

Informatikarbeit

Android development

Christian Eppler*

10. 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	3
Kapitel 2	Android System Design	5
2.1	Dalvik VM	5
2.2	Sandbox	8
2.2.1	Signatur	8
2.2.2	Manifest	10
2.3	Komponenten Design	14
2.4	Intents	15
2.5	Lifecycle	19
Kapitel 3	IDE einrichten	21
3.1	Eclipse Plugins	21
3.2	Emulator	25
Kapitel 4	Oberfläche	27
4.1	XML	27
4.1.1	Layouts	28
4.1.2	Werte	30

4.1.3	Mehrsprachig	33
4.2	Bilder	33
4.3	Layouts	35
4.3.1	Linear Layout	35
4.3.2	Absolute Layout	40
4.3.3	Relative Layout	40
4.3.4	Web View	42
4.3.5	List View	42
4.3.6	Grid View	44
4.4	Menüs	45
4.4.1	Context Menu	45
4.4.2	Option Menu	50
4.5	Dynamische Oberflächen	54
4.6	Widgets	56
4.7	Activities	65
Kapitel 5	Services	72
5.1	Arten	73
5.1.1	Gebundener Service	73
5.1.2	Remote Service	74
5.2	Broadcast receiver	79
5.3	AlarmManager	83
5.4	PowerManager	83
Kapitel 6	Persistente Datenhaltung	85
6.1	SQLite	85
6.2	Shared Preferences	97
6.3	Internal Storage	97

6.4 External Storage 97

Glossar

Abbildungsverzeichnis

1.1	Beispiel	4
2.1	Workflow	7
2.2	Beispiel eines Manifest Headers	12
2.3	Permissions	13
2.4	Activity	14
2.5	Intent-Menü	16
2.6	Intent explizit	17
2.7	Intentfilter	19
2.8	Lifecycle	20
3.1	Quelle hinzufügen	22
3.2	Plugin installieren	23
3.3	SDK installieren	24
3.4	Emulator AVD	26
4.1	Projektordner	29
4.2	Strings XML	30
4.3	Verwendung von Strings im Code	31
4.4	Verwendung von Strings in XML	32
4.5	Linear Layout	36

4.6	Linear Layout node.xml	38
4.6	Linear Layout node.xml	39
4.7	Absolute Layout	41
4.8	Relative Layout	42
4.9	Web View	43
4.10	List View	43
4.11	Grid View	44
4.12	Context Menu	46
4.13	Context Menu Code 1	47
4.14	Context Menu Code 2	49
4.15	Toast	50
4.16	Option Menu	51
4.17	Option Menu 2	51
4.18	Option Menu Code 1	52
4.19	Option Menu Code 2	53
4.20	Kontainer main.xml	56
4.20	Kontainer main.xml	57
4.21	Layout zuweisen	57
4.22	Layout dynamisch füllen	58
4.22	Layout dynamisch füllen	59
4.23	Widget	61
4.24	Widget-Manifesteintrag	62
4.25	widget.xml	63
4.26	Widget Code 1	66
4.26	Widget Code 1	67
4.27	Widget Code 2	68
4.27	Widget Code 2	69
4.28	widgetlayout.xml	70

4.28	widgetlayout.xml	70
4.29	Widget-Aufruf	71
5.1	Service-Manifesteintrag	75
5.2	Service Code	77
5.2	Service Code	78
5.3	Broadcast receiver Manifesteintrag	79
5.4	Broadcast receiver Code	81
5.4	Broadcast receiver Code	82
5.5	PowerManager Code	84
6.1	SQLite Tables	86
6.2	SQLite Datenbankschema erstellen	87
6.2	SQLite Datenbankschema erstellen	88
6.3	SQLite Datenbankschema aktualisieren	90
6.3	SQLite Datenbankschema aktualisieren	91
6.4	DB-Verbindung	92
6.5	Select-Statment	93
6.6	Insert und update-Statment	94
6.6	Insert und update-Statment	94
6.7	Delete-Statement	96

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 Experimen-

¹<http://sourceforge.net/projects/weblooker/>

tieren unter: http://weblucker.svn.sourceforge.net/viewvc/weblucker/tags/webluckera_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/>.

1.1 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 auf der nächsten Seite)

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

weblooker²

²<http://weblooker.org>

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

Abbildung 1.1: Beispiel

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 SDK³ 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

¹NDK <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>

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 8)

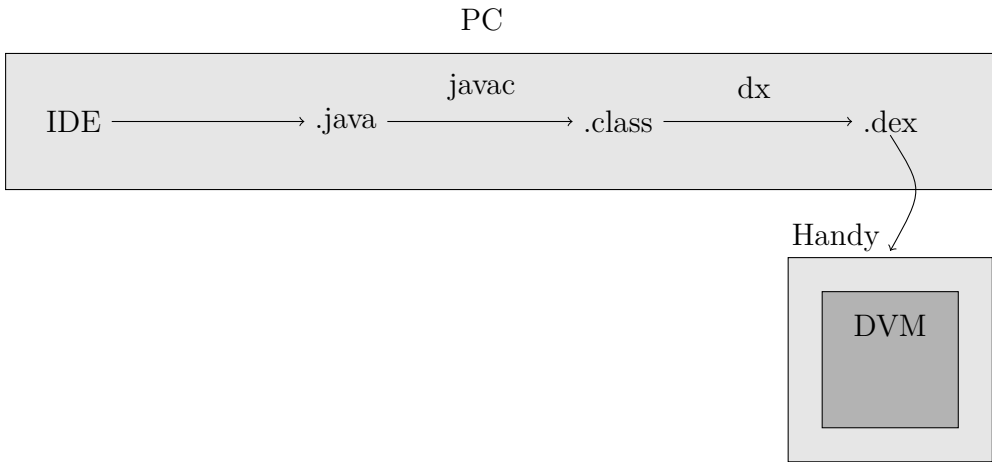


Abbildung 2.1: Workflow

2.2 Sandbox

Jede App unter Android läuft in ihrer eigenen Sandbox⁴, 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 Seite 10)

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 21)

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

⁵<https://play.google.com/store>

Die Signatur⁶ 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 14) erleichtert.

⁶http://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 erfordernten 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 benötigte Android Version und Version gegen die getestet wurde⁷

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

```
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

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

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) Ab Zeile 5 bis 8 wird ein Intentfilter angelegt, dieser spezifiziert

⁸<http://developer.android.com/reference/android/Manifest.permission.html>

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

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

Abbildung 2.3: Permissions

bei welchem Intent (siehe Abschnitt 2.4 auf Seite 15) 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. ↵
7                  intent.action.MAIN" />
8              <category android:name="android. ↵
9                  intent.category.LAUNCHER" />
          </intent-filter>
        </activity>

```

Abbildung 2.4: Activity

2.3 Komponenten Design

Android ist ein modernes auf dem Komponenten Design¹⁰ 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

¹⁰[http://de.wikipedia.org/wiki/Komponente_\(Software\)](http://de.wikipedia.org/wiki/Komponente_(Software))

¹¹<http://developer.android.com/guide/topics/providers/>

hier nicht weiter eingegangen wird.

2.4 Intents

Intents¹² 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 auf der nächsten Seite) angezeigt.

content-providers.html

¹²<http://developer.android.com/guide/components/intents-filters.html>

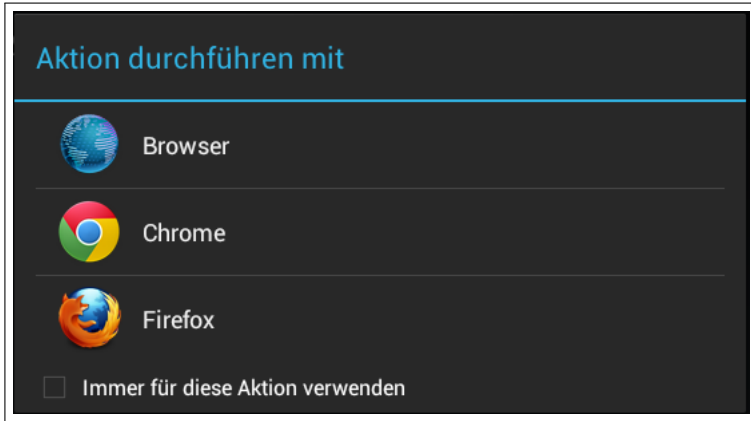


Abbildung 2.5: Intent-Menü

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 explizit

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 14 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.

¹³<http://developer.android.com/guide/topics/manifest/data-element.html>

¹⁴http://developer.android.com/reference/android/content/Intent.html#CATEGORY_DEFAULT

```
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

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 65), 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.

¹⁵<http://developer.android.com/training/basics/activity-lifecycle/index.html>

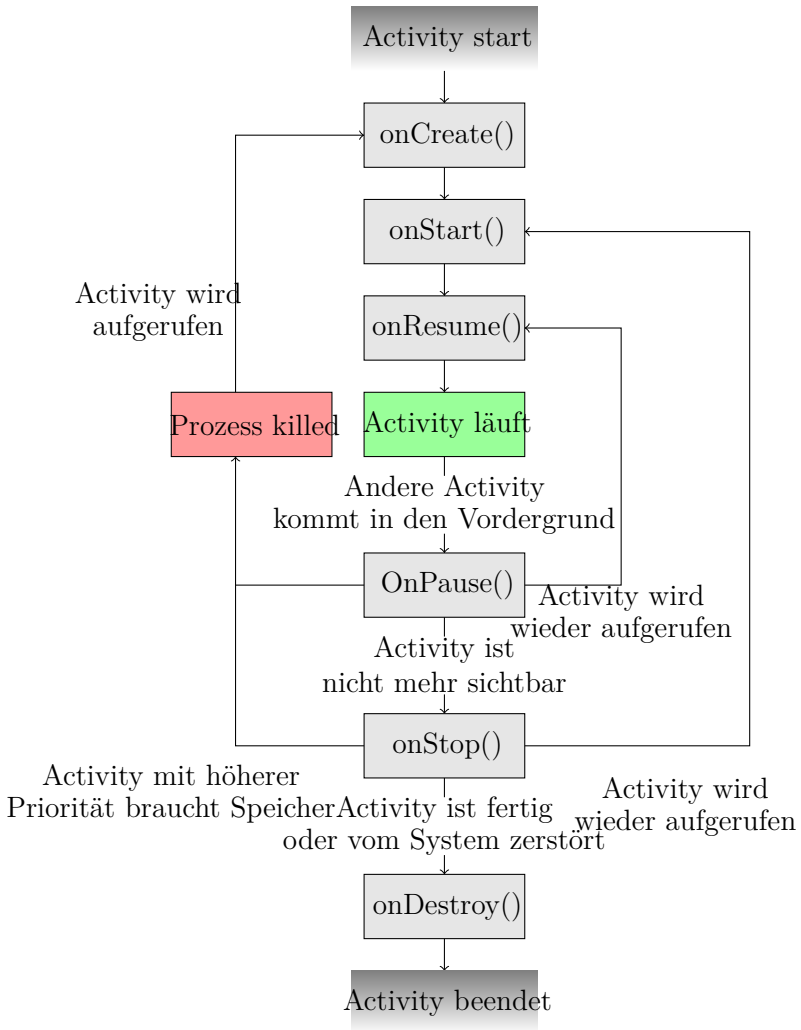


Abbildung 2.8: Lifecycle

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 auf der nächsten Seite)

¹<http://dl-ssl.google.com/android/eclipse/site.xml>

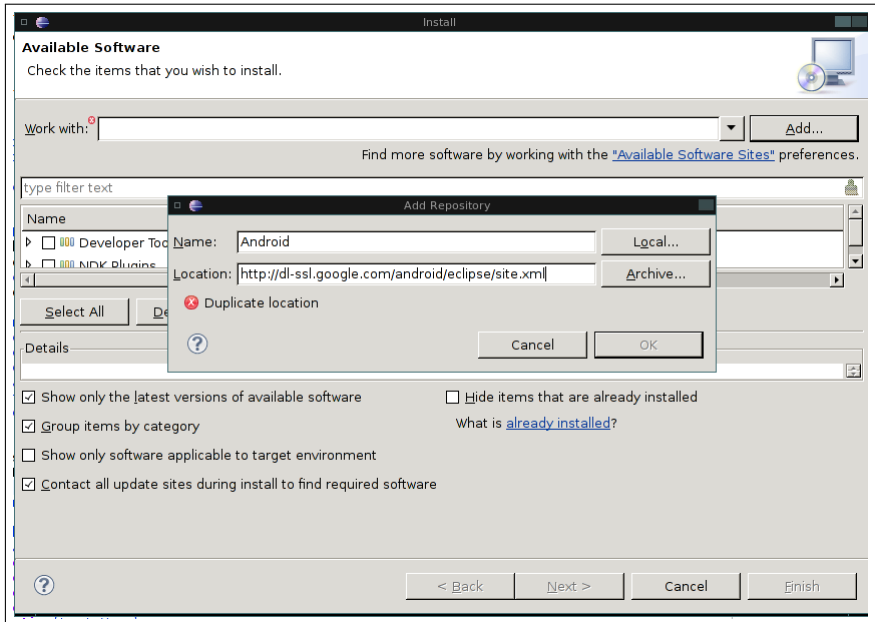


Abbildung 3.1: Quelle hinzufügen

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

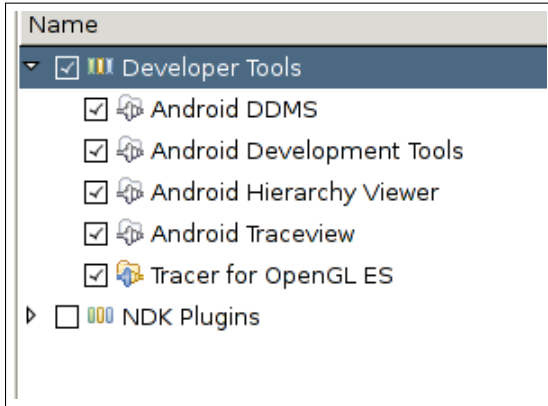


Abbildung 3.2: Plugin installieren

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

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.

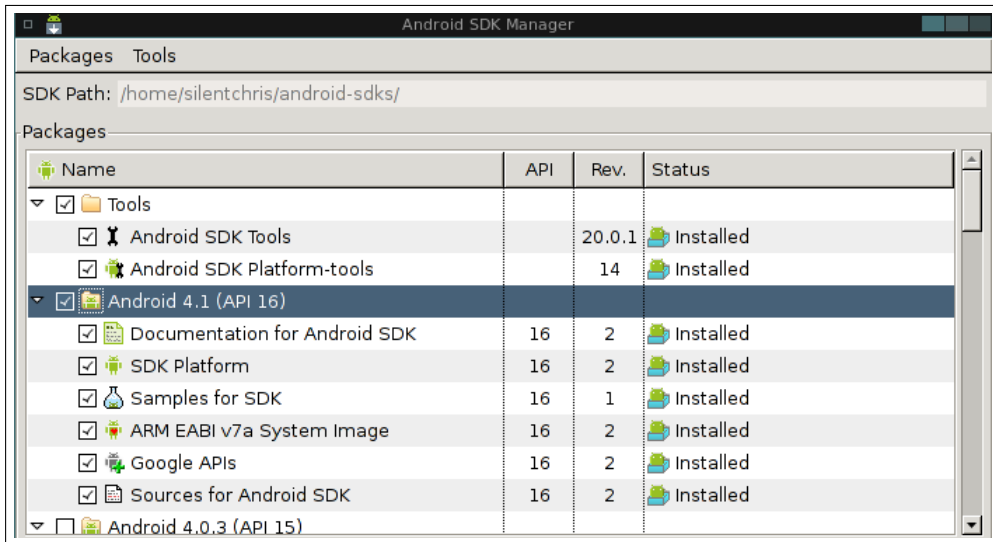


Abbildung 3.3: SDK installieren

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>

³udev <http://de.wikipedia.org/wiki/Udev>

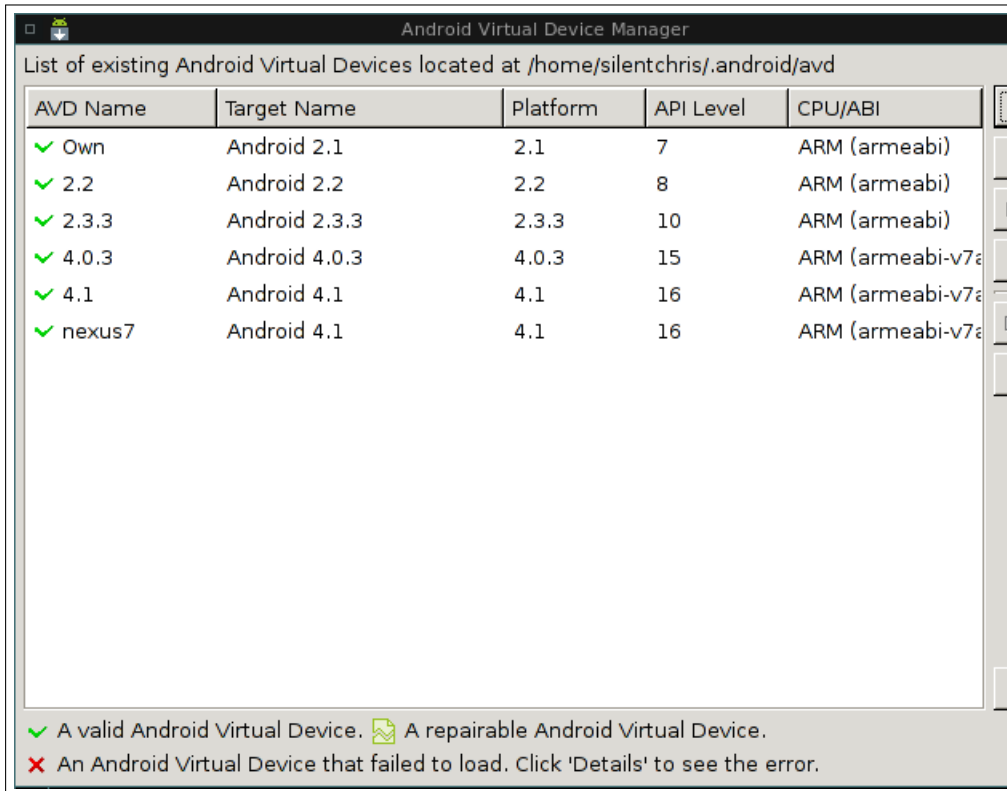


Abbildung 3.4: Emulator AVD

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 auf der nächsten Seite zu sehen ist, angelegt. Als Beispiel für eine XML-Layoutdatei sei hier auf Abbildung 4.5 auf Seite 36 verwiesen. Es können auch verschiedene Layouts für verschiedene Geräteklassen erstellt werden.¹

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

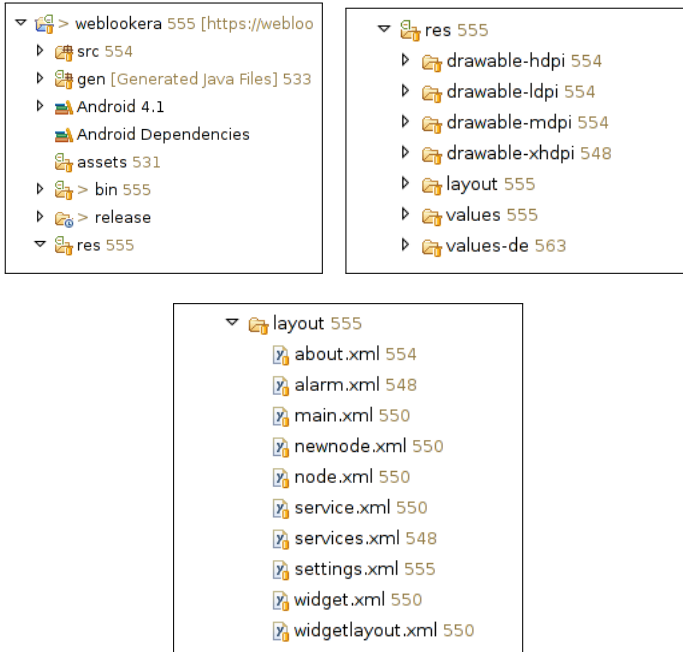


Abbildung 4.1: Projektordner

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 52)

```
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 Seite 33 besprochen wird. In Abbildung 4.3 auf der nächsten Seite in Zeile 3 wird gezeigt, wie ein String aus der XML-Datei im Code verwendet wird. Dagegen ist in Abbildung 4.4 auf Seite 32

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

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <ScrollView xmlns:android="http://schemas.android.com/ ↵
   apk/res/android"
3     android:layout_width="fill_parent"
4     android:layout_height="fill_parent">
5
6 <LinearLayout xmlns:android="http://schemas.android. ↵
   com/apk/res/android"
7     android:layout_width="fill_parent"
8     android:layout_height="wrap_content"
9     android:orientation="vertical" >
10
11 <TextView
12     android:id="@+id/host_t"
13     android:layout_width="fill_parent"
14     android:layout_height="wrap_content"
15     android:textSize="20dp"
16     android:text="@string/adrServ"
17     />
```

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 29 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 verschiedenen 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 29 zu sehen ist. Es werden im Projekt verschieden Ordner für Bilder in verschiedenen Auflösungen erstellt.² Das Android-System entscheidet dann

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

eigenständig, welche Bilder unter welchem Gerät verwendet werden.

4.3 Layouts

Nachfolgend werden verschiedene Layouts³ 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 auf der nächsten Seite rechts zu sehen. Doch soll an dieser Stelle davor gewarnt werden, Layouts mit zu vielen Ebenen zu erstellen, da dies die Performance erheblich beeinträchtigt.

Nachfolgend in Abbildung 4.6 auf Seite 39 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>

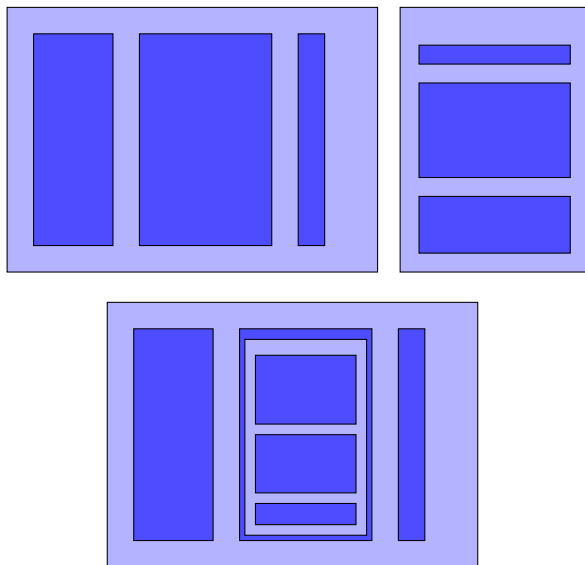


Abbildung 4.5: Linear Layout

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 28)

⁴<http://developer.android.com/reference/android/widget/ScrollView.html>

```
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  />
```

Abbildung 4.6: Linear Layout node.xml

```
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 (Fortsetzung): Linear Layout node.xml

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 nächsten Seite auf der rechten Seite, welches ein kleineres Display darstellen soll, zu sehen ist. Deswegen wird es im mobilen Umfeld eher weniger verwendet.⁵

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.

⁵<http://developer.android.com/reference/android/widget/AbsoluteLayout.html>

⁶<http://developer.android.com/reference/android/widget/RelativeLayout.html>

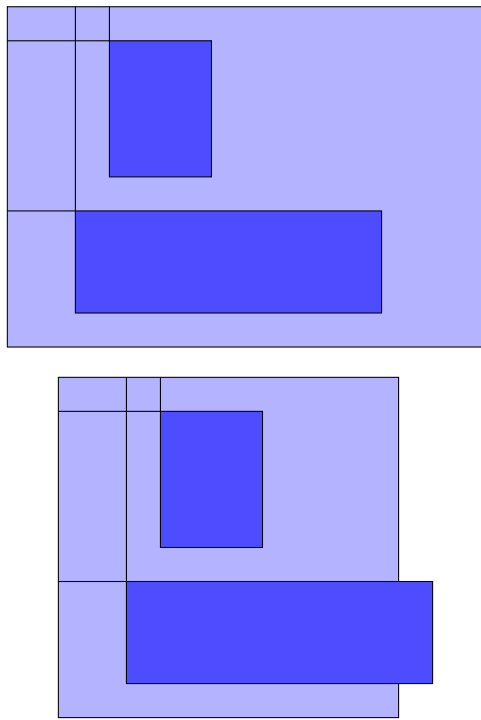


Abbildung 4.7: Absolute Layout

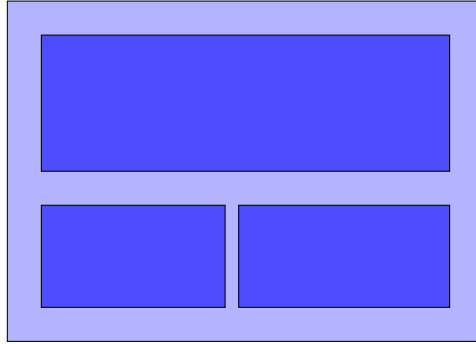


Abbildung 4.8: Relative Layout

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.

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 39.

⁷<http://developer.android.com/reference/android/widget/ListView.html>

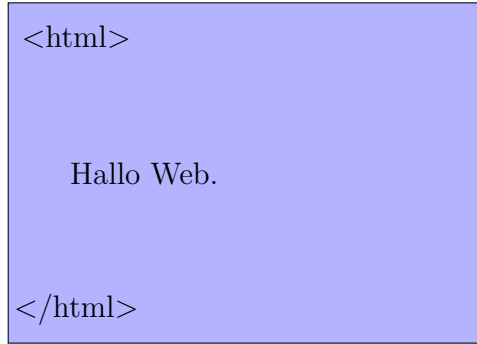


Abbildung 4.9: Web View

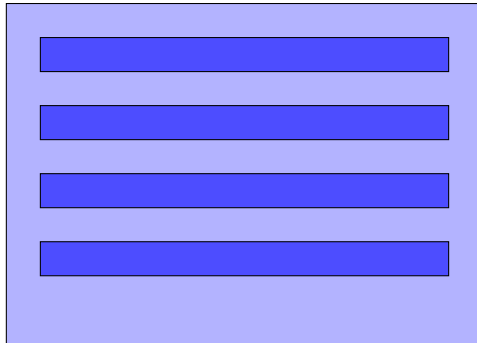


Abbildung 4.10: List View

4.3.6 Grid View

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

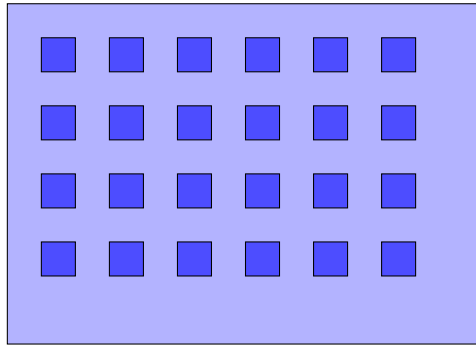


Abbildung 4.11: Grid View

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 auf der nächsten Seite.

⁹<http://developer.android.com/reference/android/view/ContextMenu.html>

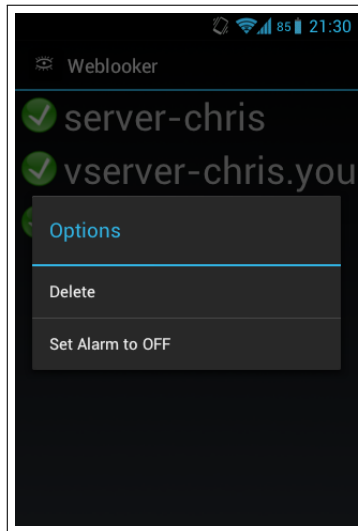


Abbildung 4.12: Context Menu

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 auf Seite 50)

¹⁰Toast <http://developer.android.com/guide/topics/ui/notifiers/toasts.html>

```
1 public boolean onOptionsItemSelected(MenuItem item)
2 {
3     if(item.toString().equals("Delete"))
4     {
5         /* Delete entry */
6         db_call.deleteNodeFromSelected(selectedNode);
7         Toast.makeText(this, "Delete_" + selectedNode,
8             Toast.LENGTH_SHORT).show();
9     }
10    else
11    {
12        /* Edit Entry */
13        Toast.makeText(this, "Change_alarm_to_" +
14            db_call.changeAlarm(selectedNode), Toast.
15            LENGTH_SHORT).show();
16    }
17    refreshView();
18    return super.onOptionsItemSelected(item);
19 }
```

Abbildung 4.14: Context Menu Code 2

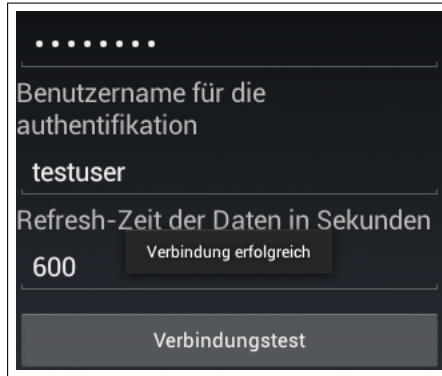


Abbildung 4.15: Toast

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 auf der nächsten Seite oben rechts zu sehen, durch eine *on-screen* Menütaste. Das Menü muss für jede Activity extra angelegt werden und ist nicht App-übergreifend.

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

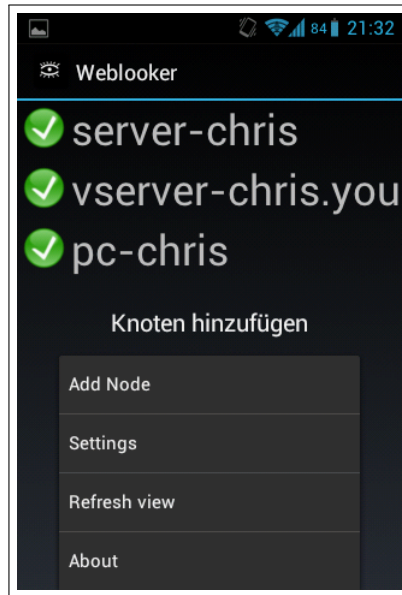


Abbildung 4.16: Option Menu

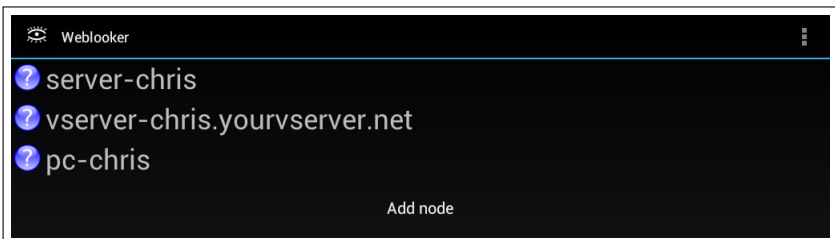


Abbildung 4.17: Option Menu 2

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

```
1 @Override
2 public boolean onCreateOptionsMenu(Menu menu)
3 {
4     menu.add(0,AD_ID,Menu.NONE,"Add_Node").setIcon(↵
5         (android.R.drawable.ic_menu_add);
6     menu.add(0,SE_ID,Menu.NONE,"Settings").setIcon(↵
7         (android.R.drawable.ic_menu_preferences);
8     menu.add(0,RE_ID,Menu.NONE,"Refresh_view").↵
9         setIcon(android.R.drawable.ic_menu_view);
10    menu.add(0,AB_ID,Menu.NONE,"About").setIcon(↵
        android.R.drawable.ic_menu_info_details);
11
12    return super.onCreateOptionsMenu(menu);
13 }
```

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 65) mit Hilfe eines Intents (siehe Abschnitt 2.4 auf Seite 15) aufgerufen.

```
1  @Override
2  public boolean onOptionsItemSelected(MenuItem item)
3  {
4      switch (item.getItemId())
5      {
6          case AD_ID:
7              final Intent node = new Intent()
8                  (this, WeblookerActivityAddNode.
9                  class);
10             startActivity(node);
11             break;
12         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 Seite 57 ist ein Teil der *main.xml* zu sehen, welche zur *WeblookeraActivity* gehört. Diese wird der Activity, wie in Abbildung 4.21 auf Seite 57 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 andeu-

¹²<http://developer.android.com/reference/android/view/ViewGroup.html>

¹³<http://developer.android.com/reference/android/view/LayoutInflater.html>

tet, ist dies bloß ein Container für den dynamisch erstellten Inhalt.

In Abbildung 4.22 auf Seite 59 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 85) 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 36). Nach dem diese mit Daten gefüllt wurde, wird die erstellte View in Zeile 25 der *ViewGroup* hinzugefügt.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <ScrollView xmlns:android="http://schemas.android.com/␣
   apk/res/android"
3     android:layout_width="fill_parent"
4     android:layout_height="wrap_content"
5     android:overScrollMode="always">
6
7     <LinearLayout xmlns:android="http://schemas.␣
   android.com/apk/res/android"
8         android:layout_width="fill_parent"
9         android:layout_height="wrap_content"
10        android:orientation="vertical"
```

Abbildung 4.20: Kontainer main.xml

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 auf Seite 61 zu sehen ist.

¹⁴<http://developer.android.com/guide/topics/appwidgets/index.html>

```
10     android:orientation="vertical"
11     android:id="@+id/main">
12
13     <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
14         android:layout_width="fill_parent"
15         android:layout_height="wrap_content"
16         android:orientation="vertical"
17         android:id="@+id/main_container">
18     </LinearLayout>
```

Abbildung 4.20 (Fortsetzung): Kontainer main.xml

```
1     public void onCreate(Bundle savedInstanceState) {
2         super.onCreate(savedInstanceState);
3         setContentView(R.layout.main);
4     }
```

Abbildung 4.21: Layout zuweisen


```
1 protected void addViews()
2 {
3   if(db_call.getNodes() == true)
4     if(db_call.nodes != null)
5       for(int i = 0; i < db_call.nodes.length; i++)
6         {
7           view = LayoutInflater.from(↵
8             getBaseContext()).inflate(R.layout.↵
9             node,null);
10          if(db_call.nodes_badStatus[i] == 1)
11            ((ImageView)view.findViewById(↵
12              R.id.status)).↵
13              setImageResource(R.drawable↵
14              .ok);
15          else if(db_call.nodes_badStatus[i] == ↵
16            2)
17            ((ImageView)view.findViewById(↵
18              R.id.status)).↵
19              setImageResource(R.drawable↵
20              .off);
21          else if(db_call.nodes_badStatus[i] == ↵
22            3)
23            ((ImageView)view.findViewById(↵
24              R.id.status)).↵
25              setImageResource(R.drawable↵
26              .away);
27
28          /* Get textview for node name */
```

Abbildung 4.22: Layout dynamisch füllen

```
16      /* Get textview for node name */
17      tview = ((TextView)view.findViewById(R.↵
           .id.txt));
18
19      /* Menu on long click*/
20      registerForContextMenu(tview);
21
22      /* Set node name */
23      tview.setText(db_call.nodes[i]);
24
25      parent.addView(view);
26  }
27 }
```

Abbildung 4.22 (Fortsetzung): Layout dynamisch füllen

Nachfolgend wird anhand eines Beispiels der Aufbau eines Widgets erläutert.

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 14).

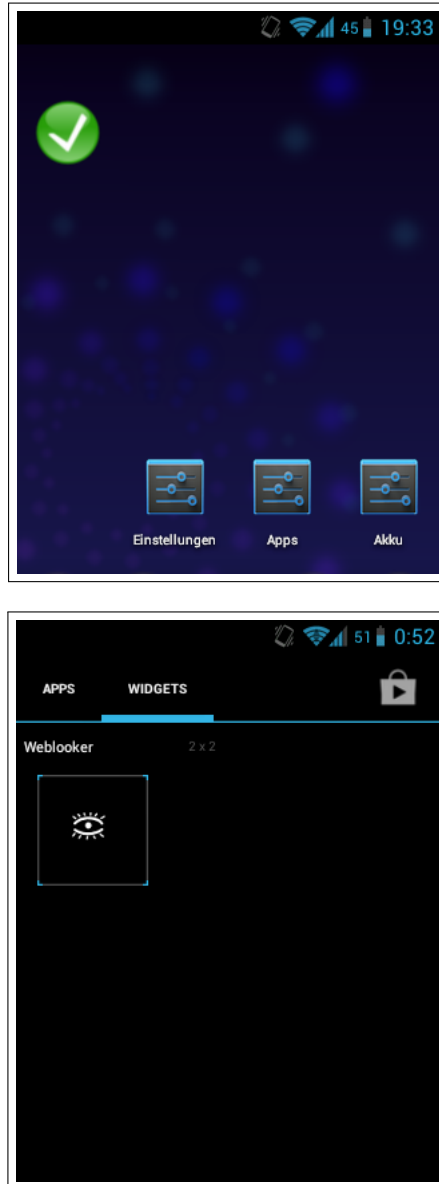


Abbildung 4.23: Widget

```
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

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

Nachfolgend, in Abbildung 4.26 auf Seite 67, 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 83) aufgerufen wird, musste die *onReceive()* Methode überschrieben werden. In Zeile 22 wird eine Instanz des *AppWidgetManager*

```
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

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 69 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 70 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 71 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 15) und in Zeile 14 als Broadcast gesendet. Wer sich für dieses Beispiel im Ganzen interessiert, der sei hier auf das Repository auf [sourceforge.org](https://weblooker.svn.sourceforge.net/svnroot/weblooker/tags/weblookera_1.0) verwiesen.¹⁵

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

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 20) zu beachten.

¹⁶<http://developer.android.com/reference/android/app/Activity.html>


```
1 public class WeblookerWidget extends AppWidgetProvider
2 {
3     SimpleDateFormat df = new SimpleDateFormat("hh:mm:ss" ↵
4         );
5     private static int status = 0;
6     public static final String ACTION_EXTRAS= "extras";
7
8     @Override
9     public void onUpdate(Context context, ↵
10         AppWidgetManager appWidgetManager, int[] ↵
11         appWidgetIds)
12     {
13
14     @Override
15     public void onReceive(Context context, Intent intent)
16     {
17         super.onReceive(context, intent);
```

Abbildung 4.26: Widget Code 1

```
16     super.onReceive(context, intent);
17
18     if(intent != null)
19     {
20         status = intent.getIntExtra(ACTION_EXTRAS,0);
21
22         AppWidgetManager am = AppWidgetManager.↵
                getInstance(context);
23         if(am != null)
24         {
25             int[] appWidgetIds = am.getAppWidgetIds(new ↵
                ComponentName(context,getClass()));
26
27             if(appWidgetIds != null && appWidgetIds.↵
                length > 0)
28                 updateWidget(context,am,appWidgetIds ↵
                    );
29         }
30     }
31 }
```

Abbildung 4.26 (Fortsetzung): 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)
```

Abbildung 4.27: Widget Code 2

```
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 (Fortsetzung): Widget Code 2

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout
3     xmlns:android="http://schemas.android.com/apk/res/ ↵
4         android"
5     android:orientation="vertical"
6     android:layout_width="match_parent"
7     android:layout_height="match_parent"
8     android:gravity="center"
9     android:layout_margin="4dp"
10    android:background="#00000000">
11
12     <ImageView android:id="@+id/image_widget"
13               android:layout_width="wrap_content"
14               android:layout_height="wrap_content"
```

Abbildung 4.28: widgetlayout.xml

```
14         android:layout_height="wrap_content"
15         android:src="@drawable/nodata"
16         android:clickable="true"
17         android:contentDescription="@string/ ↵
18             app_name"
19         android:background="#00000000"
20     />
</LinearLayout>
```

Abbildung 4.28 (Fortsetzung): widgetlayout.xml

```
1 private void widgetUpdate()
2 {
3     int value;
4     /* Get status */
5     WeblookerViewDBCalles db = new WeblookerViewDBCalles(
        (getApplicationContext()));
6     value = db.getStatusForWidget();
7     db.onDestroy();
8
9     /* Create Intent with data*/
10    Intent i = new Intent(this, WeblookerWidget.class);
11    i.putExtra(WeblookerWidget.ACTION_EXTRAS, value);
12
13    /* Send broadcast to widget */
14    sendBroadcast(i);
15 }
```

Abbildung 4.29: Widget-Aufruf

5 Services

Definitionen:

PowerManager:

Kann das Energiemanagement des Gerätes auf verschiedenen Arten beeinflussen.

Services¹ 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 83) und *PowerManager* (siehe Abschnitt 5.4 auf Seite 83) zum Lebenserhalt des Service verwendet, kann dies einen beträchtlichen Verbrauch an Energie hervorrufen.

¹<http://developer.android.com/guide/components/services.html>

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

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 79) ausgelöst wurde, gestartet werden. Entweder er teilt sich einen Prozess mit dem rest der Applikation oder er wird in einem separaten Prozess gestartet, wodurch die allgemeine Stabilität erhöht wird. Diese Art des Services werden wir uns nachfolgen näher betrachten.

In folgender Abbildung 5.1 auf der nächsten Seite 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 15). 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

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

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

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.

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

⁴http://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 ↵
5         flags, int startId)
6     {
7         super.onStartCommand(intent, flags, ↵
8             startId);
9         lockS.acquire();
10        /* Check for Internet connection*/
11        ConnectivityManager cm =(↵
12            ConnectivityManager) ↵
13            getSystemService(Context.↵
14                CONNECTIVITY_SERVICE);
15        if (!( cm.getActiveNetworkInfo() != ↵
16            null && cm.getActiveNetworkInfo().↵
17                isConnectedOrConnecting()))
18        {
19            widgetUpdate();
20            lockS.release();
21            return START_STICKY;
22        }
23        /* Start service at thread */
```

Abbildung 5.2: Service Code

```
16      /* Start service at thread */
17      new Thread(new Runnable() {
18          public void run()
19          {
20              try{
21                  if(readLoginData() == ↵
22                      false)
23                      return;
24                  /* Start service */
25                  if(service() == true)
26                  {
27                      alarm();
28                      widgetUpdate() ↵
29                      ;
30                  }
31                  /* Say goodbye to ↵
32                  server */
33                  mm.endSession();
34              }catch(Exception e){
35                  e.printStackTrace();
36              }
37              finally{
38                  lockS.release();
39              }
40          }
41      }).start();
42      return START_STICKY;
43  }
```

Abbildung 5.2 (Fortsetzung): 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 Manfesteintrag des *Broadcast receivers* zu sehen. Interessant sind hier Zeile 2 und 3, in Zeile 2 wird festgelegt, ob das System den *Broadcast receiver* instanziiieren 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 Manfesteintrag

In Abbildung 5.4 auf Seite 82 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.

```
1 public class StartupNetService extends BroadcastReceiver
2 {
3     private static final int SECS = 1000;
4     private static final boolean debug = false;
5
6     @Override
7     public void onReceive(Context arg0, Intent arg1)
8     {
9         if ("android.intent.action.BOOT_COMPLETED".equals(
10             arg1.getAction()))
11         {
12             /* Get timer */
13             WeblookerViewDBCalles db = new WeblookerViewDBCalles(
14                 arg0);
15             int time = Integer.parseInt(db.getRefresh());
16             db.onDestroy();
17
18             /* Start service with AlarmManager */
```

Abbildung 5.4: Broadcast receiver Code


```
16      /* Start service with AlarmManager */
17      /* Works always is device on sleep */
18      Calendar cal = Calendar.getInstance();
19      /* Start Service with Intent */
20      Intent in = new Intent( arg0, NetService.class )
21                  );
22      PendingIntent pi = PendingIntent.getService(
23          arg0, 0, in, PendingIntent.
24          FLAG_UPDATE_CURRENT);
25      AlarmManager alarms = (AlarmManager) arg0.
26          getSystemService(Context.ALARM_SERVICE);
27      /* Set Timer for wakeup */
28      alarms.setRepeating(AlarmManager.RTC_WAKEUP,
29          cal.getTimeInMillis(), time * SECS, pi);
30      if(debug)Toast.makeText(arg0,"Start_service",
31          Toast.LENGTH_LONG).show();
32  }
33 }
```

Abbildung 5.4 (Fortsetzung): 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 auf der nächsten Seite wird aufgezeigt, wie man an eine Referenz kommt. Mit dem *PowerManager* können dem System explizit „Befehle“ zum

⁵<http://developer.android.com/reference/android/app/AlarmManager.html>

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 78 in Zeile 7 und 35 verdeutlicht.

```
1      @Override
2      public void onCreate()
3      {
4          this.getSystemService();
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

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 auf der nächsten Seite ist ein Vorschlag zu sehen, wie die Befehle zum Erstellen der Tabellen sauber realisiert werden können.

¹<http://developer.android.com/guide/topics/data/data-storage.html>

```
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         ,\"+
7         "type_TEXT,\"+
8         "value_TEXT);";
9
10    public static final String SQL_DROP_CONNECT_DATAS =
        "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 Seite 88 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.

²<http://developer.android.com/reference/android/database/sqlite/>

```
1 public class WeblookerDB extends SQLiteOpenHelper
2 {
3     private static final String DB_NAME = "
4         weblooker.db";
5     private static final int DB_VERSION = 3;
6
7     public WeblookerDB(Context context)
8     {
9         super(context, DB_NAME, null,
10             DB_VERSION);
11     }
12
13 @Override
14 public void onCreate(SQLiteDatabase db)
15 {
16     /* Create Tables*/
17     db.beginTransaction();
18     try{
```

Abbildung 6.2: SQLite Datenbankschema erstellen

```
16     try{
17         db.execSQL(WeblookerTbl. ↵
18             SQL_CREATE_AVAILABLE_NODES);
19         db.execSQL(WeblookerTbl. ↵
20             SQL_CREATE_AVAILABLE_SERVICE);
21         db.execSQL(WeblookerTbl. ↵
22             SQL_CREATE_CONNECT_DATAS);
23         db.execSQL(WeblookerTbl. ↵
24             SQL_CREATE_NODE_STATUS);
25         db.execSQL(WeblookerTbl. ↵
26             SQL_CREATE_SELECTED_NODES);
27         db.execSQL(WeblookerTbl. ↵
28             SQL_CREATE_SERVICE_STATUS);
29         /* Insert init data*/
30         ContentValues content = new ContentValues();
31         content.put("type", "refresh");
32         content.put("value", "600");
33         db.insert("connect_datas", "_id", content);
34         content.put("type", "port");
35         content.put("value", "15001");
36         db.insert("connect_datas", "_id", content);
37         db.setTransactionSuccessful();
38     }finally{
39         db.endTransaction();
40     }
41 }
```

Abbildung 6.2 (Fortsetzung): 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 auf Seite 91 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);
```

Abbildung 6.3: SQLite Datenbankschema aktualisieren

```
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 (Fortsetzung): 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).
6                  getWritableDatabase();
7
8              time = Integer.parseInt(getRefresh())
9                  ;
10         this.context = context;
11     }
```

Abbildung 6.4: DB-Verbindung

Abbildung 6.5 auf der nächsten Seite zeigt einen einfachen „Select“ Befehl unter *SQLite* in Form eines *rawQuery*s. 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_
4         WHERE_type=?",new String[] {type});
5     if(data.getCount() == 0)
6     {
7         data.close();
8         return false;
9     }
10    data.close();
11    return true;
12 }
```

Abbildung 6.5: Select-Statment

Abbildung 6.6 auf der nächsten Seite 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", ↵
12                content);
13        else
14            db.update("connect_datas", content, "type=?", ↵
15                new String[]{"passwd"});
16        db.setTransactionSuccessful();
17    }finally
18    {
19        db.endTransaction();
20    }
21 }
```

Abbildung 6.6: Insert und update-Statment

```
20 }
```

Abbildung 6.6 (Fortsetzung): Insert und update-Statment

Zuletzt wird in Abbildung 6.7 auf der nächsten Seite 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
9              = '"+node+"'", null);
10             db.delete("available_services"
11             ,"host='"+node+"'", null);
12
13             db.setTransactionSuccessful();
14         }finally
15         {
16             db.endTransaction();
17         }
18     }
```

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 40
Activities 65
AlarmManager 83
AVD 25

B

Bilder 33
Broadcast receiver 79

C

Content Provider 14
Context Menu 45

D

DVM 5
 Dalvik Virtual Machine
 5
 dx 5
Dynamische Oberflächen 54

G

Gebundener Service 73

Grid View 44

I

Intents 15
 explizite Intents 15
 implizite Intents 15
Intentfilter 12

L

LayoutInflater 54
Layouts 35–44
 Absolute Layout 40
 Grid View 44
 Linear Layout 35
 List View 42
 Relative Layout 40
 Web View 42
Lifecycle 19
Linear Layout 35
List View 42

M

Manifest 10

-
- Menü 45
 - Context Menu 45
 - Option Menu 50
 - O**
 - Option Menu 50
 - P**
 - PowerManager 83
 - R**
 - Relative Layout 40
 - Remote Service 74
 - S**
 - Sandbox 8
 - ScrollView 37
 - SDK 23
 - Service 72
 - AlarmManager 83
 - Broadcast receiver .. 79
 - PowerManager 83
 - Services
 - Gebundener Service 73
 - Remote Service 74
 - Signatur 8
 - SQLite 85
 - Delete-Statement ... 95
 - Insert 93
 - Select 92
 - SQLiteOpenHelper . 86
 - update 93
 - START_STICKY 76
 - string.xml 30
 - V**
 - ViewGroup 54
 - W**
 - Web View 42
 - Widgets 56
 - X**
 - XML 27
 - Mehrsprachige App . 33
 - string.xml 30