Wirtschaftsuniversität Wien

Developing For Android

From 0 to 100 on Mobile and Wearable Devices

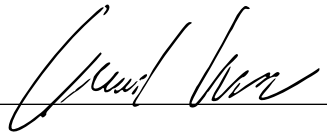
Author:
Gerald Urschitz

Supervisor:
Ronny G. Flatscher

January 7, 2016

I, Gerald Urschitz, declare that this paper titled, 'Developing for Android' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this paper has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the paper is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: January 7, 2016

Contents

1. Introduction

In this paper, I will present everything that an entry-level developer needs to know to get started with Android as well as Samsung Gear. A basic understanding of Java and XML as well as a JavaScript and HTML is assumed, this will not be part of the paper. For getting started with Java, you can use the interactive tutorial from Codecademy at `https://www.codecademy.com/learn/learn-java`, for XML you may have a look at `http://www.w3schools.com/xml/`. For JavaScript and HTML have a look at this tutorial: `https://www.codecademy.com/skills/make-an-interactive-website`
With the help of a very simple but practical example application - RecentTweets - , I will explain the important concepts of Android. This application makes use of the Twitter API to show the last 20 Tweets of a logged in Twitter User. Furthermore, I will make a little excursion into the world of Wearables and outline what is needed to get started with developing for the Gear 2 Smartwatch by Samsung, that runs Tizen as an Operating System.

1.1. Android

A lot has changed since a company named Android was founded in 2003. Only 2 years later Google, Inc. bought this very company, from which the public only knew that it made Software for mobile phones. Since then, there was not a single Android Phone out there. Since the first Phone was released in October 2008, Android has become more and more popular throughout the years. [1] In the 2nd Quarter of 2015, Android alone had a worldwide market share of around 82.8% according to IDC[2], as displayed in table 1.1.

Period	Android	iOS	Windows Phone	BlackBerry OS	Others
2015Q2	82.8%	13.9%	2.6%	0.3%	0.4%
2014Q2	84.8%	11.6%	2.5%	0.5%	0.7%
2013Q2	79.8%	12.9%	3.4%	2.8%	1.2%
2012Q2	69.3%	16.6%	3.1%	4.9%	6.1%

Table 1.1.: Q2 Market Share from 2012 - 2015

For this reason, Android simply cannot be ignored as a platform when it comes to developing

apps for Mobile Phones.
Android itself is based upon Linux and since Android L 5.0, it uses the Android Runtime (ART) as a Runtime-Environment for its Apps. Up until Android 4.4 KitKat, Android used the virtual machine Dalvik. [3] The applications are mostly written Java, but can also make use of C and C++ Libraries for performance-critical tasks. Furthermore, it should be mentioned that Android is Open Source and follows the Apache License. [1]

1.2. Example Application "RecentTweets"

The example application in this paper will be called "RecentTweets". It will make use of Twitters relatively new SDK Fabric and will display the last 20 Tweets of a signed in Twitter User. First, there will be a *LoginActivity* that simply shows a Twitter Login Button. This is necessary, as we need to obtain the authentication token for the user. In the *RecentTweetsActivity*, we will display the last 20 Tweets of the user. For this small Application, I created three Mockups, as you can see in 1.1.

(a) LoginActivity (b) RecentTweetsActivity

Figure 1.1.: Mockups for RecentTweets

2. Development Environment

In this chapter, I will describe how to set up the development environment to build applications for Android. First, I will briefly describe Android Studio, the Integrated Development Environment - short IDE - that comes directly from Google. Then, I will show you how to create an empty project and I will explain the most important files in that new project. In the end, I will explain the Gradle Build System briefly.

2.1. Android Studio

Android Studio is the Integrated Development Environment officially developed by Google and it is powered by the IntelliJ Platform of Jetbrains. Both Android Studio as well as the Android SDK can be downloaded on `https://developer.android.com/sdk/index.html`. It is available for Windows, MacOS X and Linux. For information on how to install Android Studio, please refer to appendix Installation Guides.

2.2. Creating an empty Project

Let's dive straight into it and create a new project for our application in Android Studio. Simply choose File ->New ->New Project in the menu to open the Wizard that guides you through the project creation process.

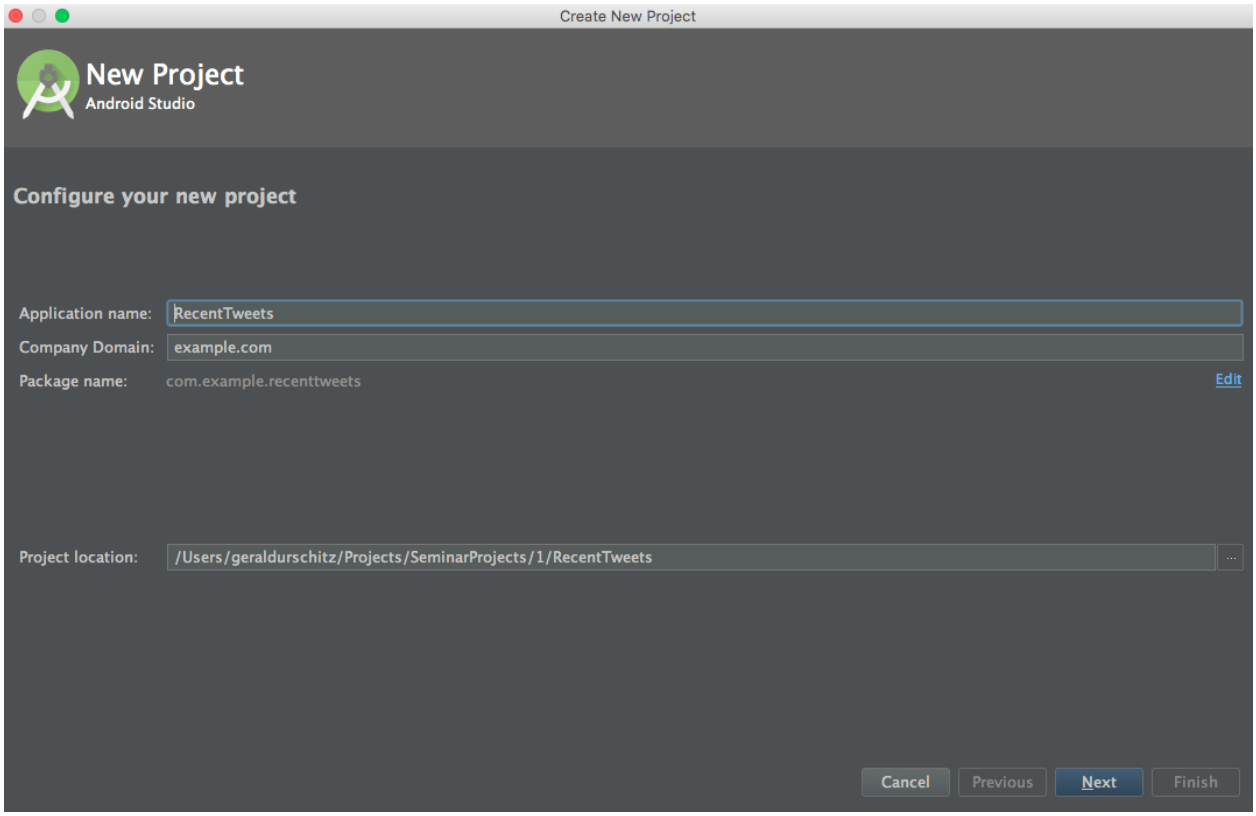


Figure 2.1.: In the first window of the Wizard, you can define the application name, the company domain and the project location.

In the first window, which is shown in figure 2.1, we can define the application name, the company domain and the project location. For the application name, we choose "RecentTweets" following the camel case naming convention of Java. The company domain will be "example.com". Out of this company domain, android studio generates the package name for our project. This package name can be edited as well, but we will keep it like that to follow the conventions on package naming. As "Project Location", we enter or choose the path that we want the project to be located at. Preferably, the folder name is the same as the projects name. After that, we click 'Next' and see a window like the one in figure 2.2.

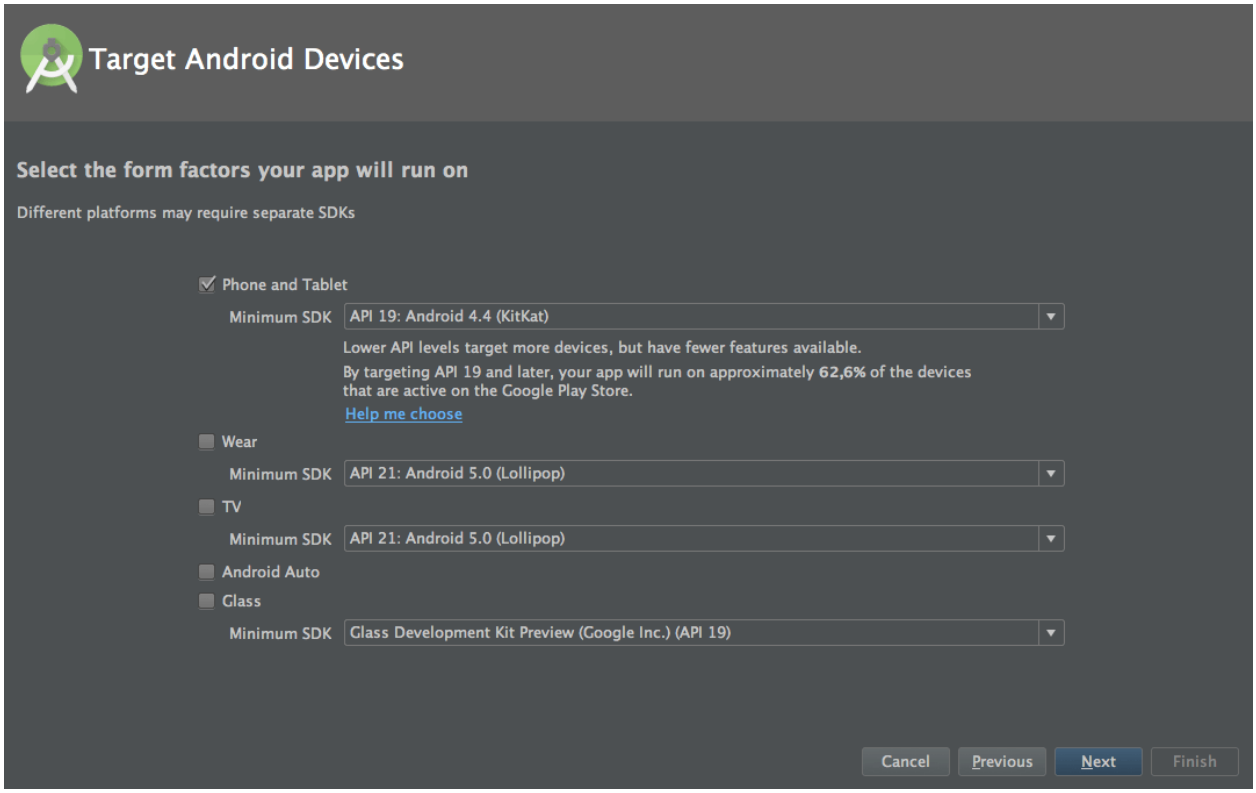


Figure 2.2.: In the second window of the Wizard, you can set the platform that you want to develop for as well as the Minimum SDK for each platform.

As we only want to develop our app for phones and tablets, we keep the default selection of the platform. Furthermore, we want to target API Level 19: Android 4.4 (KitKat), which we chose earlier in the SDK Manager as well. After that, we click 'Next'.

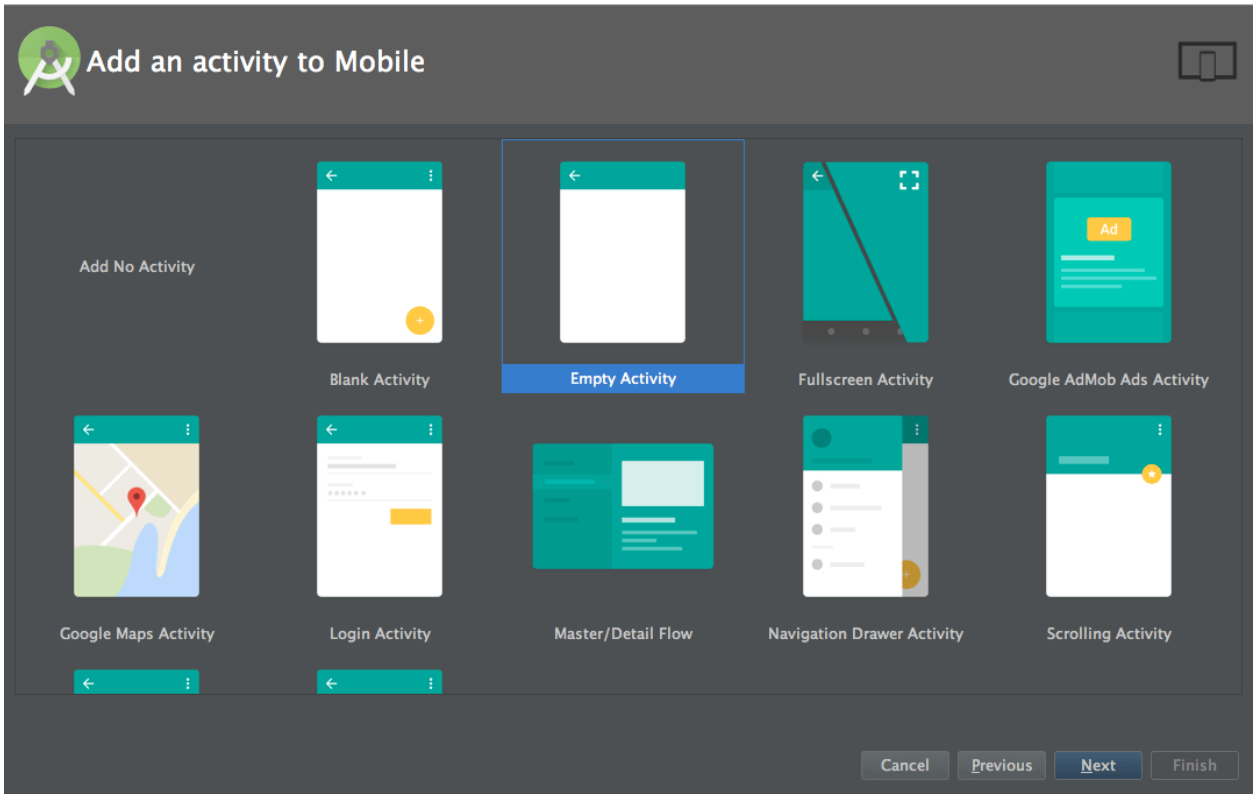


Figure 2.3.: In this window, you can choose which type of Activity you want to add to the project.

In the window that is shown in figure 2.3, we can choose which type of Activity we want to add to our project. What activities are and how we are going to use them is subject of the next chapter, right now we just want to create an "Empty Activity".

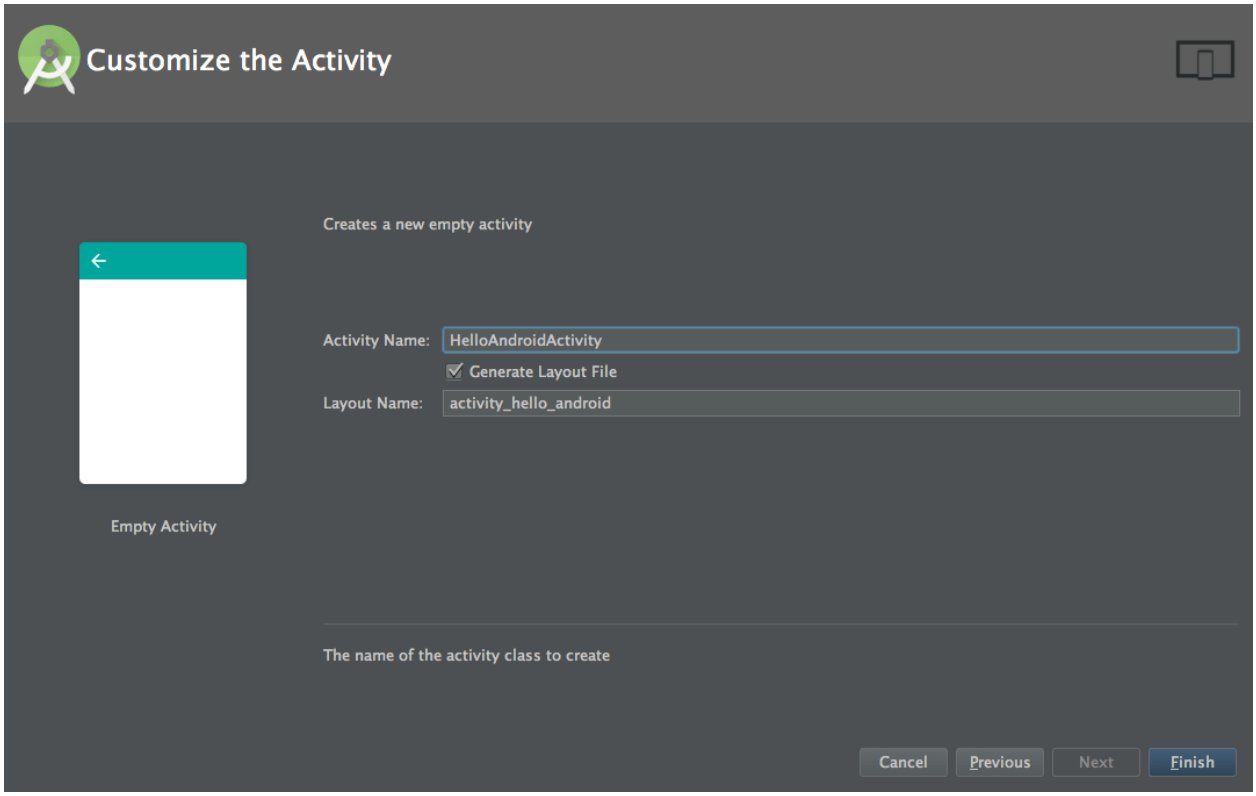


Figure 2.4.: In this window, we can give the activity, that we are about to create, a name.

Finally, as you can see in 2.4, we give our activity a name. To follow conventions, we call our activity "HelloAndroidActivity". This means, every activity has the word "Activity" in the end, and it is again written in CamelCase. Android Studio generates automatically a layout file for you, if the respective checkbox is checked. It also automatically gives the layout file the right name, following the android naming convention. For layout files, the name always starts with the type of layout, such as "activity" and then the name, all written lowercase and with underscores. Lowercase is important for internal referencing, so if you stick to these conventions, you will not counter any problems.
After clicking finish, the newly created project will load. In the project inspector on the left side of the window, we can have a view at the project in different views, which you can choose from on the top left. For our purposes, the view "Project" is sufficient. It shows us the directory structure and the including files of the whole project. In figure 2.5, I've marked all files that are important for us right now.

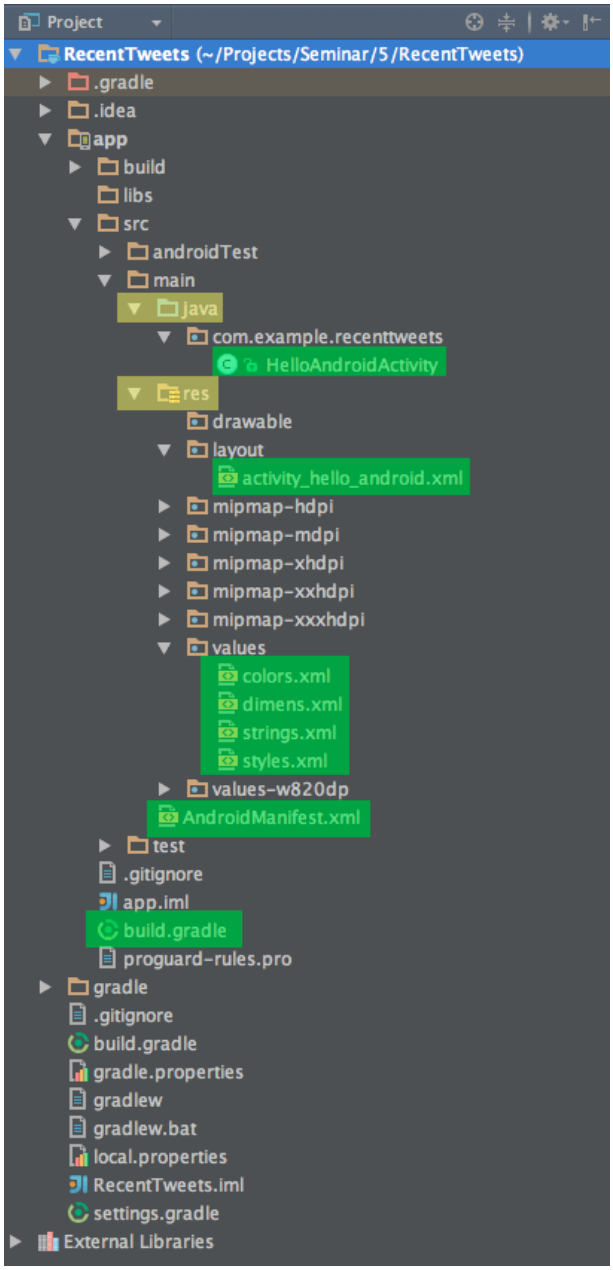


Figure 2.5.: The directory structure of our project.

2.2.1. java

In this directory, we will place all java files that contain code of our app. They will be structured in java packages.

- **HelloAndroidActivity**: This is the java file which declares our *HelloAndroidActivity* class. What activities are and specifically, how our *HelloAndroidActivity.java* looks like, is subject to the next chapter Hello Android!.

2.2.2. res

As the name might suggest, this directory contains all the resources we need for our app. In the official android docs, Google describes it this way: "Resources are the additional files and static content that your code uses, such as bitmaps, layout definitions, user interface strings, animation instructions, and more." [4]

- **activity_hello_android.xml**: This file declares the layout for our activity and defines, how our activity looks like. All layout files for all activities or fragments (or anything else, like for example a simple view) should be placed in the folder 'layout' in res. Again, this will be subject of the next chapter Hello Android!.
- **colors.xml, dimens.xml, strings.xml, styles.xml (and possibly more)**: In android, as soon as we want to define some value that we might want to reuse later again, we should think about defining a value for it. For example, let's say we have a button that has the color #2980b9. We can define that color as 'blue' by putting *<color name="blue">#2980b9</color>* in colors.xml and then just use the identifier *@colors/blue* in our layout file.

2.2.3. AndroidManifest.xml

AndroidManifest.xml is a required xml-file for each android application and defines important information of your app. Among other things, it defines the java package, declares permissions for the application, sets miscellaneous properties of the app, declares a main activity and it sets the minimum API Level. After creating the project, it looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.example.recenttweets"
          xmlns:android="http://schemas.android.com/apk/res/android">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".HelloAndroidActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>

                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>

</manifest>
```

2.2.4. build.gradle

This file configures values for the gradle build system, which will be explained in detail in the next section.

2.3. Gradle Build System

In Android, we use Gradle as our build system. The build system helps us to build, test, run and package our app and makes development and deployment very easy. It furthermore let's us declare dependencies on local and remote libraries and modules. I don't want to dive too deep into gradle and it's applications, and it is not necessary for the beginner user to know gradle by heart. In fact, you don't even have to create the gradle build files yourself, Android Studio does this for you when you create a new project. In this section, I want to show what's important for us using gradle.
After creating the new project, we have two gradle files named *build.gradle* One is the project-level build file and manages all modules in our project. We can define dependencies and configuration that apply to all our subprojects and modules. If you want to include different modules and subprojects, you need to define them in the *settings.gradle* file. The other *build.gradle* file can be found in the directory *app*. This is the one that is interesting to us, if we want to include external libraries. It looks like this right now:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 22
    buildToolsVersion "22.0.1"

    defaultConfig {
        applicationId "com.example.recenttweets"
        minSdkVersion 19
        targetSdkVersion 22
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:22.2.1'
}
```

For Gradle, we want to apply the Android plugin, which provides several tasks for android builds and enables the android element. This is done by the first line in the gradle file.[5]. In the element android, we can see that we can configure values such as the *compileSdkVersion*

as well as the *buildToolVersion*. Furthermore, we see the element *defaultConfig*. This element configures settings for the manifest file and overrides values. The *buildTypes* element can define properties for the debug and the release version the app. [5]. As already mentioned, these settings are not too much of interest for us now and are not subject of this paper. We care about the next top-level-element called *dependencies*.

2.3.1. Defining Dependencies with Gradle

There are three different types of dependencies we can have in the gradle file, which are *Local binary dependencies*, *Remote binary dependencies* and *Module dependencies*. For including local binary dependencies, let's look at the first line in our dependencies element in the gradle file.

```
1 compile fileTree(dir: 'libs', include: ['*.jar'])
```

This line tells the build system to compile everything that it finds that ends with *.jar* directory named 'libs' in the filetree. Including a whole local module looks like this:

```
1 compile project(":lib")
```

Last but not least, if we want to include and compile a remote dependency, it could look like this:

```
1 testCompile 'junit:junit:4.12'
```

This declares the testing framework *junit* as a dependency for the test module, specifically version 4.12 of *junit*. Notice here that it says *testCompile* instead of *compile*. If we want to include the dependency in the app itself, we just write *compile*. [5]

3. Hello Android!

In this chapter, I will show you how to build your first Android Application. We already created an empty project in the last chapter. In fact, this project we created was not totally empty. It generated a file named *HelloAndroidActivity.java* for us, that contains our Activity, as well as a file called *activity_hello_android.xml* that belongs to our activity and defines its layout. In this layout, there is already a TextView defined, that says "Hello World!". In this chapter we will extend that project. We want a button right underneath the TextView and when we click this button, we want the text to change from "Hello World!" to "Hello Android!". Our button will be labeled with "Change TextView". Figure 3.1 shows, how we want the final result to look like, when our app is running and we clicked the button.

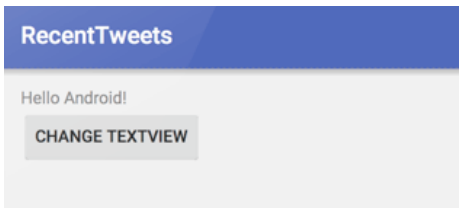


Figure 3.1.

But before we actually get to work, I will explain in the following section what activities are and dive into the activity lifecycle - the pulse of Android so to speak.

3.1. Activities

"An activity is a single, focused thing that the user can do." writes Google on the official developers references for the Activity Class. [6] Every application for Android needs a default activity, that is launched when the app is launched. This default activity is defined in *AndroidManifest.xml*, that we already discovered in chapter 2.

3.1.1. Activity Lifecycle

Even though there is also an application lifecycle, that is one level above the activity lifecycle, for us the latter is more important. Every Activity, that launches, will go through the activity lifecycle, that is displayed in figure 3.2.
All in all, there are four states that an activity can have: *running*, *paused*, *stopped* and *finshed*. These states are also shown in the figure 3.2 in the colored and rounded boxes.

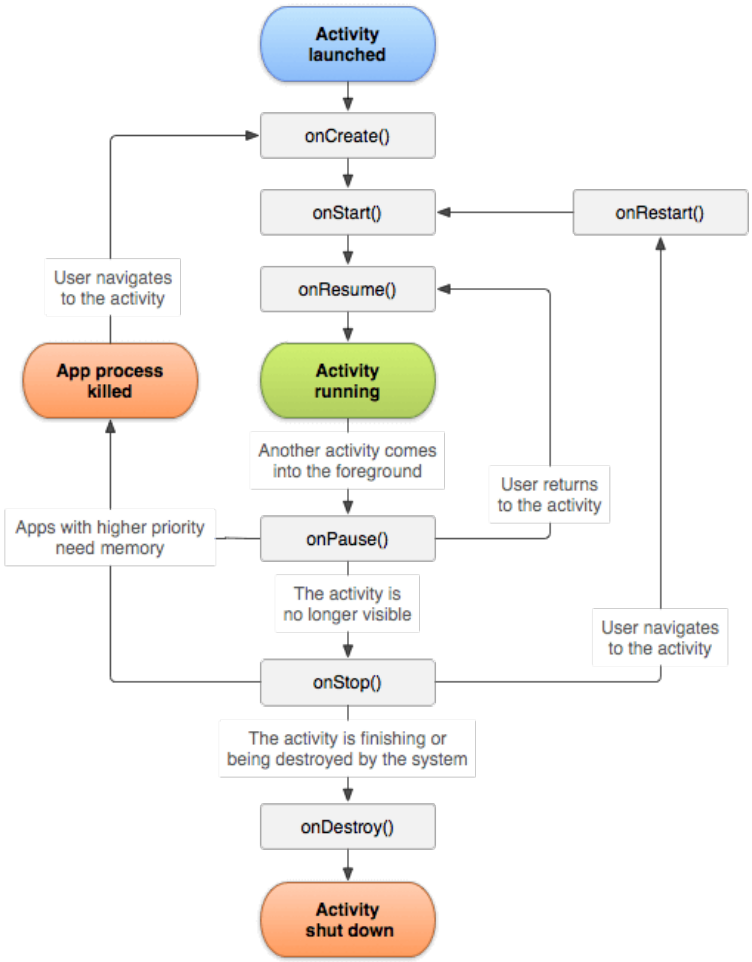


Figure 3.2.: Image from [6].

Furthermore, there are three interesting key loops:

- **The entire lifetime**: The entire lifetime starts with a call on *onCreate()* and ends at *onDestroy()*. For example, if we first start an application, it will launch the main activity and call *onCreate()* on this activity. In our case, *onCreate()* in *HelloAndroidActivity* would be called, when we start the application. Finally, when the activity is finished (for example, calling *finish()* in the activity) or when the system is finishing the activity in the background for memory-reasons.

- **The visible lifetime**: The visible lifetime starts with a call on *onStart()* and ends with a call on *onStop()*. The *onStart()*-hook is called, when the activity is visible to the user. Then, when the activity is hidden, for example because we started another activity, *onStop()* is called.

- **The foreground lifetime**: Last but not least, the foreground lifetime stats with *onResume()* and ends with *onPause()*. For example, *onResume()* is being called, when

the activity finally starts interacting with the user. *onPause()* is being called when you go into sleep mode. The activity still runs, but it is paused.

Please refer to [6] for more information.

3.2. Running the Application

Even though we didn't write a single line of code, we already can run our application. Android Studio generated a working app for us, one that displays 'Hello World!'. Let's verify that and check, if everything actually works as we expect it. On small thing, before we run the application: We should make sure, that we have the Intel Emulator Accelerator HAXM installed. You an check this under Tools ->Android ->SDK Manager ->SDK Tools. There should be a checked line similar to the one displayed in the red box in figure 3.3.

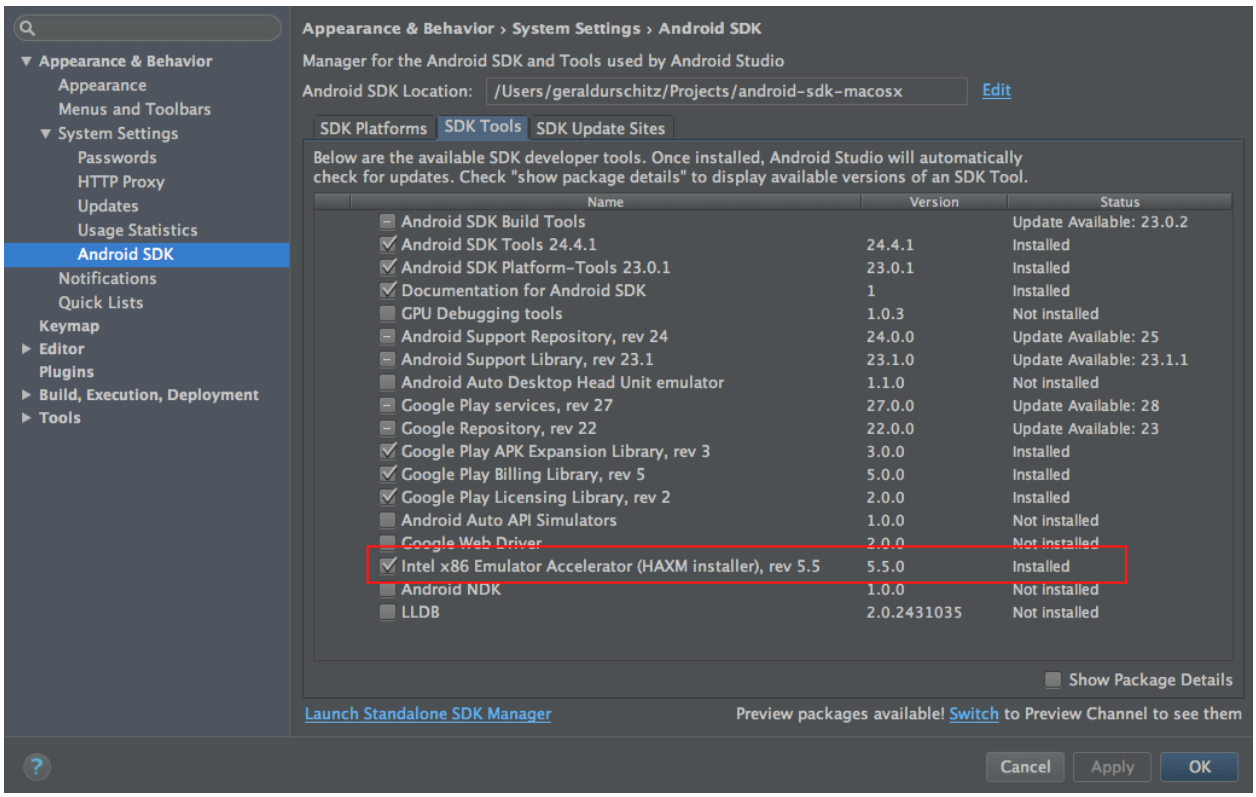


Figure 3.3.

If that's the case, just click on play next to the predefined run configuration named 'app' in the top bar. Alternatively, you can hit 'Ctrl+R' on your keyboard to run the application. See figure 3.4

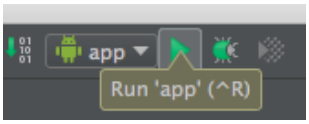


Figure 3.4.

It might happen that it still doesn't work, even if HAXM seems to be installed. To resolve this issue, go to Tools ->AVD Manager and click on the device, where the problems occur. Double clicking it should open a popup that says 'Install HAXM'. Do so and after that, the emulator should start. (Note, that you do not have to start another emulator again later, if this is the case.)
If we can successfully run the app, a popup window will ask us, if we want to use a running device (which can be an emulator or a real device), or if we want to launch a new emulator. In case you do not have a device plugged in or an emulator running, the popup window should look like the one in figure 3.5.

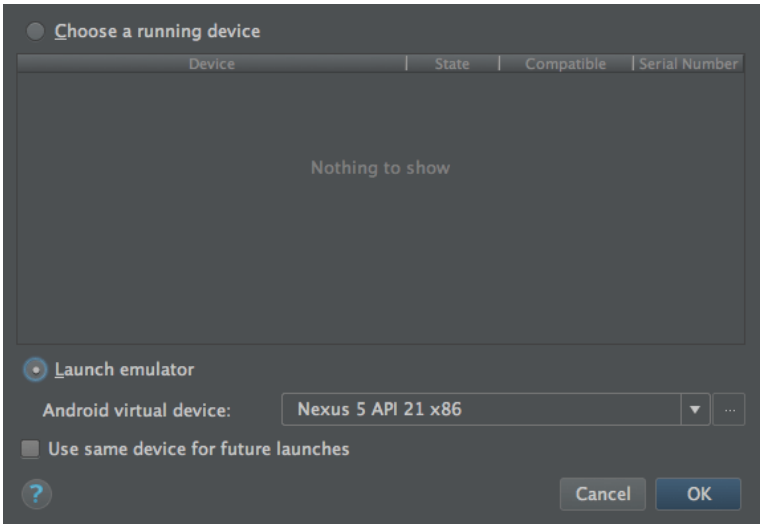


Figure 3.5.

Using the emulator, you do not require anything more than your computer, which is convenient in a lot of scenarios. However, even though emulation android applications has advanced a lot in the recent years in terms of speed and usability, you cannot fully test every aspect of your app using just the emulator. Running it on a real device is necessary at some point, if you want to develop a serious application.
For the HelloAndroidActivity, we will just go with the emulator. If you don't have an emulator running, click on 'Launch emulator'. We can also choose the emulated device, that we want to run it on, which can be helpful if we want to test different screen sizes. If you already have an emulator running, simply choose the running emulator. Launching the emulator might take a while, but when it is finished, it should automatically start our app and

look like the screen in figure 3.6.

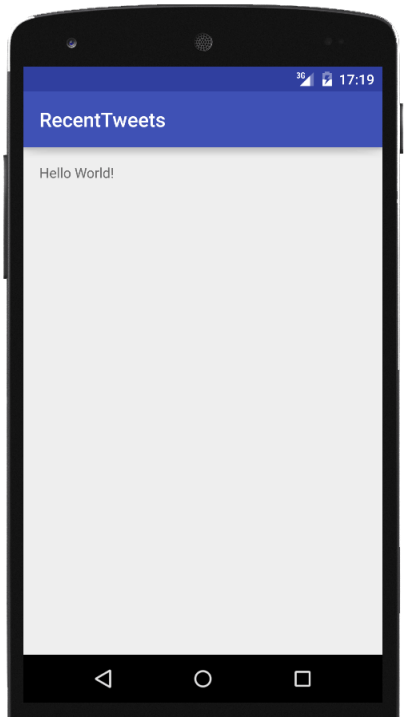


Figure 3.6.

3.3. From "Hello World!" to "Hello Android!"

As already mentioned, we want to fulfill one single goal, that is changing the TextView, that displays "Hello World!" to display "Hello Android!" in our activity, when we click a button. This sounds like a very simple task, and in fact, this is very easy, but it is important to learn how to get there. If you manage to do that you will learn some of the basic principles for programming the User Interface in Android.

3.3.1. Setting the content view

We want to add a button in the User Interface of our app. Remember, when we created our project, we created the activity HelloAndroidActivity. This activity can be found in the java-file called *HelloAndroidActivity.java*. This file will define the behavior of our activity, and almost anything, that we want our activity to do will be defined in that class in the form of java code. We also automatically generated a file called *activity_hello_android.xml*. As already mentioned, this file defines how our activity looks like. All UI-Elements will be

declared in there, so we use it to build our interface. Before we go into that, let's have a quick look at *HelloAndroidActivity.java*:

```
1  package com.example.recenttweets;

3  import android.support.v7.app.AppCompatActivity;
   import android.os.Bundle;
5
   public class HelloAndroidActivity extends AppCompatActivity {
7
       @Override
9      protected void onCreate(Bundle savedInstanceState) {
           super.onCreate(savedInstanceState);
11         setContentView(R.layout.activity_hello_android);
       }
13 }
```

When we look at the code listing, we can see that the *onCreate()*-method is overridden and in this method, there are two lines of code in this method. The first is just calling the super method *onCreate()* with the same arguments. That is necessary, and we even get an error from the IDE, if we do not do this. The second line sets the content view for the activity. In our case, the content view is set to the variable *R.layout.activity_hello_android* which resolves to an integer and links to *activity_hello_android.xml*, the file that we generated. It is set to our *Activity* and therefore is used to describe how our *Activity* looks like.

3.3.2. Layout Editor

Let's now have a look at our layout file. Click on *activity_hello_android.xml* in the directory res ->layout. The first thing you see will look something like figure 3.7

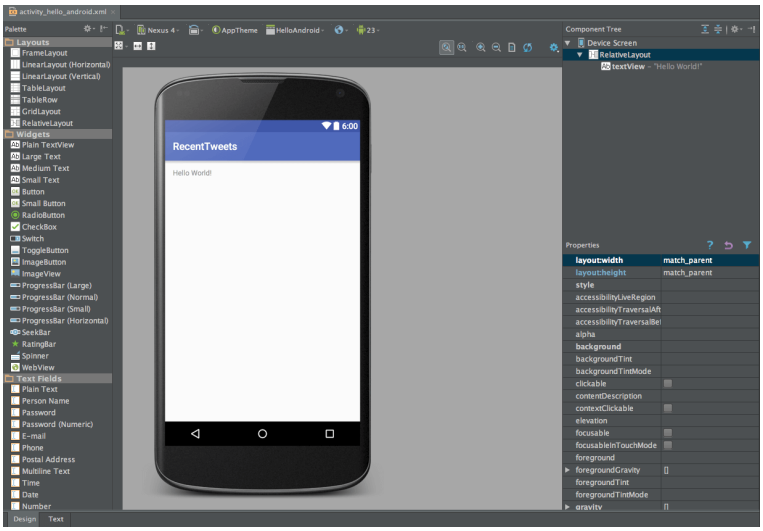


Figure 3.7.: The layout editor in Android Studio allows to design the UI via drag and drop.

This is the layout editor for Android Studio, a very powerful tool to design the User Inter-

face. It allows us to build the interface via dragging and dropping elements from the left list into our device screen. We also can inspect elements - views in this case - and set its attributes. We can furthermore simulate different devices, screen sizes, themes, translations and API Levels. Last but not least, in the bottom, we can change to the text representation of the XML-file and write our views directly using XML markups. We will come across this text representation later.

3.3.3. Views, ViewGroups and Layouts

In Android, to define how our activities look like, we use layout files. This is however just a simple explanation of what is actually going on. To show content, such a layout file needs to contain view as a root element. This can be any class, that extends the *View*-class, but android has views in place just for defining layouts. These views are actually called layouts and there are a number of different variations, such as *FrameLayout*, *LinearLayout*, *RelativeLayout* and *GridLayout* to name the most important ones. They are, in fact, all subclass of *ViewGroup*, which itself is a subclass of *View*. *ViewGroups* can have one or more views as children. Such children can be also be other *ViewGroups*. For example, a *RelativeLayout* could contain another *RelativeLayout*. Because of this, we can actually build hierarchies of *Views*, *ViewGroups* and *Layouts* that define our UI.
Our generated layout file *activity_hello_android.xml* already contains such a layout. When Android Studio generated this file, it automatically included a *RelativeLayout*. As already mentioned, you can inspect the xml by clicking on "Text" in the bottom of the layout editor, when you opened the xml file. Let's have a look at it:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.recenttweets.HelloAndroidActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:id="@+id/textView"/>

</RelativeLayout>
```

You can see see in the listing above, that the root element is a *RelativeLayout*. It has lots of attributes defined, that Android Studio also generated for us. The most important ones are *android:layout_width* and *android:layout_height*, as they are mandatory for each *View*. In this case, they are set to "match_parent", which means that the View should stretch to fit entirely

within the parent view. Instead of "match_parent", we could also choose "wrap_content", which simply put means "Be as large as you need to be". Furthermore, we could plug in concrete values, such as "100px" or "100dp", whereas dp stands for density pixels. For a root view, most of the time you want to keep it to "match_parent".
You can also see in the listing, that the *RelativeLayout* contains a *TextView* that says "Hello World". This was also generated by Android Studio and it is this very *TextView* of which we want to change the text to "Hello Android!". We could just write it into the xml definition of the *TextView* for the attribute *android:text* and we would be done, but we want to do it using a button, which means we have to programmatically change it.

3.3.4. Placing and labeling the Button

Now that we know more about *Views*, *ViewGroups* and *Layouts*, we can finally start using the layout editor and place our button into the layout file. For this, we just select the view "Button", that is in "Widgets" in the left list, and drag it onto our device screen. While dragging it to the device screen, Android Studio gives us hints about the placement and shows us several measurement values, that might be interesting for us. We want the button to be exactly under the TextView, just like in figure 3.1 in the beginning of this chapter. We just drag the button, so it is placed accordingly. It should look like in screenshot 3.8

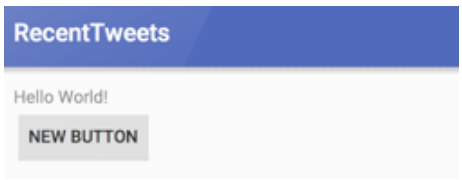


Figure 3.8.

To change the text on the button itself in the layout editor, we need select the button and change the attribute "text". We can do so on the right side of the layout editor. The attribute is the one shown in figure 3.9

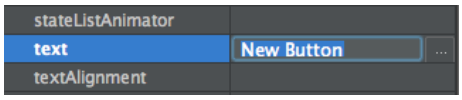


Figure 3.9.

However, it is not best practice to just enter the string, that we want to display. It is better to define the string in the *strings.xml* file, and then just put a reference into the attribute field. This way, if we ever want to translate our app into another language (the terminus for this is internationalization, or i18n), we have easy game. Android allows us to create string-files for each language, that we want to support.
To do this, we open the file *string.xml*. We already have this file generated from Android

Studio and have one string tag in there, that describes our application name. Our *strings.xml* file looks like this:

```
<resources>
    <string name="app_name">RecentTweets</string>
</resources>
```

We put our new string in xml form into the resources tag, just under the tag for the *app_name*. As a name, we can choose for example "change_button_label". The name will be then used as a reference. Our *string.xml* file should now look like this:

```
<resources>
    <string name="app_name">RecentTweets</string>
     <string name="change_button_label">Change TextView</string>
</resources>
```

Back in the layout editor for *activity_hello_android.xml*, we select the button again, and we enter *@string/change_button_label* for the attribute "text". This is how we reference strings. See figure 3.10.

Figure 3.10.

Your layout should now look like in figure 3.11.

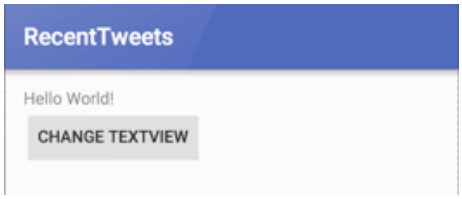


Figure 3.11.

3.3.5. Listening to the Button Click Event

Of course, we want our newly created button to do something, when it is clicked. After all, it's the purpose of a button. We could just use the attribute *onClick* of the button, and set it to a method in our Activity that we want to call. But we will do it manually. We want to set what is called a listener on the button. This listener is an interface, and we add it as an instance of an anonymous class.
To teach our button the desired behavior, we need to get a reference to the button in *HelloAndroidActivity.java*. To get this reference, the button needs to have an id. So the first thing we want to do is to set the id of our button to the id we want it to have. Let's set the id to "change_textview_button". It should look just like in figure 3.12.

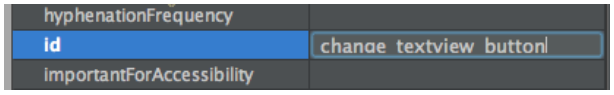


Figure 3.12.

Now that our button has an id, we can use this id to get a reference to the button object in *HelloAndroidActivity.java*. Android provides the method *findViewById(int id)* for this purpose. It is a method of the class activity and we can just call it anywhere in our *HelloAndroidActivity* because it is a subclass of *Activity*. For us, the ideal place to do this is in the *onCreate()*-hook, directly after we set the content view. We need to do this after setting the content view, because otherwise, Android would not find the button as it does not know where to look. We want it to look for the button in *activity_hello_android.xml*.
We extend the *onCreate* method as followed:

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

    // The button that we created in the layout editor
    Button changeTextViewButton = (Button) findViewById(R.id.change_textview_button);

    // setting the listener for click events
    changeTextViewButton.setOnClickListener(new View.OnClickListener(){
        @Override
        public void onClick(View view) {
            // do something
        }
    });
}
```

The method *findViewById(int id)* returns a view, therefore we have to cast the returning object into a Button. Then we set a click listener on the Button, using the method *setOnClickListener()* that implements the method *onClick(View view)*. In this method, we can then define the behavior of our button, when it is clicked. Right now, as you can see in the listing, there is just a comment in there.
Let's now create a method, that we put in *HelloAndroidActivity.java*, that changes the text of our *TextView* to a arbitrary *String*. Before we do that, we need to go to the layout editor and give our textview an id. Let's simply call it *my_textview*. Then, in our method - we can could for example call it *changeTextViewToString(String newString)* - we get a reference to the *TextView* object and change our text. When implemented, it would look something like this:

```
private void changeTextViewToString(String newString){
   TextView myTextView = (TextView) findViewById(R.id.my_textview);
   myTextView.setText(newString);
}
```

You can see, that the textView object simply offers a method *setText* that takes a *CharSequence* as a parameter. We can pass our variable *newString*, since *String* implements this

interface.

Now, we only need to call this method in the *onClick* method of our click listener.

```
changeTextViewButton.setOnClickListener(new View.OnClickListener(){
    @Override
    public void onClick(View view) {
      changeTextViewToString("Hello Android!");
    }
});
```

Now it is time to run the application and verify that it works. If everything is done correctly, it should change the textview to "Hello Android!". If so, you have made your first working application for android! It doesn't do much, but you now know some of the fundamental mechanics of Android.

4. Twitter Fabric

4.1. What is Twitter Fabric?

Twitter Fabric is Twitters own mobile platform that bundles several SDKs for different purposes, such as twitter integration, advertisement integration as well as crash tracking and analytics libraries. It was introduced in 2014 and is available for Android, iOS, OS X and Web Applications. [7] [8]

4.2. Implementing Twitter Login

4.2.1. Create LoginActivity

The first thing we need to do is to create the LoginActivity, which will include the Login Button for Twitter. It will be the first screen the user will see. To create the activity, simply right-click on the package, where you want to create your activity - in our case it's *com.example.recenttweets* - and choose New ->Java File. Enter the name of the activity in the dialog field and click on "OK". A new java file is created named *LoginActivity.java*. Furthermore, we should create an xml file in *res/layout* that is called *activity_login.xml*. You can do this by right clicking on the layout directory and click on New ->Layout resource file. In the dialog, that asks you for the name, enter the name and change the Root element from *LinearLayout* to *RelativeLayout*.
In order to actually make it an activity, we need to inherit from the Activity class. Furthermore, we want to set the content view to the layout, so we need to override *onCreate* to do this. The class should look like this:

```
public class LoginActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);
    }
}
```

We want the *LoginActivity* to be the launcher activity, therefore we need to change the Activity in the *AndroidManifest.xml*. Simply change the *android:name* attribute of the activity tag from *.HelloAndroidActivity* to *.LoginActivity*. Notice that the dot in the beginning is important. It should then look like this:

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

```
<manifest package="com.example.recenttweets"
          xmlns:android="http://schemas.android.com/apk/res/android">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".LoginActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>

                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

We can launch the App to verify, that everything works. The screen should be just white, as *activity_login.xml* is empty.

4.2.2. Including Twitter Fabric

As you might remember, in section 2 we discovered how to define dependencies for Android. Twitter Fabric would be such a dependency and if we want to do it manually, we would extend our gradle files and include the dependency on Fabric there. But Twitter made it even more simple and foolproof to include their SDK: They developed a simple plugin for Android Studio, that you just download and install, and that does pretty much the whole setup for you. The downside of this is, that you have to have a developers account for fabric, but this is needed anyways to be able to integrate Twitter.
In order to create an account for Fabric, visit `https://fabric.io/sign_up`. Fabric does a very good job in explaining every step for you, but I will explain it here as well. After Signup, you will receive an e-mail with a confirmation link in it. When you activation was successful, they will ask you for a team name. Choose a team name that you wish. After that, we want to install the plugin in Android Studio. For this, go to *Preferences ->Plugins* and click on Browse Repositories. Enter 'Fabric' in the search bar to find the plugin for Android Studio, then select it and click on 'Install Plugin'. See figures 4.1 and 4.2 for how this looks in Android Studio.
After installation, you will need to restart Android Studio to actually load the plugin.

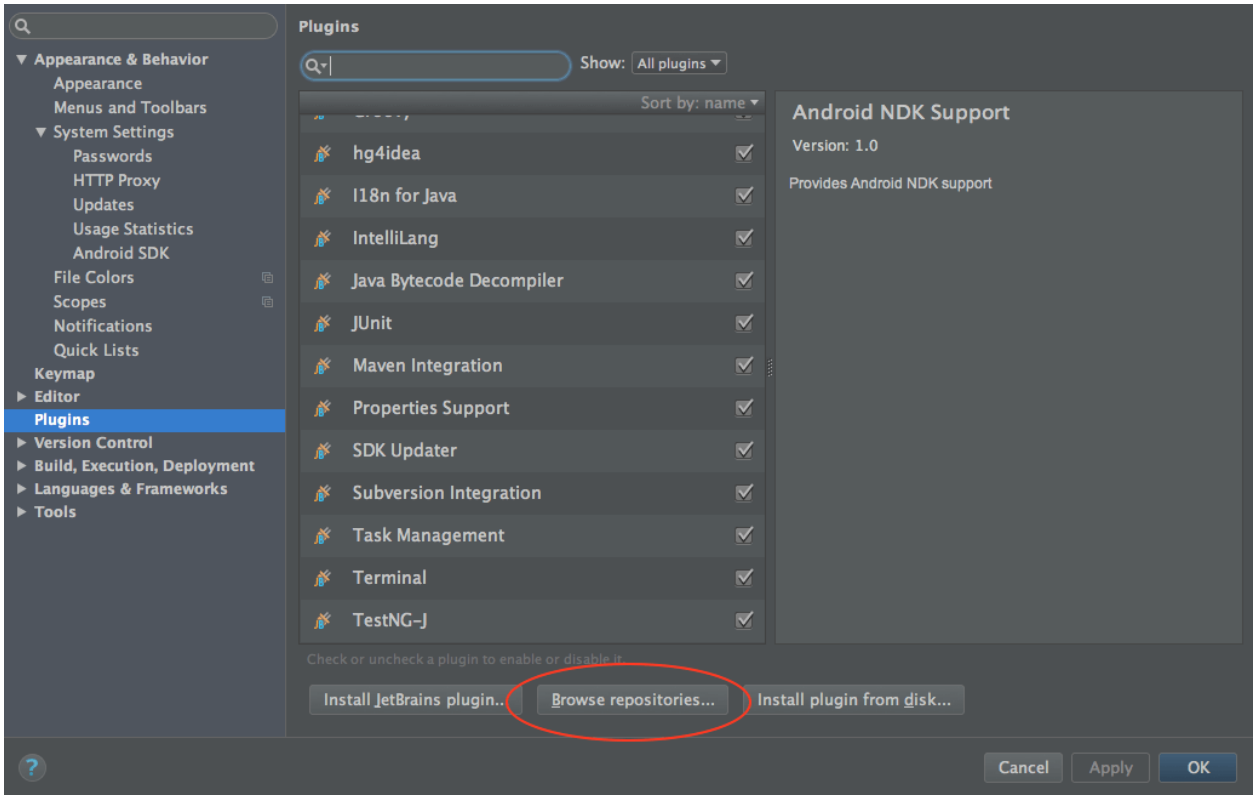


Figure 4.1.

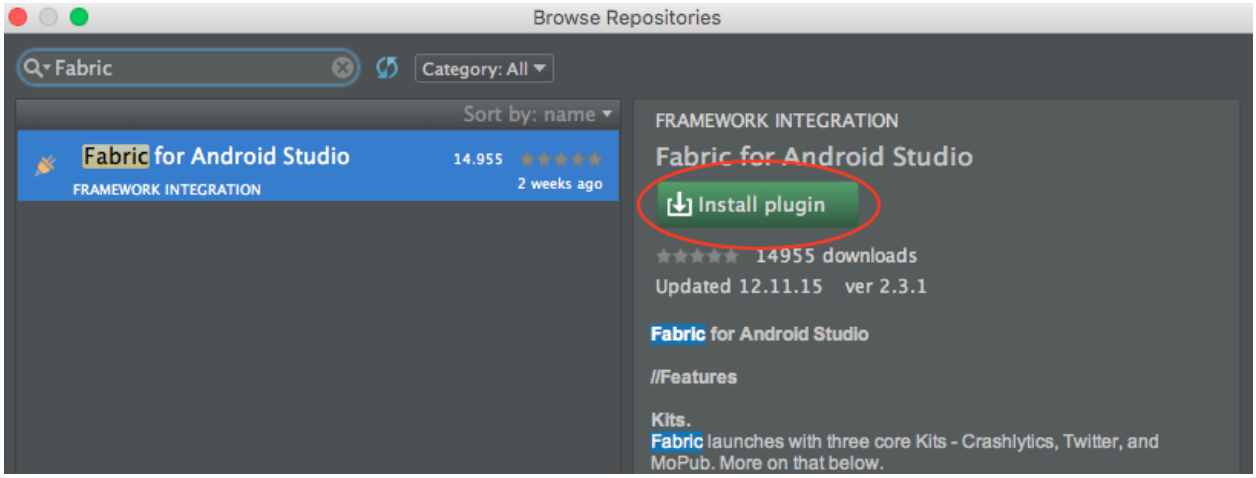


Figure 4.2.

After restarting Android Studio, you can see the Fabric logo at the right sidebar. Clicking on it will reveal a UI for the Fabric Plugin, where you need to click on the power sign. Log in with your fabric account and after success, you should see the team name that you entered in the very beginning., as shown in 4.3

Figure 4.3.

Click on 'Next' and you will come to the list of available integrations, that fabric provides. Click on 'Twitter' and install the Twitter-Kit by clicking on 'Install'. Follow the wizard and either create a new Twitter account or choose an existing one. If you need to create a new account, Twitter Fabric makes it very easy for you and automatically creates an account with your email-address for you. If you have an existing Twitter Account, you will have to fill in a Twitter Key and a Twitter Secret, that you can obtain from your account. You can do this by visiting `https://apps.twitter.com` and creating a new application.

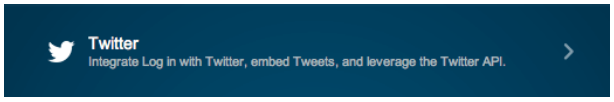


Figure 4.4.

Next, we need to add Twitter to our project. As you can see in 4.5, Twitter Fabric tells us to change the very *build.gradle* file, that we discovered in Chapter 2. Remember, it is the one in the module and not the top-level one.

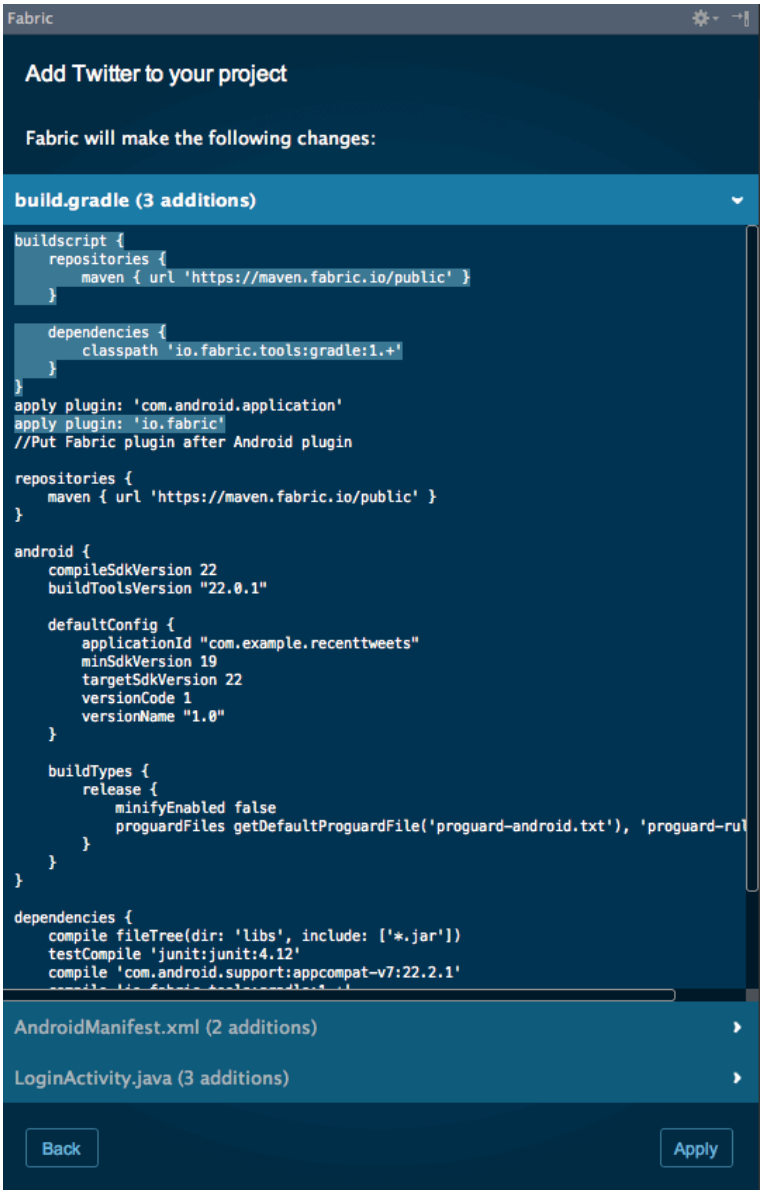


Figure 4.5.

Furthermore, it wants us to change *AndroidManifest.xml* and our Activity, that we created - *LoginActivity* - as well. Fabric shows us, how we should alter *build.gradle*, *AndroidManifest.xml* and *LoginActivity.java*. The good thing is, if we have done everything correctly when we created the LoginActivity, Twitter Fabric is actually able to apply these modifications for us. So just click Apply and everything should be done for you. If not, you need to make the modifications yourself. You can also try to create a fresh project with **??**and include Twitter again. Either way, be sure that the changes are in fact applied to your code. After including Twitter Kit into our project, you might want to launch our app and verify that Twitter is correctly configured and we don't have any problems.
Now it is time to finally add Twitter Login to our app. Twitter Fabric makes this a breeze

as well. Open Fabric again and navigate to Twitter Kit. Under "Code Examples", you can find "Log in with Twitter". You can simply follow the guide by Fabric, but I will explain every step here as well.
First, you should modify *activity_login.xml*, making it look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
                android:layout_width="match_parent"
                android:layout_height="match_parent">

    <com.twitter.sdk.android.core.identity.TwitterLoginButton
        android:id="@+id/twitter_login_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"/>

</RelativeLayout>
```

Second, we need to add a reference to the button that we just inserted in the layout, as well as adding a callback (which is similar to our listener from Chapter 3.3) We modify *LoginActivity.java* to look like this:

```
  // Note: Your consumer key and secret should be obfuscated in your source code before shipping.
    private static final String TWITTER_KEY = "xxxxx";
    private static final String TWITTER_SECRET = "xxxxx";

    private TwitterLoginButton loginButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TwitterAuthConfig authConfig = new TwitterAuthConfig(TWITTER_KEY, TWITTER_SECRET);
        Fabric.with(this, new Twitter(authConfig));
        setContentView(R.layout.activity_login);

        loginButton = (TwitterLoginButton) findViewById(R.id.twitter_login_button);
        loginButton.setCallback(new Callback<TwitterSession>() {
            @Override
            public void success(Result<TwitterSession> result) {
                // The TwitterSession is also available through:
                // Twitter.getInstance().core.getSessionManager().getActiveSession()
                TwitterSession session = result.data;
                // TODO: Remove toast and use the TwitterSession's userID
                // with your app's user model
                String msg = "@" + session.getUserName() + " logged in! (#" + session.getUserId()
                        + ")";
                Toast.makeText(getApplicationContext(), msg, Toast.LENGTH_LONG).show();
            }

            @Override
            public void failure(TwitterException exception) {
                Log.d("TwitterKit", "Login with Twitter failure", exception);
            }
        });

    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
```

```
38          // Make sure that the loginButton hears the result from any
            // Activity that it triggered.
40          loginButton.onActivityResult(requestCode, resultCode, data);
        }
```

Note that the values for the variables *TWITTER_KEY* and *TWITTER_SECRET* are just placeholders. Fabric automatically included your key and your secret, so do not change these.
Don't worry too much about what's going on there under the hood. Twitter basically does everything for you and you do not need to code the communication with Twitter yourself.
It is time to launch the app again. You should see the Twitter Login button in the center of the app.

4.3. Displaying Recent Tweets with Twitter Kit

We now come to the point, where we want to display actual data from Twitter. Thankfully, Twitter Fabric makes this a breeze as well, even though it requires a bit more than just copy and pasting code from examples.

4.3.1. Starting an Activity

The first thing we want to do is we want to create an Activity for displaying our tweets. We could just create a new Activity, but remember that we still have the *HelloAndroidActivity*, that is currently unused. We can just reuse this activity and change its name. Android Studio (which is built upon the IntelliJ platform from JetBrains) has a very nice feature called "Refactor". Just right-click on *HelloAndroidActivity.java*, select Refactor ->Rename and enter *RecentTweetsActivity*. Android Studio will rename the file and all its references correctly. You need to do the same for *activity_hello_android.xml*. Rename it to *activity_recent_tweets.xml*. Android Studio will automatically update the reference to the xml file in your java file.
After that, we need to go back to *LoginActivity*. After all, we want start the *RecentTweetsActivity* when our login was successful, so we need to change *LoginActivity*.
First, let's create a private method called *startRecentTweetsActivity()* of type void. We will later call this method at the right place to start the *RecentTweetsActivity*, but it is always good to write single-purpose methods. To start an *Activity*, we need to create a new *Intent* object. What is an *Intent*? "An intent is an abstract description of an operation to be performed" says Google in their documentation. [9] We use it to start our activity, but it could also be used to start services for example.
The constructor of *Intent* takes a *Context* instance and a *Class* instanceas parameters. Since *Activity* is a subclass of *Context*, we can just pass *this* as a reference to the current instance. The *Class* instance defines the *Activity*, that we want to start. To get this, we

can call class on the *Activity* that we want to start, which is *RecentTweetsActivity*. So we pass *RecentTweetsActivity.class* as a second parameter. Then we need to just start the *Activity* by using the method *startActivity(Intent intent)* from *Activity*. The method *startRecentTweetsActivity()* should look like this:

```
private void startRecentTweetsActivity(){
  Intent intent = new Intent(this, RecentTweetsActivity.class);
  startActivity(intent);
}
```

Now we need to call this method. Remember the *Callback*, that we set on the *loginButton* object in the *onCreate* hook? This is exactly, where we need to call *startRecentTweetsActivity()* from. Change the *Callback* to the following:

```
 loginButton.setCallback(new Callback<TwitterSession>() {
  @Override
  public void success(Result<TwitterSession> result) {
    startRecentTweetsActivity();
  }

  @Override
  public void failure(TwitterException exception) {
    Log.d("TwitterKit", "Login with Twitter failure", exception);
  }

});
```

Start your app, login and verify that it works... Except that it doesn't! You will see something like in figure 4.6.

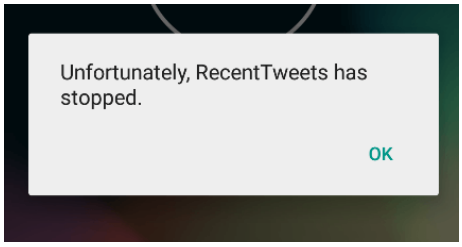


Figure 4.6.

Don't worry! This can and will happen at some point, and thanks to the logger, that is built in in Android Studio, we will soon find out why, so we can fix it.
Click on "Android Monitor" in the bottom bar of Android Studio and change the Log level to "verbose" to see the log output. You should see something like this somewhere in your output:

```
12-09 14:42:50.994 1952-1952/com.example.recenttweets E/AndroidRuntime: FATAL EXCEPTION: main
Process: com.example.recenttweets, PID: 1952
 java.lang.RuntimeException: Failure delivering result ResultInfo{who=null, request=140, result
     =-1, data=Intent { (has extras) }} to activity {com.example.recenttweets/com.example.
     recenttweets.LoginActivity}: android.content.ActivityNotFoundException: Unable to find
     explicit activity class {com.example.recenttweets/com.example.recenttweets.
     RecentTweetsActivity}; have you declared this activity in your AndroidManifest.xml?
 at android.app.ActivityThread.deliverResults(ActivityThread.java:3539)
```

```
    at android.app.ActivityThread.handleSendResult(ActivityThread.java:3582)
6   at android.app.ActivityThread.access1300(ActivityThread.java:144)
    at android.app.ActivityThreadH.handleMessage(ActivityThread.java:1327)
8   at android.os.Handler.dispatchMessage(Handler.java:102)
    at android.os.Looper.loop(Looper.java:135)
10  at android.app.ActivityThread.main(ActivityThread.java:5221)
    at java.lang.reflect.Method.invoke(Native Method)
12  at java.lang.reflect.Method.invoke(Method.java:372)
    at com.android.internal.os.ZygoteInitMethodAndArgsCaller.run(ZygoteInit.java:899)
14  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:694)
    ...
```

Event though it looks very confusing, the exception output usually tells us, what's going on. This one does too: *android.content.ActivityNotFoundException: Unable to find explicit activity class com.example.recenttweets/com.example.recenttweets.RecentTweetsActivity; have you declared this activity in your AndroidManifest.xml?*
The exception was thrown because we didn't declare our Activity in *AndroidManifest.xml*! Go to your *AndroidManifest.xml* and add the following within the *application* tag.

```
1  </activity>
     <activity android:name=".RecentTweetsActivity">
3   </activity>
```

After you added this to *AndroidManifest.xml*, try running the application again. Login with your Twitter, and if successful, you should see the very screen that we saw in chapter 3.3, when we implemented our first button.

4.3.2. Including a ListView

We want to load the last 20 Tweets of the signed in user, according to our specifications. For this, we want to add two methods in *RecentTweetsActivity*: *loadTweets(int count)*, which takes an integer for how many tweets should be loaded and *showTweets(List <Tweet >tweets)*, which takes a list of tweets that should be displayed. In both methods, we will make heavy use of the Twitter Kit from Fabric, which does a lot of the needed work for us. Before we implement those methods, we need to change our layout file to contain a *ListView*. A *ListView* is a view provided by android, that is - as the name suggests - made for displaying lists. Open *activity_recent_tweets.xml* and remove the *TextView* and the *Button*. Then drag and drop a *ListView* into the *RelativeLayout*. You should find it under "Containers" in the left list of views in the layout editor. Drag it so it is placed centered horizontally and vertically. Change the attributes *layout:width* and *layout:height* both to "match_parent". Your device screen should look like figure 4.7. You can further remove all the padding of *RelativeLayout*, if it has some.

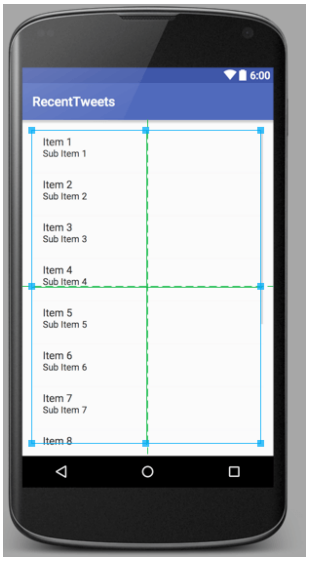


Figure 4.7.

Just like with the *Button* from chapter 3.3, we need a reference to this *ListView* object in *RecentTweetsActivity.java*. To do this, we need to set an id for the *ListView*. Set the id of the *ListView* to *recent_tweets_listview*.
Now back to *RecentTweetsActivity.java*, change your class to look like this:

```
public class RecentTweetsActivity extends Activity {

    private ListView tweetsListView;

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

        tweetsListView = (ListView) findViewById(R.id.recent_tweets_listview);
    }
}
```

Similarly to how we did it in chapter 3.3, we use *findViewById* to find the *ListView* in our layout. The only difference is that we save the reference to the *ListView* object in an instance variable in our object to use it later in our method *showTweets(List <Tweet >tweets)*.

4.3.3. Loading the Tweets

As already mentioned, we want to load the tweets that a logged in user would normally see at his home timeline on Twitter. We want to get the last 20 of them. Let's now implement a method, that takes an integer as parameter that defines how many tweets we want to load, and that loads the tweets. This is how the method looks like:

```
private void loadTweets(int count){
    TwitterApiClient twitterApiClient = TwitterCore.getInstance().getApiClient();

    StatusesService statusesService = twitterApiClient.getStatusesService();
```

```
6       statusesService.homeTimeline(count, null, null, false, true, true, true, new Callback<List<
            Tweet>>() {
            @Override
8           public void success(Result<List<Tweet>> result) {
                showTweets(result.data);
10          }

12          @Override
            public void failure(TwitterException e) {
14
            }
16      });
    }
```

This is what we do in this method: First, we get an instance of the *TwitterApiClient*. This can be done by calling the following: *TwitterCore.getInstance().getApiClient()*. We save it into our own variable of class *TwitterApiClient*. Next, on this very *TwitterApiClient*instance, we can call *getStatusesService()* to get an instance of *StatusesService*. This is a specific service class from Twitter Kit, that takes care of several tasks, such as retweeting a tweet, showing a tweet, loading the user timeline, loading the home timeline and much more. The difference between the user timeline and the home timeline is, that in the user timeline, we get all tweets, retweets and mentions from a certain user, just like when you would visit the page of a Twitter user. The home timeline is - as already mentioned - what would normally show up on the homepage of Twitter when you are logged in. And the latter is exactly what we need. Therefore, we call the specific method *homeTimeline* on *statusesService* with several parameter. The first parameter is *Integer count* and this defines, how many tweets of the home timeline should be loaded. Here, we just pass on the *count* variable that our own method has as a parameter. The second and the third parameter are two variables of type *Long*, that are rather unimportant for us: *sinceId* and *maxId*. The next parameter is *Boolean trimUser*, we set it to false because we want to get the full user object. The next parameter is *Boolean excludeReplies*, which we also set to true, because we do not want to display too many tweets. *Boolean contributeDetails* is the next parameter and we set it also to true. For more information on *StatusesService*, please to the docs. [10]. The last parameter is a *Callback*, that returns a *Result* $<$*List* $<$*Tweet* $>>$ object as a result, when the request was successful. Only when the request was successful, we take the result and extract the data (which is of type *List* $<$*Tweet* $>$) and pass it on to our method *showTweets*, that we yet have to write. We can also specify what to do, when there was a failure, but this is out of the scope of this work.

4.3.4. Displaying the Tweets

In the last subsection, we wrote the method for loading tweets. On a successful request, we called with *showTweets(result.data)* the method, that we now want to write, passing the loaded tweets that we want to display. For displaying the tweets in our *ListView*, we can write the following method in *RecentTweetsActivity.java*

```
private void showTweets(List<Tweet> tweets){
    FixedTweetTimeline timeline = new FixedTweetTimeline.Builder().setTweets(tweets).build();

    TweetTimelineListAdapter adapter = new TweetTimelineListAdapter.Builder(this)
            .setTimeline(timeline)
            .build();

    tweetsListView.setAdapter(adapter);
}
```

First, we build an object of class *FixedTweetTimeline* with its builder, and we set the tweets, that got passed as a parameter. *FixedTweetTimeline* is a class of Fabric's Twitter Kit, that can hold a fixed set of tweets, as the name suggests. Next, what we need to have to populate our *ListView*, that we included earlier, is an instance of the interface *ListAdapter*. An adapter is basically the object, that decides what data is displayed how and at what position in our *ListView*. It is basically the "glue" between the data and the *ListView*. Normally, we would write the adapter our selves and implement *ListAdapter* or extend on an existing adapter from Android, but the Twitter Kit from Fabric provides even that for us. The big advantage: We do not need to design the tweets ourselves, this is done for us. This comes in very handy, as twitter has several requirements on how to display their tweets, and they are quite strict about it. For more information, see reference[11]. With the Twitter Kit, we do not have to worry about this, as it is done for us by Twitter itself. It is implemented in the *TweetTimelineListAdapter*, and we just have to create an instance of it using the builder. We set the timeline to the instance of *FixedTweetTimeline*, that we created earlier and we build the object. Then, we just set the adapter for our *tweetsListView* to this adapter we just created, and the list should be populated with our Tweets! Now we only ned to do one thing: call *loadTweets(20)* from the *onCreate* hook. *RecentTweetsActivity.java* should now look like this:

```
public class RecentTweetsActivity extends Activity {

    private ListView tweetsListView = (ListView) findViewById(R.id.tweets_listview);

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_recent_tweets);
        tweetsListView = (ListView) findViewById(R.id.tweets_listview);
        loadTweets(20);
    }


    private void loadTweets(int limit){
        TwitterApiClient twitterApiClient = TwitterCore.getInstance().getApiClient();
        StatusesService statusesService = twitterApiClient.getStatusesService();
        statusesService.homeTimeline(limit, null, null, false, true, true, true, new Callback<
            List<Tweet>>() {
            @Override
            public void success(Result<List<Tweet>> result) {
                showTweets(result.data);
            }

            @Override
```

```
            public void failure(TwitterException e) {

            }
        });
    }

    private void showTweets(List<Tweet> tweets){
        FixedTweetTimeline timeline = new FixedTweetTimeline.Builder().setTweets(tweets).build();

        final TweetTimelineListAdapter adapter = new TweetTimelineListAdapter.Builder(this)
                .setTimeline(timeline)
                .build();

        tweetsListView.setAdapter(adapter);
    }
}
```

Compile your app and verifiy, that indeed our code does the right thing. After logging into Twitter, it should look something like in figure 4.8.

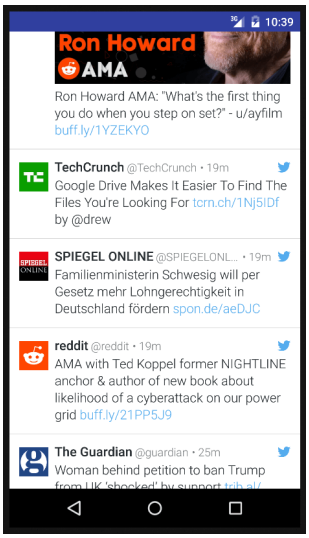


Figure 4.8.

You might wonder, how Twitter Kit knows, for which user it should load the tweets. After all, we did never explicitly set a user object. This is the beauty of Fabrics Twitter Kit: It handles even that for us. When you log in, Twitter Kit stores a session for our user. This session is then used anytime, when we call the API. So we do not need to worry about authentication or passing tokens, that we would normally have to do.

5. Developing for Samsung Gear

In this chapter, I want to briefly outline the possibility of developing an application for the Samsung Gear 2 Smartwatch in connection with an Android Application. I will give a brief overview with a quick start guide, but also provide the interested reader with the resources on how to dig deeper. Furthermore, I'll demonstrate how to quickly get started with the Samsung Accessory SDK and how to create a simple Hello-World-Application with a connection to an android host. For installing the Tizen SDK, please refer to Installation Guides. We will use the Sample that comes with the Accessory SDK. You can find the original code from Samsung in the unzipped SDK folders under Samples ->Accessory ->Samples(Web) ->HelloAccessory. In this app, our goal is to request the current time of the android device and transmit to and show it on the Samsung Gear 2. Figure 5.1 shows the Samsung Gear 2 with Tizen running on it.

Figure 5.1.: Image found at [12]

5.1. Tizen

The Samsung Gear 2 Smartwatch uses Tizen as its Operating System. Tizen is an open-source linux-based OS developed by Samsung, Intel and the Linux-Foundation and has a long history of predecessors, with relation to the operating systems MeeGo and SLP . It was introduced on the Samsung Gear in 2014, taking over from Android.
For developing applications on the Tizen-Platform, you can either use HTML5 and JavaScript

or - since version 2.0 - write native applications in C++. [13].
To connect the Samsung Gear 2 to Android, we can use Accessory SDK, a special SDK for android from Samsung. Using this SDK, we can setup our Android App either as a consumer or a provider, where the Samsung Gear 2 can be the respective counterpart, that is provider or consumer.

5.2. Hello Accessory

As already mentioned, this application is heavily inspired by the Samsung Sample Application "HelloAccessory" that comes with the Accessory SDK. [14] For the android application, I also put the respective code in Code Listings

5.2.1. Provider - Android

To setup communication between the Android application and the Samsung Gear device, you need the Samsung Accessory SDK. It enables communication between smart devices - like your android smartphone - and several accessory devices, such as a smartwatches, printing devices, car head units etc. You can find it under `http://developer.samsung.com/galaxy#accessory`.
To get started with Android, download the SDK, unzip the file and place the two *.jar*-files from the folder *libs* folder into your modules *libs* folder. Right click on the jar files and click on "Add as Library...".
Next, we need to create *ProviderService*. Create a new java class, name it *ProviderService* that inherits from *SAAgent*, which is a class from the Accessory SDK. In *ProviderService*, you can include a local private class *ServiceConnection*, that extends *SASocket*. It has the hook *onReceive*:

```
public class ServiceConnection extends SASocket {
    ...
        @Override
        public void onReceive(int i, byte[] bytes) {
            if (connection == null) {
                return;
            }
            Calendar calendar = new GregorianCalendar();
            SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy.MM.dd aa hh:mm:ss.SSS");
            String timeStr = " " + dateFormat.format(calendar.getTime());
            String strToUpdateUI = new String(bytes);
            final String message = strToUpdateUI.concat(timeStr);

            new Thread(new Runnable() {
                public void run() {
                    try {
                        connection.send(HELLOACCESSORY_CHANNEL_ID, message.getBytes());
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }).start();
```

```
        }

        ...
    }
```

As you can see in the listing, in *onReceive* we format and create our time string and send it via the connection to our device. The *onReceive* method is called, when the android host receives a message from the Gear.
Next, you will need to create a *xml*-file in the directory */res/xml* and add the following in the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<resources>
    <application name="RecentTweets">
        <serviceProfile
        role="provider"
        name="recenttweets"
        id="/app/recenttweets"
        version="1.0"
        serviceImpl="com.example.recenttweets.ProviderService"
        serviceLimit="any"
        serviceTimeout="10">
            <supportedTransports>
             <transport type="TRANSPORT_BT"/>
              <transport type="TRANSPORT_WIFI"/>
            </supportedTransports>
            <serviceChannel
            id="104"
            dataRate="low"
            priority="high"
            reliability="enable"/>
    </serviceProfile>
    </application>
</resources>
```

This is our service profile for our provider app, as the value in field *role* might suggest. Furthermore, in *serviceImpl* you need to declare the provider service class *ProviderService*. Besides these two fields, the service profile on the Tizen Web Application look pretty much the same. It is important that *name*, *id*, and the fields of *serviceChannel* match with the service profile on the Tizen Web Application. The field *role* should however be set to *consumer*.
At last, we need to add many lines to our *AndroidManifest.xml*. First, add the service reference to the *application* tag

```
    <service android:name="com.example.recenttweets.ProviderService" />
```

Next, add meta-data with a reference to your *accessoryservices.xml* and the type of Gear app you are writing, also within the *application* tag:

```
    <meta-data
        android:name="AccessoryServicesLocation"
        android:value="/res/xml/accessoryservices.xml" />
    <meta-data
        android:name="GearAppType"
        android:value="wgt" />
```

Furthermore, we add two receivers two our manifest file, that filter after two actions, that come with the Samsung Accessory SDK. These are very important. If you happen to have a typo in those, it won't work but it also won't tell you why it won't work. These receivers should be also added within the *application* tag:

```
<receiver android:name="com.samsung.android.sdk.accessory.RegisterUponInstallReceiver" >
        <intent-filter>
            <action android:name="com.samsung.accessory.action.REGISTER_AGENT" />
        </intent-filter>
    </receiver>
    <receiver android:name="com.samsung.android.sdk.accessory.
        ServiceConnectionIndicationBroadcastReceiver" >
        <intent-filter>
            <action android:name="com.samsung.accessory.action.SERVICE_CONNECTION_REQUESTED"
                />
        </intent-filter>
    </receiver>
```

At last, add these permissions to the *manifest* tag:

```
<uses-permission
    android:name="com.samsung.android.providers.context.permission.
        WRITE_USE_APP_FEATURE_SURVEY"/>
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="com.samsung.accessory.permission.ACCESSORY_FRAMEWORK" />
<uses-permission android:name="com.samsung.WATCH_APP_TYPE.Companion" />
<uses-permission android:name="com.samsung.wmanager.ENABLE_NOTIFICATION" />
```

That's it from the provider side on android. Next, we will setup the consumer application.

5.3. Consumer - Tizen

Open the Tizen IDE, it should look like in figure A.5.

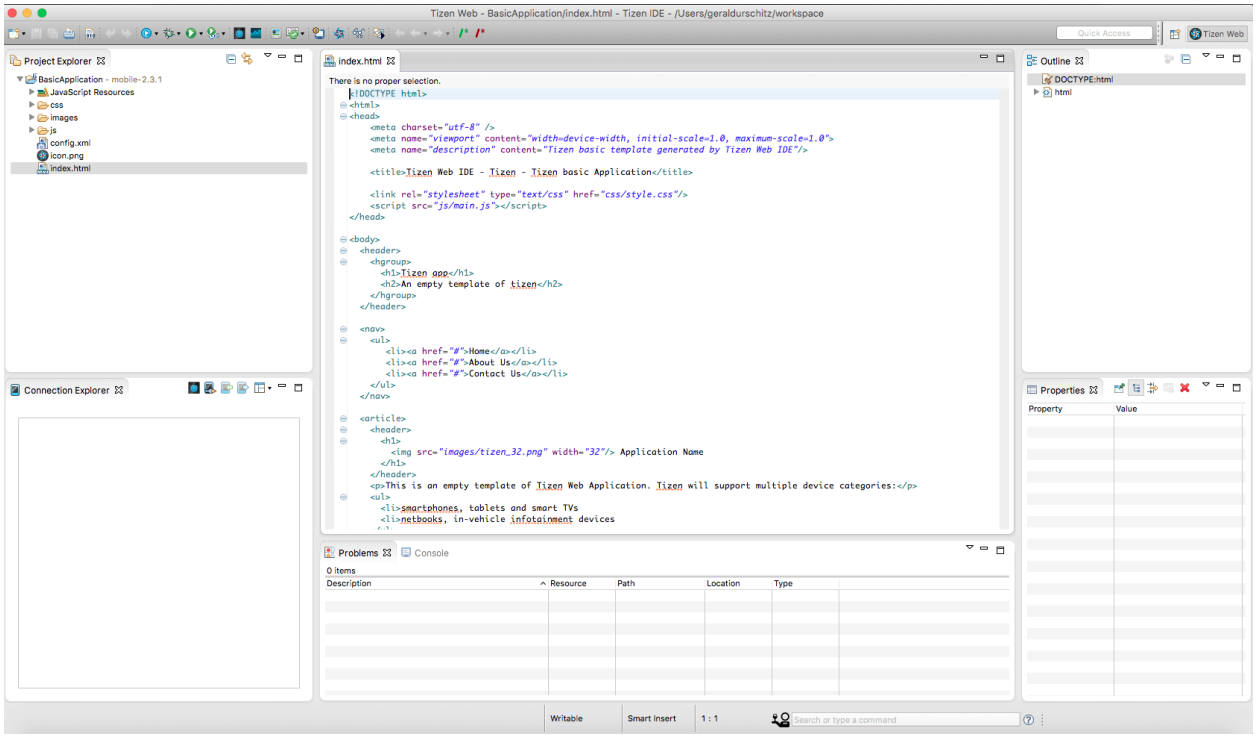


Figure 5.2.: A screenshot of the Tizen IDE for developing Tizen Applications.

As already mentioned, in Tizen, you can either write native applications in C++ or you can write your application using HTML5 and JavaScript. In this paper, we will focus on the latter. To create a new *Tizen Web Project*, you can simply go to File ->New ->Tizen Web Project, then choose the template "Basic" under the Template ->WEARABLE-2.3.1 and give your project a suitable name. See 5.3

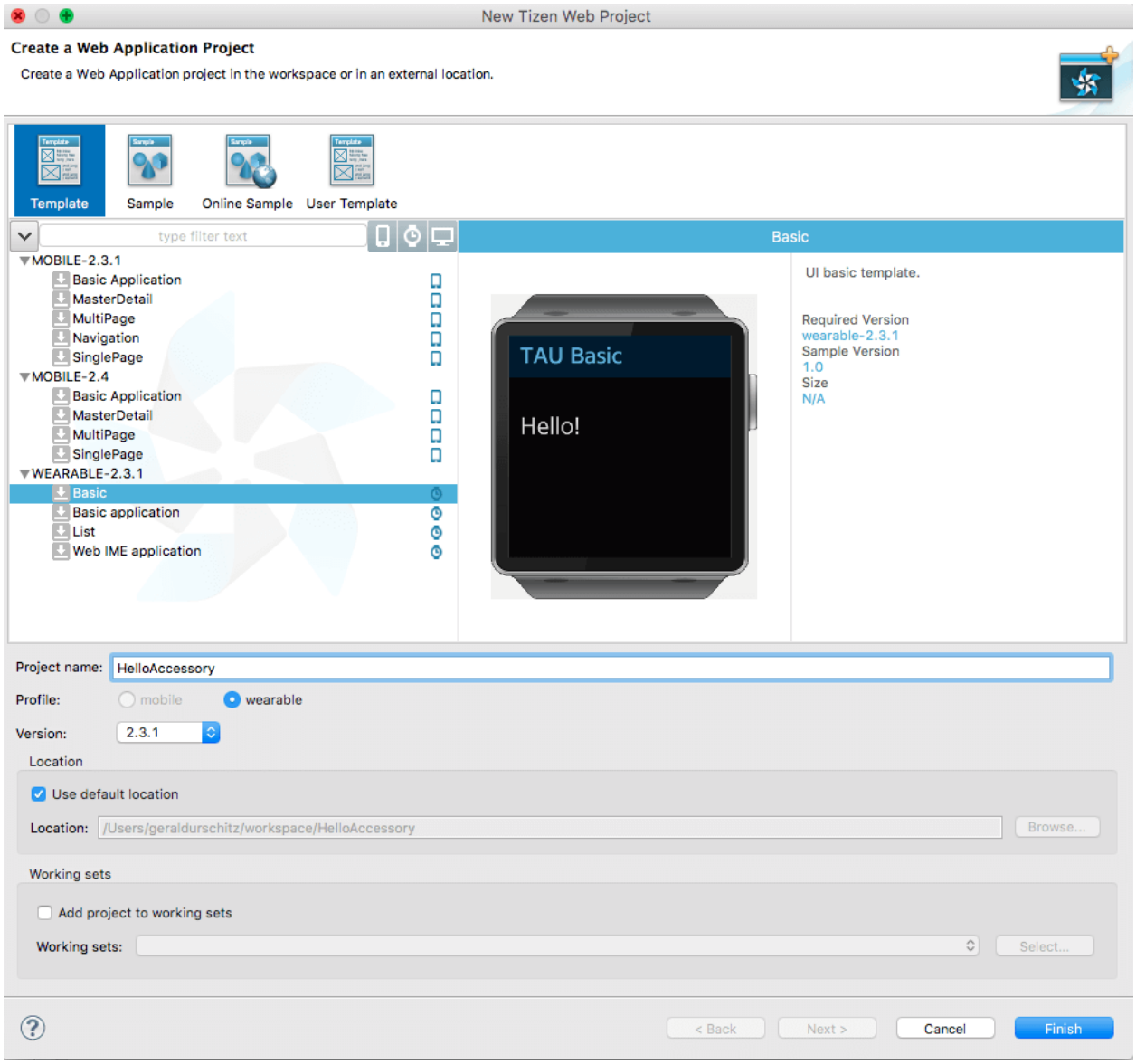


Figure 5.3.: Choose the basic template and give your project a suitable name.

Just like Android Studio, the Tizen IDE generates the whole project structure so that our application actually already works. In our case, we will however just import the sample application from Samsung and get started with this, as it already has everything set up for us. Simply click on File ->Import, then select General ->Existing Projects into Workspace. Then simply choose the sample application Tizen Web Application "HelloAccessory", that comes with the accessory SDK package. Be sure to import it from the folder, where android is the provider, and be sure to just import the consumer.
I will briefly explain the most important files:

- **index.html:** This is our main index file. It defines the html tag and includes the tags head and body, just like on a normal HTML5 web page.

In this file, we also place the UI structure. In the case of this sample application, it has a *ul*-tag with the class *ui-listview* and the three different onclick-events *connect()*, *fetch()* and *disconnect()*. These are then displayed as list buttons. Each calls the respective function in *main.js*, that we will discuss later.

```
<div class="ui-content">
        <ul class="ui-listview">
            <li><a href="#" onclick="connect();">Connect</a></li>
            <li><a href="#" onclick="fetch();">Fetch</a></li>
            <li><a href="#" onclick="disconnect();">Disconnect</a></li>
        </ul>
</div>
```

- **js/main.js:** In this JavaScript file, we do all the login on the gear sided application. Building up the connection, sending and receiving is all done in this file. Again, you can find the whole file either in the sample application or in the appendix Code Listings. I will briefly explain the *createHTML()* function, the *onreceive()* function and the *fetch()* function.

```
function createHTML(log_string)
{
    var content = document.getElementById("toast-content");
    content.textContent = log_string;
    tau.openPopup("#toast");
}
...
function onreceive(channelId, data) {
    createHTML(data);
}
...
function fetch() {
    try {
        SASocket.sendData(CHANNELID, "Hello Accessory!");
    } catch(err) {
        console.log("exception [" + err.name + "] msg[" + err.message + "]");
    }
}
```

createHTML() simply sets the context of an element with the id "toast-container" and opens it as a popup. *onreceive()* is called when new data arrives at our gear. The callback is set via the *agentCallback* with the following line: *SASocket.setDataReceiveListener(onreceive)*. In our case, onreceive simple takes the data and passes it to createHTML. This way, our String with the formatted time from android will finally lead to a popup on the Gear. The *fetch()* function is called when the user presses on "Fetch" in the UI. It simply sends the String "Hello Accessory!" via the channel ID variable. In our case, the channel ID variable is set to 104 in the top of *main.js* file.

- **jres/xml/accessoryservices.xml:** Here, we have the service profile again that identifies our connection. It is practically the same as in the android project, we just need to change the role to "consumer".

5.4. Debugging

5.4.1. Testing Gear application with the emulator

In order to run the app, you can use the emulator from Tizen. It simulates a Samsung Gear 2 that you can connect to using a real android device, that's plugged into your computer. For this to work, you need to prepare your device. Please follow this guide from Samsung to do this: `http://img-developer.samsung.com/contents/cmm/Guideline_on_Testing_Gear_applications_using_the_Emulator.pdf`

5.4.2. Certification

Furthermore, to actually run your app you need a certificate file from Samsung. To do this, you first need to get hold of your device id (either from the real device or the emulator). Any plugged device or any emulator instance shows up in the Connection Explorer. Simply do a right click and select Properties. You can then see the DUID or the device unique id. Next, click on the certificate button, that looks like in figure 5.4.

Figure 5.4.

The certification window opens up. Simply follow the guide from top to bottom and insert the respective files. You will get these files emailed to you.

5.4.3. Running the Application

If everything was right, you can now start the app.

Emulator

If you want to use the emulator, you need to start your virtual machine. You can do this using the Emulator Manager. Then, simply right click on your project, select Run As ->1 Tizen Web Application. After a while, your application should appear. See 5.5.

Figure 5.5.

Real Device

If you want to use a real device to run the application on, you simply need to plug in the device. Beware that you may need to install Samsung Kies to get your watch to show up in your device list. You can find Samsung Kies under `http://www.samsung.com/at/support/usefulsoftware/KIES/`. Your watch should show up in the Connection Explorer, if everything worked out. Then you simply need to click on the play button in the top bar to run your application.

6. Conclusion

Android provides an exiting platform for developing mobile applications for a huge number of devices. Due to its open nature and its good documentation, new developers feel at home very fast and can flourish in the community of android developers. Because of its huge market share, Android nowadays is a must for any type of company that want to provide a mobile application of their service.
As you hopefully saw in earlier chapters, even without deep knowledge, it is possible to create apps within a matter of hours with Android. You now know the basics to create a (very simple) User Interface and how to define its behavior. You know how to include external libraries such as Twitter Fabric and furthermore, you know how to include Twitter Login and display tweets using Twitter Fabric. As a bonus, you have an idea of how to get started with the Samsung Gear 2 Smartwatch, the Accessory SDK and Tizen. It probably does not feel like much, but you can now build up on that and start making great Apps for Android. Apps that might be one day the Google Play Store.

A. Installation Guides

A.0.4. Installing Android Studio

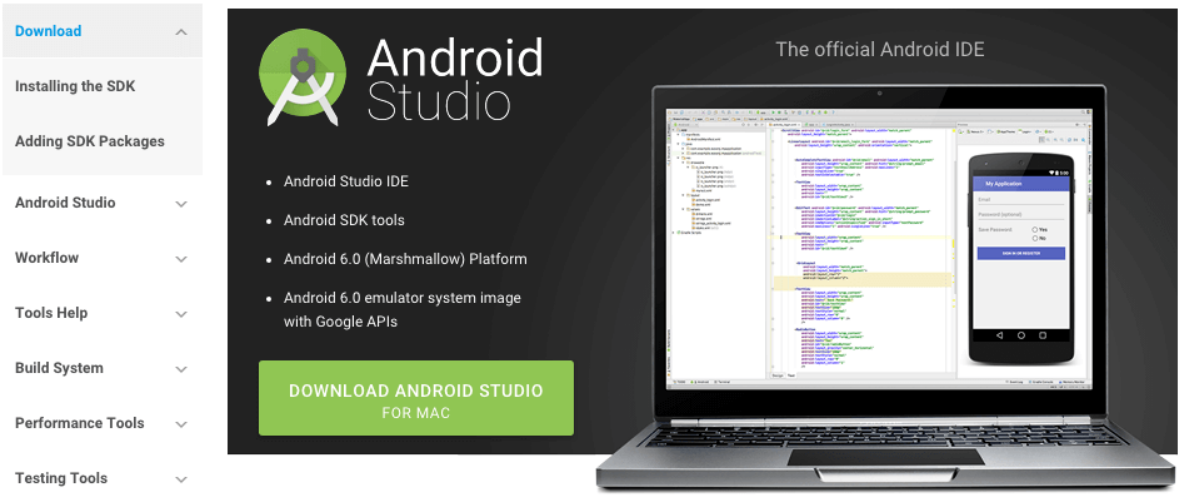


Figure A.1.: The official page where you can download Android Studio and the Android SDK.

In this paper, I will use the MacOS X Version, but there shouldn't be too much difference between the different platforms. To install it on your platform, just follow the official instructions that are provided with Android Studio.

A.1. Android SDK

The Android SDK - short for Software Development Kit - includes the following components: [15]

- Several tools such as the Android Debug Bridge (ADB) and the virtual device for emulation.
- All Libraries and APIs, that are being used for Android Development
- Documentation, Sample Codes and Tutorials

- Several extras for implementing other Google-Services

For the Android SDK, next to the name for each major version (like KitKat), there are two versioning values that are important. First, there is the version number itself, which is a number in semantic versioning. (To read more about semantic versioning, please refer to http://semver.org/) Second, the API is described by the API Level as an integer. For developers, this is the more important number, as the application code depends on the API. In practice, this means that "visible" changes (for example features for the user) are described by the version number, and changes "inside" are described by the API Level. It needs to be said, that one API Level can withstand through several versions. See A.2 to have an overview of the different Android SDKs. [16]

Version	Codename	API	Distribution
2.2	Froyo	8	0.2%
2.3.3 - 2.3.7	Gingerbread	10	3.8%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	3.3%
4.1.x	Jelly Bean	16	11.0%
4.2.x		17	13.9%
4.3		18	4.1%
4.4	KitKat	19	37.8%
5.0	Lollipop	21	15.5%
5.1		22	10.1%
6.0	Marshmallow	23	0.3%

Data collected during a 7-day period ending on November 2, 2015.
Any versions with less than 0.1% distribution are not shown.

Figure A.2.: The versions and api levels of all the different SDKs as of November 2015. [16]

A.1.1. Configuring the SDK

After the installation completed, you will see a window like the one on the left side in figure A.3. To configure the Android SDK, simply click on Configure and then on SDK Manager.

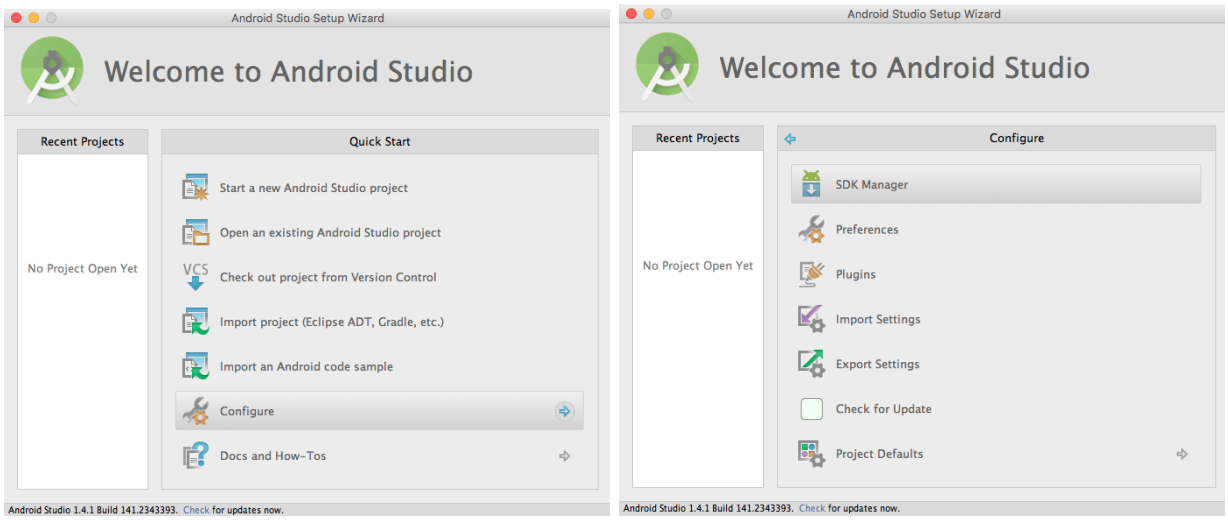


Figure A.3.: To configure the SDK click on Configure ->SDK Manager.

This will open the built-in SDK Manager of Android Studio. There is also a standalone version of this SDK Manager, but for our purposes, the built-in one is sufficient. For our Application we will use Android 4.4 KitKat, as we currently reach the most devices with this version with 37.8%. In the SDK Manager, choose Android 4.4.2 and click on Apply. This will download and install the Android SDK in the version 4.4.2 with API Level 19. See figure A.4.

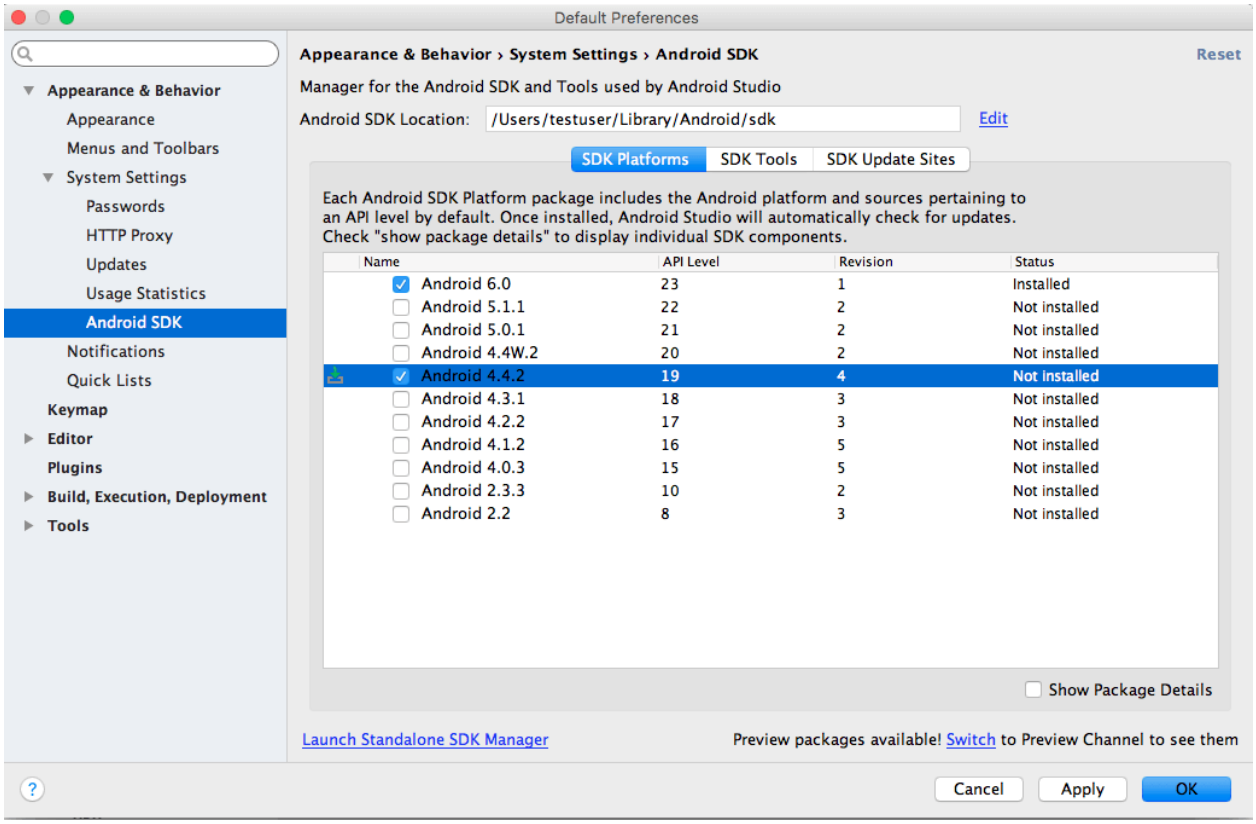


Figure A.4.: Choose 4.4.2 with API Level 19.

A.2. Install Tizen SDK

In order to develop applications for Tizen, you need to install the Tizen SDK, which includes a special IDE for Tizen apps, an emulator, sample code, several tools as well as documentation.

The Tizen IDE is built upon Eclipse and - just like Android Studio - makes it easy for you to create Tizen projects. See figure A.5 for a screenshot of the Tizen IDE.

To get started, download the Tizen SDK. You can find it under `https://developer.tizen.org/development/tools/download?langswitch=en` for the platforms Windows, Ubuntu and Mac OS X. After installation, the Tizen Update Manager asks you what tools you want to install. Be sure to install everything for the Wearable platform, the Tizen SDK tools as well as the *Certificate Extension* and *Tizen Wearable Extension for 2.3.1*. Please especially make sure to install the *Emulator* and the *Emulator Manager*.

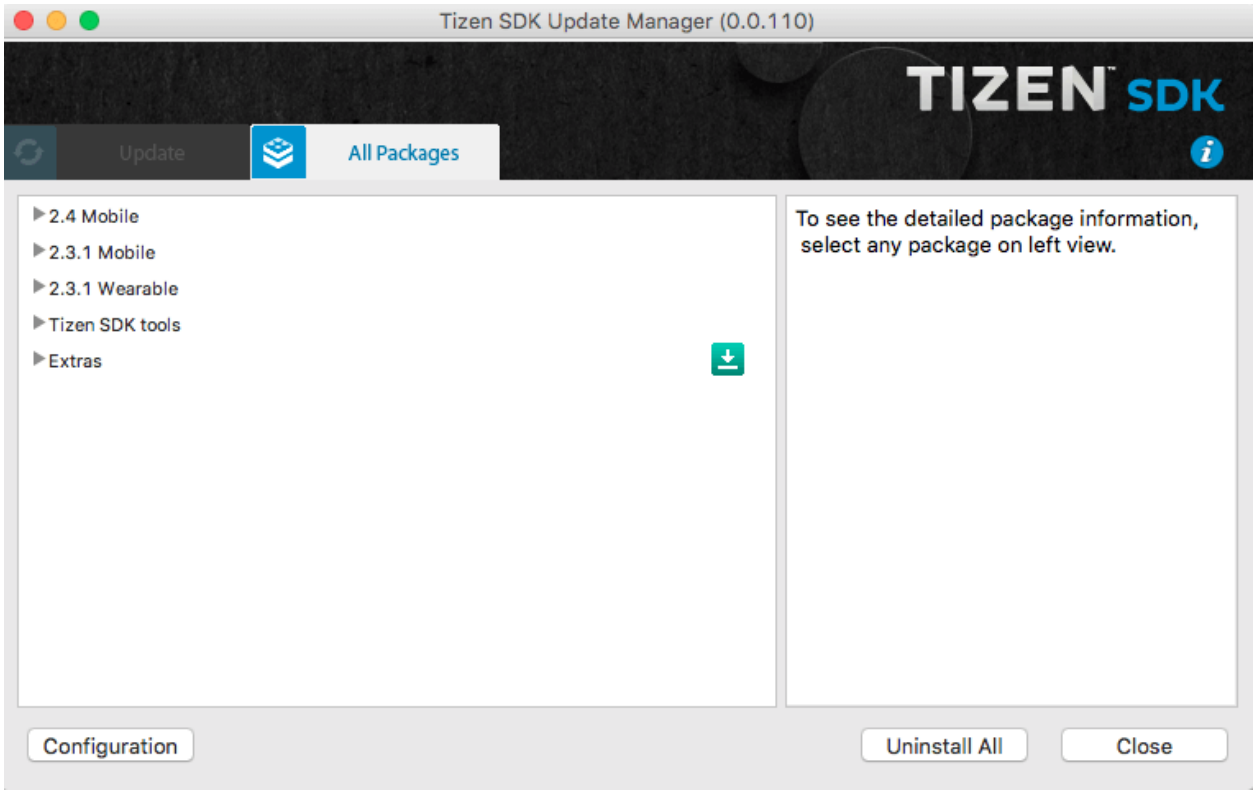


Figure A.5.: Installing the necessary tools for Samsung Gear development happens in the Tizen Update Manager.

B. Code Listings

B.1. HelloAccessory - Provider

B.1.1. AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.example.recenttweets"
          xmlns:android="http://schemas.android.com/apk/res/android">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <service android:name="com.example.recenttweets.ProviderService" />

        <receiver android:name="com.samsung.android.sdk.accessory.RegisterUponInstallReceiver" >
            <intent-filter>
                <action android:name="com.samsung.accessory.action.REGISTER_AGENT" />
            </intent-filter>
        </receiver>
        <receiver android:name="com.samsung.android.sdk.accessory.
            ServiceConnectionIndicationBroadcastReceiver" >
            <intent-filter>
                <action android:name="android.accessory.service.action.
                    ACCESSORY_SERVICE_CONNECTION_IND" />
            </intent-filter>
        </receiver>

        <meta-data
            android:name="AccessoryServicesLocation"
            android:value="/res/xml/accessoryservices.xml" />
        <meta-data
            android:name="GearAppType"
            android:value="wgt" />

        <activity android:name=".LoginActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>

                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
            <activity android:name=".RecentTweetsActivity">
        </activity>
        <meta-data
            android:name="io.fabric.ApiKey"
            android:value="7be76ea568c99d51cb168933fe9de596341406d2" />
```

```
    </application>


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

    <uses-permission
        android:name="com.samsung.android.providers.context.permission.
            WRITE_USE_APP_FEATURE_SURVEY"/>
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-permission android:name="com.samsung.accessory.permission.ACCESSORY_FRAMEWORK" />
    <uses-permission android:name="com.samsung.WATCH_APP_TYPE.Companion" />
    <uses-permission android:name="com.samsung.wmanager.ENABLE_NOTIFICATION" />
</manifest>
```

B.1.2. ProviderService.java

```
public class ProviderService extends SAAgent {
    private static final String TAG = "HelloAccessory(P)";
    private static final Class<ProviderConnection> SASOCKET_CLASS = ProviderConnection.class;
    private static final int HELLOACCESSORY_CHANNEL_ID = 104;
    private ProviderConnection connection;
    private final IBinder binder = new LocalBinder();

    public ProviderService(){
        super(TAG, SASOCKET_CLASS);
    }

    @Override
    public IBinder onBind(Intent intent) {
        return binder;
    }

    public class LocalBinder extends Binder {
        public ProviderService getService() {
            return ProviderService.this;
        }
    }

    @Override
    public void onCreate() {
        super.onCreate();
        SA accessory = new SA();
        try {
            accessory.initialize(this);
        } catch (SsdkUnsupportedException e) {
            // you can handle SsdkUnsupportedException here
        } catch (Exception e1) {
            e1.printStackTrace();
            /*
             * Your application can not use Samsung Accessory SDK. Your application should work
                 smoothly
             * without using this SDK, or you may want to notify user and close your application
                 gracefully
             * (release resources, stop Service threads, close UI thread, etc.)
             */
            stopSelf();
        }
    }

```

```
43      @Override
        protected void onServiceConnectionResponse(SAPeerAgent peerAgent, SASocket thisConnection,
            int result) {
45          if (result == CONNECTION_SUCCESS) {
                if (thisConnection != null) {
47                  connection = (ProviderConnection) thisConnection;
                }
49          }else if (result == CONNECTION_ALREADY_EXIST) {
                    Log.e(TAG, "CONNECTION_ALREADY_EXIST");
51              }
        }
53

55      class ProviderConnection extends SASocket {

57          protected ProviderConnection(String s) {
                super(s);
59          }

61          @Override
            public void onError(int i, String s, int i1) {
63

            }
65
            @Override
67          public void onReceive(int i, byte[] bytes) {
                if (connection == null) {
69                  return;
                }
71              Calendar calendar = new GregorianCalendar();
                SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy.MM.dd aa hh:mm:ss.SSS");
73              String timeStr = " " + dateFormat.format(calendar.getTime());
                String strToUpdateUI = new String(bytes);
75              final String message = strToUpdateUI.concat(timeStr);

77              new Thread(new Runnable() {
                    public void run() {
79                      try {
                            connection.send(HELLOACCESSORY_CHANNEL_ID, message.getBytes());
81                      } catch (IOException e) {
                            e.printStackTrace();
83                      }
                    }
85              }).start();
            }
87
            @Override
89          protected void onServiceConnectionLost(int i) {
                connection = null;
91              Log.w(TAG,"Connection lost");
            }
93      }
    }
```

B.2. HelloAccessory - Consumer

B.2.1. index.html

```
/*
```

```
 * Copyright (c) 2014 Samsung Electronics Co., Ltd.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 *     * Redistributions of source code must retain the above copyright
 *        notice, this list of conditions and the following disclaimer.
 *     * Redistributions in binary form must reproduce the above
 *       copyright notice, this list of conditions and the following disclaimer
 *       in the documentation and/or other materials provided with the
 *       distribution.
 *     * Neither the name of Samsung Electronics Co., Ltd. nor the names of its
 *       contributors may be used to endorse or promote products derived from
 *       this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

var SAAgent = null;
var SASocket = null;
var CHANNELID = 104;
var ProviderAppName = "HelloAccessoryProvider";

function createHTML(log_string)
{
    var content = document.getElementById("toast-content");
    content.textContent = log_string;
    tau.openPopup("#toast");
}

function onerror(err) {
    console.log("err [" + err + "]");
}

var agentCallback = {
    onconnect : function(socket) {
        SASocket = socket;
        createHTML("HelloAccessory Connection established with RemotePeer");
        SASocket.setSocketStatusListener(function(reason){
            console.log("Service connection lost, Reason : [" + reason + "]");
            disconnect();
        });
        SASocket.setDataReceiveListener(onreceive);
    },
    onerror : onerror
};

var peerAgentFindCallback = {
    onpeeragentfound : function(peerAgent) {
        try {
```

```
64              if (peerAgent.appName == ProviderAppName) {
                    SAAgent.setServiceConnectionListener(agentCallback);
66                  SAAgent.requestServiceConnection(peerAgent);
                } else {
68                  createHTML("Not expected app!! : " + peerAgent.appName);
                }
70          } catch(err) {
                console.log("exception [" + err.name + "] msg[" + err.message + "]");
72          }
        },
74      onerror : onerror
    }
76
    function onsuccess(agents) {
78      try {
            if (agents.length > 0) {
80              SAAgent = agents[0];

82              SAAgent.setPeerAgentFindListener(peerAgentFindCallback);
                SAAgent.findPeerAgents();
84          } else {
                createHTML("Not found SAAgent!!");
86          }
        } catch(err) {
88          console.log("exception [" + err.name + "] msg[" + err.message + "]");
        }
90  }

92  function connect() {
        if (SASocket) {
94          createHTML('Already connected!');
            return false;
96      }
        try {
98          webapis.sa.requestSAAgent(onsuccess, function (err) {
                console.log("err [" + err.name + "] msg[" + err.message + "]");
100         });
        } catch(err) {
102         console.log("exception [" + err.name + "] msg[" + err.message + "]");
        }
104 }

106 function disconnect() {
        try {
108         if (SASocket != null) {
                SASocket.close();
110             SASocket = null;
                createHTML("closeConnection");
112         }
        } catch(err) {
114         console.log("exception [" + err.name + "] msg[" + err.message + "]");
        }
116 }

118 function onreceive(channelId, data) {
        createHTML(data);
120 }

122 function fetch() {
        try {
124         SASocket.sendData(CHANNELID, "Hello Accessory!");
        } catch(err) {
```

```
126         console.log("exception [" + err.name + "] msg[" + err.message + "]");
    }
128 }

130 window.onload = function () {
    // add eventListener for tizenhwkey
132     document.addEventListener('tizenhwkey', function(e) {
        if(e.keyName == "back")
134             tizen.application.getCurrentApplication().exit();
    });
136 };

138 (function(tau) {
    var toastPopup = document.getElementById('toast');
140     toastPopup.addEventListener('popupshow', function(ev){
        setTimeout(function(){tau.closePopup();}, 3000);
142     }, false);
})(window.tau);
144 \end{stlisting}

146 \subsection{js/main.js}
\begin{lstlisting}
148 /*
 * Copyright (c) 2014 Samsung Electronics Co., Ltd.
150  * All rights reserved.
 *
152  * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
154  * met:
 *
156  *     * Redistributions of source code must retain the above copyright
 *        notice, this list of conditions and the following disclaimer.
158  *     * Redistributions in binary form must reproduce the above
 *       copyright notice, this list of conditions and the following disclaimer
160  *       in the documentation and/or other materials provided with the
 *       distribution.
162  *     * Neither the name of Samsung Electronics Co., Ltd. nor the names of its
 *       contributors may be used to endorse or promote products derived from
164  *       this software without specific prior written permission.
 *
166  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
168  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
170  * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
172  * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
174  * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
176  * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
178
var SAAgent = null;
180 var SASocket = null;
var CHANNELID = 104;
182 var ProviderAppName = "HelloAccessoryProvider";

184 function createHTML(log_string)
{
186     var content = document.getElementById("toast-content");
    content.textContent = log_string;
```

```
188     tau.openPopup("#toast");
    }
190
    function onerror(err) {
192     console.log("err [" + err + "]");
    }
194
    var agentCallback = {
196     onconnect : function(socket) {
            SASocket = socket;
198         createHTML("HelloAccessory Connection established with RemotePeer");
            SASocket.setSocketStatusListener(function(reason){
200             console.log("Service connection lost, Reason : [" + reason + "]");
                disconnect();
202         });
            SASocket.setDataReceiveListener(onreceive);
204     },
        onerror : onerror
206 };

208 var peerAgentFindCallback = {
        onpeeragentfound : function(peerAgent) {
210         try {
                if (peerAgent.appName == ProviderAppName) {
212                 SAAgent.setServiceConnectionListener(agentCallback);
                    SAAgent.requestServiceConnection(peerAgent);
214             } else {
                    createHTML("Not expected app!! : " + peerAgent.appName);
216             }
            } catch(err) {
218             console.log("exception [" + err.name + "] msg[" + err.message + "]");
            }
220     },
        onerror : onerror
222 }

224 function onsuccess(agents) {
        try {
226         if (agents.length > 0) {
                SAAgent = agents[0];
228
                SAAgent.setPeerAgentFindListener(peerAgentFindCallback);
230             SAAgent.findPeerAgents();
            } else {
232             createHTML("Not found SAAgent!!");
            }
234     } catch(err) {
            console.log("exception [" + err.name + "] msg[" + err.message + "]");
236     }
    }
238
    function connect() {
240     if (SASocket) {
            createHTML('Already connected!');
242         return false;
        }
244     try {
            webapis.sa.requestSAAgent(onsuccess, function (err) {
246             console.log("err [" + err.name + "] msg[" + err.message + "]");
            });
248     } catch(err) {
            console.log("exception [" + err.name + "] msg[" + err.message + "]");
```

```
250     }
    }
252
    function disconnect() {
254     try {
            if (SASocket != null) {
256             SASocket.close();
                SASocket = null;
258             createHTML("closeConnection");
            }
260     } catch(err) {
            console.log("exception [" + err.name + "] msg[" + err.message + "]");
262     }
    }
264
    function onreceive(channelId, data) {
266     createHTML(data);
    }
268
    function fetch() {
270     try {
            SASocket.sendData(CHANNELID, "Hello Accessory!");
272     } catch(err) {
            console.log("exception [" + err.name + "] msg[" + err.message + "]");
274     }
    }
276
    window.onload = function () {
278     // add eventListener for tizenhwkey
        document.addEventListener('tizenhwkey', function(e) {
280         if(e.keyName == "back")
                tizen.application.getCurrentApplication().exit();
282     });
    };
284
    (function(tau) {
286     var toastPopup = document.getElementById('toast');
        toastPopup.addEventListener('popupshow', function(ev){
288         setTimeout(function(){tau.closePopup();}, 3000);
        }, false);
290 })(window.tau);
```

Bibliography

[1] Wikipedia, "Android (betriebssystem)." https://de.wikipedia.org/w/index.php?title=Android_(Betriebssystem)&oldid=147243252. Retrieved: 2015-10.

[2] IDC, "Idc: Smartphone os market share 2015, 2014, 2013, and 2012." http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Retrieved: 2015-10.

[3] Wikipedia, "Android runtime." https://de.wikipedia.org/w/index.php?title=Android_Runtime&oldid=145850639. Retrieved: 2015-10.

[4] G. Inc., "App resources." http://developer.android.com/guide/topics/resources/index.html. Retrieved: 2015-12.

[5] Google, "Configuring gradle builds." http://developer.android.com/tools/building/configuring-gradle.html. Retrieved: 2015-12.

[6] Google, "Activity." http://developer.android.com/reference/android/app/Activity.html. Retrieved: 2015-11.

[7] D. o. P. Jeff Seibert, "Introducing fabric." https://blog.twitter.com/2014/introducing-fabric. Retrieved: 2015-12.

[8] T. Inc., "Fabric docs." https://docs.fabric.io/. Retrieved: 2015-12.

[9] Google, "Intent." http://developer.android.com/reference/android/content/Intent.html. Retrieved: 2015-10.

[10] T. Inc., "Fabric docs: Statusesservice." https://docs.fabric.io/javadocs/twitter-core/1.4.1/com/twitter/sdk/android/core/services/StatusesService.html. Retrieved: 2015-12.

[11] T. Inc., "Display requirements." https://about.twitter.com/company/display-requirements/. Retrieved: 2015-12.

[12] S. Wong, "Samsung unveils tizen-powered gear 2 and gear 2 neo smart watches." http://www.hardwarezone.com.sg. Retrieved: 2015-12.

[13] Wikipedia, "Tizen." https://en.wikipedia.org/w/index.php?title=Tizen&oldid=692447785. Retrieved: 2015-12.

[14] Samsung, "Samsung accessory sdk." http://developer.samsung.com/galaxy#accessory. Retrieved: 2015-12.

[15] Wikipedia, "Android software development." https://en.wikipedia.org/w/index.php?title=Android_software_development&oldid=689857190. Retrieved: 2015-11.

[16] Google, "Dashboards." http://developer.android.com/about/dashboards/index.html. Retrieved: 2015-11.