Android App Development

This is an introduction to Android App development on Linux.

Working on pure 64-bit Slackware

***Updated 20170823**: I have successfully developed an Android App on pure 64-bit Slackware machine. I only need to compile 'mksdcard' (discussed below). So, this sub-section is left here only for historical purposes* 😎

***Note**: Although the SDK can be run on pure 64-bit Linux system some of the tools are actually still 32-bit versions. This is not a problem if the Linux system is multilib (ability to run 32-bit binaries as well as 64-bit) like Windows system. I run this on a 32-bit chroot environment on my pure 64-bit system. It is a hassle... but it works.*

Refer this.

- so far, got studio to run and load old projects after getting mksdcard.c from here
  - look for `emulator/mksdcard/mksdcard.c`
  - install to `android-sdk-linux/tools`

Development Environment

The official development environment is Android Studio (IDE). It bundles the SDK together with Gradle build tool in a single download file. I also started with this when I built my first Android App.

I call the above user-friendly environment. I now go for a more developer-friendly environment. So, I just download the Android SDK and the Gradle build tool. Naturally, a Java Development Kit (JDK) is also required.

- Get Android SDK - look for command-line tool
- Get Gradle build tool - refer here for help
- Get Java Development Kit - I use OpenJDK on Slackware Linux

Android SDK

Assuming the downloaded SDK tarball (android-sdk_rXX.X.X-linux.tgz) is in $HOME/download/, unpack it to an installation folder (e.g. /home/share/tool/):

```
# cd /home/share/tool
# tar xf $HOME/download/android-sdk_rXX.X.X-linux.tgz
# export SDK_PATH=/home/share/tool/android-sdk-linux
```

Notice that an environment variable SDK_PATH is set. Of course, the variable name can be something else, like ANDROID_SDK_PATH. Append this path to your PATH variable,

```
# export PATH=$PATH:$SDK_PATH
```

This will enable execution of Android SDK tools from any path.

Everything can be done from command line, but it also has a GUI interface SDK Manager for the more visual-centric developer. To use that, simply run

```
# android
```

As mentioned here, every installation requires an SDK Tool (get latest?), an SDK Platform-tool (get latest?), an SDK Build-tool (get latest?) and at least one SDK Platform. Clicking your way around should be quite self-explanatory.

To work from command line, this is what should be done first

```
# android --help
```

but, notice that that there are no indications whatsoever on how to execute it in command line mode only. Apparently, to do that we need the switch `--no-ui`. So, to list the available packages for download,

```
# android list sdk --no-ui
```

Note that for some commands, like this, the `--no-ui` option is ON by default, so running

```
# android list sdk
```

would provide the same output. The available packages for download is indexed from 1 (we will/can use this number to select packages for download). Somehow the tool shows a package's latest version only (which is not what I want since I want to avoid preview@RC packages). So, to view ALL available packages, do a

```
# android list sdk --all
```

To install those packages (or keep the SDK updated), run

```
# android update sdk --no-ui
```

However, as mentioned here, this will download ALL the available packages - which is A LOT! So, starting with SDK R12 (in this case, we are way past that! :p), it is possible to filter that selection using (duh!) the `--filter` switch. So, to update the SDK Tools and SDK Platform-tools,

```
# android update sdk --no-ui --filter tool,platform-tool
```

Somehow, this does not include a build-tool. So, from the list, select a build-tool and a platform. So, if a list shows

```
Packages available for installation or update: 151
...
   5- Android SDK Build-tools, revision 24 rc3
   6- Android SDK Build-tools, revision 23.0.3
   7- Android SDK Build-tools, revision 23.0.2
...
  28- Documentation for Android SDK, API 23, revision 1
  29- SDK Platform Android 6.0, API 23, revision 3
  30- SDK Platform Android N Preview, revision 2
  31- SDK Platform Android 5.1.1, API 22, revision 2
  32- SDK Platform Android 5.0.1, API 21, revision 2
  33- SDK Platform Android 4.4W.2, API 20, revision 2
  34- SDK Platform Android 4.4.2, API 19, revision 4
...
  56- ARM EABI v7a System Image, Android API 23, revision 3
...
 103- Google APIs, Android API 23, revision 1
 104- Google APIs, Android API 22, revision 1
 105- Google APIs, Android API 21, revision 1
...
 107- Google APIs, Android API 19, revision 18
```

and the needed packages are build-tool, platforms 6.0 (API 23), 5.0.1 (API 21) and 4.4.2 (API 19), run

```
# android update sdk --no-ui --all --filter 6,29,32,34,103,105,107
```

Notice the use of '–all' switch here if you want to include packages that are only visible with that switch when running `list sdk`.

These should be enough to get started coding on a project. However, it is recommended that we also have an emulated environment to test our software. For that a system image is needed and the preferred choice here is the generic ARM EABI v7a System Image with Android API 23.

```
# android update sdk --no-ui --all --filter 56
```

To sum up, I got

- Android SDK Tools
- Android SDK Platform-tools
- Android *.* (API ??)
    - I have Android 6.0 (API 23), Android 5.0.1 (API 21), Android 4.4.2 (API 19)
    - For each, I got `SDK Platform` and `Google API`
    - For the latest API, I also got `ARM EABI System Image`
- Android Support Repository (Extras)

That should do it.

Update20170905 It's been quite a while, and Google has changed SDK management tools. The `android` binary is being deprecated, replaced by `sdkmanager`. To get the above components,

- `tools` is there with initial download

- `platform-tools`

```
sdkmanager "platform-tools"
```

- `build-tools`

```
sdkmanager "build-tools;26.0.1"
```

  - the current version is 26.0.1 - modify accordingly
- `platforms`

```
sdkmanager "platforms;android-23"
```

- (optional) `docs`

```
sdkmanager "docs"
```

- (optional) `ndk-bundle`

```
sdkmanager "ndk-bundle"
```

The component names can be viewed by

```
sdkmanager --list
```

One annoying thing is a warning error will appear on first run for missing `~/.android/repositories.cfg` file. To be fair, maybe the SDK does not want to create something behind our back... but still, just ask and create it already on first run! Anyways, just do a

```
touch ~/.android/repositories.cfg
```

That should solve the problem.

Gradle

It is the build tool of choice for Android SDK-based development. Only the binary distribution is needed. Assuming the downloaded ZIP file (gradle-X.X-bin.zip) is in $HOME/download/, unpack it to an installation folder (e.g. /home/share/tool/):

```
# cd /home/share/tool
# unzip $HOME/download/gradle-X.X-bin.zip
# export GRADLE_PATH=/home/share/tool/gradle-X.X
```

Notice that an environment variable SDK_PATH is set. Append this path to your PATH variable,

```
# export PATH=$PATH:$GRADLE_PATH
```

This will enable execution of gradle build tool from any path.

The rest is up to the configuration in Andoird SDK gradle plugin and our project file.

Android App From Scratch

Choose a project path (e.g. $HOME/project/test/)

```
$ mkdir -p $HOME/project/test
$ cd $HOME/project/test
```

Create basic source code path structure

```
$ mkdir -p src/main/java/org/my1matrix/my1testapp/
$ mkdir -p src/main/res/values/
$ mkdir -p src/main/res/layout/
```

Let us create a simple 'Hello' program in `src/main/java/org/my1matrix/my1testapp/`

TestActivity.java

```
package org.my1matrix.my1testapp;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class TestActivity extends Activity {
    @Override
    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);
        setContentView(R.layout.layout_main);
    }
    @Override
    public void onStart() {
        super.onStart();
        TextView textView = (TextView) findViewById(R.id.text_view);
        textView.setText("Hello, You!");
    }
}
```

An activity is basically a view on an Android system. You need to understand OOP class concept to really understand this code. Anyways, the layout for our main activity (i.e. main view) is named `layout_main` and will be defined/written in another file (below). As can be seen in the code above, a text view on our main activity will be changed to “Hello, You!” once the app has been started.

Now, to define a layout for the activity, create a file in `src/main/res/layout/`

layout_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
    <TextView android:id="@+id/text_view"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"/>
</LinearLayout>
```

This is a simple layout (vertical orientation that fills up the display), with a text display that will fill the whole layout).

Create a manifest file in `src/main/`

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="org.my1matrix.my1testapp"
  android:versionCode="1"
  android:versionName="1.0.0">
    <application
      android:theme="@android:style/Theme.Holo.Light.DarkActionBar"
      android:icon="@mipmap/ic_launcher"
      android:label="@string/app_name">
        <activity android:name=".TestActivity"
          android:theme="@android:style/Theme.Holo.Light.DarkActionBar"
          android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category
android:name="abdroid.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

This selected theme is something I remembered seeing from files generated by Android Studio. Two things to note here (1) a string constant app_name and (2) an icon definition (ic_launcher). The icon is actually optional and I will not bother to explain about it here.

To define the string constant, create a file in `src/main/res/values/`

strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
```

```
    <string name="app_name">MY1 Test</string>
</resources>
```

That is about it.

UPDATE20190530 *This NO LONGER WORKS with the latest Android SDK & gradle. Still trying to figure it out* 😕

Building APK using Gradle

In short, just run

```
$ gradle
```

to prepare/download all the necessary stuff to build. Once done, run `gradle tasks` to list all possible commands for various build configurations. If we simple want to build the APK, run

```
$ gradle build
```

If everything goes well, the built APK will be available at `build/output/apk`. The one we want would be named `<app-name>-release-unsigned.apk`.

That is all... in a nutshell.

Virtual Test Environment

Get a list of available targets

```
$ android list target
```

which should get us a list - something like the list shown below

```
Available Android targets:
----------
id: 1 or "android-19"
     Name: Android 4.4.2
     Type: Platform
     API level: 19
     Revision: 4
     Skins: HVGA, QVGA, WQVGA400, WQVGA432, WSVGA, WVGA800 (default),
WVGA854, WXGA720, WXGA800, WXGA800-7in
 Tag/ABIs : no ABIs.
----------
```

```
id: 2 or "android-21"
     Name: Android 5.0.1
     Type: Platform
     API level: 21
     Revision: 2
     Skins: HVGA, QVGA, WQVGA400, WQVGA432, WSVGA, WVGA800 (default),
WVGA854, WXGA720, WXGA800, WXGA800-7in
 Tag/ABIs : no ABIs.
----------
id: 3 or "android-23"
     Name: Android 6.0
     Type: Platform
     API level: 23
     Revision: 3
     Skins: HVGA, QVGA, WQVGA400, WQVGA432, WSVGA, WVGA800 (default),
WVGA854, WXGA720, WXGA800, WXGA800-7in
 Tag/ABIs : default/armeabi-v7a
----------
...
```

So, to create a virtual device with Android 6.0 as target platform and label it as MyDevice, run

```
android create avd -n MyDevice -t 3
```

To start the virtual device, run

```
$ emulator64-arm -avd myDevice
```

To install our app on the virtual device, use `adb`

```
$ adb install -r build/outputs/apk/<app-name>-debug.apk
```

Some extra info

Any running AVD can be accessed using telnet (localhost:{port} where {port} is the number displayed on AVD window).

To simulate GPS fix: run `geo fix {long} {lat}` in a telnet environment

Signing Android App

From here.

To create a keystore (and a private key) use a Java tool (`keytool`)

```
$ keytool -genkey -v -keystore mykeys.keystore -alias theKey -keyalg RSA -
keysize 2048 -validity 10000
```

It will prompt you for a store password and key password. Validity of 10000 days would give you about 27 years... should be enough, don't you think?

To sign an APK, use another Java tool (`jarsigner`)

```
$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore
mykeys.keystore app.apk theKey
```

This will also prompt you for passwords. The tool will complain about missing `-tsa` option, so we can insert

```
-tsa http://timestamp.digicert.com
```

into the command. I am not sure if this is a free thingy - from what I see, we only need to pay if we want a full certification. Timestamping is a free service? (Guys from digicert, please correct me if I'm wrong! :p)

To verify if the APK has actually been signed,

```
$ jarsigner -verify -verbose -certs app.apk
```

Finally, use Android SDK tool (`zipalign`) to realign bytes,

```
$ zipalign -v 4 app-unaligned.apk app.apk
```

***Note**: The zipalign tool is in build-tools/<version>/. Add this path to system PATH to make things easier.*

That is all.

IDE: Android Studio

This is the recommended development environment - using Android Studio.

Most of the stuff here are based on what I read from the internet - kudos to them.

Notes on starting new project

dumped...

android-studio_init.txt

```
creating new project: post-create checklist

- refactor (change module name)
- modify app's build.gradle
```

```
import java.text.DateFormat
import java.text.SimpleDateFormat

static def getDateTime() {
    DateFormat that = new SimpleDateFormat("yyyyMMddHHmmss")
    return that.format(new Date())
}

android {
    signingConfigs {
        debug {
            storeFile file('/home/azman/.keystore/my1keys.jks')
            storePassword 'testkey'
            keyAlias = 'testKey'
            keyPassword 'testkey'
        }
    }
    ... (compileSdkVersion,buildToolsVersion)
    def name = archivesBaseName
    def code = getDateTime()
    def vers = "0.1"
    defaultConfig {
        ...
        versionCode code.toBigInteger()
        versionName "${vers}"
        signingConfig signingConfigs.debug
    }
    buildTypes {
        applicationVariants.all { variant ->
            variant.outputs.all {
                if (variant.name == "release") {
                    outputFileName = "${name}-${vers}-${code}.apk"
                } else {
                    outputFileName = "${name}-${variant.name}-
${code}.apk"
                }
            }
        }
        ...
    }
}

- find file>settings (disable spelling check in code inspection)
- enable version control (git) & commit
```

Re-Use library module

Assuming current project is *project2*, and the library module *libmod1* is in project *project1*.

- edit `settings.gradle` file in *project2* root path

```
include ':libmod1'
project(':libmod1').projectDir = new File('/path/to/project1/libmod1')
```

- edit `build.gradle` (for calling module)

```
dependencies {
    implementation project(path: ':libmod1')
}
```

Create Build Numbers (Code Version)

It is nice to have a build number that is automatically incremented for every build.

Edit the build.gradle file in 'app' module, and add the lines shown below. A file (in this case, 'version.properties') will be created if not already there. Naturally, the build number will start with 1.

build.gradle

```
android {
    ...
    def that = file('version.properties')
    Properties prop = new Properties()
    prop.load(new FileInputStream(that))
    if (that.canRead()) {
        prop.load(new FileInputStream(that))
    }
    def code = (prop.getProperty('VERSION_CODE') ?: "0").toInteger() +
1
    if (!that.exists()) {
        that.createNewFile()
        if (that.canWrite())
            prop.store(that.newWriter(), null)
    }
    ...
}
```

Adding Map to Activity other than MapActivity

I noticed the default MapActivity class implements OnMapReadyCallback interface. But simply adding

that interface definition does not automatically load the required imports. The trick is to add the following line into app module's `build.gradle`.

build.gradle

```
android {
}
dependencies {
    ...
    compile 'com.google.android.gms:play-services-maps:11.0.4'
    ...
}
```

Of course, the version number may differ. Just create a new project using MapActivity and checkout the version used.