg2IQE

http://android-developers.blogspot.com.ar/2009/05/painless-threading.html

http://logc.at/2011/11/08/the-hidden-pitfalls-of-asynctask/

http://developer.android.com/reference/android/content/AbstractThreadedSyncAdapter.html

http://www.youtube.com/watch?v=xHXn3Kg2IQE&feature=youtu.be

http://developer.android.com/guide/topics/providers/content-provider-creating.html

http://naked-code.blogspot.com/2011/05/revenge-of-syncadapter-synchronizing.html

http://developer.android.com/reference/android/content/AbstractThreadedSyncAdapter.html

https://www.stackmob.com/

ANDROID/MOBILE

50 Android Hacks

Carlos Sessa

Hacks. Clever programming techniques to solve thorny little problems. Ten lines of code that save you two days of work. The little gems you learn from the old guy in the next cube or from the geniuses on Stack Overflow. That's just what you'll find in this compact and infinitely useful book.

The name **50 Android Hacks** says it all. Ranging from the mundane to the spectacular, each self-contained, fully illustrated hack is just a couple pages long and includes annotated source code. These practical techniques are organized into twelve collections covering layout, animations, patterns, and more.

What's Inside

• Hack 3	Creating a custom ViewGroup
• Hack 8	Slideshow using the Ken Burns effect
• Hack 20	The Model-View-Presenter pattern
• Hack 23	The SyncAdapter pattern
• Hack 31	Aspect-oriented programming in Android
• Hack 34	Using Scala inside Android
• Hack 43	Batching database operations
• Plus 43 more!	

Most hacks work with Android 2.x and greater. Version-specific hacks are clearly marked.

Carlos Sessa is a passionate professional Android developer. He's active on Stack Overflow and is an avid hack collector.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/50AndroidHacks

Free eBook
SEE INSERT

"How to solve common problems that arise in Android development."

—From the Foreword by Jake Wharton, Android Engineer

"One of the best how-to books I've read!"

—Christian Badenas
Android and .NET Developer

"Packed with useful Android development tidbits not found in the official documentation."

—Matthias Käppler, SoundCloud

"A great resource for creating nontrivial user experiences for the Android platform."

—Frank Ableson
Coauthor of *Android in Action*

ISBN 13: 978-1-617290-56-5
ISBN 10: 1-617290-56-4
53499
9 781617 290565

MANNING

$34.99 / Can $36.99 [INCLUDING eBOOK]