Informatikarbeit

Android development

Christian Eppler[*]

4. Januar 2013

Hochschule Ravensburg-Weingarten

[*]Vielen Dank möchte ich an dieser Stelle an Frau Prof. Dr.-Ing. Keller aussprechen, Die diese Arbeit betreut hat.

Inhaltsverzeichnis

Kapitel 1 Einleitung **1**
1.1 Konventionen 2

Kapitel 2 Android System Design **3**
2.1 Dalvik VM 3
2.2 Sandbox 5
2.2.1 Signatur 5
2.2.2 Manifest 6
2.3 Komponenten Design 8
2.4 Intents 8
2.5 Lifecycle 11

Kapitel 3 IDE einrichten **12**
3.1 Eclipse Plugins 12
3.2 Emulator 14

Kapitel 4 Oberfläche **16**
4.1 XML 16
4.1.1 Layouts 16
4.1.2 Werte 17
4.1.3 Mehrsprachig 18
4.2 Bilder 18
4.3 Layouts 19
4.3.1 Linear Layout 19
4.3.2 Absolute Layout 21
4.3.3 Relative Layout 21
4.3.4 Web View 22
4.3.5 List View 22
4.3.6 Grid View 23
4.4 Menüs 24
4.4.1 Context Menu 24
4.4.2 Option Menu 27
4.5 Dynamische Oberflächen 29
4.6 Widgets 32
4.7 Activities 37

Kapitel 5 Services **38**

5.1 Arten 38
5.1.1 Gebundener Service 38
5.1.2 Remote Service 39
5.2 Broadcast receiver 42
5.3 AlarmManager 44
5.4 PowerManager 44

Kapitel 6 Persistente Datenhaltung **45**
6.1 SQLite 45
6.2 Shared Preferences 51
6.3 Internal Storage 51
6.4 External Storage 51

Glossar

Abbildungsverzeichnis

1.1 Beispiel 2

2.1 Workflow 4
2.2 Beispiel eines Manifest Headers 6
2.3 Permissions 7
2.4 Activity 7
2.5 Intent-Menü 8
2.6 Intent exlpizit 9
2.7 Intentfilter 10
2.8 Lifecycle 11

3.1 Quelle hinzufügen 12
3.2 Plugin installieren 13
3.3 SDK installieren 13
3.4 Emulator AVD 15

4.1 Projektordner 16
4.2 Strings XML 17
4.3 Verwendung von Strings im Code 17
4.4 Verwendung von Strings in XML 18
4.5 Linear Layout 19
4.6 Linear Layout node.xml 20
4.7 Absolute Layout 21
4.8 Relative Layout 21
4.9 Web View 22
4.10 List View 22
4.11 Grid View 23
4.12 Context Menu 24
4.13 Context Menu Code 1 25
4.14 Context Menu Code 2 26
4.15 Toast 26
4.16 Option Menu 27
4.17 Option Menu 2 27
4.18 Option Menu Code 1 28
4.19 Option Menu Code 2 28
4.20 Kontainer main.xml 30
4.21 Layout zuweisen 30
4.22 Layout dynamisch füllen 31

4.23 Widget 32
4.24 Widget-Manifesteintrag 32
4.25 widget.xml 33
4.26 Widget Code 1 34
4.27 Widget Code 2 35
4.28 widgetlayout.xml 36
4.29 Widget-Aufruf 36

5.1 Service-Manifesteintrag 39
5.2 Service Code 41
5.3 Broadcast receiver Manifesteintrag 42
5.4 Broadcast receiver Code 43
5.5 PowerManager Code 44

6.1 SQLite Tables 45
6.2 SQLite Datenbankschema erstellen 46
6.3 SQLite Datenbankschema aktualisieren 47
6.4 DB-Verbindung 48
6.5 Select-Statment 48
6.6 Insert und update-Statment 49
6.7 Delete-Statement 50

1 Einleitung

Dieses Dokument soll dem Leser einen kurzen Überblick über wichtige Bestandteile der Android Entwicklung vermitteln. Jedoch ist Android ein sich schnell weiter entwickelndes System mit einem sehr großen Spektrum an verschiedenen Bereichen.

Im gegebenen Zeitumfang dieser Informatikarbeit können nicht alle Themen behandelt werden und in vielen Fällen auch nicht so tief, wie es nötig wäre. Sinn dieses Dokumentes ist es, dem Leser einen kurzen Überblick über die Android Plattform und der Entwicklung von Anwendungen sog. Apps für Android zu geben. Weder liegt es im Sinne des Verfassers, noch in der Natur des Dokuments ein ausführliches *Lehrwerk* zu ersetzen. Somit ist der Leser selbst dazu aufgefordert, sich weiter aus anderen Quellen zu informieren. Sollte Ihnen dieses Dokument als PDF vorliegen ist zu beachten das Referenzierungen klickbar sind.

Viele Code-Beispiele, die in diesem Dokument verwendet werden, entstammen aus einem bestehenden Projekt: weblooker[1]. Gerne können Sie sich den Code zum Experimentieren unter: `http://weblooker.svn.sourceforge.net/viewvc/weblooker/tags/weblookera_1.0/` downloaden.

This work is licensed under the Creative Commons 3.0 by-nc-sa To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-sa/3.0/`.

[1] `http://sourceforge.net/projects/weblooker/`

1.1 Konventionen

Am Anfang eines Abschnitts werden, wenn benötigt, Begriffe, die aus dem Umfeld der Android Entwicklung stammen, die erst später vollständig erklärt werden, kurz erläutert. Weiterhin finden Sie alle Definitionen auf den letzten Seiten dieses Dokumentes im Glossar.

Definitionen:

Android:
Von Google entwickeltes, auf dem Linux Kernel basierendes, für den mobilen Einsatz optimiertes, Betriebssystem.

Beispielcode wird in einem grauen Kasten dargestellt, der eine Abbildungsnummer besitzt und zeilenweise nummeriert ist. In den dazu gehörigen Erklärungen wird auf die Abbildung sowie auf die wichtigen Zeilennummern verwiesen. (siehe Abbildung 1.1)

```
1  <uses-permission
2      android:name="android.permission.ACCESS_NETWORK_STATE" />
3  <uses-permission
```

Abbildung 1.1: Beispiel

Viele Themen können nicht vollständig behandelt werden. Ist dies der Fall, gibt es eine Fußnote zu weiterführenden Links.

weblooker[2]

[2] http://weblooker.org

2 Android System Design

2.1 Dalvik VM

Definitionen:

App:

Abkürzung für Applikation. Im Sprachgebrauch wird der Begriff aber meist nur für kleinere Programme verwendet, welche meist im mobilen Umfeld zu finden sind.

Applikationen für Android sog. Apps werden i.d.R. in Java geschrieben. Performancekritische oder systemnahe Apps werden meist im nativen C++ geschrieben, bei dem die Dalvik VM komplett umgangen wird.[1]

Im Gegensatz zur ursprünglichen Java VM (JVM)[2] wurde die *Dalvik Virtual Machine* speziell für Android entwickelt und optimiert. Das Android SDK[3] wandelt die vom Java Compiler erstellten .class Dateien in für die DVM ausführbaren *Dex-Bytecode* um. Genau genommen erledigt dies das Tool dx, welches den bestehenden Java Bytecode in den dex Code umwandelt. (siehe Abbildung 2.1 auf der nächsten Seite) Das Android-SDK bzw. das Eclipse Plugin führen alle nötigen Schritte automatisch aus. Dieser *Dex-Bytecode* ist im Gegensatz zum normalen Java-Byte Code auf die Architektur des Zielsystemes optimiert. Eine dieser Optimierungen ist die Nutzung der Register der Ziel CPU. Des weiteren wurde die DVM darauf optimiert, performant mehrfach nebeneinander zu laufen. (siehe Abschnitt 2.2 auf Seite 5)

[1] NDK `http://developer.android.com/tools/sdk/ndk/index.html`
[2] `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`
[3] `http://developer.android.com/sdk/index.html`

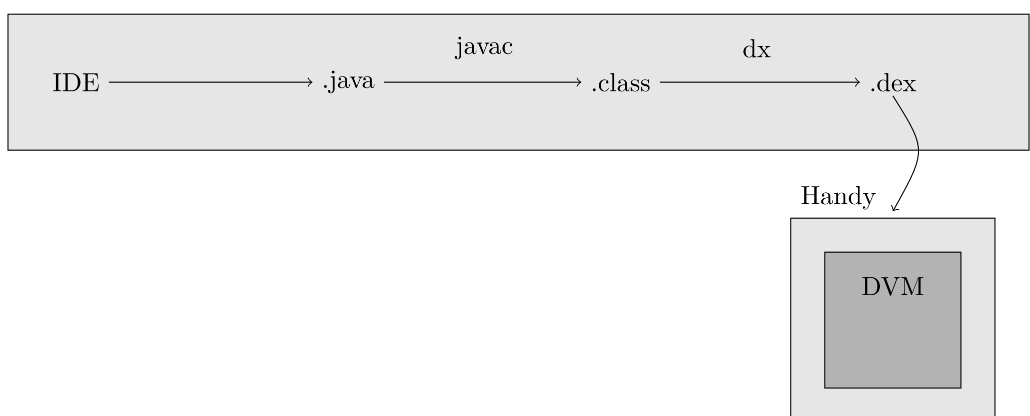


Abbildung 2.1: Workflow

2.2 Sandbox

Jede App unter Android läuft in ihrer eigenen Sandbox[4], was sich positiv auf die Sicherheit sowie die Stabilität des Systems auswirkt. Die durch den resultierenden Overhead entstehenden Performance-Einbußen spielen angesichts dieser Vorteile eine untergeordnete Rolle.

Da jede App ihren eigenen Speicher sowie ihre eigene DVM hat, ist der Wirkungsbereich einer abstürzenden App bzw. einer böswilligen App begrenzt. Des weiteren ist es durch das Sandbox-Prinzip leichter möglich, ein sehr granulares Rechtemanagement zu realisieren. (siehe Abschnitt 2.2.2 auf der nächsten Seite)

2.2.1 Signatur

Apps müssen bevor sie auf einem Android-System installiert werden digital signiert werden. Beim Entwickeln der Anwendung und direktem Installieren der App über ein im Debuggingmodus angeschlossenes Gerät übernimmt dies das Eclipse Plugin mit einem mitgeliefertem Standardzertifikat.

Soll die App jedoch im fertigen Zustand in den *Google-Play-Store*[5] geladen werden, muss ein eigenes Zertifikat erstellt werden. Dies kann wieder mit dem Eclipse Plugin erledigt werden und muss nicht von Hand in einer Konsole erstellt werden. (siehe Abschnitt 3.1 auf Seite 12)

Die Signatur[6] wird unter Android dazu verwendet Apps, die mit der gleichen Signatur signiert sind zu identifizieren und einem Entwickler zuzuordnen. Die vom gleichen Zertifikat signierten Apps werden in der gleichen Sandbox ausgeführt, wodurch die Kommunikation zwischen diesen Applikationen vereinfacht wird. Außerdem wird dadurch die Wiederverwendung von Komponenten (siehe Abschnitt 2.3 auf Seite 8) erleichtert.

[4] http://en.wikipedia.org/wiki/Sandbox_(computer_security)

[5] https://play.google.com/store

[6] http://de.wikipedia.org/wiki/Elektronische_Signatur

2.2.2 Manifest

Definitionen:

Komponente:
Unter Android wird das Komponentendesign verfolgt, um somit Teile einer Applikationen leicht wiederverwenden zu können.

Activity:
Wird zur Darstellung der grafischen Bedienoberfläche verwendet. Zum Start besitzt sie den ganzen Bildschirm, um mit dem Benutzer zu interagieren.

Intent:
Vergleichbar mit einem Signal, welches unter Java verwendet wird, um eine Klasse aufzurufen, die den passenden Listener implementiert hat.

Intentfilter:
Wie unter Java hören Listener nur auf bestimmte Signale dies wird unter Android durch Intentfilter gesteuert.

Das Manifest spielt unter Android eine wichtige Rolle, denn hier müssen alle in der Applikation erstellten Komponenten bekannt gegeben werden. Zudem müssen alle zur Ausführung erforderlichen Spezialberechtigungen in der Manifest Datei definiert werden. Bei der Installation der App muss der Anwender diese Berechtigungen bestätigen, um die Applikation installieren zu können. Stimmt der Anwender den erforderten Berechtigungen nicht zu, bricht die Installation ab.

Folgend ist ein Beispiel einer Manifest Datei gelistet. Die Manifest Datei beginnt mit der Definition des XML Headers. Nachfolgend die Erklärung der weiteren Zeilen.

3. Paketname / Präfix
4. Code Version für interne Verwendung
5. Versionsnummer welche im *Google-Play-Store* angezeigt wird
6. Der Installationsort, an dem die App installiert werden soll
8. Minimal benötige Android Version und Version gegen die getestet wurde[7]

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="weblooker.andro"
4     android:versionCode="3"
5     android:versionName="1.0"
6     android:installLocation="auto">
7
8     <uses-sdk android:minSdkVersion="7" android:targetSdkVersion="16" />
```

Abbildung 2.2: Beispiel eines Manifest Headers

[7] http://developer.android.com/guide/topics/manifest/uses-sdk-element.html

In diesem weiteren Ausschnitt wird der Applikation die Berechtigung erteilt, den Netzwerkstatus zu erfragen. Da es unter Android viele Berechtigungen gibt, werden diese hier nicht alle einzeln aufgezählt.[8]

```
1     <uses-permission
2         android:name="android.permission.ACCESS_NETWORK_STATE" />
3     <uses-permission
```

Abbildung 2.3: Permissions

Nachfolgend wird ein Ausschnitt der Manifest Datei dargestellt, in der die Activity *WeblookerActivity* dem System bekannt gegeben wird. Der Punkt vor dem Namen der Activity in Zeile 2 dient als Platzhalter für den Paketnamen und wird automatisch eingefügt.[9] (siehe Abbildung 2.2 auf der vorherigen Seite) Ab Zeile 5 bis 8 wird ein Intentfilter angelegt, dieser spezifiziert bei welchem Intent (siehe Abschnitt 2.4 auf der nächsten Seite) diese Activity reagieren soll.

```
1         <activity
2             android:name=".WeblookeraActivity"
3             android:label="@string/app_name" >
4 
5                 <intent-filter>
6                     <action android:name="android.intent.action.MAIN" />
7                     <category android:name="android.intent.category.LAUNCHER" />
8                 </intent-filter>
9         </activity>
```

Abbildung 2.4: Activity

[8] http://developer.android.com/reference/android/Manifest.permission.html

[9] http://weblooker.svn.sourceforge.net/viewvc/weblooker/tags/weblookera_1.0/AndroidManifest.xml?revision=561&view=markup

2.3 Komponenten Design

Android ist ein modernes auf dem Komponenten Design[10] basierendes System, welches sich dadurch auszeichnet, dass Komponenten einer Applikation in einer weiteren Applikation wieder verwendet werden können. Genannt sei hier z. B. das Adressbuch, welches mit den benötigten Berechtigungen von einer eigenen App mitverwendet werden kann.

Außer denen vom Android-System bereitgestellten Komponenten können auch eigene Komponenten für die Verwendung durch Dritte freigegeben werden. Dies wird unter Android mit sogenannten *Content Providern*[11] erreicht auf die hier nicht weiter eingegangen wird.

2.4 Intents

Intents[12] dienen unter Android zur Kommunikation zwischen Komponenten und können dabei auch Daten zwischen ihnen übertragen. Es gibt zwei Arten von Intent, die *expliziten* sowie die *impliziten* Intents. Die expliziten Intents werden mit dem vollständigen Namen der Klasse aufgerufen und dienen meist zur internen Kommunikation einer Applikation. Im Gegensatz dazu werden implizite Intents meist für Aufrufe von Komponenten verwenden, welche außerhalb der eigenen Applikation liegen. Für implizite Intents spielen die *Intentfilter* eine wichtige Rolle, da das System durch diese entscheidet, welche Komponente aufgerufen wird. Gibt es mehrere Intentfilter, welche zum aufrufenden Intent passen, wird ein Auswahlmenü (siehe Abbildung 2.5) angezeigt.

Abbildung 2.5: Intent-Menü

[10] http://de.wikipedia.org/wiki/Komponente_(Software)
[11] http://developer.android.com/guide/topics/providers/content-providers.html
[12] http://developer.android.com/guide/components/intents-filters.html

Nachfolgend ist ein Beispiel eines expliziten Intents aufgezeigt, welcher außer dem eigentlichen Aufruf noch Daten an die aufgerufene Komponente übergibt. In diesem Beispiel wird die Activity *WeblookerActivityNode* aufgerufen, was in der Zeile 6 durch den 2. Parameter ersichtlich wird. In Zeile 9 werden Zusatzinformationen in Form eines Wertepaares übergeben. Abschließend wird in Zeile 11 der Intent durch *startActivity()* abgesendet.

```
1    public void onClick(View arg0)
2    {
3        /* Get node name */
4        String node = ((TextView) arg0).getText().toString();
5
6        final Intent intent = new Intent(this,WeblookerActivityNode.class);
7
8        /* Data for called activity */
9        intent.putExtra("node",node);
10
11               startActivity(intent);
12   }
```

Abbildung 2.6: Intent exlpizit

Nachfolgend sollen nun die impliziten Intents näher betrachtet werden. Wie unten ersichtlich, besteht der Intentfilter aus zwei XML-Einträgen.

1. <action> Intent auf den dieser Filter reagieren soll
2. <category> Wie die Komponente aufgerufen werden soll
3. <data> (In diesem Beispiel nicht vorhanden)[13]

In diesem Beispiel reagiert der Filter auf den Intent, den das System im Anschluss an einen vollständigen Bootvorgang aussendet. In Zeile 6 wird hier die Kategorie *DEFAULT*[14] gewählt. Diese Kategorie wird bei Intentfiltern vorausgesetzt, wenn diese auf Implizite Intent reagieren sollen.

Als weiteres Beispiel kann hier die Abbildung 2.4 auf Seite 7 betrachtet werden, in der spezifiziert wird, dass dieser Intentfilter auf den Intent *MAIN* reagiert. Der Intent *MAIN* deklariert den Haupteinstiegspunkt der Applikation. Außerdem wurde die Komponente in die Kategorie *LAUNCHER* eingeteilt, was dem System sagt, das diese Activity beim Klicken des App Icons aufgerufen werden soll.

```
1 <receiver android:name="weblooker.andro.net.StartupNetService"
2           android:enabled="true"
3           android:exported="true">
4         <intent-filter>
5                 <action android:name="android.intent.action.BOOT_COMPLETED" />
6                 <category android:name="android.intent.category.DEFAULT" />
7         </intent-filter>
8 </receiver>
```

Abbildung 2.7: Intentfilter

[13] http://developer.android.com/guide/topics/manifest/data-element.html
[14] http://developer.android.com/reference/android/content/Intent.html#CATEGORY_DEFAULT

2.5 Lifecycle

Da Android ein System für den mobilen Einsatz ist und in diesem Umfeld Ressourcen wie Speicher nicht in unbegrenzter Menge zur Verfügung stehen ist die Ressourcen Verwaltung unter Android sehr restriktiv gestaltet. Dies hat zur Folge das Komponenten z. B. eine Activity (siehe Abschnitt 4.7 auf Seite 37), die sich nicht gerade im Vordergrund aufhält, beendet werden kann, wenn ein Mangel an Ressourcen vorliegt.[15] Nachfolgend ist der Lifecycle dargestellt, welcher beim Erstellen von Apps immer beachtet werden sollte.

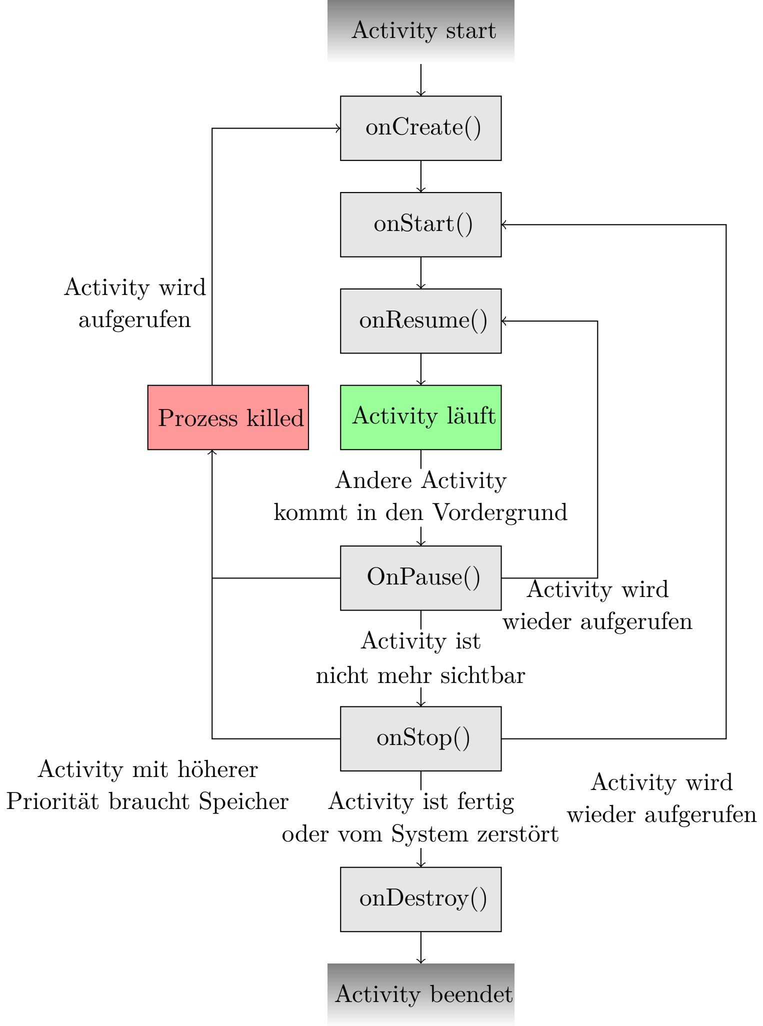


Abbildung 2.8: Lifecycle

[15] http://developer.android.com/training/basics/activity-lifecycle/index.html

3 IDE einrichten

Da der Verfasser dieses Dokumentes Eclipse als Entwicklungsumgebung verwendet, bezieht sich der folgende Inhalt dieses Kapitels auf Eclipse. Verständnis der grundlegenden Funktionen von Eclipse wird in diesem Kapitel vorausgesetzt.

3.1 Eclipse Plugins

Das Eclipse Plugin erleichtert dem Entwickler eine Menge von Aufgaben. Darunter fallen, nur um ein Paar zu nennen:

1. Projektstruktur anlegen
2. Erstellen eines Zertifikates
3. Debugging
4. Emulator
5. Updaten des SDKs
6. ...

Um diese Vorteile in Eclipse nutzen zu können, ist es nur nötig eine neue Quelle in Eclipse einzutragen.[1] (siehe Abbildung 3.1)

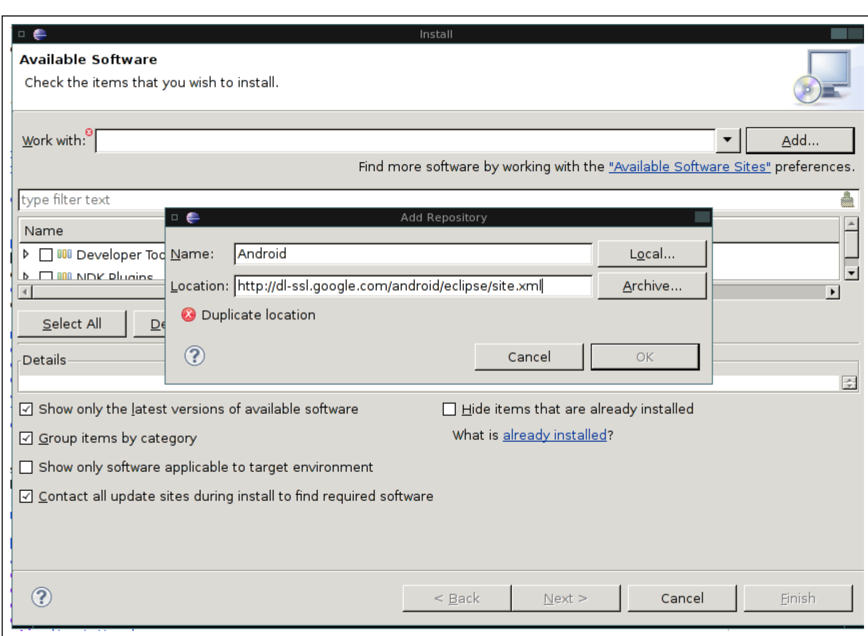


Abbildung 3.1: Quelle hinzufügen

[1] http://dl-ssl.google.com/android/eclipse/site.xml

Ist die neue Quelle eingetragen, können die IDE Plugins installiert werden.

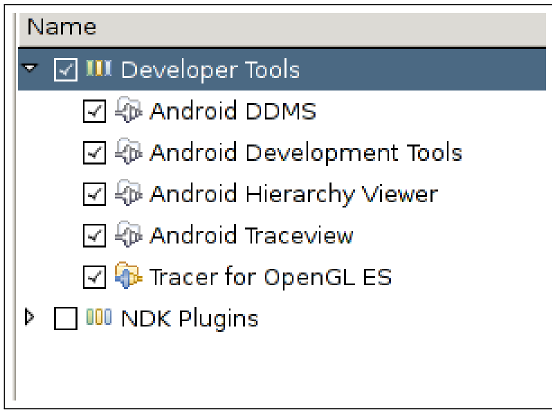


Abbildung 3.2: Plugin installieren

Nach dem Neustart der IDE muss nun noch das SDK mit zugehörigen Tools installiert werden. Dies muss aber nicht manuell gemacht werden, sondern das erledigt das Plugin. Window->Android SDK Manager

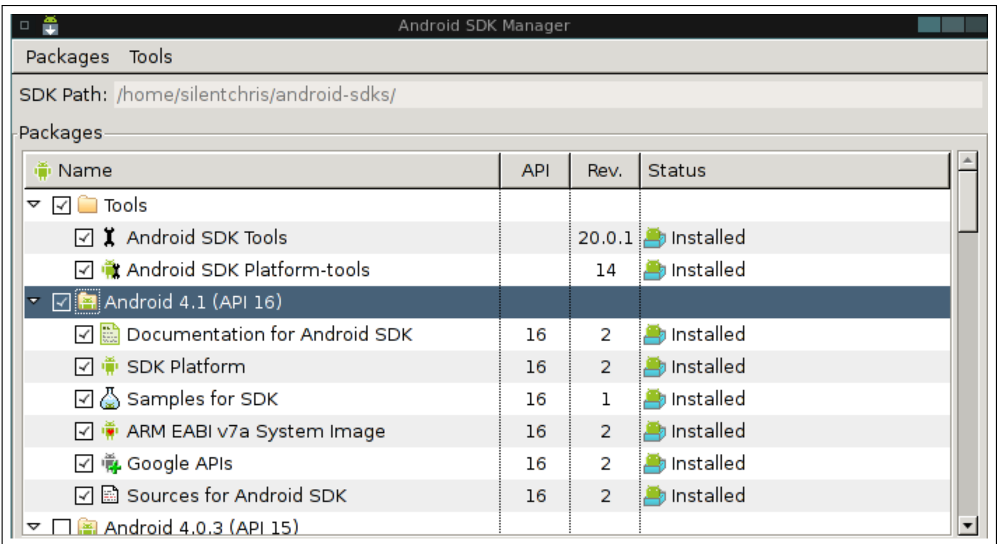


Abbildung 3.3: SDK installieren

Nach dem die Installation des SDKs und der dazu gehörigen Tools beendet ist. Sind alle benötigten Installationen getätigt, um mit Eclipse die ersten Android-Applikationen zu entwickeln.

3.2 Emulator

Mit der Installation des SDKs wurde auch ein Emulator installiert, welcher für viele Testszenarien ausreichend ist. Einige Situationen können leider nicht im Emulator getestet werden. Darunter fällt das Verhalten der Applikation im Standbymodus des Handys, da diese Funktionalität im Emulator nicht unterstützt wird.[2] Um auch ältere Versionen zu testen, müssen die jeweiligen SDK Versionen installiert werden. (siehe Abbildung 3.3 auf der vorherigen Seite) Ist dies gemacht, kann für jede Androidversion ein eigener Emulator oder auch AVD genannt erstellt werden. (siehe Abbildung 3.4 auf der nächsten Seite)

Wenn unter Linux entwickelt wird und es zu Problemen bei der Verbindung der AVD oder Handys kommt, könnte Folgendes helfen:

```
  ./adb kill-server
* service udev stop
  ./adb start-server
  ./adb devices
```

* Dieser Service sollte später wieder gestartet werden, wenn die automatische Erkennung von Geräten erwünscht ist.[3]

[2] http://developer.android.com/tools/help/emulator.html
[3] udev http://de.wikipedia.org/wiki/Udev

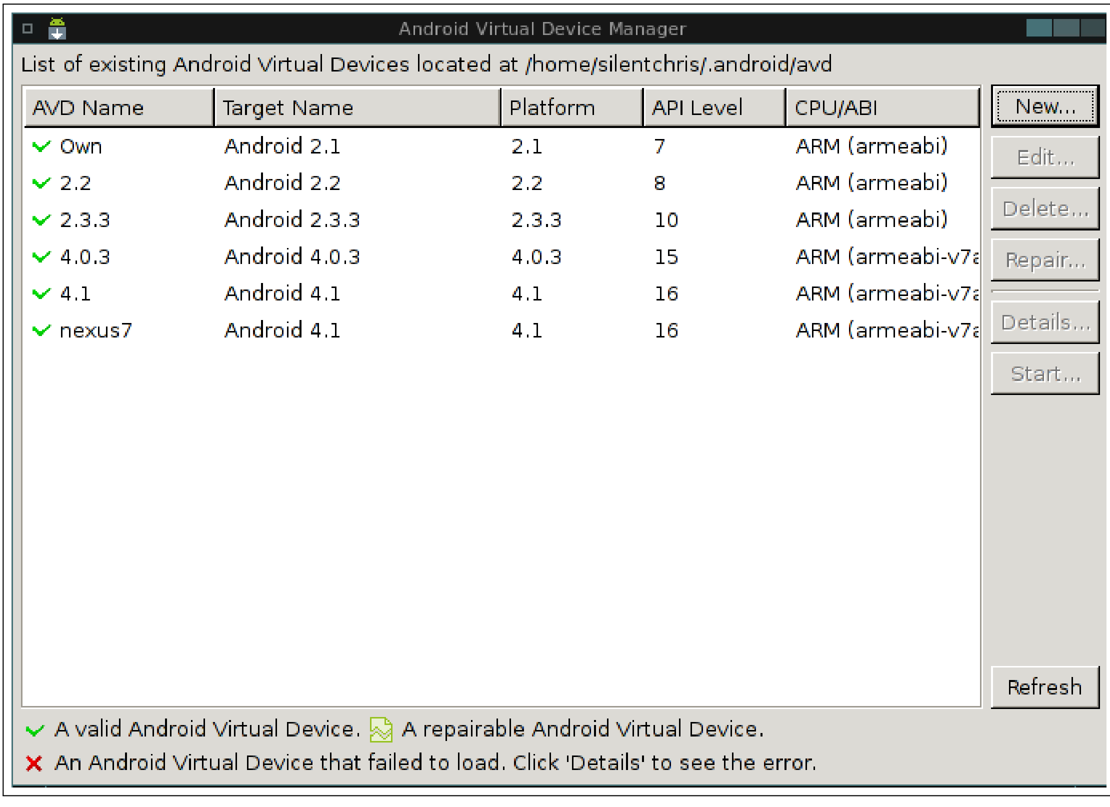


Abbildung 3.4: Emulator AVD

4 Oberfläche

Unter Android bestehen mehrere Möglichkeiten der Oberflächengestaltung. Die Oberfläche kann entweder vollständig dynamisch ohne XML Dateien oder fast ausschließlich mit XML Dateien realisiert werden. In diesem Kapitel werden mehrere Möglichkeiten genannt. Im ersten Teil werden grundlegende Themen, wie die Projektstruktur oder Internationalisierung sowie die Verwendung von Bildern behandelt. Danach werden im zweiten Teil verschiedene grundlegende Layouts vorgestellt. Des weiteren werden die Möglichkeiten zum Erstellen eines statischen sowie eines dynamischen Layouts erläutert.

4.1 XML

In Android werden viele Eigenschaften der Oberfläche in XML Dateien gespeichert. Es ist zwar möglich Oberflächen im Java Code zu erstellen, doch sollte in den meisten Fällen davon abgesehen werden, da dies zu undurchsichtigem Code führt.

4.1.1 Layouts

Unter Android werden Layouts der Oberfläche unter einem extra Ordner *layout*, wie in Abbildung 4.1 zu sehen ist, angelegt. Als Beispiel für eine XML-Layoutdatei sei hier auf Abbildung 4.5 auf Seite 19 verwiesen. Es können auch verschiedene Layouts für verschiedene Geräteklassen erstellt werde.[1]

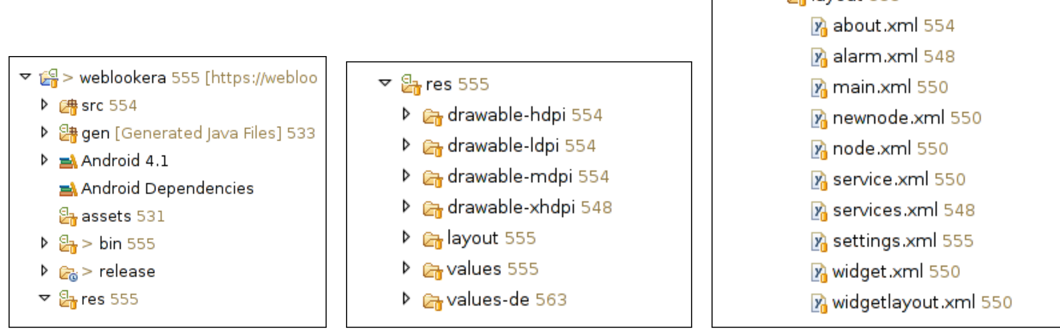


Abbildung 4.1: Projektordner

[1] `http://developer.android.com/guide/practices/screens_support.html`

4.1.2 Werte

Auch die Beschriftungsinformationen werden unter Android in XML Dateien gespeichert, wenn auch es möglich ist alle Oberflächen im Code zu beschriften. (siehe Abbildung 4.18 auf Seite 28)

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3
4     <string name="alarmSetting">Set the alarm for this node</string>
5     <string name="oneNode">Nodes which you can add</string>
6     <string name="statusicon">Status icon</string>
```

Abbildung 4.2: Strings XML

Die bessere Alternative ist die Strings in der *string.xml* zu speichern. Außer geordnetem Code hat dies einen weiteren großen Vorteil, der im Abschnitt 4.1.3 auf der nächsten Seite besprochen wird. In Abbildung 4.3 in Zeile 3 wird gezeigt, wie ein String aus der XML-Datei im Code verwendet wird. Dagegen ist in Abbildung 4.4 auf der nächsten Seite in Zeile 16 zu sehen, wie Strings in einer XML-Datei statisch verwendet werden.

```
1 /* Send Notification*/
2 int icon = android.R.drawable.stat_notify_error;
3 CharSequence tickerText = getResources().getText(R.string.alarmF)+db.nodes[i];
4 long when = System.currentTimeMillis();
```

Abbildung 4.3: Verwendung von Strings im Code

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical" >

   <TextView
        android:id="@+id/host_t"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:text="@string/adrServ"
        />
```

Abbildung 4.4: Verwendung von Strings in XML

4.1.3 Mehrsprachig

Da eine App im *Goggle-Play-Store* eine große Anwenderzahl erreicht, sollte darüber nachgedacht werden eine App mehrsprachig auszulegen. Dies ist unter Android einfach zu erreichen. Es müssen einfach wie in Abbildung 4.1 auf Seite 16 zwei values Ordner angelegt werden. Der value Ordner ohne Länderkennung ist die Standardsprache welche verwendet wird, wenn die gewünschte Sprache nicht vorhanden ist. Die Sprache wird anhand der im Android-System eingestellten Standardsprache automatisch ausgewählt. Die anderen value Ordner müssen mit *values-Länderkennung* erstellt werden. In diesen Ordnern werden dann einfach die *string.xml* Dateien in verschieden Sprachen angelegt. Deswegen sollten, wenn möglich, alle Strings in der *string.xml* gespeichert werden.

4.2 Bilder

Die Verwendung von Bildern ist unter Android nicht so einfach wie unter einem Desktop System. Durch die großen Unterschiede der Bildschirmgröße beim Handy bis zum Tablet muss beachtet werden das Bildinhalte, welche auf dem Handy gut skalieren bei einem Tablet völlig unzureichend dargestellt werden könnten. Doch auch hier gibt es eine Lösung, wie in Abbildung 4.1 auf Seite 16 zu sehen ist. Es werden im Projekt verschieden Ordner für Bilder in verschiedenen Auflösungen erstellt.[2] Das Android-System entscheidet dann eigenständig, welche Bilder unter welchem Gerät verwendet werden.

[2] http://developer.android.com/guide/practices/screens_support.html

4.3 Layouts

Nachfolgend werden verschiedene Layouts[3] unter Android vorgestellt. Da die XML-Implementierungen sich meist sehr ähneln, werden nicht zu allen Layouts separate Code-Beispiele aufgezeigt.

4.3.1 Linear Layout

Das *Linear Layout* ist ein gut skalierendes Layout, welches horizontal sowie vertikal eingesetzt werden kann. Natürlich können auch mehrere Ebenen verschachtelt werden, wie in Abbildung 4.5 rechts zu sehen. Doch soll an dieser Stelle davor gewarnt werden, Layouts mit zu vielen Ebenen zu erstellen, da dies die Performance erheblich beeinträchtigt.

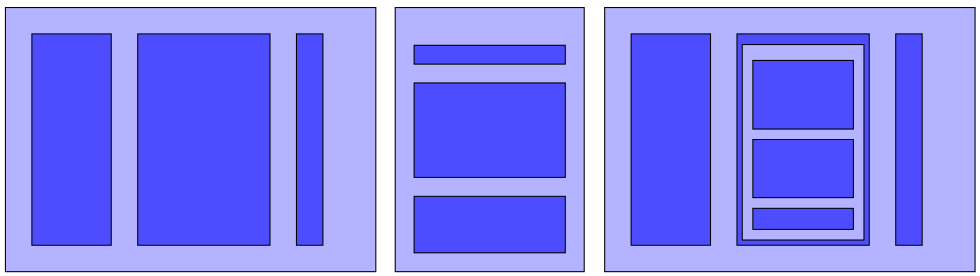


Abbildung 4.5: Linear Layout

Nachfolgend in Abbildung 4.6 auf der nächsten Seite ein Code Beispiel dazu. In Zeile 2 wird das Linear Layout deklariert, welches bis zur Zeile 28 reicht. Interessant sind die Zeilen 3 und 4, welche in den meisten weiteren Layouts auch existieren. Das ist in Zeile 17 und 18 zu erkennen. Für diese Zeilen gibt es 3 Werte:

1. FILL_PARENT
2. MATCH_PARENT
3. WRAP_CONTENT

[3] http://developer.android.com/guide/topics/ui/declaring-layout.html

Wobei Nr.2 nur eine neue Version von Nr.1 ist, was den effektiven Wertebereich auf zwei unterschiedliche Werte einschränkt. Nr.2 und Nr.1 bewirken, wie es schon der Name sagt, dass das Eltern Element ausgefüllt wird. Anders sieht dies bei Nr.3 aus, welches nur den Platz in Anspruch nimmt, den sein Inhalt erfordert.

Da die *ScrollView* nicht extra behandelt wird, sei diese hier kurz erwähnt. Da es im mobile Bereich Displays in verschiedensten Variationen gibt und nicht jeder Benutzer ein 4Zoll+ Display besitzt. Ist die ScrollView[4] eine gute Absicherung, dass auch Benutzer mit kleinem Display Inhalte vollends sehen können, auch wenn etwas umständlicher. Natürlich könnten auch verschiedene Layouts für diese Gerätekategorien erstellt werden. (siehe Abschnitt 4.1.1 auf Seite 16)

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:layout_width="wrap_content"
4      android:layout_height="wrap_content"
5      android:orientation="horizontal" >
6
7       <ImageView
8       android:layout_width="35dp"
9           android:layout_height="35dp"
10      android:src="@drawable/nodata"
11      android:layout_margin="5dp"
12      android:id="@+id/status"
13      android:contentDescription="@string/statusicon"
14      />
15
16     <HorizontalScrollView xmlns:android="http://schemas.android.com/apk/res/
           android"
17         android:layout_width="fill_parent"
18         android:layout_height="wrap_content"
19         android:overScrollMode="always">
20              <TextView
21              android:layout_width="wrap_content"
22              android:layout_height="50dp"
23              android:textSize="35dp"
24              android:id="@+id/txt"
25              android:onClick="onClick"
26              android:clickable="true"/>
27     </HorizontalScrollView>
28 </LinearLayout>
```

Abbildung 4.6: Linear Layout node.xml

[4]http://developer.android.com/reference/android/widget/ScrollView.html

4.3.2 Absolute Layout

Wie der Name schon sagt, werden die Elemente bei diesem Layout *absolut* positioniert. Dies hat zur Folge, dass dieses Layout schwer zu handhaben ist, wie es in Abbildung 4.7 auf der rechten Seite, welches ein kleineres Display darstellen soll, zu sehen ist. Deswegen wird es im mobilen Umfeld eher weniger verwendet.[5]

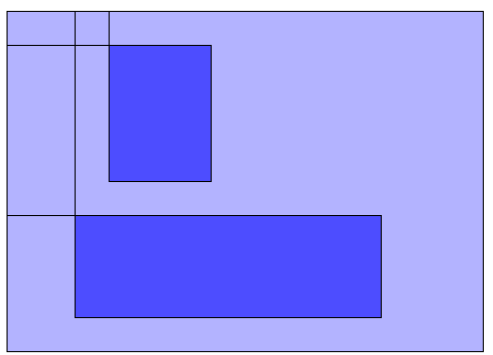
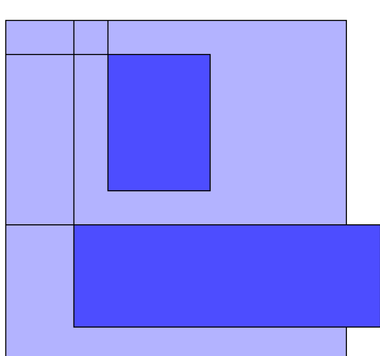



Abbildung 4.7: Absolute Layout

4.3.3 Relative Layout

Das *relative Layout*[6] ist ein für den mobile Bereich gut geeignetes Layout da es auf Displays verschiedenster Größe gut skaliert. Wie der Name es schon andeutet, werden bei diesem Layouts Elemente relativ zueinander angeordnet. Dies bedeutet, dass Positionsdaten wie z. B. ein Abstand von oben am darüberliegenden Element errechnet werden.

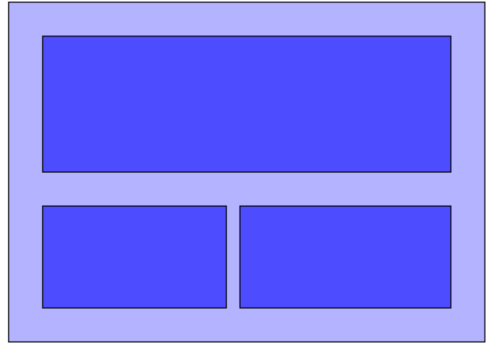


Abbildung 4.8: Relative Layout

[5] http://developer.android.com/reference/android/widget/AbsoluteLayout.html
[6] http://developer.android.com/reference/android/widget/RelativeLayout.html

4.3.4 Web View

Die nachfolgenden Layouts sind keine reinen Layouts mehr, sondern stellen eine Mischung zwischen View und Layout dar. Sie stellen ein fertiges Layout, für verschiedene Arten von Informationen, zur Verfügung.

Zu dieser View muss nicht viel gesagt werden, da sie wie der Name es sagt Web-Inhalte darstellen kann.

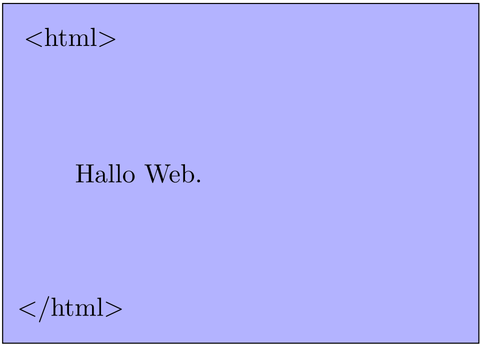


Abbildung 4.9: Web View

4.3.5 List View

Die *List View*[7] ist gut für größere Auflistungen geeignet, da sie automatisch vertikal scrollbar ist, sobald der Inhalt dies erforderlich macht. Wenn andere Layouts scrollbar gemacht werden sollen, siehe Abbildung 4.6 auf Seite 20.

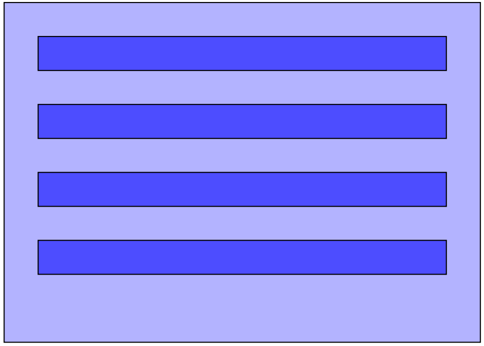


Abbildung 4.10: List View

[7] `http://developer.android.com/reference/android/widget/ListView.html`

4.3.6 Grid View

Bei dieser View werden die Elemente in einem Gitter angezeigt, Scrollbars werden automatisch angezeigt wenn nötig.[8]

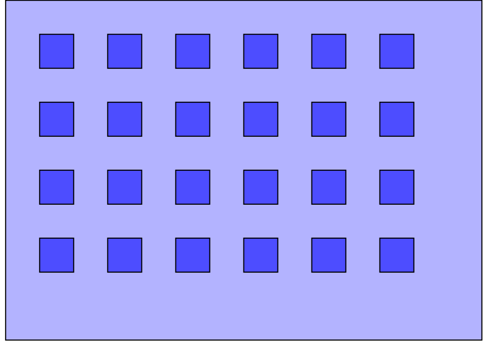


Abbildung 4.11: Grid View

[8] http://developer.android.com/reference/android/widget/GridView.html

4.4 Menüs

Im nachfolgenden Kapitel werden zwei wichtige Menüarten unter Android vorgestellt das *Context Menu* und das *Option Menu*.

4.4.1 Context Menu

Das *Context Menu*[9] wird bei einem langen Klick auf ein klickbares Element geöffnet siehe Abbildung 4.12.

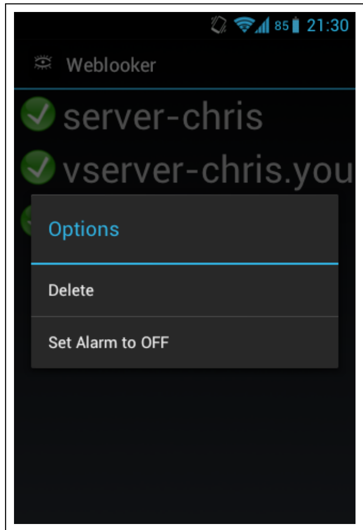


Abbildung 4.12: Context Menu

[9] http://developer.android.com/reference/android/view/ContextMenu.html

Folgend ist zu sehen, wie das *Context Menu* erstellt wird, sobald ein langer Klick auf ein klickbares Element ausgeführt wird. Weiter ist in Zeile 4 ersichtlich, dass der überschriebenen Methode die Referenz auf die View mit übergeben wurde, von der das *Context Menu* aufgerufen wurde. Somit können Informationen über diese ausgelesen werden. In Zeile 5 bis 7 wird dann das sichtbare Menü erstellt.

```
1  public void onCreateContextMenu(ContextMenu menu, View v,ContextMenuInfo
       menuInfo)
2  {
3      super.onCreateContextMenu(menu, v, menuInfo);
4      selectedNode = ((TextView)v.findViewById(R.id.txt)).getText().toString();
5      menu.setHeaderTitle("Options");
6      menu.add(0, v.getId(), 0, "Delete");
7      menu.add(1, v.getId(), 1, "Set Alarm to "+db_call.
           getChangeAlarmStatusOfNode(selectedNode));
8  }
```

Abbildung 4.13: Context Menu Code 1

Wurde ein Element des *Context Menus* angeklickt, wird die nachfolgende Methode ausgeführt. In der, durch den übergebenen Parameter *item*, kann entschieden werden, welche Option der Benutzer gewählt hat, um darauf die passende Aktion durchzuführen. Interessierten Lesern, denen der Befehl *Toast*[10] aufgefallen ist sei kurz gesagt das *Toast* eine Nachricht auf dem Display ausgibt. (siehe Abbildung 4.15)

```
1  public boolean onContextItemSelected(MenuItem item)
2  {
3   if(item.toString().equals("Delete"))
4   {
5          /* Delete entry */
6          db_call.deleteNodeFromSelected(selectedNode);
7          Toast.makeText(this,"Delete␣"+selectedNode, Toast.LENGTH_SHORT).show();
8   }
9   else
10  {
11         /* Edit Entry */
12         Toast.makeText(this,"Change␣alarm␣to␣"+ db_call.changeAlarm(selectedNode),
               Toast.LENGTH_SHORT).show();
13  }
14
15  refreshView();
16  return super.onContextItemSelected(item);
17 }
```

Abbildung 4.14: Context Menu Code 2

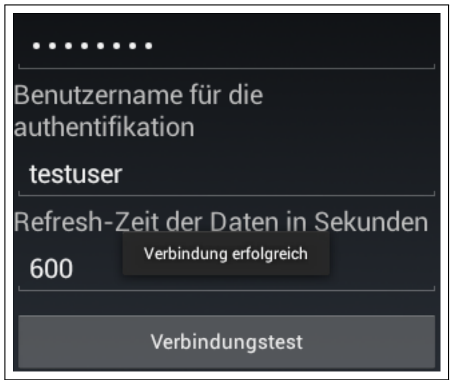


Abbildung 4.15: Toast

[10]Toast http://developer.android.com/guide/topics/ui/notifiers/toasts.html

4.4.2 Option Menu

Das *Option Menu*[11] wird durch das Betätigen einer physisch vorhandene Taste ausgelöst oder in neueren Versionen des Android-Systems, wie in Abbildung 4.17 oben rechts zu sehen, durch eine *on-screen* Menütaste. Das Menü muss für jede Activity extra angelegt werden und ist nicht App-übergreifend.

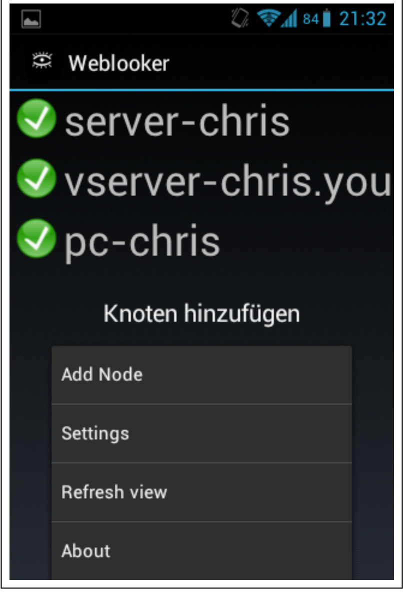


Abbildung 4.16: Option Menu

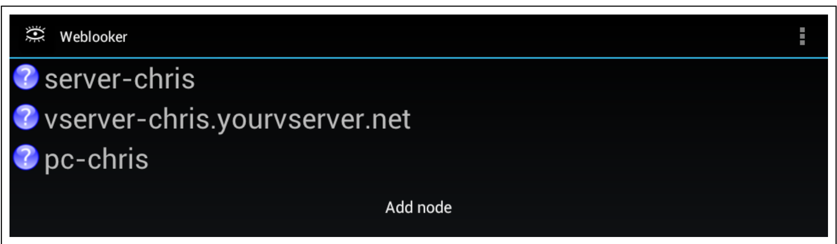


Abbildung 4.17: Option Menu 2

[11] http://developer.android.com/guide/topics/ui/menus.html#options-menu

Wie beim *Context Menu* wird hier das *Option Menu* beim Aufrufen erstellt und die Menü Einträge festgelegt.

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
        menu.add(0,AD_ID,Menu.NONE,"Add Node").setIcon(android.R.drawable.
            ic_menu_add);
        menu.add(0,SE_ID,Menu.NONE,"Settings").setIcon(android.R.drawable.
            ic_menu_preferences);
        menu.add(0,RE_ID,Menu.NONE,"Refresh view").setIcon(android.R.drawable.
            ic_menu_view);
        menu.add(0,AB_ID,Menu.NONE,"About").setIcon(android.R.drawable.
            ic_menu_info_details);

        return super.onCreateOptionsMenu(menu);
}
```

Abbildung 4.18: Option Menu Code 1

Nach dem eine Option ausgewählt wurde, wird die nachstehende Methode aufgerufen, in der nachfolgend die jeweiligen Aktionen für den Menüeintrag ausgeführt werden. In diesem Beispiel wird eine andere *Activity* (siehe Abschnitt 4.7 auf Seite 37) mit Hilfe eines Intents (siehe Abschnitt 2.4 auf Seite 8) aufgerufen.

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item)
    {
        switch (item.getItemId())
                {
                case AD_ID:
                        final Intent node = new Intent(this,
                            WeblookerActivityAddNode.class);
                        startActivity(node);
                        break;
                case AB_ID:
```

Abbildung 4.19: Option Menu Code 2

4.5 Dynamische Oberflächen

Definitionen:

ViewGroup:
Eine besondere Art der View, welche mehrere Kinder Views enthalten kann.

LayoutInflater:
Wird verwendet um eine grafische Bedienoberfläche dynamisch, aus vordefinierten XML-Teilen, zusammenzusetzen.

SQLite:
Relationales Datenbanksystem, welches für den Einsatz im Embedded Bereich entworfen ist.

Auch wenn mit XML-Dateien viel gemacht werden kann, ist es manchmal doch nötig eine GUI dynamisch bzw. *teilweise dynamisch* zu erstellen. Auch dies ist möglich und soll nachfolgend anhand eines Beispiels verdeutlicht werden. In diesem Beispiel wird dafür eine *ViewGroup*[12] sowie der *LayoutInflater*[13] verwendet.

In Abbildung 4.20 auf der nächsten Seite ist ein Teil der main.xml zu sehen, welche zur *WeblookeraActivity* gehört. Diese wird der Activity, wie in Abbildung 4.21 auf der nächsten Seite in Zeile 3 zu sehen ist, zugewiesen. Interessant sind an dieser Stelle die Zeilen 13 bis 18 in der main.xml. Wie der Name schon andeutet, ist dies bloß ein Container für den dynamisch erstellten Inhalt.

In Abbildung 4.22 auf Seite 31 ist zu sehen, wie der dynamische Inhalt aus der *WeblookeraActivity* erstellt wird. Hier werden Daten aus einer SQLite Datenbank (siehe Abschnitt 6.1 auf Seite 45) ausgelesen und somit dynamisch der Inhalt der Activity erstellt. Um diesen Inhalt zu erstellen, wird mit dem *LayoutInflater* eine weitere XML-Datei geladen, welche mit Daten gefüllt wird. In diesem Fall die node.xml (siehe Abbildung 4.5 auf Seite 19). Nach dem diese mit Daten gefüllt wurde, wird die erstellte View in Zeile 25 der *ViewGroup* hinzugefügt.

[12] http://developer.android.com/reference/android/view/ViewGroup.html
[13] http://developer.android.com/reference/android/view/LayoutInflater.html

```xml
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:overScrollMode="always">

        <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:id="@+id/main">

                <LinearLayout xmlns:android="http://schemas.android.com/apk/res/
                    android"
                    android:layout_width="fill_parent"
                    android:layout_height="wrap_content"
                    android:orientation="vertical"
                    android:id="@+id/main_container">
                </LinearLayout>
```

Abbildung 4.20: Kontainer main.xml

```java
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
```

Abbildung 4.21: Layout zuweisen

```java
protected void addViews()
{
 if(db_call.getNodes() == true)
  if(db_call.nodes != null)
        for(int i = 0;i < db_call.nodes.length;i++)
        {
                view = LayoutInflater.from(getBaseContext()).inflate(R.layout.node
                    ,null);
                if(db_call.nodes_badStatus[i] == 1)
                        ((ImageView)view.findViewById(R.id.status)).
                            setImageResource(R.drawable.ok);
                else if(db_call.nodes_badStatus[i] == 2)
                        ((ImageView)view.findViewById(R.id.status)).
                            setImageResource(R.drawable.off);
                else if(db_call.nodes_badStatus[i] == 3)
                        ((ImageView)view.findViewById(R.id.status)).
                            setImageResource(R.drawable.away);


                /* Get textview for node name */
                tview = ((TextView)view.findViewById(R.id.txt));

                /* Menu on long click*/
                registerForContextMenu(tview);

                /* Set node name */
                tview.setText(db_call.nodes[i]);

                parent.addView(view);
        }
}
```

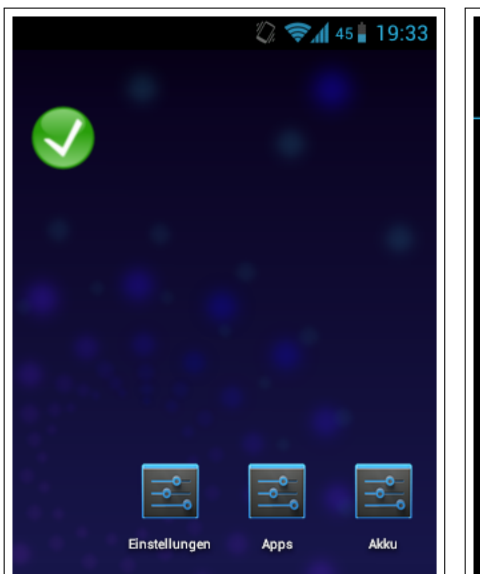
Abbildung 4.22: Layout dynamisch füllen

4.6 Widgets

Definitionen:

AlarmManager:
Wird verwendet um periodische Aktionen durchzuführen, selbst wenn das Gerät im Ruhemodus versetzt ist.

Widgets[14] sind kleine Programme, die unter Android in andere Programme integriert werden können. Genannt sei hier der meist gebräuchlichste Verwendungszweck, der *Home screen*, wie in Abbildung 4.23 zu sehen ist. Nachfolgend wird anhand eines Beispiels der Aufbau eines Widgets erläutert.



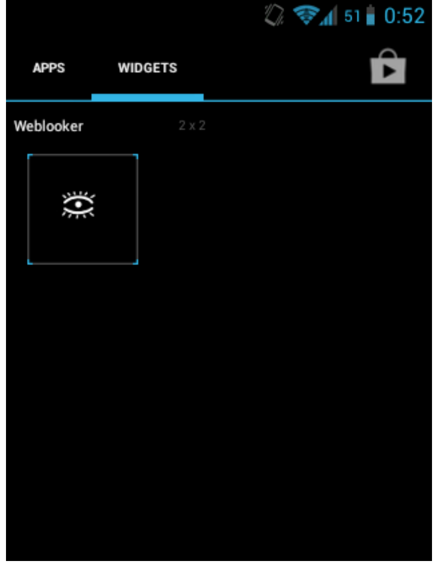


Abbildung 4.23: Widget

Als Erstes der Eintrag in der *Manifest-Datei*. Wie zu sehen ist, unterscheidet sich dieser Eintrag fast nicht von dem der *WeblookeraActivity* (siehe Abbildung 2.4 auf Seite 7).

```
1         <activity android:name=".widget.weblooekrWidget"
2                                   android:label="@string/app_name">
3          <intent-filter>
4           <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
5                   <category android:name="android.intent.category.LAUNCHER" />
6          </intent-filter>
```

Abbildung 4.24: Widget-Manifesteintrag

[14] http://developer.android.com/guide/topics/appwidgets/index.html

Das nachfolgende XML spezifiziert die grundlegenden Daten des Widget, wie die minimale Größe und das Initial Layout des Widgets.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
3     android:minWidth="72dp"
4     android:minHeight="72dp"
5     android:initialLayout="@layout/widgetlayout">
6 </appwidget-provider>
```

Abbildung 4.25: widget.xml

Nachfolgend, in Abbildung 4.26 auf der nächsten Seite, ist der Kern des *Widgets* zu sehen. Die Methode *onReceive()* wird durch den im Manifest angelegten *Intentfilter* aufgerufen sobald der passende Intent versendet wurde. In der *onReceive()* wird danach der Intent abgefertigt und wenn nötig die *onUpdate()* Methode aufgerufen. Da in diesem Beispiel das Widget mit dem Hintergrundservice aktualisiert wird, der durch einen AlarmManager (siehe Abschnitt 5.3 auf Seite 44) aufgerufen wird, musste die *onReceive()* Methode überschrieben werden. In Zeile 22 wird eine Instanz des *AppWidgetManager* geholt, der Informationen zu vorhanden Widget Instanzen bereitstellt. Da ein Widget in mehreren Instanzen auf dem *Homescreen* vorhanden sein kann, wird für jede Instanz die Id ermittelt und folgend die *updateWidget()* Methode, die in Abbildung 4.27 auf Seite 35 zu sehen ist, aufgerufen.

Diese Methode dient nur zur Ordnung und muss nicht erstellt werden. Wenn es gewünscht ist, kann alles was in dieser Methode passiert auch in der *onReceive()* durchgeführt werden. Nun werden alle Instanzen des *Widgets* erneuert. In der Zeile 8 bis 10 wird ein Intent für das Widget erstellt, der die *WeblookeraActivity* aufruft, wenn der Intent abgeschickt wird. In Zeile 13 wird das Layout für das Widget geladen, das in Abbildung 4.28 auf Seite 36 zu sehen ist. Danach wird in Zeile 14 der Intent gesetzt, damit beim Klick auf das Widget der Intent verschickt wird. In den Zeilen 16 bis 23 wird das Layout mit Daten befüllt. Nachdem dies alles erledigt ist, wird das Update in Zeile 24 an das Widget gesendet.

Aber wie wird das Widget nun erneuert? Dazu sollte nun die Methode in Abbildung 4.29 auf Seite 36 betrachtet werden. Diese wird mit dem Service, der regelmäßig ausgeführt wird, aufgerufen. In Zeile 3 bis 7 werden Daten für das Widget aus der SQLite Datenbank ausgelesen. Folgend wird in Zeile 10 bis 11 ein expliziter Intent mit zusätzlichen Daten erstellt (siehe Abschnitt 2.4 auf Seite 8) und in Zeile 14 als Broadcast gesendet. Wer sich für dieses Beispiel im Ganzen interessiert, der sei hier auf das Repository auf sourceforge.org verwiesen.[15]

[15] https://weblooker.svn.sourceforge.net/svnroot/weblooker/tags/weblookera_1.0

```java
public class WeblookerWidget extends AppWidgetProvider
{
 SimpleDateFormat df = new SimpleDateFormat("hh:mm:ss");
 private static int status = 0;
 public static final String ACTION_EXTRAS= "extras";

 @Override
 public void onUpdate(Context context, AppWidgetManager appWidgetManager,int[]
     appWidgetIds)
 {

 }

 @Override
 public void onReceive(Context context, Intent intent)
 {
        super.onReceive(context, intent);

        if(intent != null)
        {
         status = intent.getIntExtra(ACTION_EXTRAS,0);

         AppWidgetManager am = AppWidgetManager.getInstance(context);
         if(am != null)
         {
          int[] appWidgetIds = am.getAppWidgetIds(new ComponentName(context,
              getClass()));

          if(appWidgetIds != null && appWidgetIds.length > 0)
                  updateWidget(context,am,appWidgetIds);
         }
        }
 }
```

Abbildung 4.26: Widget Code 1

```
1  private void updateWidget(Context context,AppWidgetManager am,int[] appWidgetIds )
2  {
3      final int N = appWidgetIds.length;
4      for (int i = 0; i < N; i++)
5      {
6       int appWidgetId = appWidgetIds[i];
7
8       /* On click to main activity */
9       Intent intent = new Intent(context, WeblookeraActivity.class);
10      PendingIntent pendingIntent = PendingIntent.getActivity(context, 0, intent,
           0);
11
12      /* Get view of widget */
13      RemoteViews views = new RemoteViews(context.getPackageName(),R.layout.
           widgetlayout);
14      views.setOnClickPendingIntent(R.id.image_widget, pendingIntent);
15
16      if(status == 3)
17         views.setImageViewResource(R.id.image_widget,  R.drawable.away);
18      else if(status == 2)
19         views.setImageViewResource(R.id.image_widget, R.drawable.off);
20      else if(status == 1)
21         views.setImageViewResource(R.id.image_widget, R.drawable.ok);
22      else
23         views.setImageViewResource(R.id.image_widget, R.drawable.nodata);
24      am.updateAppWidget(appWidgetId, views);
25     }
26 }
```

Abbildung 4.27: Widget Code 2

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="center"
  android:layout_margin="4dp"
  android:background="#00000000">


    <ImageView android:id="@+id/image_widget"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/nodata"
            android:clickable="true"
            android:contentDescription="@string/app_name"
            android:background="#00000000"
            />
</LinearLayout>
```

Abbildung 4.28: widgetlayout.xml

```java
private void widgetUpdate()
{
  int value;
  /* Get status */
  WeblookerViewDBCalles db = new WeblookerViewDBCalles(getApplicationContext());
  value = db.getStatusForWidget();
  db.onDestroy();

  /* Create Intent with data*/
  Intent i = new Intent(this, WeblookerWidget.class);
  i.putExtra(WeblookerWidget.ACTION_EXTRAS, value);

  /* Send broadcast to widget */
  sendBroadcast(i);
}
```

Abbildung 4.29: Widget-Aufruf

4.7 Activities

Activities[16] sind die Komponenten unter Android, die normalerweise zum Erstellen von Oberflächen verwendet werden. Jede *Activity* hat beim Start den kompletten Bildschirm zur Verfügung, um Inhalte darzustellen und mit dem Benutzer zu kommunizieren. Jede App hat eine *Main Activity*, die beim Starten der App aufgerufen wird. In diesem Fall ist es die *WeblookeraActivity*. Weitere *Activities* können durch eigene Intents aufgerufen werden.

Werden weitere *Activities* aufgerufen, ist der *Lifecycle* (siehe Abbildung 2.8 auf Seite 11) zu beachten.

[16] `http://developer.android.com/reference/android/app/Activity.html`

5 Services

Definitionen:

PowerManager:
Kann das Energiemanagement des Gerätes auf verschiedenen Arten beeinflussen.

Services[1] unter Android sind eine nicht ganz triviale Angelegenheit. Da Android ein mobiles Betriebssystem darstellt, ist es nicht garantiert, dass ein einmal gestarteter Service dauerhaft läuft. Es ist eher sehr wahrscheinlich, dass dieser irgendwann beendet wird. Sei es durch Standby des Handys oder durch Ressourcenmangel. Außerdem spielt beim Erstellen eines Services eine weitere Ressource, der Akku neben Speicher und CPU, eine große Rolle. Wird ein Service zu exzessiv genutzt sowie *AlarmManager* (siehe Abschnitt 5.3 auf Seite 44) und *PowerManager* (siehe Abschnitt 5.4 auf Seite 44) zum Lebenserhalt des Service verwendet, kann dies einen beträchtlichen Verbrauch an Energie hervorrufen.

5.1 Arten

Unter Android gibt es grob aufgeteilt zwei Arten von *Services*, den *gebundenen Service* und den *remote Service*, die nachfolgend kurz aufgezeigt werden.

5.1.1 Gebundener Service

Ein gebundener Service läuft nur so lange, wie andere Komponenten diesen Service an sich *gebunden* haben. Sie werden meist verwendet, wenn Interaktionen zwischen Komponenten zum Beispiel einer Activity und einem Service benötigt werden. Mehr Informationen entnehmen Sie der Android Dokumentation.[1]

[1] `http://developer.android.com/guide/components/services.html`

5.1.2 Remote Service

Definitionen:

BroadcastReceiver:
Empfängt systemweit gesendete Intents, die von anderen Applikationen über sendBroadcast() oder vom System selbst gesendet wurden.

Ein Remote Service kann theoretisch „ewig“ laufen. Er kann von einer Activity gestartet werden oder durch einen Intent, der von einem BroadcastReceiver (siehe Abschnitt 5.2 auf Seite 42) ausgelöst wurde, gestartet werden. Entweder er teilt sich einen Pozess mit dem rest der Applikation oder er wird in einem seperaten Prozess gestartet, wodurch die allgemeine Stabilität erhöht wird. Diese Art des Services werden wir uns nachfolgen näher betrachten.

In folgender Abbildung 5.1 ist der Ausschnitt der Manifestdatei dargestellt, der den Eintrag für den Service darstellt. Interessant sind dabei die Zeilen 2 und 3, die restlichen Zeilen entsprechen den bereits besprochenen Einträgen (siehe Abschnitt 2.4 auf Seite 8). In Zeile 2 wird der Service so konfiguriert, dass er *privat* ist. Dies bedeutet er ist nur von unserer App ansprechbar. In Zeile 3 wird gesagt, wie der Service ausgeführt werden soll. In diesem Beispiel wird der Service in einem eigenen privaten Prozess ausgeführt.[2]

Die drei Eigenschaften des Namens, die dies spezifizieren, sind nachfolgend gelistet:

Eigener Prozess „:“
Privater Xname
Globale xname

Durch einen anführenden Doppelpunkt, wird ein eigener Prozess verwendet. Wird der erste Buchstabe großgeschrieben, ist der Service privat. Wird er dagegen kleingeschrieben, ist der Service global.

```
1        <service android:name="weblooker.andro.net.NetService"
2                 android:exported="false"
3                 android:process=":NetService">
4                 <intent-filter>
5                        <action android:name="weblooker.andro.net.NetService" />
6                 </intent-filter>
7        </service>
```

Abbildung 5.1: Service-Manifesteintrag

[2] http://developer.android.com/guide/topics/manifest/application-element.html#proc

Nun zum eigentlichen Service. Wichtig ist hierbei, dass dieser von der Klasse *Service* erbt. Als Nächstes müssen mehrere Methoden überschrieben werden, die nicht alle gelistet sind, da hier der *Remote service* behandelt wird. Interessierte Leser seien hier wieder auf Sourceforge.net verwiesen.[3] Die *onStartCommand()* wird, wie es der Name schon andeutet, beim Aufruf des Services ausgeführt. In dieser Methode sollte folgend für länger dauernde Aktionen ein Thread erstellt werden. Wichtig in dieser Methode ist ebenfalls der Rückgabewert, welcher hier *START_STICKY*[4] ist. Dieser Rückgabewert macht bei einem Service, der periodisch explizit aufgerufen wird, Sinn. Da garantiert ist, dass die *onStartCommand()* Methode beim Erstellen der nächsten Service Instanz aufgerufen wird. Im nächsten Abschnitt wird behandelt, wie der Service aufgerufen wird.

[3] http://weblooker.svn.sourceforge.net/viewvc/weblooker/tags/weblookera_1.0/source/weblooker/andro/net/NetService.java?revision=561&view=markup

[4] http://developer.android.com/reference/android/app/Service.html#START_STICKY

```
1  public class NetService extends Service
2  {
3          @Override
4          public int onStartCommand(Intent intent, int flags, int startId)
5          {
6                  super.onStartCommand(intent, flags, startId);
7                  lockS.acquire();
8                  /* Check for Internet connection*/
9                  ConnectivityManager cm =(ConnectivityManager) getSystemService(
                       Context.CONNECTIVITY_SERVICE);
10                 if (!( cm.getActiveNetworkInfo() != null &&  cm.
                       getActiveNetworkInfo().isConnectedOrConnecting()))
11                 {
12                  widgetUpdate();
13                  lockS.release();
14                  return START_STICKY;
15                 }
16                 /* Start service at thread */
17                 new Thread(new Runnable() {
18                  public void run()
19                  {
20                         try{
21                                 if(readLoginData() == false)
22                                         return;
23                                 /* Start service */
24                                 if(service() == true)
25                                 {
26                                         alarm();
27                                         widgetUpdate();
28                                 }
29                                 /* Say goodbye to server */
30                                 mm.endSession();
31                         }catch(Exception e){
32                                 e.printStackTrace();
33                         }
34                         finally{
35                                 lockS.release();
36                         }
37                  }
38                 }).start();
39                 return START_STICKY;
40         }
```

Abbildung 5.2: Service Code

5.2 Broadcast receiver

Broadcast receiver empfangen Broadcasts vom System oder anderen Applikationen, selbst wenn der Rest der Applikation nicht aktiv ist. Dies macht ihn dafür geeignet Services beim Start des Systems aufzurufen.

In Abbildung 5.3 ist der Manifesteintrag des *Broadcast receivers* zu sehen. Interessant sind hier Zeile 2 und 3, in Zeile 2 wird festgelegt, ob das System den *Broadcast receiver* instanziieren kann. In Zeile 3 wird festgelegt, ob der *Broadcast receiver* auf Broadcasts, die von außerhalb der eigenen Applikation kommen, reagieren soll.

```
1 <receiver android:name="weblooker.andro.net.StartupNetService"
2           android:enabled="true"
3           android:exported="true">
4         <intent-filter>
5                 <action android:name="android.intent.action.BOOT_COMPLETED" />
6                 <category android:name="android.intent.category.DEFAULT" />
7         </intent-filter>
8 </receiver>
```

Abbildung 5.3: Broadcast receiver Manifesteintrag

In Abbildung 5.4 auf der nächsten Seite ist der *Broadcast receiver* aufgezeigt. Wichtig hierbei ist, dass dieser von der Klasse *BroadcastReceiver* erbt und die *onReceive()* überschrieben werden muss. In Zeile 9 wird auf den erwarteten Broadcast geprüft, wenn dieser eintrifft, wird der Service mit dem *AlarmManager* gestartet.

```java
public class StartupNetService extends BroadcastReceiver
{
 private static final int SECS = 1000;
 private static final boolean debug = false;

@Override
public void onReceive(Context arg0, Intent arg1)
{
 if ("android.intent.action.BOOT_COMPLETED".equals(arg1.getAction()))
 {
        /* Get timer */
        WeblookerViewDBCalles db = new WeblookerViewDBCalles(arg0);
        int time = Integer.parseInt(db.getRefresh());
        db.onDestroy();

        /* Start service with AlarmManager */
        /* Works always is device on sleep */
        Calendar cal = Calendar.getInstance();
        /* Start Service with Intent */
        Intent in = new Intent( arg0, NetService.class);
        PendingIntent pi = PendingIntent.getService( arg0, 0, in, PendingIntent.
            FLAG_UPDATE_CURRENT);
        AlarmManager alarms = (AlarmManager) arg0.getSystemService(Context.
            ALARM_SERVICE);
        /* Set Timer for wakeup */
        alarms.setRepeating(AlarmManager.RTC_WAKEUP, cal.getTimeInMillis(), time
            *  SECS, pi);
        if(debug)Toast.makeText(arg0,"Start service", Toast.LENGTH_LONG).show();
 }
}
```

Abbildung 5.4: Broadcast receiver Code

5.3 AlarmManager

Der *AlarmManager*[5] ist dafür da, Aktionen periodisch auszuführen. Ein großer Vorteil gegenüber anderen Methoden ist, dass der *AlarmManager* auch ausgeführt wird, wenn das Handy im Standby liegt, was normalerweise bedeutet, dass die CPU ausgeschaltet ist und kein Code ausgeführt wird. Deswegen ist der *AlarmManager* das richtige Werkzeug, wenn ein Service auch im Standby des Handys arbeiten soll, wie es bei einer Monitoring Applikation der Fall ist. In Abbildung 5.4 auf der vorherigen Seite ist in Zeile 18 bis 24 zu sehen, wie der *AlarmManager* angewandt wird. Doch damit ist nicht garantiert, dass der somit aufgerufen Service, seine Arbeit vollständig beenden kann.

5.4 PowerManager

Ein Service, der durch einen *AlarmManager* im Standby aufgerufen wird, hat nicht die Garantie, dass er genug CPU-Zeit bekommt, um seine Arbeit zu beenden. Es kann passieren, dass die CPU sich schon vor dem Beenden des Services wieder schlafen legt. Doch dafür gibt es eine Lösung: Der *PowerManager*. In Abbildung 5.5 wird aufgezeigt, wie man an eine Referenz kommt. Mit dem *PowerManager* können dem System explizit „Befehle“ zum Energiemanagement erteilt werden.[6] Es sollte an dieser Stelle aber erwähnt werden, dass durch extensive Verwendung des *PowerManagers* ein exorbitanter Energieverbrauch entstehen kann. Die Verwendung des *PowerManagers* wird in Abbildung 5.2 auf Seite 41 in Zeile 7 und 35 verdeutlicht.

```
1       @Override
2       public void onCreate()
3       {
4               this.getApplicationContext();
5
6               /* Get power Manager to save service before asleep */
7               pm = (PowerManager)getSystemService(Context.POWER_SERVICE);
8               lockS = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,LOCK);
```

Abbildung 5.5: PowerManager Code

[5]http://developer.android.com/reference/android/app/AlarmManager.html
[6]http://developer.android.com/reference/android/os/PowerManager.html

6 Persistente Datenhaltung

Unter Android gibt es verschiedene Möglichkeiten der *persistenten Datenhaltung*[1]. Nachfolgend werden mehrere davon vorgestellt.

6.1 SQLite

SQLite ist eine auf den mobilen Bereich optimierte Datenbank. Im Gegensatz zu den großen bekannten Datenbanksystemen fehlen viele Funktionalitäten. Diese Funktionalitäten werden jedoch im mobilen Bereich meist überhaupt nicht gebraucht, da die Datenbank unter Android meist nur zum Verwalten von kleinen Datenmengen gebraucht wird. Nachfolgend wird aufgezeigt, wie unter Android, *SQLite* verwendet wird.

In Abbildung 6.1 ist ein Vorschlag zu sehe, wie die Befehle zum Erstellen der Tabellen sauber realisiert werden können.

```
1  public final class WeblookerTbl
2  {
3   public static final String SQL_CREATE_CONNECT_DATAS =
4                 "CREATE␣TABLE␣connect_datas␣("+
5                 "_id␣INTEGER␣PRIMARY␣KEY␣AUTOINCREMENT,"+
6                 "type␣TEXT,"+
7                 "value␣TEXT);";
8
9   public static final String SQL_DROP_CONNECT_DATAS =
10                "DROP␣TABLE␣IF␣EXISTS␣connect_datas";
```

Abbildung 6.1: SQLite Tables

Wenn es darum geht, unter Android beim Update eine Applikation das Tabellenschema zu ändern, ist in Abbildung 6.2 auf der nächsten Seite eine wichtige Klasse zu sehen. Wichtig ist hier in Zeile 1, dass die eigene Klasse von *SQLiteOpenHelper*[2] abgeleitet wird. *SQLiteOpenHelper* hilft einem Entwickler beim Management verschiedener Versionen des Tabellenschemas. Wie in Zeile 8 zu sehen ist, erstellt der *SQLiteOpenHelper* für uns die Datenbank und übernimmt das Verwalten der Versionen. Wird eine Applikation zum ersten Mal installiert, wird die Datenbank angelegt und die *onCreate()* Methode aufgerufen, in der dann wie in Zeile 12 bis 35 zu sehen ist, die Tabellen angelegt werden können.

[1] http://developer.android.com/guide/topics/data/data-storage.html

[2] http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html

```
public class WeblookerDB extends  SQLiteOpenHelper
{
        private static final String DB_NAME = "weblooker.db";
        private static final int DB_VERSION     = 3;

        public WeblookerDB(Context context)
        {
                super(context, DB_NAME, null, DB_VERSION);
        }

@Override
public void onCreate(SQLiteDatabase db)
{
        /* Create Tables*/
        db.beginTransaction();
        try{
         db.execSQL(WeblookerTbl.SQL_CREATE_AVAILABLE_NODES);
         db.execSQL(WeblookerTbl.SQL_CREATE_AVAILABLE_SERVICE);
         db.execSQL(WeblookerTbl.SQL_CREATE_CONNECT_DATAS);
         db.execSQL(WeblookerTbl.SQL_CREATE_NODE_STATUS);
         db.execSQL(WeblookerTbl.SQL_CREATE_SELECTED_NODES);
         db.execSQL(WeblookerTbl.SQL_CREATE_SERVICE_STATUS);
         /* Insert init  data*/
         ContentValues content = new ContentValues();
         content.put("type","refresh");
         content.put("value","600");
         db.insert("connect_datas","_id", content);
         content.put("type","port");
         content.put("value","15001");
         db.insert("connect_datas","_id", content);
         db.setTransactionSuccessful();
        }finally{
        db.endTransaction();
        }
}
```

Abbildung 6.2: SQLite Datenbankschema erstellen

Wird die Applikation von einer früheren Version auf eine neue aktualisiert und die *DB_VERSION* wurde inkrementiert, wird vom *SQLiteOpenHelper* die in Abbildung 6.3 dargestellte Methode *onUpgrade()* aufgerufen. In dieser können die vorigen Tabellen gelöscht, geändert oder existierende Daten gesichert und nach dem Update des Tabellenschemas wieder zurückgespielt werden.

```
1  public void onUpgrade(SQLiteDatabase wr, int oldVersion, int newVersion)
2  {
3          String passwd   = "";
4          Cursor data;
5          /* Read login data */
6          data = wr.rawQuery("SELECT value FROM connect_datas WHERE type='passwd'",
               null);
7          if(data.getCount() != 0)
8          {
9                  data.moveToNext();
10                 passwd = data.getString(0);
11         }
12         data.close();
13                 .
14         /* Delete old tables*/
15         wr.execSQL(WeblookerTbl.SQL_DROP_AVAILABLE_NODES);
16         wr.execSQL(WeblookerTbl.SQL_DROP_AVAILABLE_SERVICE);
17         wr.execSQL(WeblookerTbl.SQL_DROP_CONNECT_DATAS);
18         wr.execSQL(WeblookerTbl.SQL_DROP_NODE_STATUS);
19         wr.execSQL(WeblookerTbl.SQL_DROP_SELECTED_NODES);
20         wr.execSQL(WeblookerTbl.SQL_DROP_SERVICE_STATUS);
21
22         /* Create new tables */
23         onCreate(wr);
24         /* Insert data*/
25         ContentValues content = new ContentValues();
26         content.put("type","passwd");
27         content.put("value",passwd);
28         wr.insert("connect_datas","_id", content);
29                 .
30 }
```

Abbildung 6.3: SQLite Datenbankschema aktualisieren

Nachstehend in Abbildung 6.4 wird aufgezeigt, wie über die *SQLiteOpenHelper* Klasse eine Verbindung zur Datenbank geöffnet wird.

```
1         public WeblookerViewDBCalles(Context context)
2         {
3                 /* Create connection to db */
4                 if(db == null)
5                  db = new WeblookerDB(context).getWritableDatabase();
6
7                  time = Integer.parseInt(getRefresh());
8
9                  this.context = context;
10        }
```

Abbildung 6.4: DB-Verbindung

Abbildung 6.5 zeigt einen einfachen „Select“ Befehl unter *SQLite* in Form eines *rawQerys*. Mit dem ersten Parameter wird das SQL-Statment übergeben. Das „?“ dient als Platzhalter für den im zweiten Parameter übergebenen Wert.

```
1  private boolean checkEntry(String type)
2  {
3   data = db.rawQuery("SELECT␣value␣FROM␣connect_datas␣WHERE␣type=␣?",new String[] {
       type});
4   if(data.getCount() == 0)
5   {
6          data.close();
7          return false;
8   }
9   data.close();
10  return true;
11 }
```

Abbildung 6.5: Select-Statment

Abbildung 6.6 zeigt in Zeile 3 bis 5, wie der Inhalt für ein Insert bzw. Update in *SQLite* vorbereitet wird. In Zeile 13 ist zu sehen, wie in SQLite ein *Insert* aufgebaut ist. Als ersten Parameter wird der Tabellenname übergeben, als zweiter Parameter können die Spalten angegeben werden welche mit NULL initialisiert werden sollen. Hier ist dies der Primärschlüssel, der mit *Auto increment* deklariert ist. Im dritten Parameter können „Where Klauseln“ mit Fragezeichen als Platzhalter für die Werte übergeben werden. Der letzte Parameter, übergibt die Werte für die Platzhalter in den „Where Klauseln“.

```
1  public void setPasswd(String passwd)
2  {
3          ContentValues content = new ContentValues();
4          content.put("type","passwd");
5          content.put("value",passwd);
6  
7          db.beginTransaction();
8          try
9          {
10          if( checkEntry("passwd") == false)
11                 db.insert("connect_datas","_id", content);
12          else
13          db.update("connect_datas", content, "type=?",new String[]{"passwd"});
14 
15          db.setTransactionSuccessful();
16         }finally
17         {
18                 db.endTransaction();
19         }
20 }
```

Abbildung 6.6: Insert und update-Statment

Zuletzt wird in Abbildung 6.7 ein „Delete-Statement“ aufgezeigt. Dieses hat 3 Parameter:

1. Tabellenname
2. Where Klausel
3. Werte für die Where Klausel

Wie an diesem Beispiel zu sehen ist, müssen die Werte nicht in einem extra Parameter übergeben werden, sondern können in die „Where Klausel“ mit eingebettet werden.

```
1         public void deleteNodeFromSelected(String node)
2         {
3                 db.beginTransaction();
4                 try
5                 {
6                         db.delete("selected_nodes","name='"+node+"'", null);
7                         db.delete("service_status","host='"+node+"'", null);
8                         db.delete("node_status","name='"+node+"'", null);
9                         db.delete("available_services","host='"+node+"'", null);
10
11                        db.setTransactionSuccessful();
12                }finally
13                {
14                        db.endTransaction();
15                }
16        }
```

Abbildung 6.7: Delete-Statement

6.2 Shared Preferences

Shared Preferences[3] sind geeignet um primitive Datentypen, über die Laufzeit der Applikation hinweg abzuspeichern.

6.3 Internal Storage

Internal Storage[4] ist geeignet um Dateien abzuspeichern, die nur von der eigenen Applikation erreichbar sein sollen und beim Deinstallieren der Applikation auch mit gelöscht werden sollen.

6.4 External Storage

External Storage[5] ist geeignet, um große Dateien abzuspeichern. Dabei sollte jedoch beachtet werden, dass jede Applikation Daten, die auf dem *External Storage* gespeichert sind, auch lesen und verändern kann.

[3] http://developer.android.com/guide/topics/data/data-storage.html#pref
[4] http://developer.android.com/guide/topics/data/data-storage.html#filesInternal
[5] http://developer.android.com/guide/topics/data/data-storage.html#filesExternal

Literatur

Google. *Android Dokumentation.* `http://developer.android.com`.

Künneth, Thomas. *Android 4 Apps entwickeln mit dem Android SDK*. Galileo Computing, 2012.

Marcus Pant, Arno Becker und. *Android 2 Grundlagen und Programmierung.* dpunkt.verlag, 2010.

Glossar

Activity
Wird zur Darstellung der grafischen Bedienoberfläche verwendet. Zum Start besitzt sie den ganzen Bildschirm, um mit dem Benutzer zu interagieren.

AlarmManager
Wird verwendet um periodische Aktionen durchzuführen, selbst wenn das Gerät im Ruhemodus versetzt ist.

Android
Von Google entwickeltes, auf dem Linux Kernel basierendes, für den mobilen Einsatz optimiertes, Betriebssystem.

App
Abkürzung für Applikation. Im Sprachgebrauch wird der Begriff aber meist nur für kleinere Programme verwendet, welche meist im mobilen Umfeld zu finden sind.

BroadcastReceiver
Empfängt systemweit gesendete Intents, die von anderen Applikationen über sendBroadcast() oder vom System selbst gesendet wurden.

Dalvik VM
Eine für den mobilen Einsatz optimierte virtuelle Maschine.

Dex-Bytecode
Für die Zielplattform optimierter Code.

Emulator
Emuliert ein Android Gerät um Applikationen ohne physikalisches Gerät zu testen.

Intent
Vergleichbar mit einem Signal, welches unter Java verwendet wird, um eine Klasse aufzurufen, die den passenden Listener implementiert hat.

Intentfilter
Wie unter Java hören Listener nur auf bestimmte Signale dies wird unter Android durch Intentfilter gesteuert.

Komponente
Unter Android wird das Komponentendesign verfolgt, um somit Teile einer Applikationen leicht wiederverwenden zu können.

LayoutInflater
Wird verwendet um eine grafische Bedienoberfläche dynamisch, aus vordefinierten XML-Teilen, zusammenzusetzen.

Lifecycle
Da eine mobile Anwendung durch verschiedene Ereignisse wie z.B. Anrufe oder andere Anwendungen unterbrochen werden kann, müssen mehrere Einstiegspunkte sowie Ausstiegspunkt beachtet werden.

Manifest
Bekanntgabe der Komponenten sowie Anforderung der Spezialberechtigungen dem System gegenüber.

PowerManage
Kann das Energiemanagement des Gerätes auf verschiedenen Arten beeinflussen.

Sandbox
Sicherheitskonzept, das eine Anwendung in einem gekapselten Bereich ausführt und die Kommunikation mit dem System und anderen Applikationen genau steuert.

SQLite
Relationales Datenbanksystem, welches für den Einsatz im Embedded Bereich entworfen ist.

ViewGroup
Eine besondere Art der View, welche mehrere Kinder Views enthalten kann.

Widget
Eine kleine Applikation, die in andere Applikationen integriert werden kann.

Index

A
Absolute Layout . . . 21
Activities . . . 37
AlarmManager . . . 44
AVD . . . 14

B
Bilder . . . 18
Broadcast receiver . . . 42

C
Content Provider . . . 8
Context Menu . . . 24

D
DVM . . . 3
- Dalvik Virtual Machine . . . 3
- dx . . . 3

Dynamische Oberflächen . . . 29

G
Gebundener Service . . . 38
Grid View . . . 23

I
Intents . . . 8
- explizite Intents . . . 8
- implizite Intents . . . 8
- Intentfilter . . . 7

L
LayoutInflater . . . 29
Layouts . . . 19–23
- Absolute Layout . . . 21
- Grid View . . . 23
- Linear Layout . . . 19
- List View . . . 22
- Relative Layout . . . 21
- Web View . . . 22

Lifecycle . . . 11
Linear Layout . . . 19
List View . . . 22

M
Manifest . . . 6
Menü . . . 24
- Context Menu . . . 24
- Option Menu . . . 27

O
Option Menu . . . 27

P
PowerManager . . . 44

R
Relative Layout . . . 21
Remote Service . . . 39

S
Sandbox . . . 5
ScrollView . . . 20
SDK . . . 13
Service . . . 38
- AlarmManager . . . 44
- Broadcast receiver . . . 42
- PowerManager . . . 44

Services
- Gebundener Service . . . 38
- Remote Service . . . 39

Signatur . . . 5
SQLite . . . 45
- Delete-Statement . . . 50
- Insert . . . 49
- Select . . . 48
- SQLiteOpenHelper . . . 45
- update . . . 49

START_STICKY . . . 40
string.xml . . . 17

V
ViewGroup . . . 29

W

Web View 22
Widgets 32

X

XML 16
    Mehrsprachige App 18
    string.xml 17