Saarland University
Information Security & Cryptography Group
Prof. Dr. Michael Backes

Android Security Lab WS 2014/15

Lab 1: Android Application Programming

M.Sc. Sven Bugiel

Version 1.0 (October 6, 2014)

Login on Lab Machines

Username:	smartseclab
Password:	android

Introduction

In this Lab, we provide a simple introduction to application programming on Android, with focus on familiarizing the students with the most important IDE elements and on inter-application communication based on *Intents* and *Services*.

1 Development environment

In this lab we use the*Android Development Tools* (ADT)[1], which are based on the Eclipse IDE. Alternative IDEs exist, e.g., Android Studio based on IntelliJ IDEA[2].

1. Open the ADT from your desktop
2. As workspace chose the `/usr/local/home/smartseclab/androidlab` directory
3. On the bottom you see Android development specific tools such as Logcat, Emulator control, etc., which you will use later.

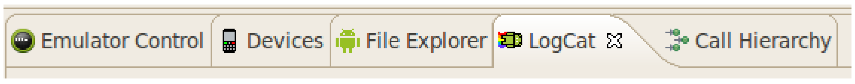


Figure 1: Android development specific tools

4. In the top right corner you can switch to DDMS, the debugging view for Android **if required**.
5. Further, in the Menu Bar, you see two new entries:
   (a) for the Android SDK Manager (left)
   (b) for the Android AVD (Android Virtual Device) Manager (right)

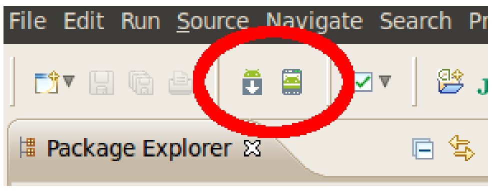


Figure 2: Android Menu Bar entries

[1]`http://developer.android.com/sdk/index.html`
[2]`http://developer.android.com/sdk/installing/studio.html`

2 Creating an Android Virtual Device

1. Open the AVD Manager.
2. Use the wizard to create a new **Intel Atom (x86) Android v4.3** AVD by either configuring it according to your wishes (first tab) or using a pre-configured device definition (second tab).
3. Start your new virtual device
4. Open a terminal (using the panel entry on top of your screen or from the desktop link) and use the *Android Debug Bridge* (adb) to observe the device's log output:

```
$ adb logcat
```

For a full list of the adb features (e.g., emulator control), please refer to the adb help page.

5. Once the device has been fully booted, switch to the DDMS view in ADT and *briefly* familiarize yourself with available options (e.g., overview of device's processes, colored logcat view at the bottom, etc.). In particular, use the emulator control to send an SMS message to your device and verify your virtual device has received the message.
6. The emulator control is essentially just a GUI for a telnet-based service to send commands to the emulator. Sending an SMS message could, thus, also be accomplished from the terminal using a telnet client:

```
$ telnet localhost 5554
$ sms send 5556 "Your␣message"
```

where *5554* is the ID of the emulator (check the emulator window name) and *5556* the sender number of the SMS.

7. Create and boot a second virtual device using the same configuration for SDK version as for the first device.
8. Once this second device has booted, go to the SMS (*"Messaging"*) app and send a test SMS to the first emulator device. The phone number of each device equals the number (e.g., 5554) in the title bar of the emulator window.
9. To end emulators, simply close their windows, no special shutdown procedure is required.
10. Close both emulators and the AVD window.

3 Creating a first Android App

1. Switch back to the Java view in Eclipse.
2. Create a new Android Application Project (cf. Figure 3).

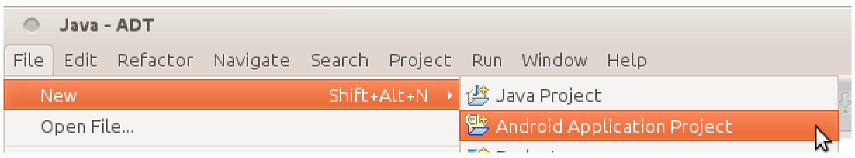


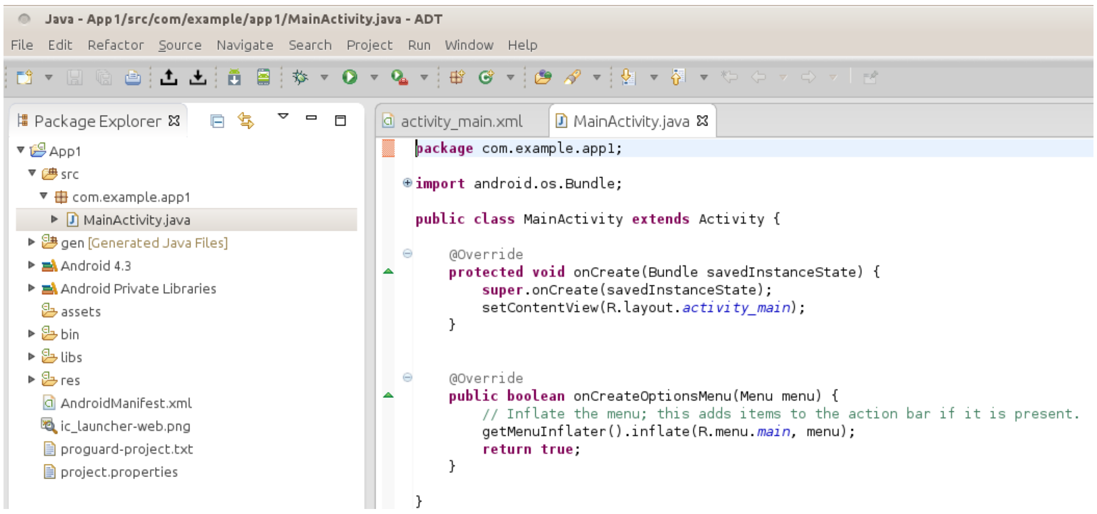
Figure 3: Creating a new Android Application Project

3. Since we will extend this app in the next lab session, please use the following settings:
   (a) Application Name "App1"

   (b) Minimum required SDK API 8 (Froyo)
   (c) Leave the rest in this option dialog and all following dialogs as default

4. Once the project has been created, you should see a screen similar to Figure 4, which shows a simple application skeleton with a "Hello World" Activity.

5. To quickly test your app, start one of your virtual devices and wait until it has booted up. Then right-click on your App1 project and chose "Android Application" in the "Run as" context menu (cf. Figure 5). When there is only one emulator running, the app will be automatically started on this one. Otherwise a menu pops up in which you can chose the target virtual device.



```
package com.example.app1;

import android.os.Bundle;

public class MainActivity extends Activity {

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


    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }

}
```

Figure 4: New app template

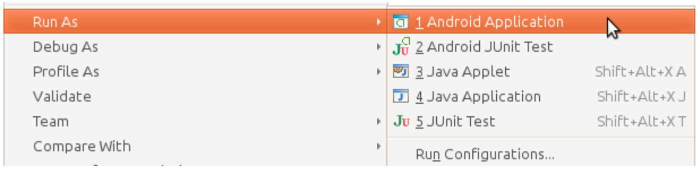


Figure 5: Run as Android application

6. In the following, we will extend this app with a simple Service and a BroadcastReceiver component.

3.1 Adding a BroadcastReceiver component

1. Create a new class "Receiver.java" in your package (i.e., `com.example.app1.Receiver`)

2. The following Listing presents a minimal implementation of the BroadcastReceiver component:

```
package com.example.app1;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class Reciever extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    // This code is called when the receiver is triggered
  }
}
```

3. Add the following line of code to the `onReceive` method in order to log the received *Intent* message:

```
Log.d("App1.Receiver", "Called␣with␣Intent␣"+intent.toString());
```

4. To actually register the BroadcastReceiver in the system upon app installation, we have to extend the `application` entry of the `AndroidManifext.xml` of App1. You could do this using the GUI of ADT. For now, however, we will edit the XML directly by adding the following lines of XML:

```
<application ...>
...
  <receiver android:name="com.example.app1.Receiver">
    <intent-filter>
      <action android:name="com.example.intent.action.BROADCAST"/>
    </intent-filter>
  </receiver>
...
</application>
```

This registers the BroadcastReceiver component in this system. It defines an Intent filter, which means the BroadcastReceiver will only receive broadcasted Intents that match the action string `com.example.intent.action.BROADCAST`.

5. To quickly test your new BroadcastReceiver, re-install your modified app by running it again on the virtual device. Start monitoring the logcat output, either in the DDMS view or in a terminal by filtering for the string App1. Run in a (second) terminal the following command to instruct the *ActivityManager* on the device to send a broadcast that fits the Intent filter of the receiver component:

```
$ adb shell am broadcast -a com.example.intent.action.BROADCAST
```

6. If done correctly, you should be able to see debug output similar to:

```
D/App1.Receiver( 2136): Called with Intent Intent{ act=com.example.intent.action.BROADCAST flg↷
    =0x10 cmp=com.example.app1/.Reciever }
```

7. A broadcast receiver can also be registered dynamically at runtime through an application's context. However, at this point, we refer you to the documentation[3] for more information on this.

3.2 Adding a simple Service component

1. Next, we will add a simple Service component to the application to which other apps can bind and then interact with in a client-server fashion. First, define the interface of the new service through the *Android Interface Definition Language* (AIDL)[4]. Create in your App1 source code package a new file `IRemoteService.aidl` with the following content:

```
package com.example.app1;

interface IRemoteService {
  int getPID();
}
```

[3]http://developer.android.com/reference/android/content/BroadcastReceiver.html
[4]http://developer.android.com/guide/components/aidl.html

2. Now we will implement this interface in a dedicated Service component (cf. Figure 8 and Figure 9 in Appendix A for an illustration of a Service's life-cycle). Thus, add a new class to your App1 package called `RemoteService.java`. The following shows its implementation:

```
package com.example.app1;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.Process;
import android.os.RemoteException;

public class RemoteService extends Service {

  @Override
  public IBinder onBind(Intent intent) {
    // Object to be returned to caller apps for RPC
    return mBinder;
  }

  // Stub of our service, i.e., implementation of the interface
  private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
    @Override
    public int getPID() throws RemoteException {
      // We use android.os.Process
      return Process.myPid();
    }
  };
}
```

The `Stub` implements the Service interface. The only remotely callable function `getPID()` simply returns the service's process ID to the caller. Callers that bind to this service, receive an `IBinder` interface object for this service, which they can use for RPC.

3. Of course, we also have to register this service in the system through the `AndroidManifest.xml` upon installation:

```
<application ...>
  <service android:name="com.example.app1.RemoteService">
    <intent-filter>
      <action android:name="com.example.app1.IRemoteService"/>
    </intent-filter>
  </service>
  ...
</application>
```

4. To test this service, we create a simple second caller application.

3.3 Creating a second Android App

1. Repeat the steps for the creation of App1, however, call this new app App2.
2. Since App2, as a caller to the service of App1, has to know the Service interface, we have to copy the AIDL file from the package of App1 to the package of App2 as shown in the following Figure 6.
3. For this setup, we keep binding and calling to App1's service very simple and integrate them strictly into the life-cycle of App2's main activity (cf. Figure 7 in Appendix A for an illustration of the Activity life-cycle). Add the following code to the `MainActivity.java` of App2:

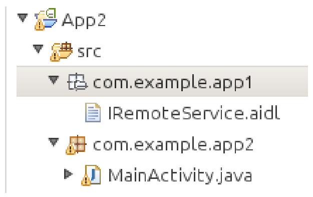


Figure 6: Setup for App2 to be able to create a `Proxy` for App1's service.

```
IRemoteService mIRemoteService;

private ServiceConnection mConnection = new ServiceConnection() {
  // Called when the connection with the service is established
  public void onServiceConnected(ComponentName className, IBinder service) {
    // This gets an instance of the IRemoteInterface, which we can use to call on the service
    mIRemoteService = IRemoteService.Stub.asInterface(service);
    int pid = -1;
    try {
        pid = mIRemoteService.getPID();
    } catch (RemoteException e) {
        Log.e("App2.MainActivity", "Error␣calling␣getPID()", e);
    }
    Log.d("App2.MainActivity", "getPID()="+pid);
  }

  // Called when the connection with the service disconnects unexpectedly
  public void onServiceDisconnected(ComponentName className) {
    mIRemoteService = null;
  }
};

@Override
protected void onResume()
{
  super.onResume();

  if(bindService(new Intent(IRemoteService.class.getName()), mConnection, Context.↪
      BIND_AUTO_CREATE))
  {
    Toast.makeText(this, "Binding␣to␣successful", Toast.LENGTH_LONG).show();
  } else {
    Toast.makeText(this, "Binding␣to␣failed!", Toast.LENGTH_LONG).show();
  }
}

@Override
protected void onPause()
{
  super.onPause();
  if(mIRemoteService != null)
  {
    unbindService(mConnection);
  }
}
```

This code will bind to the service every time the Activity comes back to the foreground on the screen (`onResume()`) and will directly use the IBinder object received upon binding to call the remote service. The received PID from App1 is logged. The result of the binding operation is briefly shown as a pop-up (*Toast*) on the screen to the user. To correctly cast the IBinder object to the interface of App1's service, App2 requires the AIDL file. To avoid leaking service connections, the activity unbinds from the service as soon as the activity is not longer in the foreground on the screen (`onPause()`).

4. Install first App1 and afterwards App2 by running them as Android applications. Observe the screen for successful binding and the logcat output for the logged PID.

5. For simpler service implementations, such as `IntentService`s, we refer to the Android documentation[5].

[5] `http://developer.android.com/reference/android/app/IntentService.html`

A Activity and Service Life-Cycle

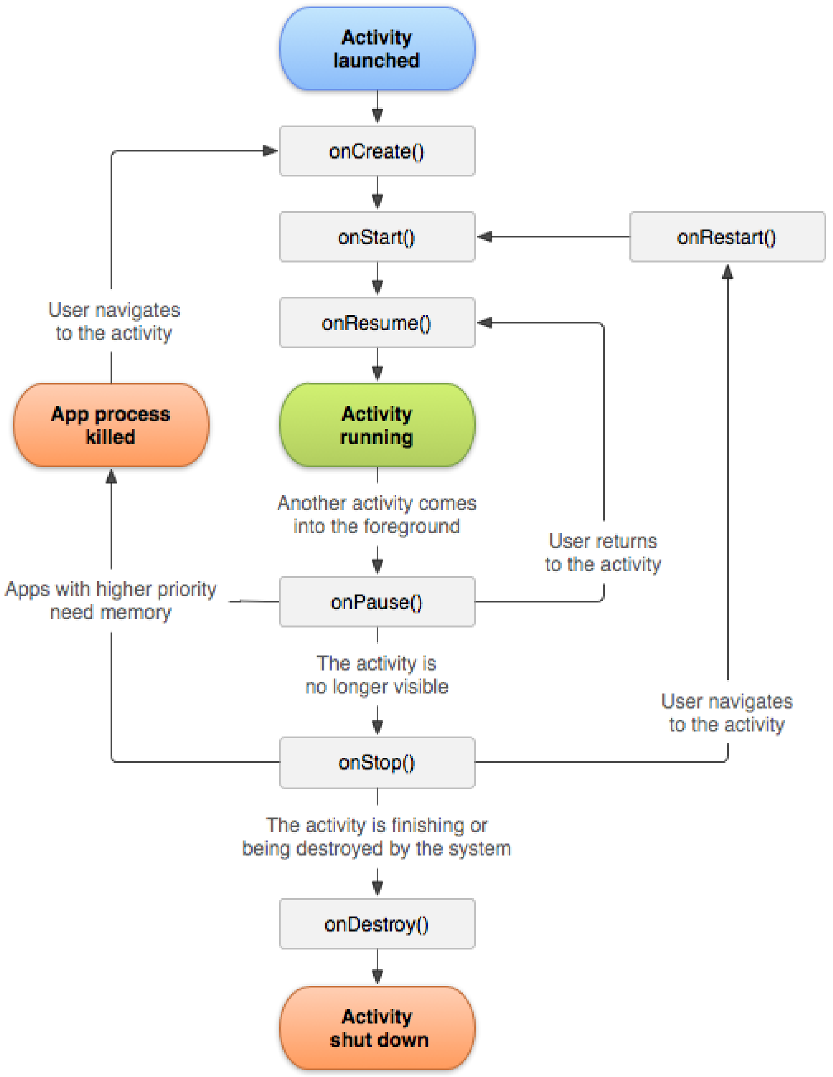

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 7: Activity lifecycle (Source `http://developer.android.com/reference/android/app/Activity.html`)

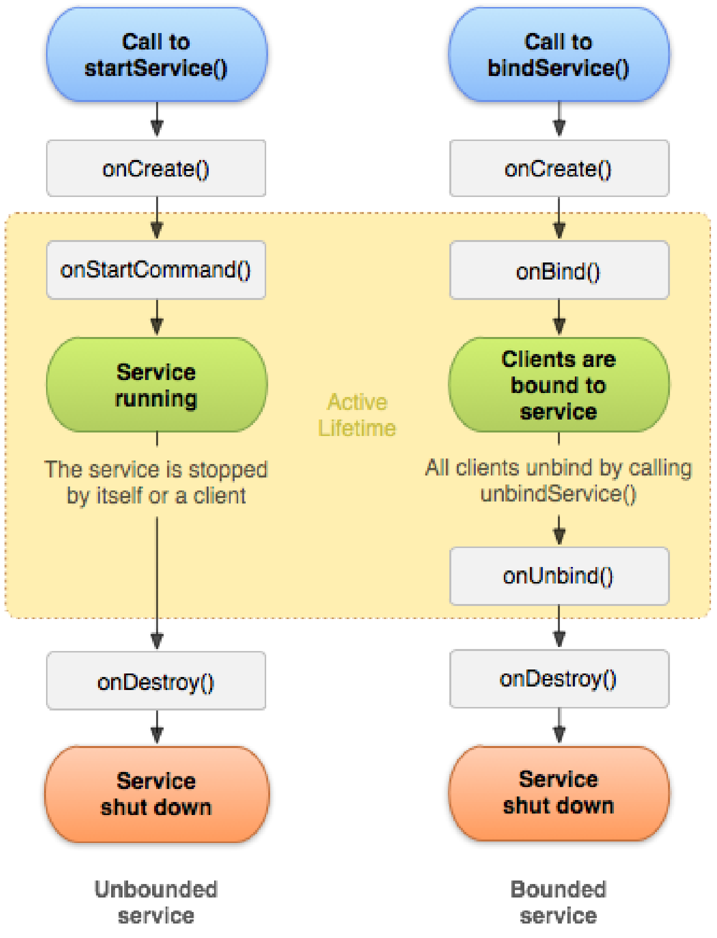


Figure 8: General Service life-cycle (left: service creation with `startService()`; right: service creation with `bindService()`; source `http://developer.android.com/guide/components/services.html`)

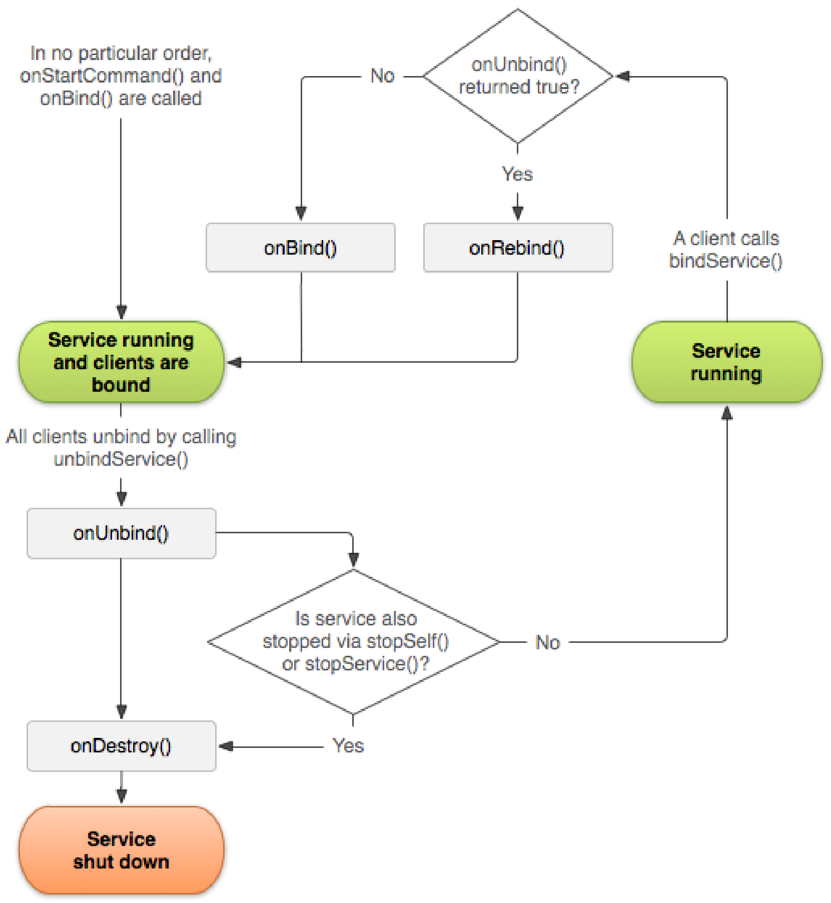

In no particular order, onStartCommand() and onBind() are called
No
onUnbind() returned true?
Yes
onBind()
onRebind()
A client calls bindService()
Service running and clients are bound
Service running
All clients unbind by calling unbindService()
onUnbind()
Is service also stopped via stopSelf() or stopService()?
No
onDestroy()
Yes
Service shut down

Figure 9: Bound-Services life-cycle (Source `http://developer.android.com/guide/components/bound-services.html`).