JAVA Mag

Java magazin™

Java | Architektur | Software-Innovation

JAVA EE 8

DIE MACHT DER ACHT

ra2studio/Shutterstock.com

SLF4J und Logback in Android

Android Logging für Java-Profis

Wenn man als Java-Entwickler in die Android-Entwicklung einsteigt, ist zunächst vieles aus Java bekannt. Einige Aspekte sind aber auch anders gelöst, beispielsweise das Thema Logging. Hier bringt Android sein eigenes API mit, das ohne weitere Konfiguration bequem verwendet werden kann. Bei fortgeschrittenen Problemen hat dieser Ansatz jedoch seine Grenzen. Hier können Entwickler auf vorhandenes Wissen und Bibliotheken aus der Java-Welt zurückgreifen.

von Johannes Schnatterer

Für Neulinge erfreulich, lassen sich in Android ohne weitere Konfiguration die statischen Methoden der Klasse *android.util.Log* aufrufen, was zu einem Eintrag im systemweiten Log führt. Dieses kann mittels der Anwendung Logcat über den Android-Debug-Server (DDMS) oder direkt über eine Android Debug Bridge Shell (ADB) eingesehen werden. Während *android.util.Log* grundlegende Logging-Funktionalität bequem bereitstellt, bietet es keine Lösungen für fortgeschrittene Probleme wie Konfiguration, Integration der Logs von Bibliotheken oder Zugriff auf Logs aus der Anwendung. Viele Java-Entwickler würden solche Herausforderungen mittels SLF4J und Logback meistern.

Gerade durch sein einfaches API und die Möglichkeit der einheitlichen Konfiguration aller Log-Statements, auch von Third-Party-Code, hat sich die Simple Logging Facade for Java (SLF4J) in Java zum De-facto-Standard-Logging-API entwickelt [1]. SLF4J stellt ein schmales API bereit, gegen das man im Code Log-Statements absetzt. Zusätzlich existieren Bridges, um Log-Statements, die mit anderen Logging-Frameworks abgesetzt wurden, auf SLF4J umzuleiten. Welches Logging Framework, also welche Implementierung, das SLF4J-API verwendet, muss erst beim Deployment entschieden werden.

Abbildung 1 zeigt die Schichten, Module und deren Zusammenhänge im Kontext von SLF4J. Die einzelnen Module stehen als separate JAR-Dateien zur Verfügung. Die Bezeichnungen der Schichten sind aus der SLF4J-Do-

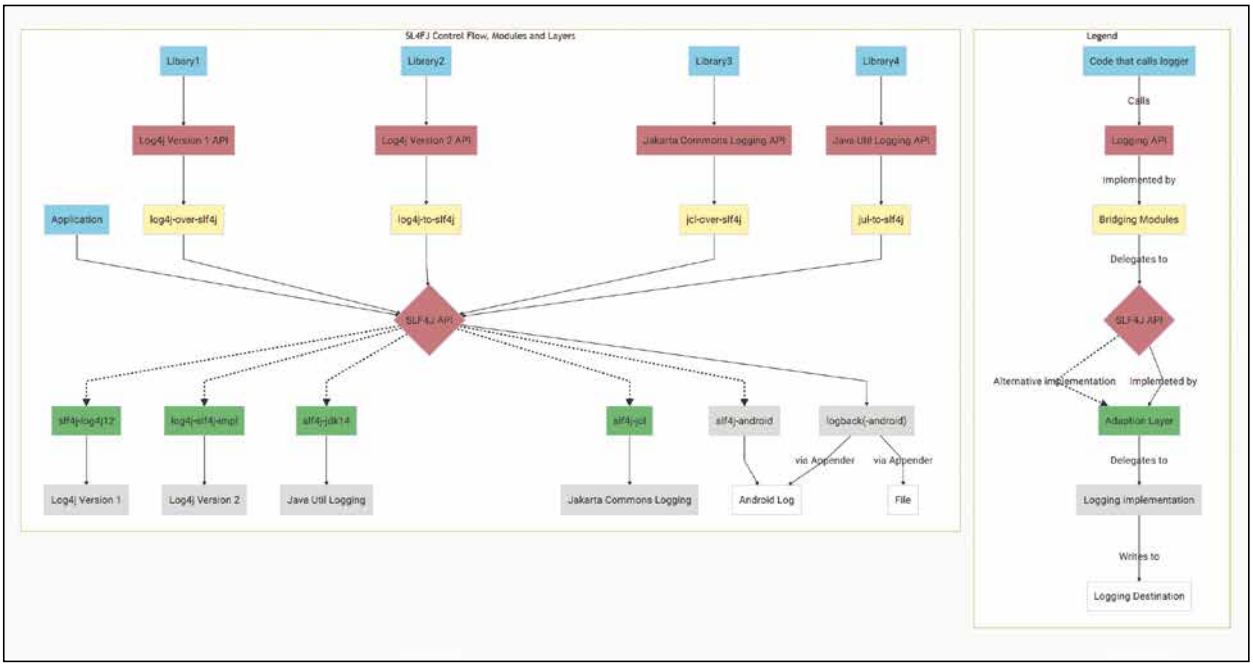

Abb. 1: SLF4J-Kontrollfluss, -Module und -Schichten

kumentation übernommen [2]. Vorsicht: Im Gegensatz dazu stehen die Bezeichnungen, die Log4J in der Version 2.0 seinen Modulen gibt: Das Bridging-Modul log4j-to-slf4j heißt Adapter und das Modul aus dem Adpation Layer log4j-slf4j-impl heißt Binding oder Bridge [3].

Unter anderem existieren Bridging-Module für Log4j 1.x (log4j-over-slf4j), Log4j 2.x (log4j-to-slf4j), Jakarta Commons Logging (jcl-over-slf4j) und Java Util Logging (jul-to-slf4j). Als eigentliche Implementierung des SLF4J-API stehen viele Logging-Frameworks zur Auswahl. Beispiele hierfür sind SLF4Js native Implementierungen Logback, dessen Fork logback-android, SLF4J Android, aber auch Adaptermodule für Log4j 1.x (slf4j-log4j12), Log4j 2.x (log4j-slf4j-impl), Java Util Logging (slf4j-jdk14) oder Jakarta Commons Logging (slf4j-jcl). Die Abstraktion über das SLF4J-API erlaubt es, das eigentliche Logging-Framework leicht auszutauschen, ohne den Code anzupassen.

Durch das Umleiten aller Log-Statements auf das einheitliche SLF4J-API ist es möglich, das Logging aller eingesetzten Bibliotheken durch das Framework der Wahl einheitlich zu konfigurieren. Denn die Konfiguration ist nicht Teil von SLF4J, sondern wird durch das Logging-Framework realisiert. Im Fall von Logback und logback-android ordnet man den im Code definierten Loggern Datensenken zu, so genannte Appender, beispielsweise Dateien, Standard Out oder das systemweite Android-Log. Die Formatierung der Log-Statements erfolgt innerhalb der Appender. Eine Filterung, welche Log-Level ausgegeben werden, lässt sich pro Appender oder im Code definierte Logger vornehmen.

Listing 1: build.gradle

```
compile 'org.slf4j:slf4j-api:1.7.21'
apk('com.github.tony19:logback-android-classic:1.1.1-6') {
  // workaround issue #73
  exclude group: 'com.google.android', module: 'android'
}
```

Listing 2: Absetzen eines Log-Statements

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

class MainActivity {
  private static final Logger
LOG = LoggerFactory.
getLogger(MainActivity.class);

  public void someMethod() {
    LOG.info("SLF4J info");
  }
}
```

SLF4J in Android

Die Vorzüge von SLF4J können auch mit Android genutzt werden. Java Util Logging (JUL) ist seit Java 1.4 Teil des Java Development Kits und auch im Android Software Development Kit (SDK) vorhanden. SLF4J hingegen bietet eine leichtgewichtige Implementierung des eigenen API an (slf4j-android), die alle Statements einfach an das Android Logging weiterleitet [4]. Es existiert eine auf Android angepasste Implementierung von Log4j 1.x (android-logging-log4), die nicht mehr aktiv weiter entwickelt zu werden scheint [5]. Außerdem gibt es eine auf Android angepasste Implementierung von Logback (logback-android) [6].

An dieser Stelle soll die kontrovers geführte Diskussion über Für und Wider von logback oder Log4j nicht geführt werden. Vor dem Hintergrund, dass Log4j 2.x für Android noch nicht verfügbar ist [7], ist logback-android als native Implementierung die vielversprechendste Lösung, um Wissen und gewohnte Funktionalitäten

Im Kern handelt es sich bei logback-android um einen Fork von Logback, aus dem Abhängigkeiten zu Klassen entfernt wurden, die im Android SDK nicht verfügbar sind.

zum Thema Logging aus dem Java-Umfeld auch in Android nutzen zu können.

logback-android vs. Logback

Im Kern handelt es sich bei logback-android um einen Fork von Logback, aus dem Abhängigkeiten zu Klassen, die im Android SDK nicht verfügbar sind, entfernt wurden (beispielsweise JMS, JMX, JNDI oder Servlets). Dafür stellt logback-android zusätzliche Features für Android bereit, wie *LogcatAppender* und *SQLiteAppender* [8]. Die aktuelle Version von logback-android 1.1.1-6 basiert auf Logback Version 1.1.1, das lässt sich immer an den ersten drei Stellen der Versionsnummer erkennen.

Alle Bestandteile sind Open Source. Das API SLF4J steht unter permissiver MIT-Lizenz. logback-android selbst erbt das duale Lizenzmodell von Logback aus der Eclipse Public License (EPL) – diese wird auch von JUnit verwendet – und der GNU Lesser General Public License (LGPL). Das soll einen möglichst unkomplizierten Einsatz ermöglichen. Durch die LGPL werden Inkompatibilitäten mit der GPL vermieden. Die EPL erlaubt den Einsatz in Bereichen, in denen Einschränkungen durch die LGPL befürchtet würden [9].

Das grundlegende Aufsetzen in Android verhält sich wie bei Logback. Es werden die Abhängigkeiten zum Logging-API (SLF4J) und zur Implementierung (in diesem Falle logback-android) benötigt (Listing 1).

Es ist empfehlenswert, logback-android im *apk*-Scope zu deklarieren – vergleichbar mit dem *runtime* Scope in Maven. Dadurch stehen die Klassen bei der Entwicklung nicht zur Verfügung, sondern erst zur Laufzeit. Das stellt sicher, dass stets gegen das SLF4J-API programmiert wird. In Sonderfällen kann aber davon abgewichen werden. Log-Statements können dann wie gewohnt im Code gegen das SLF4J-API erfolgen (Listing 2).

Die Konfiguration erfolgt in der von Logback bekannten *logback.xml*, die bei Android im Verzeichnis *assets* abgelegt wird. Listing 3 zeigt ein einfaches Beispiel: Hier werden die Log-Statements aller Logger, die mit Level *DEBUG* oder höher abgegeben wurden, an das systemweite Android-Log übergeben. Android speichert neben der eigentlichen Nachricht eines Log-Statements die folgenden Metadaten [10]:

- Package-Name der App (konfiguriert in der *AndroidManifest.xml* der App)
- Log-Level, Synonympriorität (vorgegeben durch den Aufruf im Code)
- Tag (durch logback-android konfigurierbar)
- Prozess-ID (PID) und Thread-ID (TID)
- Zeitstempel (werden durch das System gesetzt)

In Listing 3 wird der Tag mittels *tagEncoder* auf den Namen des Loggers auf zwölf Zeichen gekürzt gesetzt. Im *encoder*-Tag kann man die eigentliche Nachricht formatieren. Hier bieten Logback und logback-android die Möglichkeit, die eigentliche Nachricht mit weiteren Informationen anzureichern [11]. Da Android bereits viele Metadaten abspeichert, wird in Listing 3 darauf verzichtet und nur die eigentliche Nachricht übergeben. Auch der bei anderen Appendern notwendige Zeilenumbruch (*%n*) ist hier nicht notwendig. Die Konfiguration kann alternativ direkt in der *AndroidManifest.xml* oder im Code erfolgen. Letzteres erfordert Scope *compile* in *build.gradle* [12].

Der in Listing 2 gezeigte Aufruf, erzeugt mit der Konfiguration aus Listing 3 beispielsweise folgende Ausgabe im Android-Log (kopiert aus Android Studio):

```
01-22 12:23:33.800 24191-24191/info.schnatterer.logbackandroiddemoI/
                                                    i.s.l.MainActivity: SLF4J info
```

Diese folgt folgendem Format:

```
<Zeitstempel> <PID>-<TID>/<Package Name> <Log Level>/<Tag>: <Nachricht>
```

Mit diesem Konfigurationsmechanismus lassen sich auch komplexere Anwendungsfälle realisieren.

Fortgeschrittene Anwendungen umsetzen

Dadurch, dass Android-Apps Clientanwendungen sind, bestehen dort zum Teil andere Anforderungen an das Logging als beispielsweise bei einer Java-EE-Anwendung, die auf einem Server läuft. Man möchte beispielsweise Logs in der Anwendung anzeigen, sie mit der Anwendung versenden, das Log-Level zur Laufzeit ändern oder Crash Reports versenden, um die Ursache von Fehlern zu erörtern. Die Umsetzung dieser Anforderungen existiert zwar nicht out of the Box, man kann sie aber mit wenig Zusatzaufwand realisieren. Die Beispielanwendung zu diesem Artikel [13] erlaubt es, auf Knopfdruck Logs gegen SLF4J und JUL abzusetzen (**Abb. 2**).

Die Anwendung zeigt, wie man in der *logback.xml* das Schreiben auf täglich wechselnde Logdateien konfigu-

riert, Inhalte von Logdateien in einer *TextView* anzeigt, das Log-Level für den Root Logger sowie Logcat- und File-Appender zur Laufzeit (per *Preference*) einstellt, alle Logdateien per E-Mail versendet und die aktuelle Logdatei in einem externen Editor anzeigt. Die dabei benötigte Logik, die von logback-android abhängt, ist in der überschaubaren Klasse *Logs* gekapselt, die in ein eigenes wiederverwendbares Projekt [14] ausgelagert wurde. Da diese Funktionalität nicht von dem SLF4J-API bereitgestellt wird, muss der Scope der Abhängigkeit zu logback-android in der *build.gradle* auf *compile* verändert werden.

An dieser Stelle ist es auch wichtig, sich die Hierarchie der Logger zu vergegenwärtigen: Der Root Logger ist das Elternelement aller Logger. Stellt man ihn auf einen größeren Log-Level als den Logcat Appender, hat der Root Logger Vorrang. An einem Beispiel:

- Root-Log-Level: *WARN*
- Logcat-Log-Level: *INFO*
- Ergebnis: Es werden auch bei Logcat nur Statements mit Level ab *WARN* geloggt.

Der einfachste Weg ist es, hier die gleiche Einstellung für alle Appender zu verwenden und sie über den Root Logger zu steuern. Wenn unterschiedliche Level für die Appender verwendet werden sollen, ist es am einfachsten, den Root Logger auf *ALL* und die Log-Levels pro Appender einzustellen. Ansonsten muss stets darauf geachtet werden, dass der Root Logger ein gleich großes oder größeres Level als die Appender hat, um unerwartetes Verhalten zu vermeiden.

Was in diesem Beispiel nicht gezeigt wird, ist die Anbindung an einen Crash-Reporting-Anbieter. Mit der Bibliothek ACRA ist dies jedoch relativ einfach möglich. Hier kann man deklarativ die Logdatei an einen Crash Report anhängen [15]. Das Backend, an den der Crash Report versendet wird, lässt sich in ACRA konfigurieren. Es stehen selbstgehostete Open-Source-Lösungen (z. B. Acralyzer) oder kommerzielle Dienste (z. B. Hockeyapp) zur Verfügung [16].

Zu guter Letzt soll an dieser Stelle noch auf eine Einschränkung von logback-android hingewiesen werden: Es gibt derzeit keine Möglichkeit, Log-Statements, die direkt auf das Android-Log geschrieben werden, auf SLF4J umzuleiten. Dadurch fallen Log-Statements, die von Android-Fremdbibliotheken oder von Android selbst mit dem Package-Name der App geschrieben werden, nicht unter die Kontrolle von logback-android. Sie werden deshalb weder durch die konfigurierten Log-Levels eingeschränkt noch an die Appenders weitergeleitet.

ProGuard und Lint

Nutzt man ProGuard, um das APK zu optimieren, sollten die *proguard-rules.pro* entsprechend der Anleitung erweitert werden, um zur Laufzeit noch alle benötigten Klassen zur Verfügung zu haben [17]. Erfahrungsgemäß ist das Hinzufügen des folgenden zusätzlichen Statements hilfreich: *-dontwarn org.slf4j.***. Aufwendiger wird es bei der Verwendung des Android Linters. Die von Logback übernommene Abhängigkeit zu *javax.mail* steht im Android SDK nicht zur Verfügung. Hier gibt es mehre Lösungen. Wenn man den *SMTPAppender* nicht verwendet, kann man das logback-android JAR in dem entsprechenden Check des Linters mittels *lint.xml* ignorieren [18]. Alternativ kann man den Check in der *build.gradle* komplett deaktivieren. Von Letzterem ist jedoch abzuraten, da damit auch Funde bei anderen Abhängigkeiten ausgeschlossen werden. Verwendet man den *SMTPAppender*, fügt man eine Implementierung für *javax.mail* zu den Abhängigkeiten hinzu. Eine Möglichkeit ist android-mail [19].

Abb. 2: Darstellung des Logs innerhalb der Anwendung

Alternativen und Größe

In den letzten Jahren wurden viele Logging-Frameworks speziell für Android entwickelt, z. B. Timber [20] und Logger [21]. Dabei handelt es sich um kleine, leichtgewichtige Frameworks, welche die Verwendung von *android.util.Log* komfortabler machen. Die Leichtgewichtigkeit ist ihr Vorteil gegenüber logback-android. Sie sind schneller eingebaut und konfiguriert, nur wenige Kilobyte groß und haben nur wenige hundert Methoden. Letzteres ist aufgrund der Beschränkung der Dalvik-Executable-(DEX-)Dateien auf 65 536 Methoden nicht

Listing 3: logback.xml

```
<configuration>
  <appender name="logcat" class="ch.qos.logback.classic.android.LogcatAppender">
    <tagEncoder>
      <pattern>%logger{12}</pattern>
    </tagEncoder>
    <encoder>
      <pattern>%msg</pattern>
    </encoder>
  </appender>
  <root level="DEBUG">
    <appender-ref ref="logcat" />
  </root>
</configuration>
```

unerheblich. Zum Vergleich: logback-android und seine Abhängigkeiten haben ca. 4 500 Methoden (gemessen mit dex-method-counts [22] als Differenz zweier gleicher APKs mit und ohne logback-android v1.1.1-6). Die Vorteile von logback-android liegen in der umfangreicheren Funktionalität und Konfigurierbarkeit sowie der Wiederverwendung vorhandenen Wissens aus der Java-Welt. Gemessen an den GitHub-Stars sind Timber und Logger mit mehreren Tausend Stars bekannter als logback-android mit weniger als 500. Wie repräsentativ diese Zahlen sind, ist jedoch fragwürdig, da selbst Logback und SLF4J, obwohl seit Jahren De-facto-Standard in der Java-Welt, weniger als tausend Stars aufweisen.

Fazit

Dieser Artikel basiert auf den Erfahrungen, die bei der Entwicklung der App nusic erlangt wurden [23]. Hier ist logback-android seit Jahren mit den oben beschriebenen Funktionalitäten erfolgreich im Einsatz und ermöglicht die einheitliche Konfiguration des Log-Levels, von Fremdbibliotheken und Logging in Dateien. Das ermöglicht die Anzeige in der App und das Versenden per E-Mail. Wer die aus Java gewohnten Logging-Funktionalitäten und seine Erfahrungen auch unter Android einsetzen möchte und nicht am DEX-Methodenlimit kratzt, dem ist logback-android zu empfehlen. Durch die Abstraktionsschicht SLF4J wird zudem ein potenzieller späterer Austausch des Logging-Frameworks vereinfacht. Auch für reine Android-Entwickler ohne Java-Historie können die Funktionalitäten von logback-android von Interesse sein. In jedem Fall ist es sinnvoll, vor der Entscheidung für ein Logging-Framework die Alternativen zu evaluieren. Einen besonderen Fokus verdient hier die Integration des Logging-Frameworks mit dem verwendeten Crash-Reporting-Anbieter.

Johannes Schnatterer ist Solution Architect bei der TRIOLOGY GmbH in Braunschweig. Technologisch ist er dort in den Bereichen JavaEE und Web tätig und versucht mit besonderem Fokus auf Qualität, Open-Source-Enthusiasmus, einem Hauch von Pedantismus und der Pfadfinderregel die IT-Welt jeden Tag ein bisschen besser zu machen.

Johannes.schnatterer@triology.de

Links & Literatur

[1] Weiss, Tal: „We Analyzed 30,000 GitHub Projects – Here Are The Top 100 Libraries in Java, JS and Ruby“: http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/

[2] Legacy APIs: http://slf4j.org/legacy.html

[3] log4j2 Dependencies: https://logging.apache.org/log4j/2.x/runtime-dependencies.html

[4] slf4j-android: http://www.slf4j.org/android/

[5] Logging with Log4J in Android: https://code.google.com/archive/p/android-logging-log4j/

[6] logback-android: http://tony19.github.io/logback-android/

[7] How to use log4j2 with Android (Logcat)?: https://issues.apache.org/jira/browse/LOG4J2-951

[8] logback android FAQ: https://github.com/tony19/logback-android/wiki/FAQ

[9] logback-Lizenz: http://logback.qos.ch/license.html

[10] Logcat: https://developer.android.com/studio/command-line/logcat.html

[11] logback-Konfiguration: http://logback.qos.ch/manual/configuration.html

[12] logback-android-Manifest: https://github.com/tony19/logback-android/wiki#androidmanifestxml

[13] logback-android-Demo: https://github.com/schnatterer/logback-android-demo

[14] logback-android-utils: https://github.com/schnatterer/logback-android-utils

[15] ACRA-Logdatei: https://github.com/ACRA/acra/wiki/AdvancedUsage#adding-your-own-log-file-extracts-to-reports

[16] ACRA-Backends: https://github.com/ACRA/acra/wiki/Backends

[17] ProGuard: https://github.com/tony19/logback-android/wiki#proguard

[18] logback-android-Demo lint: https://github.com/schnatterer/logback-android-demo/blob/master/app/lint.xml

[19] logback-android lint Errors: https://github.com/tony19/logback-android/issues/113

[20] Timber: https://github.com/JakeWharton/timber

[21] Logger: https://github.com/orhanobut/logger

[22] dex-method-counts: https://github.com/mihaip/dex-method-counts

[23] nusic: https://github.com/schnatterer/nusic