Java aktuell

iJUG Verbund
www.ijug.eu

Anwendungen schneller machen
Java-Performance-Analyse mit YourKit

Der neue Trend
Domain-driven Design Patterns mit Java EE 8 / Jakarta EE

Tipps vom Experten
Hilfe, ich muss JavaScript programmieren!

Ausgabe 05/2018
Java aktuell
Praxis. Wissen. Networking. Das Magazin für Entwickler
D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR
4 191978 304903 05


Early Bird
bis zum
28. Sep.
2018
DOAG
Konferenz + Ausstellung
20. - 23. November
in Nürnberg
2018.doag.org
PROGRAMM
ONLINE
mit rund 450 Vorträgen
Eventpartner:
AOUG
SOUG
swiss oracle user group
iJUG
Verbund
ORACLE

Probleme beim Umstieg auf Java 9 und höher

Bei Java geht es jetzt Schlag auf Schlag. Bereits sechs Monate nach Veröffentlichung von Java 9 ist das Release 10 herausgekommen. Die Migration auf diese neuen Versionen kann mit Schwierigkeiten verbunden sein.

„Write Once Run Anywere" lautet der Grundsatz der Programmiersprache Java. Ein Programm ist also auf allen Plattformen ohne Einschränkung ablauffähig und alte Programme laufen in der Regel auch mit neueren Java-Versionen. Dieses wichtige Prinzip wurde bei Java 9 und höher vernachlässigt.

Vorteil der neuen Versionen ab Java 9 ist die Möglichkeit zur Modularisierung. Wer diese Eigenschaften nutzen möchte, braucht allerdings entsprechend modularisierte Bibliotheken. So wird beispielsweise Java Architecture for XML Binding (JAXB), eine Programmschnittstelle in Java, um Daten aus einer XML-Schema-Instanz heraus automatisch an Java-Klassen zu binden, von Java 9 blockiert und muss entweder extra per Kommandozeile freigeschaltet, oder besser noch, durch ein externes Modul ersetzt werden, da es schon ab Java 11 definitiv nicht mehr im JDK enthalten sein wird.

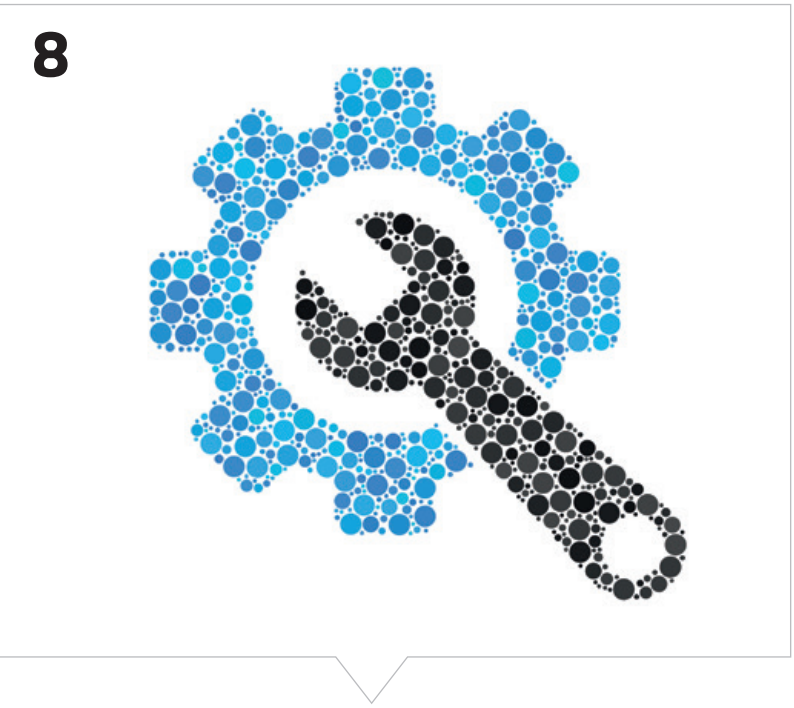
Bisher mussten Tool-Hersteller nicht viel unternehmen, wenn ein neues Java-Release herauskam. Das ist nun beim Umstieg auf das modularisierte Java anders. Deshalb sind sie aufgefordert, mit künftigen Java-Versionen klarzukommen. Oracle bietet keine verbindliche Liste, damit ein Entwickler die fehlenden Module ersetzen kann.

Viel Gluck beim Umstieg auf die neuen Java-Releases,

Ihr

Wolfgang Taschner
Chefredakteur Java aktuell

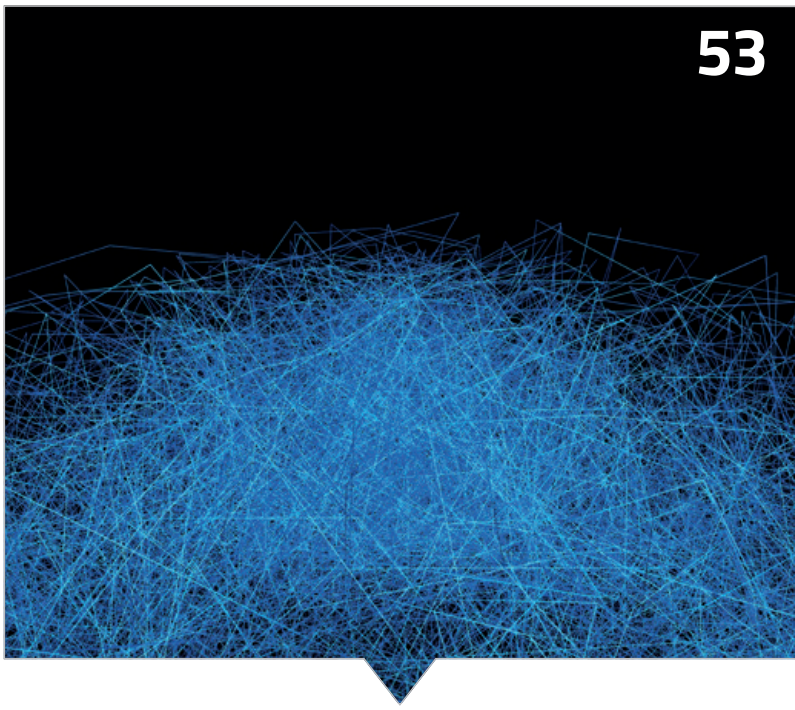
Inhalt



Umfassende Informationen zum kürzlich gestarteten JSR 382, der Konfiguration standardisieren soll

Domain-driven Design mit Java EE oder dem nahenden Jakarta EE praktisch umsetzen

Gestengesteuerte und benutzerfreundliche JavaFX-Oberflächen mit Touch-Bedienung und Anbindung der Eingabegeräte



Interessante Ansätze und Tools, um bei Legacy-Anwendungen Regressionstests anhand des Quellcodes zu generieren

Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community

Das Java-Tagebuch gibt einen Überblick über die wichtigsten Geschehnisse rund um Java.

24. April 2018

SE 10 für Raspberry Pi – Liberica

Liberica ist eine Variante des OpenJDK, von der ich gerade zum ersten Mal gehört habe, ein auf den Raspberry Pi zugeschnittener Build des OpenJDK (9 und 10) inklusive JavaFX und des Device-I/O-API. Das erste Release auf GitHub ist vom Dezember 2017 (JDK 9) – ein weiterer kleiner Hersteller, der seine Nische im OpenJDK-Umfeld gefunden hat.
github.com/bell-sw/Liberica

25. April 2018

Release-Müdigkeit?

JAXenter hat bekannte Gesichter der Java-Community zu Java 10 und insbesondere zu dem neuen Release-Zyklus von sechs Monaten befragt. Viele der Antworten zu Rhythmus und Support-Zeiträumen waren geteilt bis kritisch. Greg Luck von Hazelcast zum Beispiel ist überzeugt, dass die meisten Java-Nutzer erstmal bei Java 8 bleiben und abwarten, insbesondere wie sich die Oracle-Position hinsichtlich Support und Gebühren entwickelt. Markus Eisele von Lightbend glaubt, dass die schnelleren Zyklen für Unternehmen eine zu bewältigende Herausforderung darstellen, dass es aber für Library-Entwickler und Open-Source-Projekte besonders hart werden wird, wenn sie eine größere Anzahl von JDK-Versionen parallel unterstützen müssen. Multi-Release-Jars könnten in seinen Augen eine Lösung für viele Projekte sein. Viele fordern einen Kompromiss – ein Release alle zwei bis drei Jahre sei zu wenig, zwei pro Jahr seien wiederum zu viel. Der Release-Train und das drohende „Support Cliff"-Ende im Januar 2019 (letzter „Public Update" für Java 8) werden die Community noch eine Weile beschäftigen.
https://jaxenter.de/11-experten-java-10-teil-1-69359

3. Mai 2018

Oracle antwortet auf Fragen zum neuen Release-Zyklus

Donald Smith, Senior Director of Product Management bei Oracle, antwortet auf die Kritik am Release-Train und an den Support-Zeiträumen. Es gebe keine „Major Releases" alle sechs Monate, sondern „Feature Releases": Die Folge Java 9, 10, 11 sei eher mit 8, 8u20, 8u40 vergleichbar, die auch ungefähr im Sechs-Monats-Rhythmus herauskamen. Durch den geringeren Änderungsumfang zwischen Releases sei ein Upgrade eine geringere Herausforderung – und die Download-Zahlen sprächen deutlich dafür, dass auch Java 10 angenommen werde. Für Nutzer, die statt neuer Features Stabilität und Bug-Fixes möchten, stellt er nochmal Java 11 heraus sowie den damit einhergehenden mindestens achtjährigen (allerdings kostenpflichtigen) Long-Term-Support. Er betont jedoch auch, dass andere Organisationen sich bereits in der Vergangenheit um nicht mehr von Oracle unterstützte Releases gekümmert haben, insbesondere durch Back-Ports von Fixes aus den neueren Releases; Oracle unterstütze dies.
https://blogs.oracle.com/java-platform-group/update-and-faq-on-the-java-se-release-cadence

6. Mai 2018

Lightbend macht bei Jakarta EE mit

In der letzten Ausgabe des Tagebuchs hatte ich erwähnt, dass Microsoft jetzt bei MicroProfile mitmischt (nicht nur des Namens wegen). Unterschlagen hatte ich, dass Lightbend, die Firma hinter Akka, Scala, Play & Co., praktisch gleichzeitig dazu-gestoßen ist. James Roper von Lightbend ist gerade MicroProfile Committer geworden und stimmt auf „dzone" eine Lobeshymne darüber an, wie hervorragend die Zusammenarbeit läuft (hoffen wir, dass sie auch bald zum Release 1.4 beziehungsweise 2.0 führt). Seiner Ansicht nach ist damit die Antwort auf die „JCP-Frage" gegeben: „Funktioniert Hersteller-Zusammenarbeit bei Spezifikationen im Enterprise-Application-Development-Umfeld generell nicht oder war es konkret der JCP, der nicht funktionierte? Es sei nur der JCP gewesen – und das wiederum sei ein gutes Zeichen für Jakarta EE.

Ich bin noch skeptisch, ob man von MicroProfile einfach auf Jakarta EE und seine kommenden Standardisierungsbemühungen schließen kann. Der Neuanfang hilft jedoch auf jeden Fall, produktivere Formen der Zusammenarbeit zu finden – und wenn MicroProfile mit seiner sehr unkomplizierten und offenen wie öffentlichen Kollaboration da als Vorlage dienen kann, ist schon viel gewonnen.
https://dzone.com/articles/how-the-microprofile-community-will-shape-jakarta

23. Mai 2018

Jakarta-EE-Fortschritt

Mike Milinkovich äußert sich im Eclipse-Blog zum Fortschritt bei Jakarta EE gegenüber den Hauptpunkten des ursprünglichen Plans. Der erste Punkt – Java EE und GlassFish mit allen Lizenzen, TCKs und Referenz-Implementierungen von Oracle übertragen – ist zum großen Teil geschafft: 39 Projekte sind bei der Eclipse Foundation unter EE4J entstanden und mit einer dualen Lizenz (EPL 2.0 sowie GPL V2 mit Classpath Exception) versehen. Punkt 2 – bei Eclipse eine EE-8-kompatible Implementierung bauen – wird laut Mike noch eine Weile dauern und enormen Aufwand erfordern. Punkt 3, die Markenstrategie, sieht zumindest bei zwei zentralen Dingen schon besser aus: neuer Name, neues Logo – check. Der Spezifikations-Prozess, die eigentliche Ablösung des JCP, ist Punkt 4: Die Schaffung der Jakarta EE Working Group im März war ein großer Schritt, aber auf ihre einzelnen Committees wartet noch viel Arbeit. Letzter Punkt: Neue Mitglieder finden. Fujitsu, Payara und Tomitribe sind inzwischen „strategic members"; „participating members" sind unter anderem SAP, Microsoft und Pivotal (Spring!) bis hin zur London Java Community und einer Reihe kleinerer Unternehmen. Daneben beteiligen sich

viele einzelne Entwickler aus der Java-EE-Community am Nachfolger. Es ist nicht alles perfekt, resümiert Mike, hinsichtlich der Komplexität sei man jedoch von dem bisherigen Fortschritt begeistert.
https://www.eclipse.org/ee4j/status.php

30. Mai 2018

WildFly 13 mit „heimlicher" EE-8-Untersützung

Der WildFly Server unterstützt in der gerade freigegebenen Version 13 den vollen Funktionsumfang von Java EE 8. Offiziell (und mit Zertifizierung) soll allerdings erst die im nächsten Quartal erscheinende Version 14 darauf umschwenken. Bis dahin bleibt EE 7 der Standard, was sich aber beim Start über den Parameter „ee8.preview.mode" ändern lässt.
http://wildfly.org/news/2018/05/30/WildFly13-Final-Released

31. Mai 2018

Jakarta-EE-Newsletter

Der neue Eclipse-Newsletter widmet sich ausschließlich Jakarta EE. Wer etwa mal genau wissen möchte, was beim Transfer der Projekte von Oracle so alles passiert – und warum das bitte schön so lange dauert, kann das unter „Jakarta EE Challenges" nachlesen.
https://www.eclipse.org/community/eclipse_newsletter

1. Juni 2018

Ein Shebang für Java

Java 11 soll „JEP 330 – Launch Single-File Source-Code Programs" enthalten, also den direkten Aufruf von (einzelnen) Java-Source-Files wie einem Skript „java HelloWorld.java". Das soll insbesondere Java-Anfängern den Einstieg und das Ausprobieren erleichtern. Hinzugefügt wurde während der Ausarbeitung die Unterstützung des „Shebang" („#!" beziehungsweise hier „#!/path/to/java"), mit dem sich das File dann direkt ohne Java-Kommando ausführen lässt. Daraufhin startete eine Diskussion darüber, ob der Java-Compiler das Shebang verarbeiten können muss oder ob es zwei Typen von Files geben soll: Java-Skript-Files und gültige Java-Source-Files (ohne Shebang). Alternativen wurden auch diskutiert, etwa eine als Java-Kommentar maskierte Version („//!/path/to/java" – dafür ist jedoch Betriebssystem-Unterstützung erforderlich). Nach einer einwöchigen Fristverlängerung für die Diskussion soll jetzt die ursprüngliche Variante, mit reinen Skript-Files im Unterschied zu Source-Files, aufgenommen werden.
https://dzone.com/articles/shebang-coming-to-java

2. Juni 2018

Das Nashorn soll aussterben

Ein weiterer – für Java 11 geplanter – JEP (335) hat zum Ziel, die JavaScript-Engine auf die Liste der aussterbenden Module zu setzen: „@Deprecated(forRemoval=true)". Als Grund wird die hohe Änderungsgeschwindigkeit der Sprachdefinition ECMAScript und ihrer APIs genannt und die damit einhergehenden Anpassungsaufwände im Nashorn-Projekt. Tatsächlich entfernt werden sollen Engine und „jjs"-Tool mit einem weiteren JEP in einem noch nicht näher definierten zukünftigen Release. Ganz sicher scheinen sich die Verantwortlichen nicht zu sein: Die Breite der Nashorn-Nutzung sei schwer zu beziffern, daher wird um Feedback gebeten. Bei eindeutigem (rechtzeitigem) Feedback könne JEP 335 noch vor der Integration in Java 11 gestoppt oder in einem zukünftigen JEP zurückgenommen werden. Wer das Nashorn vor dem Aussterben bewahren will, kann es zum Beispiel über die Mailing-Liste versuchen.
http://mail.openjdk.java.net/mailman/listinfo/nashorn-dev

6. Juni 2018

EE4J-PMC-News

Das EE4J PMC hat ein „Technical Direction"-Dokument veröffentlicht, in dem grundlegende Prinzipien für die Projekte definiert werden. Beispielsweise sollen sich die Projekte nach Möglichkeit auf die Nutzung des Modulsystems (JPMS) vorbereiten, auch wenn das erste Release noch JDK8-basiert sein wird (Vorsicht, Support Cliff!). Alte und selten genutzte Technologien sollen in optionale Komponenten ausgelagert werden. Projekte, die noch „ant" für den Build benutzen, sollen auf den De-facto-Standard „maven" umschwenken". (Wer hat da „gradle" gerufen? Setzen!). Interessant ist, dass eine Empfehlung für einen jährlichen Release-Zyklus – mit vierteljährlichen Minor-Releases – ausgesprochen wird. Wobei das PMC ausdrücklich schreibt, dass dies nicht als tatsächliche Roadmap interpretiert werden soll.
https://projects.eclipse.org/projects/ee4j/pmc

Andreas Badelt
stellv. Leiter der DOAG Java Community
andreas.badelt@doag.org

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich in der DOAG Deutsche ORACLE-Anwendergruppe e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit dem Jahr 2015 ist er stellvertretender Leiter der neugegründeten Java Community innerhalb der DOAG. Beruflich hat er seit dem Jahr 1999 als Entwickler und Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet und ist seit dem Jahr 2016 als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

Die Kunst der Konfiguration (JSR 382)

Anatole Tresch, Trivadis AG

Im Oktober letzten Jahres wurde der JSR 382 [1] gestartet, der Konfiguration standardisieren soll. Höchste Zeit, uns den JSR etwas näher anzuschauen.

Was lange währt, wird endlich gut ...

Einen so zentralen Aspekt zu standardisieren, ist nicht einfach, denn die Meinungen darüber, wie Applikationen sinnvollerweise zu konfigurieren sind, gehen weit auseinander. Bereits die Repräsentation von Konfiguration als Schlüssel-Wertepaare („Map<String,String>“) gegenüber der Repräsentation als Baum (wie „java.util.Preferences“) kann unter Experten leicht zu abendfüllenden Diskussion führen.

Nimmt man dann noch bestehende System- und Umgebungsvariablen, Programm-Argumente, Speicherorte, -formate, Prioritäten und Übersteuerungsmechanismen hinzu, kann man erahnen, welche Vielfalt an Fragestellungen sich auftun. Entsprechend steht der aktuelle JSR am Ende einer Reihe von Versuchen, einen Standard für diese Problem-Domäne zu etablieren.

Im Alltag sind vermutlich viele mit Spring *[2]* und Java EE bereits in Berührung: Spring kennt beispielsweise mit „Environment“ und „PlaceHolderConfigurer“ aufgrund der Historie gleich mehrere SPIs und Java EE lässt standardmäßig nur Konfiguration zur Deployment-Zeit zu. Zudem existieren zahlreiche Projekte, die jedes für sich interessante Ansätze aufzuweisen haben. Die vielversprechendsten Ansätze, die auch gewisse Verbreitung gefunden haben, sind Apa-

che Tamaya *[3]* und MicroProfile Config *[4]*. Konsequenterweise finden sich Ideen aus beiden Projekten auch im JSR wieder. Weitere Details können dem Blog *[5]* entnommen werden.

Was ist Konfiguration?

Wie wird nun Konfiguration modelliert? Ganz einfach: als String-basierte Schlüssel-Wertepaare. Dieser Ansatz bringt gleich mehrere Vorteile mit sich:

- Das Konzept ist denkbar einfach
- Umgebungs- und System-Variablen sowie Programm-Argumente lassen sich einfach damit abbilden
- Der Ansatz ist kompatibel mit „java.util.Properties" und weitgehend auch mit den meisten anderen gängigen Formaten („xml", „ini", „yaml" und „json")
- Konfigurationswerte lassen sich einfach vergleichen und können problemlos serialisiert, gecacht, gespeichert oder übertragen werden
- Die Repräsentation als Text ist Plattform- und Sprach-unabhängig

Im neuen Standard werden Konfigurations-Properties über eine Instanz von „Config" abgefragt, die über das „ConfigProvider"-Singleton bezogen werden kann. Dabei kann auch ein „ClassLoader" übergeben werden, was vor allem in komplexeren Szenarien wie klassischem Java EE wichtig ist *(siehe Listing 1)*.

Konkrete Konfigurationswerte lassen sich mit drei Methoden abfragen: „getValue(String,Class)" setzt das Existieren eines entsprechenden Konfigurationseintrages voraus und wirft eine „NoSuchElementException", wenn kein Eintrag gefunden werden konnte. Ist ein Wert nicht zwingend, so sollte man „getOptionalValue(String,Class)" benutzen. Die Methode „access(String)" wird später im Zusammenhang mit dynamischer Konfiguration diskutiert. Möchte man wissen, welche Schlüssel aktuell definiert sind, lassen sich diese mit „getPropertyNames()" abfragen. Allerdings können nicht alle Konfigurationsquellen Auskunft über die definierten Schlüssel geben, sodass die zurückgelieferte Schlüsselliste nicht zwingend vollständig ist *(siehe Listing 2)*.

Den aufmerksamen Leserinnen und Lesern wird nun nicht entgangen sein, dass Werte als String repräsentiert werden, wir aber im obigen Beispiel die Konfiguration typisiert abfragen können. Hier fehlt also noch ein Baustein, der diese Lücke schließt, nämlich „Converter" *(siehe Listing 3)*. Alle gängigen Java-Basistypen sind bereits standardmäßig unterstützt. Will man weitere benutzerdefinierte Typen unterstützen, so lassen sich entsprechende Converter-Klassen ganz einfach mit den „ServiceLoader" registrieren.

Konfigurationsquellen und Ordinals

Wie setzt sich nun eine konkrete Konfiguration zusammen? Welche Konfigurationsquellen werden berücksichtigt, wie können Konfigurations-Properties einer Quelle die Properties einer anderen Quelle übersteuern? Dazu ist die „ConfigSource"-Schnittstelle genauer zu betrachten, die von allen Konfigurationsquellen implementiert werden muss *(siehe Listing 4)*.

```
// Access using current Thread classloader
Config config = ConfigProvider.getConfig();

// Access using explicit classloader
ClassLoader classloader = …
Config config = ConfigProvider.getConfig(classloader);
```

Listing 1

```
Iterable<String> getPropertyNames();

<T> T getValue(String key, Class<T> type);
<T> Optional<T> getOptionalValue(String key, Class<T>
type);
<T> ConfigValue<T> access(String key);
```

Listing 2

```
public interface Converter<T>{
    T convert(String value);
}
```

Listing 3

```
public interface ConfigSource {
   String CONFIG_ORDINAL = "config_ordinal";
   int DEFAULT_ORDINAL = 100;

   String getName();
   default int getOrdinal() {...}

   String getValue(String propertyName);
   default Set<String> getPropertyNames() {...}

   Map<String, String> getProperties();

   default void setOnAttributeChange(Consumer<Set<Str
ing>>
            reportAttributeChange) {...}
}
```

Listing 4

Wie zu erwarten, liefert eine „ConfigSource" Konfigurations-Properties. Wir sehen auch, dass die Properties wie erwähnt als „Strings" modelliert sind. Zudem besitzt jede Konfigurationsquelle einen Namen, der eindeutig sein sollte. Interessant ist die „getOrdinal"-Methode; sie definiert die Signifikanz einer Konfigurationsquelle.

Werte einer Quelle mit einem höheren Ordinal-Wert übersteuern Werte mit gleichem Schlüssel von Quellen mit tieferem Ordinal. Haben zwei Konfigurationsquellen denselben Ordinal-Wert, wird der vollqualifizierte Klassenname als zusätzliches Kriterium hinzugezogen, um immer eine definierte Reihenfolge garantieren zu können. Analog zu Convertern können eigene Konfigurationsquellen mit dem Java-„ServiceLoader" registriert werden. In vielen Fällen ist das jedoch gar nicht nötig, da standardmäßig bereits folgende Quellen mit entsprechenden Default-Ordinals automatisch zur Verfügung gestellt und registriert sind *(siehe Tabelle 1)*.

Da gewisse Betriebssysteme Umgebungsvariablen mit einem Punkt nicht unterstützen, sieht der JSR für Umgebungsvariablen (und nur für diese) einen erweiterten Lookup-Mechanismus vor. Der Schlüssel „com.ACME.size“ wird beispielsweise wie folgt aufgelöst (dabei wird der erste gefundene Wert als Resultat zurückgeliefert):

1. Exakter Match: „com.ACME.size“
2. Alle „.“ durch „_“ ersetzen: „com_ACME_size“
3. Alle „.“ durch „_“ ersetzen und zu Uppercase konvertieren: „COM_ACME_SIZE“

Nun könnte man den Eindruck erhalten, dass das alles doch sehr statisch ist. Dazu ein genauerer Blick auf die Default-Implementation der „ConfigSource#getOrdinal()“-Methode *(siehe Listing 5)*.

Wir können also den Ordinal-Wert unserer Konfigurationsquelle beeinflussen, indem wir einen Eintrag unter dem Schlüssel „config_ordinal“ mit dem gewünschten ganzzahligen Ordinal-Wert ablegen. Möchten wir zum Beispiel die Signifikanz von System-Properties gegenüber Environment-Properties umkehren (etwa um in einem Docker-Container Environment-Properties zum Übersteuern zu nutzen), so genügt es, mit „java -Dconfig_ordinal=200 ...“ ein entsprechendes System-Property zu setzen. Der Mechanismus ist natürlich analog auch auf andere Konfigurationsquellen anwendbar.

Quelle	Ordinal
System-Properties	400
Environment-Properties	300
META-INF/javaconfig.properties (Classpath Resource)	100

Tabelle 1

```
default int getOrdinal() {
   String configOrdinal = getValue(CONFIG_ORDINAL);
   if(configOrdinal != null) {
       try {
           return Integer.parseInt(configOrdinal);
       }
       catch (NumberFormatException ignored) {

       }
   }
   return DEFAULT_ORDINAL;
}
```

Listing 5

```
ConfigValue<String> value = …;
ConfigValue<Integer> intVal = value.as(Integer.class);
```

Listing 6

```
// Default value setzen und abfragen
ConfigValue<Integer> intVal =
                    value(as(Integer.class).withDefault(8080);
intVal = value.as(Integer.class).withStringDefault(“8080”);
Integer defaultIntValue = intVal.getDefaultValue();

// Abfrage Schlüssel und Werte
System.out.println(“Key  : “ + intVal.getKey());
System.out.println(“Value: “ + intVal.getValue());

Optional<Integer> optIntValue = intValue.getOptionalValue();
```

Listing 7

Dynamische Konfiguration und Konfigurationsänderungen

In vielen Fällen ist Konfiguration eine relativ statische Angelegenheit. In verteilten Systemen muss allerdings manchmal sehr rasch auf Konfigurationsänderungen reagiert werden können. Es kann auch sein, dass man eine gewisse Zeit vor Änderungen geschützt sein will. In beiden Fällen kann ein „ConfigValue“ benutzt werden, der über die bereits vorher erwähnte Methode „Config#access(String)“ mit „ConfigValue<String> value = ConfigProvider.getConfig().access (“app.host.port”);“ bezogen werden kann. „ConfigValue“ implementiert ein Builder-Muster, das im Zusammenhang mit einem Konfigurationswert folgende Aktionen ermöglicht:

- Typisiert (optional mit eigenen Converter) oder als „Set“ beziehungsweise „List“ zugreifen
- Mit einen Default-Wert versehen
- Mithilfe von Platzhaltern auflösen
- Mithilfe einer Lookup-Chain über mehrere Schlüssel abfragen
- Cachen
- Callback-Listener fur Konfigurationsanderungen hinzufügen

Zuerst ein genauer Blick auf die Methoden für den typisierten Zugriff. Dazu startet man mit der „ConfigValue“-Instanz aus dem vorigen Beispiel und typisiert den Wert als Integer *(siehe Listing 6)*.

```
ConfigValue<List<Integer>> listVals = intVal.asList();
ConfigValue<Set<Integer>> setVals = intVal.asList();
```

Listing 8

```
Converter<PortList> portListConverter = …;
ConfigValue<PortList> configuredPortList =
                    value.useConverter(portListConverter);
```

Listing 9

```
ConfigValue<T> value = …;
value = value.evaluateVariables(true);
```

Listing 10

```
String tenant = getCurrentTenant();
Integer timeout = config.access("some.server.url")
               .withLookupChain(tenant, ${projectStage})
               .getValue();
```

Listing 11

Ähnlich wie mit „Config" kann man den aktuellen Schlüssel beziehungsweise Wert als erforderlichen oder optionalen Wert abfragen. Wie erwähnt lässt sich auch ein Default-Wert setzen, und zwar entweder als String oder typisiert *(siehe Listing 7)*. Sind in der Konfiguration mehrere Werte durch Komma getrennt konfiguriert, kann man auf einen Wert auch als „List" oder „Set" zugreifen *(siehe Listing 8)*. Man kann explizit für diesen Konfigurationswert einen eigenen „Converter" setzen *(siehe Listing 9)*.

Um einen Konfigurationswert temporär zu cachen, kann die „ConfigValue#cacheFor(long,TimeUnit)"-Methode benutzt werden. Bei „ConfigValue<String> val = value.cacheFor(5,TimeUnit.MINUTES);" wird sich der Wert für die nächsten zehn Minuten nicht mehr verändern.

Platzhalter und Lookup-Chains

Der JSR unterstützt auch Platzhalter im UNIX-Stil, die ihrerseits wiederum auf andere Konfigurationseinträge zeigen können. Dazu ist auf einem „ConfigValue" die Methode „evaluateVariables(boolean)" anzuwenden, die einen neuen „ConfigValue" erzeugt *(siehe Listing 10)*. Bei Lookup-Chains werden mehrere Schlüssel in definierter Art und Weise evaluiert, um so typische Übersteuerungsmechanismen abzubilden. *Listing 11* zeigt dazu ein Beispiel.

Angenommen, „tenant" ist „myTenant" und „${projectStage}" evaluiert zu „Production", ergibt sich folgende Lookup-Reihenfolge:

1. „some.server.url.myTenant.Production"
2. „some.server.url.myTenant"
3. „some.server.url.Production"
4. „some.server.url"

Der effektiv aufgelöste Schlüssel lässt sich mit „ConfigValue #getResolvedKey()" abfragen.

```
interface ConfigChanged {
        <T> void onValueChange(String key, T oldValue,
T newValue);
}

ConfigValue<Integer> port = …;
port.onChange((k, o, n) -> { … });
```

Listing 12

```
ConfigBuilder builder = ConfigProvider.getInstance()
                              .getConfigBuilder();
Config config = builder
                        .withDefaultConverters()
                        .withDefaultSources()
                        .withDiscoveredSources()
                        .withSource(new MyConfig-
Source())
                        .build ;
```

Listing 13

```
@ConfigProperty
private Integer intVal;

@ConfigProperty(name="a.b.prop")
private Optional<Integer> optionalIntVal;

@ConfigProperty(name="a.b.prop", defaultValue="1234")
private Provider<Integer> providedIntVal;
```

Listing 14

Callbacks und „ConfigBuilder"

Schließlich kann auf einem „ConfigValue" auch ein Callback registriert werden, um über Konfigurationsänderungen informiert zu werden *(siehe Listing 12)*. In gewissen Szenarien möchte man den Lifecycle der „Config"-Instanzen nicht zentral verwalten lassen. Auch hierzu bietet der JSR eine Lösung an: den „ConfigBuilder". Mit diesem API können „Config"-Instanzen Schritt für Schritt definiert und geladen werden. Die erzeugten Konfigurationen sind vollkommen unabhängig und lassen sich nach Bedarf benutzen oder wieder dereferenzieren. Aus Platzgründen wird nicht weiter im Detail auf das API eingegangen, *Listing 13* zeigt allerdings ein kurzes Beispiel, wie es benutzt wird.

„@Config Injection"

Bisher haben wir uns auf das Java-SE-API konzentriert; der aktuelle JSR definiert jedoch auch ein Injection-API, das mit CDI entsprechend Java-EE-kompatibel ist. Die wichtigste Annotation ist dabei „@ConfigProperty". Mit ihr können Konfigurations-Properties in-

```
Config config = ConfigProvider.getConfig();
String host = config.getValue("service.host", String.class);
int port = config.getValue("service.port", int.class);
```

Listing 15

```
Consumer<Set<String>> configChangedListener = …;
config.registerConfigChangedListener(configChangedListener);
```

Listing 16

jiziert werden. Auch ist es möglich, Default-Werte zu setzen oder auf Werte als „Optional"- und „Provider"-Instanzen zuzugreifen. Ob auch „ConfigValue"-Instanzen injiziert werden können, ist aktuell noch offen *(siehe Listing 14)*.

Atomarität von Konfiguration

Zum Abschluss noch ein etwas weniger einfach ersichtlicher Aspekt: die Atomarität von Konfiguration beziehungsweise Konfigurationszugriffen. Dazu stellt man sich ein Szenario wie in *Listing 15* vor.

Wie gesehen, ist die aktuelle Konfiguration durch eine nach Ordinal geordnete Liste von „ConfigSource"-Instanzen definiert. Diese Liste wird bei jedem Zugriff durchlaufen, um den aktuell richtigen, konkreten Wert zu evaluieren. Nun stellt sich die Frage, wie sich sicherstellen lässt, dass zwischen dem Zugriff auf „host" und dem auf „port" keine Konfigurationsänderung in einer „ConfigSource" stattfindet, die unter Umständen in einer inkonsistenten „host:port"-Kombination enden kann. Der JSR macht hierzu selbst keine Garantien, allerdings kann auf einer „Config"-Instanz ein Listener registriert werden, der bei Änderungen entsprechend informiert wird *(siehe Listing 16)*.

Somit kann betroffener Client-Code auf relevante Änderungen hören. Damit das funktioniert und während des Aufrufs eines Listeners selbst keine weiteren Konfigurationsänderungen möglich sind (der Zugriff während dieser Zeit also atomar ist), muss die Implementation der „Config"-Klasse transitiv auch von den registrierten „ConfigSources" Atomaritätsgarantien erhalten. Dies führt uns zur Methode „ConfigSource#onAttributeChanged(Consumer <Set<String>>)". Diese Methode erlaubt es, Änderungen in einer registrierten „ConfigSource" an die entsprechende „Config"-Instanz anzuzeigen.

Ob eine Änderung effektiv zu einer veränderten Konfiguration führt, hängt natürlich von Anzahl, Inhalt und Reihenfolge (Signifikanz/Ordinal) aller „ConfigSource"-Instanzen ab. Für die umgebende „Config"-Instanz ist dieser Mechanismus allerdings ausreichend. Wird nämlich während der Evaluation eines Wertes eine Änderung einer „ConfigSource" angezeigt, muss die „Config"-Instanz den evaluierten Wert verwerfen und zuerst alle registrierten Listener über eine allfällige Änderung informieren. Somit ist die Atomarität grundsätzlich gegeben. Dies erfordert aktuell einiges an Client-Code. Deshalb werden diese Fragestellungen derzeit im JSR intensiv diskutiert und es bleibt abzuwarten, ob hier noch das API erweitert wird oder man für diese Anwendungsfälle auf Vendor-spezifische APIs zurückgreifen muss.

Kritik und Ausblick

Grundsätzlich macht der Standard gute Fortschritte und es ist zu erwarten, dass in naher Zukunft ein „Early Draft Review" verfügbar sein wird. Das definierte API ist recht einfach gehalten und es kann eine Mehrheit der Anwendungsfälle unterstützt werden. Natürlich gibt es noch einiges an Feinarbeit zu leisten, und dazu ist auch ein breites Feedback der Community wichtig. Aber es scheint so, dass Java es nach vielen Startschwierigkeiten in naher Zukunft tatsächlich schafft, als erstes Ecosystem diesen wichtigen Cross-Cutting-Concern zu standardisieren.

Weiterführende Links

[1] *https://jcp.org/en/jsr/detail?id=382*
[2] *https://cloud.spring.io/spring-cloud-config/*
[3] Apache Tamaya: *http://tamaya.incubator.apache.org*
[4] *https://github.com/eclipse/microprofile-config*
[5] *http://javaeeconfig.blogspot.ch/2014/08/overview-of-existing-configuration.html*
[6] *https://github.com/eclipse/ConfigJSR*

Anatole Tresch
anatole.tresch@trivadis.com

Anatole Tresch war nach dem Wirtschaftsinformatik-Studium an der Universität Zürich mehrere Jahre lang als Managing Partner, Berater, Lead Engineer und technischer Architekt tätig. Er sammelte weitreichende Erfahrungen in allen Bereichen des Java-Ökosystems von IOT bis Java EE. Aktuell ist er Principal Consultant bei der Trivadis AG und beschäftigt sich dort mit Cloud-Architektur, Resilient Design, DevOps und verteilten Systemen allgemein. Zudem ist er JCP-Star-Specification-Lead und aktiver Apache-PPMC-Member.

Zielführende Fragmentierung mit Microservices und Docker vs. Herausforderungen des Monitorings – die Bedeutung des Journald von Linux

Roland Grieder, GRADUAL.CONSULTING

Während immer mehr Konsens darüber entsteht, dass Software-Entwicklung mit (Micro-) Services effektiv ist, ergeben sich seitens des System-Managements zusätzliche Herausforderungen: Wie überwacht man am besten die Vielzahl von Komponenten, deren Zahl sich auch dynamisch ändern kann, insbesondere mit Docker. Der Artikel zeigt, wie man mit Bordmitteln von Linux, insbesondere Journald, hochwertig (nicht nur) Logs zentralisieren und auswerten kann. Darüber hinaus leitet der Autor eine schlanke, moderne Architektur für das Monitoring ab, die auf Journald aufbaut – und in der Praxis funktioniert.

Über Microservice-Architekturen und flankierende Maßnahmen wie DevOps oder Transformation zu agilen Organisationsformen gibt es mittlerweile detaillierte Erfahrungen. Die Beteiligten unterstützen den allgemeinen Konsensus, dass mit Maß verteilte Services in einem passenden Kontext attraktiv sind.

Je mehr Systeme verteilt sind, desto aufwendiger ist es, diese zu überwachen. Das Monitoring genauer betrachtet ergibt, dass sich heutzutage eine Vielzahl von Systemen etabliert hat, wobei die Funktionalität überlappt, sodass sich der Aufwand pro Tool und Komponente unbefriedigend multipliziert: Alarmierung, Logs zentralisieren und korrelieren, Systeme für Audit-Trails; in Bezug auf Security gehören im weiteren Sinne Konsolen-Logger dazu, hinzu kommen Überwachungstools für Prozesse auf dem Betriebssystem (nebst Systemd) etc. Einige dieser Komponenten können nicht ohne konzeptionelle Überlegungen zu entsprechender Spezialkonfiguration mit dynamisch gestarteten beziehungsweise gestoppten Komponenten umgehen.

Verschiedene Arten des Monitorings

Monitoring lässt sich mit einer Reihe von Attributen qualifizieren, wobei sich die gängig verwendeten Begriffe überschneiden. Nebst einer vollständigen Sicht lässt sich der Zielumfang für die Monitoring-Architektur abstecken, die dieser Artikel herleitet:

- **Aktiv vs. passiv**
  Eine Applikations-Komponente wird von einer aktiven Komponente nach deren Status gefragt oder ein Benutzer löst betrieblich relevante Log-Records aus, die ein Monitoring-System (passiv) einsammelt
- **Whitebox vs. Blackbox**
  Innenleben oder Außenwirkung
- **Negativ, positiv**
  Fehlermeldungen vs. Status-Information; horizontal entlang eines Ablaufs vs. vertikal, pro Server
- **Layer**
  Information kann aus Ablauf- beziehungsweise Benutzer-Perspektive abgegriffen werden (Probe), pro Applikations-Komponente, auf Ebene von Middleware, System-Prozessen oder Kernel inklusive Netzwerk
- **Log-Level beziehungsweise Zweck**
  Von Alarmierung, Warnung, Information, Hilfe beim Debuggen bis zu detaillierter Ablauf-Analyse (Trace)
- **Technisches Format**
  Gängig sind Files, Meldungen (SNMP, Syslog etc.), Messaging und seit einigen Jahren Journald
- **Veränderbarkeit**
  Monitoring-Information lässt sich gegen nachträgliche Änderungen schützen
- **Struktur**
  Text, strukturierter Kopf mit „Tail" oder gänzlich strukturiert, wobei die Struktur im Record mitgeliefert wird oder implizit gegeben ist
- **Situation einer Komponente**
  Information zur Installation, Starten/Stoppen oder Laufzeit
- **Art der überwachten Kenngröße**
  Neben der Information über das Ablaufverhalten von Programmen werden auch Kapazitäten überwacht (auf verschiedenen Layern) mit Schwellwerten sowie spezifische Systemgrößen (CPU nice, Angaben zu Paketverlusten auf dem Netzwerk etc.)
- **Aufbereitung**
  Direkt abgegriffene Information, aufbereitet und ergänzt, gerechnete Indikatoren oder korrelierte Information
- **Mengengerüst**
  Monitoring-Information kann diskret erfolgen, also Ereignis-basiert, periodisch, als Zeitreihe oder laufend (stream)
- **Technische Domäne**
  Applikation, Betriebssystem, System-Ressourcen, Netzwerk, Security etc.
- **Zeitversatz**
  Monitoring in Echtzeit oder unter Umständen verzögert

Ein Einschub: Besonders hinweisen möchte der Autor im Kontext von Microservices auf passive Probes, ein Instrument, mit dem SLAs unmittelbar ausgewertet werden können inklusive Ursachen-Analyse, statistisch oder pro Einzelfall. Benutzer lösen Logs aus, die IDs zur Session, zum Geschäftsablauf, Request, aufrufender beziehungsweise aufgerufener Komponenten etc. enthalten. Passive Probes geben also Auskunft über die geschäftliche Nutzung oder Auswirkung einer Störung bis hinunter auf die Ebene der Komponenten-Interaktion; zum Beispiel lässt sich das Antwortzeitverhalten einfach als etappierte Sicht von Round-Trips auswerten. Dynamisch ist also deutlich effizienter und zuverlässiger, als Bottom-up-Impacts zu korrelieren, zudem nützlich für die Security.

Praktische Erfahrung mit Log-Zentralisierung – die gute Seite …

Angefangen hat alles in einer Bank, die den Großrechner durch ein modernes Kernbankensystem abgelöst hat, mit einer verteilten Architektur für Front-Komponenten und Zusatzdienste. Es kamen Linux-Systeme zum Einsatz, das Deployment wurde vollständig mit RPM und insbesondere Ansible automatisiert.

Der Aufwand für die Fehler-Abklärungen wäre mit „remote Login", „find" für Logs, „tar" und trivialen Werkzeugen wie „grep" zu groß geworden, deshalb hat das Engineering-Team die Logs zentralisiert. Dabei wurde eines der gängigen Tools eingesetzt mit Agent, Server (Logs parsen), No-SQL-Datenbank und Portal zur Auswertung. Es wurde festgestellt, dass die Tools in der Grund-Architektur austauschbar sind; Ansatz, Nutzen wie auch die Probleme sind allerdings überall gleich.

Doch zunächst zum Nutzen; alle Logs wurden sorgfältig getaggt: Umgebung, Applikation, Unterkomponente, Version, Start/Run/Stop, Server-Name etc. Damit war man tatsächlich in der Lage, technische Incidents inklusive Problem-Analyse innerhalb einer Viertelstunde zu lösen. Man stand kurz davor, dass die Entwickler per Self-Service ihre Tickets effizient bearbeiten können, statt Konstellationen oft mühsam mit zusätzlichen Daten in ihrer Umgebung zu reproduzieren beziehungsweise iterativ Logs vom Operating-Team zu bestellen.

…und die schlechte – das war zu lösen

Das Engineering-Team meldete, der Log-Server sei bereit. Dann geschah es: „Bis wann läuft das eBanking (auf der Abnahmeumgebung) wieder?" Die Antwort lautete „Hm … wir schauen umgehend und melden uns." Ein sehr unangenehmes „Hm", es gab keinerlei

Anzeichen einer Störung. Noch schlimmer eine Woche später: Der Job-Scheduler hatte sich verabschiedet, wortreich wie eine Diva (2 GB Logs), worauf der Log-Server sich verschluckt hatte, er wurde instabil. Das Engineering-Team hat wieder nichts davon erfahren. Auch das Problem-Management klappte nicht, da diese interessanten Daten teilweise unterwegs verloren gingen – inakzeptabel.

Die erste Regung bestand darin, den Code der Open-Source-Lösung anzuschauen, um Bug-Fixes beizusteuern. Dabei wurde festgestellt, dass diese Art von Log-Tools als architektonisches Prinzip die Applikationen möglichst nicht stören soll, was sie mit Zuverlässigkeit und damit Wertschöpfung für den Betreiber bezahlen:

- Als Protokoll kommt standardmäßig das verlustbehaftete UDP zum Einsatz, wobei die Log-Tools auf dem applikatorischen Layer den Datenverkehr nicht zusätzlich absichern. Bei viel Netzverkehr profitiert also TCP ein Stück weit davon, dafür werden Logs zunehmend unzuverlässiger übertragen. Also TCP.
- Die Agents arbeiten mit eher kleinen, flüchtigen Puffern, die interne Verarbeitung im Server (ohne Not) auch. Überlauf oder Ausfälle führen unmittelbar zu Datenverlust.
- Durch das Parsen ist der Server recht langsam. Die vielen Retries der wartenden Agenten haben ihn soweit nachvollziehbar instabil gemacht. Der Log-Server muss generell hoch dimensioniert sein, sonst baut sich bei Last der Eingangs-Puffer immer mehr auf. Wenn das Telefon klingelt, bevor ein Alarm eintrifft, kann man zu Recht über den Sinn der Lösung diskutieren. Wenn Logs mit etwas zufälligem Zeitverzug eintreffen, stellt sich auch die Frage, wie zuverlässig die sofortige Log-Korrelation noch funktioniert.
- Um täglich ein Log-File zu hinterlassen, wird das laufend gefüllte Log-File umbenannt. Damit nun der Log-Agent das neue File nicht neu auswertet, arbeitet er nicht mit Dateinamen, sondern mit den dahinterliegenden IDs, den „Inodes". Diese bleiben nämlich bei diesem Vorgang stabil. Konzeptionell unsauber wird es, wenn man alte Log-Files löscht: Das File ist weg, der Inode entsprechend frei zur erneuten Vergabe, aber die Buchführung des Agent erfährt davon nichts. Das Engineering-Team fühlte sich trotz geringer Wahrscheinlichkeit auch mit dieser Entdeckung schlecht, da die künftige Betriebsfirma hohe Ausfallentschädigungen riskiert; niemand würde doch einen Webshop ohne Transaktionen für Bestellungen und Zahlungen implementieren, wo es um deutlich geringere Beträge geht.

Noch eine andere Eigenschaft ließ am Ansatz dieser gängigen Lösungen zweifeln: In den Java-Applikationen kommt Log4J V.2 mit einem sauber gepflegten Thread-Context zum Einsatz. Diese direkt maschinell auswertbare Information wird im ersten Schritt zu Fließtext umgewandelt, also massiv entwertet, und das teilweise irreversibel: Je nach Situation zum Zeitpunkt eines Absturzes sehen Thread-Context wie Log-Message anders aus. Es gibt also keine klare Spezifikation, sondern einen Medienbruch: Man soll-

te auf dem Log-Server einen Parser schreiben, der – generell gedacht pro Version der Applikations-Komponente – irgendwie mit der Vielfalt der möglichen Log-Meldungen klarkommen sollte. Die Realität hat gezeigt, dass es möglich ist, einige Felder zu extrahieren, aber irgendwann der Rest eines Log-Record im Textformat belassen werden musste.

Es schien, als seien die Logs einst erfunden worden, das Thema jedoch dann lieblos gehandhabt wurde. Um nur zwei, aber wesentliche Faktoren zu nennen: Betreiber stellen traditionell keine Anforderungen an Logs, geschäftsferne Logik wird zurückhaltend finanziert. Denn eigentlich wäre eine saubere Lösung doch gar nicht so schwierig …

Journald – ein strukturierter, störungsresistenter und sich selbst verwaltender Puffer

Das Ziel ist, mit möglichst nur einer hochverfügbaren Lösung diverse Monitoring-Informationen – strukturiert und direkt maschinell verwertbar – zuverlässig und in Echtzeit zentral auszuwerten. Starten wir mit dem Problem des Puffers: Weder soll die Geschäftsfunktionalität der Applikations-Komponente durch mögliche Staueffekte für das Monitoring beeinträchtigt werden, noch soll das Memory des Agenten überlaufen. Also brauchen wir einen persistenten, störungsresistenten Puffer für strukturierte Daten. Dieser möge sich selbst verwalten, ohne Skript-Wunderwerk, um alte Logs zu managen, und idealerweise zentral sein.

Hier kommt nun der Clou, ein großer Schritt für das Monitoring, noch erstaunlich wenig bekannt, aber in unserem Zusammenhang bestens geeignet: Alle Linux-Derivate haben sich auf Journald geeinigt als zentrales Subsystem für Logging mit mehr Benefits als nur den erwähnten Charakteristika. Ein Journal ist wie eine Datei, die für diverse Szenarien von Störungen gesichert ist. Neben den genannten Kriterien verhindert das Subsystem nachträgliche Änderungen, ideal für Audit-Trails. Mit Rechten lassen sich Bedürfnisse unterschiedlicher Leser konfigurieren.

Journald erfährt umfassend, was auf dem System geschieht. Man kann etwa nachvollziehen, wer sich eingeloggt und wann welche Commands beziehungsweise Komponenten aufgerufen hat; dank der technischen Güte schwierig kompromittierbar. Journald arbeitet mit Key-Value Pairs unterschiedlicher Datentypen. Nützliche Felder wie Hostname, Zeitstempel etc. fügt es selbst hinzu. Außerdem ist Journald die sauberste Lösung im Umgang mit Docker (Varianten gibt es mit automatischen Managern wie Kubernetes), indem die Log-Information lokal geschrieben, aber zentral auf das unterliegende Betriebssystem abgelegt wird; insbesondere dafür wurde Journald ursprünglich erfunden.

Die Logs sind nun also strukturiert persistiert, man muss sich um nichts mehr kümmern, da Journald alte Records automatisch gemäß konfigurierbaren Regeln löscht. Wichtig für die weitere Verarbeitung: Das Niveau der technischen Güte halten. Nur so kann man Auditing, Konsolen-Logger, Alarming etc. zusammenlegen. Also keine Files mehr einbauen. Auch hier hilft Journald: Records können in ein beliebiges Journal (eben nicht File) ausgelagert oder von einem Server auf andere Journalds konzentriert werden. Eine umfassende Log-Zentralisierung nur mit Board- Mitteln von Linux.

Was das mit Java zu tun hat

Log4J Version 2 und höher ist auf einem guten Stand, wir müssen nur noch den wohlstrukturierten Thread-Kontext ins Journald übertragen, wofür Log4J einen speziellen „Appender" vorsieht. Dazu nimmt man den Logger des Applikations-Servers und konfiguriert einen Appender dazu für Journald. Dann legt man einen Log4J-Logger zu Journald für alle Applikationen zusammen an, als geteilte Ressource. Als Basis für den erwähnten Appender kann eine Open-Source-Minimalversion aus dem Internet geladen werden. Um diese zu integrieren, schaut man im Internet für seinen Application Server unter dem Stichwort „custom appender" nach. Achtung, Stolperstein: Ein Java-Appender integriert die Adapter-Library für Journald mit JNA. Der Adapter wird in einem anzugebenden Verzeichnis zwischengespeichert, das ausführbar sein muss; für einen speziellen User finden wir dazu eine gute Lösung. *Listing 1* zeigt die Konfiguration des Pfads.

Der Autor hat in seiner Implementation zusätzlich Funktionalität eingebaut, um Keys vorzugeben, zu denen er die Werte (Values) erhalten möchte, oder um alle verfügbaren Felder abzurufen. Außerdem fügte er Metadaten-Felder hinzu: Umgebung, Applikationskomponente, Version etc. Einige Daten lassen sich direkt beschaffen (Manifest-File für den Applikationsnamen sowie die Version), andere mit Regeln ableiten (das Namenskonzept der Server gab Hinweise zum Verwendungszweck einer Umgebung) und einige Felder setzt man statisch (Run-Log). Fazit: Ein Appender und vernetzte Journalds – und ein Entwickler hat alle Informationen beisammen, die er zum Debuggen braucht.

Journald für Entwickler

Nachdem die Information im Journald liegt, wäre es gut zu wissen, wie man sie wieder ausliest. Im Internet gibt es eine Vielzahl von Tutorials – für Administratoren. Für die Bedürfnisse der Entwickler folgen die wesentlichen Kommandozeilen, sodass man Journald gleich ausprobieren kann.

Wichtig: Zunächst erhält ein Benutzer mit dem zentralen Command „journalctl" meist gar nichts zurück, weil ihm die Berechtigungen fehlen. Der erste Schritt besteht also darin, zum Administrator zu

```
JAVA_OPTS="$JAVA_OPTS -Djna.tmpdir=/path/to/loaddir"
JAVA_OPTS="$JAVA_OPTS - Djava.io.tmpdir=/path/to/loaddir
```

Listing 1

```
Sudo nano /etc/system/journald.conf #Im Konfigurations-File:
…
[Journal]
Storage=persistent    #Damit wird alle Information persistiert.
```

Listing 2

```
ExecStartPost=/bin/sh -c 'tail -f "/var/logs/legacyfile.log" | systemd-cat -t "beispiel-app"' #-t für Attribute, “tags”.
```

Listing 3

```
Journalctl -r #Umgekehrte Reihenfolge, neueste Einträge zuerst (mein Favorit…).
journaldctl -n 20 #Analog tail, die letzten n Einträge.
journalctl --since “2018-03-28” --until “20187-01-11 17:21”
journalctl --since yesterday --until “1 hour ago”
journalctl _UID=55 _PID=8088 #Eigne, reservierte Felder Journald beginnen mit dem Zeichen “_”.
journalctl /user/bin/bash #Commands gefiltert nach Verzeichnis.
journalctl -p err #Priority level, wie Syslog: 0 emerg, 1 alert, 2 crit, 3 err, 4 warning, 5 notice, 6 info, 7 debug.
Journalctl -p 0..4 #Die höchsten fünf Severity-Stufen.
Journalctl -no-pager #Output am Stück für Weiterverarbeitung.
jouirnalctl -u nginx -u elastic.service --since today -o json
#Liefert JSON-Records, -o json-pretty (tatsächlich) schön formatiertes JSON im Zeilenformat.
Journalctl -f #Wie tail -f, laufendes Auslesen neuer Einträge.
```

Listing 4

gehen. Dabei ist zu überprüfen, dass Journald persistent aufgesetzt ist *(siehe Listing 2)*. *Listing 3* zeigt, wie man Legacy-Logs aus Services mit „systemd-cat“ ins Journald umleitet (mäßig strukturiert), und *Listing 4* die Abfragen.

Die Zukunft der traditionellen Log-Tools

Wenn man zu einem traditionellen Log-Tool einen Agenten für Journald sucht, findet man den nicht. Logisch: Traditionelle Log-Server können Logs parsen, was ja mit Log4J zu Journald wegfällt, aber nicht zuverlässig und unverfälscht übertragen; das macht viel vom Wertangebot des Journald aus. Falls es also doch einmal einen solchen Agenten geben sollte, ist Vorsicht geboten. Mit dieser Feststellung wird unmittelbar die Zukunft der aktuellen Architektur dieser Tools fraglich.

Im Verbund mit Journald ergibt eine No-SQL-Datenbank mit Portal weiterhin Sinn: Man kann feiner granular als nur ganze Feld-Inhalte suchen, auch mit Wildcards oder statistisch. Außerdem erhöht eine grafische Benutzerschnittstelle die Effizienz sowie die von Dritten deutlich. Zudem sind flankierende Maßnahmen zu treffen, um die technische Güte auch in dieser Komponente so hochzuhalten, dass eine moderne Log-Architektur ihren breiten Einsatzzweck ohne Bedenken erfüllen kann *(siehe Abbildung 1)*.

Zur Implementation: Mit „At-least-once Guarantee“ (mehr geht mit einer No-SQL-Datenbank aufgrund deren Konsistenz-Mechanismen nicht, sonst kann man beispielsweise JSON in PostgreSQL verwenden) verbindet ein schlanker Agent in Go Journald durch eine ebenfalls sehr schlanke Server-Komponente mit dem Bulk-Interface der No-SQL-Datenbank ElasticSearch. Der Agent kann Legacy-Logs parsen, Werte anpassen (etwa die Severity von Testsystemen deckeln), ein Datenpaket aus Journald zum Transfer anbieten und bei Störungen den Server wechseln.

Der Server vermittelt über viele Agenten hinweg den Datentransfer in das Bulk- Interface, wobei erst das Commit der Datenbank an die Agenten zurückgesendet wird. Zudem sortiert der Server bereits Alarme aus. Beim Datentransfer wurde darauf geachtet, dass Paketgrößen zu denen auf Netzwerk-Ebene passen, um die Effizienz weiter zu erhöhen. Einige positive Charakteristika dieser Vorgehensweise sind:

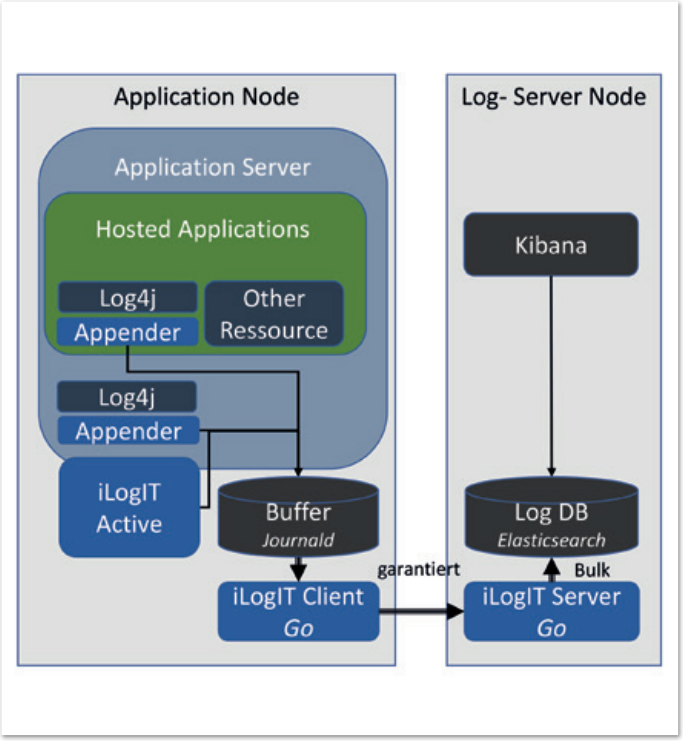


Abbildung 1: Die Komponenten einer modernen, konsolidierten Monitoring-Architektur

- Der Durchsatz dieses Log-Servers ist enorm (gegenüber einem parsenden Server im Leerlauf bereits Faktoren von mehr als 10.000), daher kaum Retries. Der Server selbst ist mit seinen wenigen Code-Zeilen robust.
- Dank der getakteten Übertragung in Echtzeit und des Verzichts auf einen weiteren Buffer auf dem Server sind die Probleme mit Zeitverzug wie Synchronität, um unmittelbar eintreffende Logs zu korrelieren, gelöst.
- Auch der Server ist in Go implementiert, sodass er direkt auf den Knoten von ElasticSearch installiert werden kann, ohne relevante technische Abhängigkeiten. Der Server und Inserts in Elastic sind abwechselnd tätig; man spart eigene VMs für den Server.
- Mit dem Bulk-Interface indexiert ElasticSearch wesentlich effizienter, die Bandbreite nimmt massiv zu, der Ressourcen-Bedarf beziehungsweise die Kosten ab.

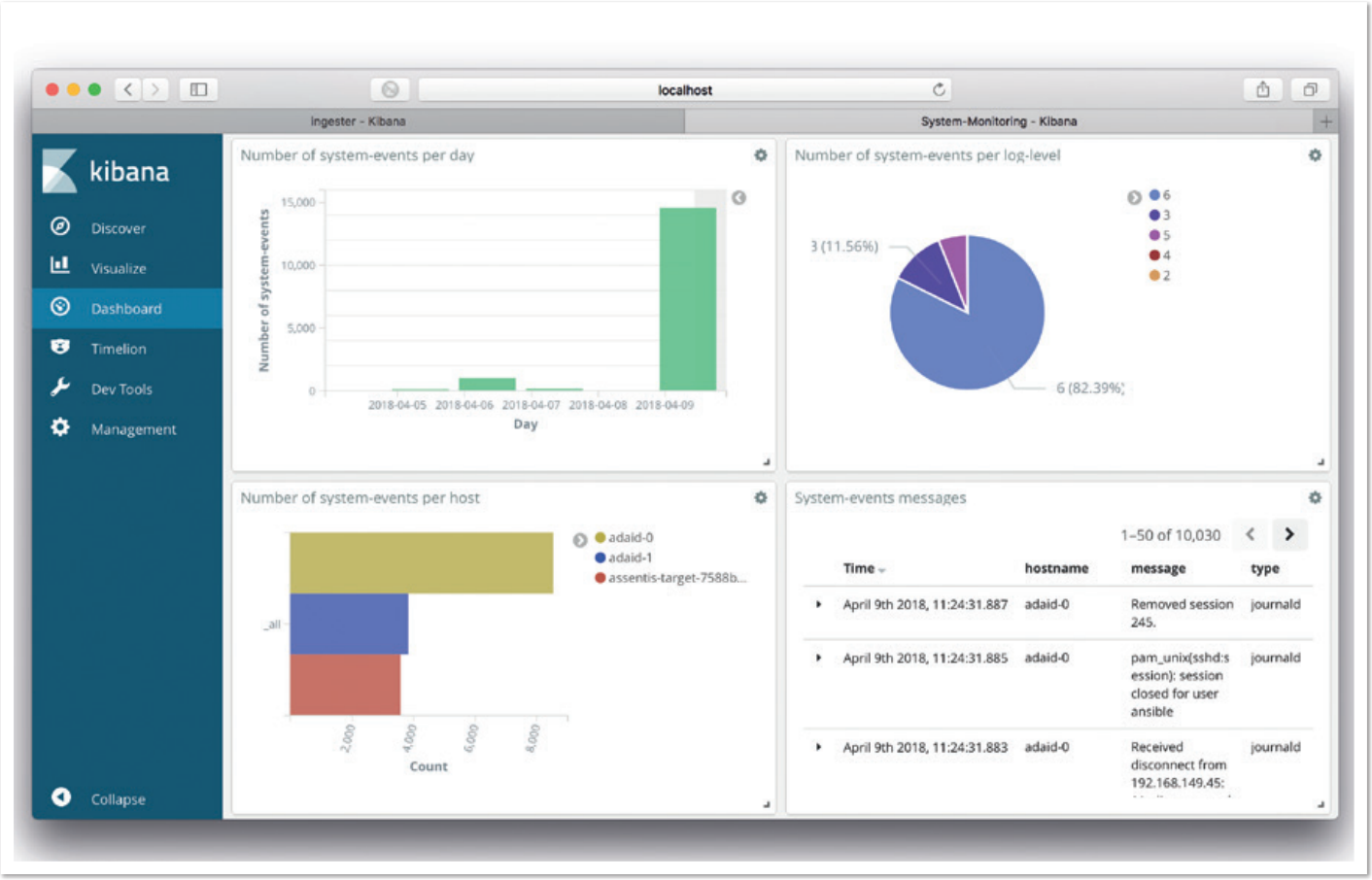


Abbildung 2: Mit wenigen Mausklicks lassen sich solche Dashboards herstellen

- Ohne Logs und Audit-Trails brauchen die Online-Services das Dateisystem gar nicht mehr, sobald diese installiert sind. Nicht nur applikatorisch sind die Services damit state-less, sondern auch betrieblich. Im Sinne der Komplexitätsreduktion wurde gleich auch auf Server-Backups verzichtet; geht auf einer VM etwas schief, wird diese neu angelegt und automatisiert installiert.

Der moderne Log-Server wird zum Systems-Management-Tool

Da nun umfassende Information pro Server zuverlässig, unveränderlich und in Echtzeit zentralisiert vorliegt, kann man auf diverse Spezialtools verzichten. Mit dem Essen kommt der Appetit: Was kann man sonst noch weglassen? Beim Blick auf die Liste mit Eigenschaften des Monitorings sind alle Aspekte abgedeckt, außer aktives Monitoring, da ist noch eine abfragende Komponente erforderlich: JMX, Messaging- Kapazitäten, Betriebssystem-Prozesse und der Status des Kernels inklusive Netzwerk. Eine aktive Komponente ist attraktiver, als eine traditionelle Alarmierungslösung für diesen Zweck aufzubauen, mit großer funktionaler Überlappung, wodurch mehrfach konfiguriert werden muss.

Diese letzte Komponente der Lösung fragt und schreibt beziehungsweise rechnet also periodisch Statuswerte ins Journald. Sie überwacht direkte oder gerechnete Messgrößen gegenüber Schwellwerten. Was fehlt noch? Die Funktion, die entscheidet, welche Prozesse laufen dürfen, sollen oder verhindert werden. Dann bleiben die File-Systeme, die aber meist in anderer Zuständigkeit und für eine Log-freie Installation auch nicht mehr wichtig sind. Deren Zustandsüberwachung noch zu integrieren, wäre sehr einfach.

Damit ist bereits der Stand einer ausgereiften Systems-Management-Suite in Bezug auf Monitoring erreicht – sogar übertroffen *(siehe Abbildung 2)*. In der Praxis hat sich diese moderne Monitoring-Architektur bislang durchweg bewährt. Sie schont das Nervenkostüm, Aufwände und Betriebskosten; daher heißt das Tool freundlich „iLogIT".

Roland Grieder
roland.grieder@gradual.consulting

Roland Grieder hat an der ETH Zürich Elektrotechnik studiert, bei IBM unter anderem die Lehrgänge zu Consulting und Architektur absolviert sowie an der HSG einen Abschluss für General Management, CSA, absolviert. Eine Spezialisierung ist die (Neu-) Planung von Rechenzentren zusammen mit der Architektur für Betriebsautomatisierung, so als Head of Projects einst bei Swisscom Wholesale zur Überwachung des Schweizerischen Festnetzes. Diverse Projekte folgten bei Banken und Versicherungen, die die IT komplett umgestellt haben.

Domain-driven Design Patterns mit Java EE 8 / Jakarta EE

Sebastian Daschner, Sebastian Daschner – IT-Beratung

Domain-driven Design (DDD), das in Eric Evans Buch beschrieben ist, macht sich zum Ziel, Softwaremodelle zu definieren, die möglichst akkurat die fachliche Domäne abbilden. DDD beschreibt besonders die Notwendigkeiten, Fachwissen mit Experten in der Domäne abzustimmen und zu kommunizieren, von allen geteilte Domänen-Fachbegriffe zu verwenden sowie das Verständnis der Domäne immer weiter zu vertiefen und daraufhin die Repräsentation in der Software zu verfeinern. Der Artikel zeigt, ob es möglich ist, diese Ideen mit Java EE oder dem nahenden Jakarta EE umzusetzen.

```
@Entity
public class ElectricGuitar {

    @Id
    private long id;

    private Model model;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public Model getModel() {
        return model;
    }

    public void setModel(Model model) {
        this.model = model;
    }
}
```

Listing 1

```
@Embeddable
public class Model {

    @Basic(optional = false)
    private String brand;

    @Basic(optional = false)
    private String name;

    protected Model() {
        // required by JPA
    }

    public Model(String brand, String name) {
        this.brand = brand;
        this.name = name;
    }

    public String getBrand() {
        return brand;
    }

    public String getName() {
        return name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
return false;
        Model model = (Model) o;
        return Objects.equals(brand, model.brand)
                && Objects.equals(name, model.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(brand, name);
    }

    @Override
    public String toString() {
        return brand + ", " + name;
    }
}
```

Listing 2

Zuerst ein Blick auf die Konzepte und darauf, wie das Java-EE-Programmiermodell beim Bauen der Domänen-Modelle unterstützt. Die grundlegenden Konzepten beginnen mit „Bounded contexts", die die Zuständigkeiten, Grenzen und Bedeutungen einer Applikation im System beschreiben. Eine bestimmte Domänen-Entität wie „Customer" ist in darin enthalten. Grenzen, Zuständigkeiten und potenzielle Überschneidungen sind in der „context map" des Systems festgehalten. In einer Microservice-Architektur ist ein „Bounded context" typischerweise als einzeln deploybare Applikation abgebildet.

Sogenannte „Entities" repräsentieren die Entitäten der fachlichen Domäne. Eine wichtige Eigenschaft ist dabei, dass sie innerhalb ihrer Domäne identifizierbar sind. Die Identität der „Entities" entscheidet, welches Objekt verwendet wird.

Die folgenden Beispiele behandeln eine Instrumentenfabrik-Applikation. Ein gefertigtes Instrument ist eine identifizierbare Entität – als Java-Klasse implementiert. *Listing 1* zeigt, wie die „JPA @Entity"- Annotation die identifizierbaren Entitäten auf die Datenbank abbildet. Sie zwingt uns Entwickler, für die Entität auch einen „Identifier" zu bestimmen, hier mit „@Id" abgebildet. Das Modell des Instruments ist ein „value object", das als „embedded JPA"-Feld abgebildet wird.

Value objects

„Value objects" sind Typen der fachlichen Domäne, die nicht-identifizierbare Werte widerspiegeln. Für diese Typen ist es irrelevant, welche Objekt-Instanz innerhalb des Business-Prozesses verwendet wird. Prominente Beispiele dafür sind Adressen, Währungswerte oder Java-Enums. Idealerweise sind „Value objects" immutable, was uns ermöglicht, Instanzen bedenkenlos wiederzuverwenden. Das Instrumenten-„Model" ist ein Beispiel dafür. Die Modelle sind durch ihre Marke und den Modellnamen identifiziert und können von verschiedenen Stellen gleichzeitig referenziert werden.

„Value objects" sind in JPA als „embeddable" abgebildet – „Entities" würden erfordern, dass wir einen „Identifier" definieren. Die Datenbank-Tabelle „ElectricGuitar" wird alle nicht-transienten „embeddable" Felder wie zum Beispiel „Model" als Spalten einbetten *(siehe Listing 2)*.

„Value objects" implementieren typischerweise „equals()" und „hashCode()", um sicherzustellen, dass fachlich identische Instanzen auch als solche erkannt werden. Aufmerksame Leser haben vermutlich schon festgestellt, dass die Implementierung nicht vollständig „immutable" ist. Das ist der JPA-Spezifikation geschuldet, die uns zwingt, einen parameterlosen Konstruktor mit mindestens „protected" Sichtbarkeit zu definieren. Manche Mapping-Frameworks wie zum Beispiel Hibernate ermöglichen es, die Sichtbarkeit dieses Konstruktors auf „private" zu verringern. Das ist jedoch nicht JPA-konform und führt zu nicht-portablen Anwendungen.

Services

„Services" sind dafür zuständig, Geschäftslogik auszuführen, die nicht Teil spezifischer Entitäten oder „Value Objects" sind. Sie definieren die Einstiegspunkte der Business Use Cases und verwalten die Entitäten. Sie halten die separaten Schritte des Prozesses zusammen.

```
@Stateless
public class InstrumentCraftShop {

    @Inject
    InstrumentMaker instrumentMaker;

    @PersistenceContext
    EntityManager entityManager;

    public ElectricGuitar craftInstrument() {
        ElectricGuitar instrument = instrumentMaker.build();
        return entityManager.merge(instrument);
    }
}
```

Listing 3

In der Java-EE-Welt sind „Services" als „managed beans" implementiert, entweder mit CDI oder als EJBs. „Services", die die Use-Case-Einstiegspunkte darstellen, manchmal auch „Boundaries" genannt, sind gewöhnlich als EJBs realisiert. EJBs haben von Haus aus schon oft benötigte technische Funktionalitäten wie zum Beispiel Transaktionsmanagement.

Der „InstrumentCraftShop Service" repräsentiert den Einstiegspunkt, um neue Musikinstrumente zu erstellen *(siehe Listing 3)*. Die „Boundary"-Einstiegspunkte delegieren typischerweise komplexere Geschäftslogik an weitere „Services". Diese „Delegates" wie zum Beispiel „InstrumentMaker" stehen dank Dependency Injection in den Beans zur Verfügung.

Aggregates

„Aggregates" repräsentieren komplexere Entitäten, die selbst aus mehreren Entitäten beziehungsweise „Value Objects" bestehen. Sie werden durch ein einzelnes „Root"-Objekt als Kollektiv behandelt und verwaltet. Dadurch sind Integrität und Konsistenz sichergestellt.

In JPA werden die Persistenz-Operationen auf dem „root entity" eines Aggregate angewendet. Die Operationen kaskadieren dann zu allen anderen Objekten des Kollektivs. Im Beispiel in *Listing 4* behandeln wir einen „GuitarBody"-Typ, der Teil einer gesamten E-Gitarre ist. Er repräsentiert in dieser Form somit den „root entity" des Aggregate.

In unserer Domäne bestehen E-Gitarren aus einem einzelnen Korpus, der zu Zwecken der Nachvollziehbarkeit eine identifizierbare Entität ist *(siehe Listing 5)*. Die „OneToOne"-Relation definiert, dass alle Persistenz-Operationen kaskadieren, die auf dem E-Gitarren-Objekt aufgerufen werden. Damit ist garantiert, dass der Zustand der involvierten Entitäten konsistent bleibt.

Repositories

Alle genannten Persistenz-Operationen sind irgendwie auf die Entitäten anzuwenden. Auf gleiche Weise müssen wir die Entitäten vom Persistenz-Provider aus der Datenbank laden können. DDD-„Repositories" sind dafür zuständig, die Persistenz der Entitäten zu verwalten. Sie kapseln diese Funktionalität, um den Rest der Applikation von Persistenz-Implementierungsdetails fernzuhalten. Nur Entitäten, die innerhalb der Domäne eine eindeutige Identität besitzen, werden durch „Repositories" persistiert und verwaltet.

In Java EE beziehungsweise JPA erfüllt der „EntityManager" bereits diese Funktion. Er wird benutzt, um Domänen-Entitäten zu persistieren, zu laden und zu verwalten. Die Bedingung, dass JPA-„Entities" einen „Identifier" definieren müssen, passt ideal zur Idee von identifizierbaren DDD-„Entities". „Services" können den „EntityManager" „injecten" und verwenden *(siehe Listing 6)*.

```
@Entity
public class GuitarBody {

    @Id
    private long id;

    @Enumerated(EnumType.STRING)
    @Basic(optional = false)
    private Material material;

    @Enumerated(EnumType.STRING)
    @Basic(optional = false)
    private Color color;

    protected GuitarBody() {
    }

    public GuitarBody(Material material, Color color) {
        this.material = material;
        this.color = color;
    }

    public enum Material {
        MAPLE, MAHOGANY
    }

    public enum Color {
        BLACK, RED
    }
}
```

Listing 4

Factories

Das Erstellen der Domänen-Objekte kann komplexere Logik erfordern, als nur einen Konstruktor aufzurufen. Dazu definiert DDD das Konzept der „Factories". Die Idee ist, die Logik zum Erzeugen komplexerer Objekte in separate Methoden oder Klassen auszulagern. Je nachdem, wie sehr die Logik mit bestehenden Instanzen verbunden ist, kann es sinnvoll sein, die „Factories" als Methoden von „Entity"- oder „Value Object"-Typen zu definieren. Für unsere Beispiel-Applikation machen wir jetzt etwas Musik mit unseren Instrumenten. Ein „Value Object" „Music" ist wie in *Listing 7* defi-

```
@Entity
public class ElectricGuitar {

    // id, model, getters & setters from previous definition ...

    @OneToOne(cascade = CascadeType.ALL, optional = false)
    private GuitarBody body;

}
```

Listing 5

```
@Stateless
public class InstrumentCraftShop {

    @Inject
    InstrumentMaker instrumentMaker;

    @PersistenceContext
    EntityManager entityManager;

    public ElectricGuitar craftInstrument() {
        ElectricGuitar instrument = instrumentMaker.build();
        return entityManager.merge(instrument);
    }

    public ElectricGuitar retrieveInstrument(long identifier) {
        return entityManager.find(ElectricGuitar.class,
          identifier);
    }
}
```

Listing 6

```
public class Music {

    private final String description;

    public Music(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }
}
```

Listing 7

```
public class ElectricGuitar {

    // ...

    public Music play() {
        return new Music("Let's rock!");
    }
}
```

Listing 8

```
public class ElectricGuitarCrafted {

    private final Instant instant;
    private final Model model;

    public ElectricGuitarCrafted(Model model) {
        this.model = model;
        instant = Instant.now();
    }

    // getters
}
```

Listing 9

niert. Die Erzeugung dieser „Value Objects" ist stark an den Instrumenten-Typ gekoppelt und wird daher als Methode darin definiert. Das Gleiche gilt, wenn die Erzeugung Informationen aus der Instanz benötigt *(siehe Listing 8)*.

CDI-„Producer" sind ein weiterer Weg, um „Factories", die lose an die Domänen-Objekte gekoppelt sind, zu implementieren. Die CDI-„Producer"-Methoden oder -Felder exponieren die Instanzen, die dann in Beans injiziert werden. Somit repräsentieren CDI „Producer Factories".

Domänen-Events

Domänen-Events treten während der Ausführung der Geschäftslogik auf. Sie haben geschäftslogikspezifische Semantik und entstehen gewöhnlich aus den Business Use Cases. Beispiele für Events sind „InstrumentCrafted" oder „ArticlePurchased". Domänen-Events sind als „value objects" implementiert, die die Informationen zum Event enthalten. In Java implementieren wir Events gewöhnlich als „immutable plain old Java objects" (POJO). Da die Events in der Vergangenheit passiert sind, dürfen sie sich später nicht mehr ändern. Der Typ „ElectricGuitarCrafted" repräsentiert ein Domänen-Event *(siehe Listing 9)*.

Java EE kommt mit einer Funktionalität, die es uns erlaubt, Events abzuschicken und zu konsumieren, nämlich CDI-„Events". Diese tre-

```
@Stateless
public class InstrumentCraftShop {

    @Inject
    InstrumentMaker instrumentMaker;

    @Inject
    Event<ElectricGuitarCrafted> instrumentCreated;

    @PersistenceContext
    EntityManager entityManager;

    public ElectricGuitar craftInstrument() {
        ElectricGuitar instrument = instrumentMaker.build();

        instrumentCreated.fire(
            new ElectricGuitarCrafted(instrument.getModel()));

        return entityManager.merge(instrument);
    }

    // retrieveInstrument() ...
}
```

Listing 10

```
public class CraftedBrandRecorder {

    public void onCraftedInstrument(@Observes ElectricGuitarCrafted event) {
        Model model = event.getModel();
        System.out.println("new instrument crafted for model: " + model);
    }
}
```

Listing 11

ten für gewöhnlich innerhalb der Geschäftslogik auf *(siehe Listing 10)*. Der Typ „Event<T>" wird in unsere Beans injiziert und benutzt, um jegliche Event-Typen abzuschicken, wie zum Beispiel „ElectricGuitarCrafted". Das Event wird in einer CDI-„Observer"-Methode konsumiert, vom Rest der Geschäftslogik entkoppelt *(siehe Listing 11)*.

Seit Java EE 8 ist es durch die „Event#fireAsync()"-Methode und die „@ObservesAsync"-Annotation möglich, direkt mit CDI ein Event asynchron zu behandeln. Das Event wird dann in einem separaten Thread konsumiert.

Fazit

Modernes Java EE macht es möglich, Enterprise-Anwendungen mit Fokus auf der Geschäftslogik zu entwickeln. Anders als es bei J2EE der Fall war, setzt diese Technologie nur minimale Beschränkungen auf die Domänen-Logik. Domänen-Typen müssen keine spezifischen Java-EE-Typen ableiten oder Interfaces implementieren.

Der einfachste Weg, Businesslogik zu schreiben, ist in reinem Java. Die benötigten querschneidenden Aspekte werden durch Annotationen konfiguriert. Die Flexibilität speziell der CDI- und JPA-Spezifikationen ermöglicht es Entwicklern, sich darauf zu fokussieren, was einen Mehrwert für die Applikation schafft: die Geschäftslogik.

Jakarta EE, der Nachfolger von Java EE, wird auf Java EE 8 basieren. Damit behalten die hier vorgestellten Konzepte und Ideen auch in der Zukunft weiterhin ihre Gültigkeit.

Sebastian Daschner
mail@sebastian-daschner.com

Sebastian Daschner ist selbstständiger Java-Consultant und Trainer und programmiert begeistert mit Java (EE). Seine Kunden unterstützt er mit Workshops und Beratung in Enterprise-Themen. Er ist Autor des Buchs „Architecting Modern Java EE Applications". Sebastian Daschner nimmt am Java Community Process teil, ist in den JAX-RS, JSON-P und Config Expert Groups vertreten und entwickelt an diversen Open-Source-Projekten. Für seinen Beitrag in der Java Community und Java-Ökosystem wurde er mit den Titeln „Java Champion", „Oracle Developer Champion" und „JavaOne 2016 Rockstar" ausgezeichnet. Er spricht auf internationalen IT-Konferenzen wie JavaLand, JavaOne oder Jfokus. Neben Java und Java EE benutzt Sebastian Daschner intensiv Linux und Container-Technologien wie Docker und Kubernetes. Er evangelisiert Java- und Programmier-Themen unter „https://blog.sebastian-daschner.com" und auf Twitter unter „@DaschnerS". Wenn er nicht gerade mit Java arbeitet, bereist er auch gerne die Welt — entweder per Flugzeug oder per Motorrad.

JavaFX mit MVVM-Pattern, Usability und Gestensteuerung für Leitstände

M.Sc. Mark Gebler, B.Sc. Hannes Walz und Prof. Dr. Gudrun Görlitz, Beuth Hochschule für Technik Berlin, Fachbereich Informatik und Medien

Der Beitrag führt in JavaFX ein, zeigt das Konzept von MVVM in Verbindung mit mvvmFX am realen Beispiel aus der Logistik und gibt einen Ausblick auf gestengesteuerte und nutzerfreundliche JavaFX-Oberflächen mit Touch-Bedienung und Anbindung der Eingabegeräte Leap Motion und der Thalmic Myo.

Die Beuth Hochschule forscht in einem Kooperationsprojekt *(siehe „https://projekt.beuth-hochschule.de/kopgeo")* zu einem universell nutzbaren Gesten- und Interaktionskonzept für Leitstände mit einem situativ geführten, berührungssensitiven, aber auch berührungslosen Bediensystem. Das Ziel besteht in der Entwicklung eines neuartigen Softwaresystems (Dispositionssystems), das die heutigen Anforderungen an die Steuerung und das Monitoring von und in Leitständen erfüllt.

Die Kernaufgabe der Disposition in der Entsorgungslogistik ist die effiziente Routenplanung der verschiedenartigen Entsorgungsfahrzeuge. Der Kooperationspartner „Gesellschaft für Informationssysteme und Prozessautomatisation mbH" (GIPA), der Software für die Entsorgungslogistik entwickelt und vertreibt, befasst sich im Kooperationsprojekt mit der Entwicklung eines Frameworks zur Anbindung von gestengesteuerten User-Interfaces für komplexe verteilte Prozesse auf Basis von Microservices. Komplexe Applikationen sollen sich dabei aus einfachen Bestandteilen flexibel zusammensetzen. Es kommen Straßenkarten-Darstellungen und GPS-verortete Objekte zum Einsatz.

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

<?import javafx.scene.control.Label?>
<?import javafx.scene.layout.StackPane?>
<StackPane xmlns="http://javafx.com/javafx"
           xmlns:fx="http://javafx.com/fxml"
           fx:controller="example2.HelloFXMLController"
           prefHeight="200.0" prefWidth="200.0">

    <Label fx:id="myLabel" text="Hello JavaFX again" />

</StackPane>
```

Listing 1

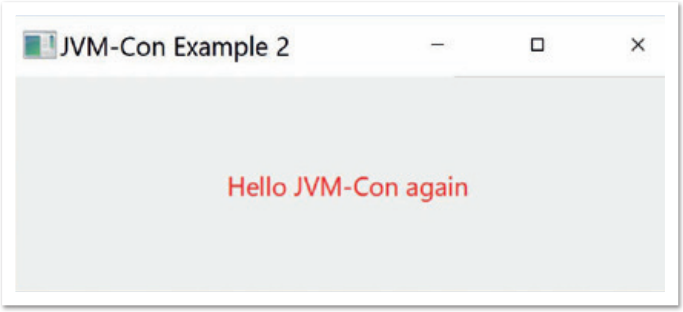


Abbildung 1: Beispiel einer Anwendungsoberfläche mit einer FXML-Datei und Controller

Im Enterprise-Bereich ist Java eine weitverbreitete Programmiersprache im Backend-Bereich. Es ist daher naheliegend, Java mit modernen Konzepten und Design-Pattern auch im Frontend einzusetzen, um unter anderem das Unternehmenspersonal in einer gewohnten Programmiersprache arbeiten zu lassen. Nicht zuletzt ist auch die plattformübergreifende Ausführung aller JavaFX-Komponenten ein fast unschlagbares Argument, wenn es um die zahlreichen Systeme und Systemkomponenten geht, die ein Fahrer im Fahrzeug bedienen muss.

JavaFX

JavaFX ist ein Framework zur Erstellung von Benutzeroberflächen in Java. Es war die Oracle-Antwort auf die im Jahr 2000 aufkommenden neuen UI-Technologien wie Silverlight von Microsoft und Flex von Adobe. Wie die Mehrzahl der aktuellen UI-Frameworks erlaubt es, Benutzeroberflächen aus einer Markup-Repräsentation aufzubauen und nur die dahinterliegende Funktionalität als Code in einer Programmiersprache zu entwickeln. Diese XML-Beschreibung ist grundsätzlich vergleichbar mit XML-Views, wie man sie aus Android kennt, oder mit Microsoft XAML. In einem einfachen Beispiel soll der grundsätzliche Aufbau einer JavaFX-Anwendung aus Application-Starter, FXML-Beschreibung und View Controller gezeigt werden *(siehe Listings 1 bis 3* und *Abbildung 1)*.

In der JavaFX-Welt entspricht ein Fenster einer Stage. Die initiale Stage bekommt der Application-Starter in seine „start"-Methode übergeben. Auf dieser Stage kann die Anwendung einen Szenengraph anlegen. Die Szenen in einem Fenster lassen sich über die „setScene"-Methode austauschen oder initial setzen. Einer Szene werden dann die UI-Elemente, die auch Nodes genannt werden, hinzugefügt. Da eine Szene nur ein Root-Element haben kann, muss es sich dabei um einen Container handeln, der die Unterelemente in einem Layout anordnet.

In unserem Beispiel laden wir die UI-Elemente – bestehend aus dem Root-Element, der „StackPane", und dem Label – aus unserer

```
public class HelloFXMLApplication extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        // UI-Graphen aus FXML-Datei laden
        Parent root = FXMLLoader.load(getClass().getResource("hello_javafx.fxml"));

        Scene scene = new Scene(root, 200, 200);

        primaryStage.setTitle("JavaFX Example 2");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 2

```
public class HelloFXMLController {
    @FXML
    private Label myLabel;

    public void initialize() {
        // Die Methode wird vom FXMLLoader aufgerufen, wenn die Elemente bereit stehen (s. fx:controller)
        myLabel.setTextFill(Color.RED);
    }
}
```

Listing 3

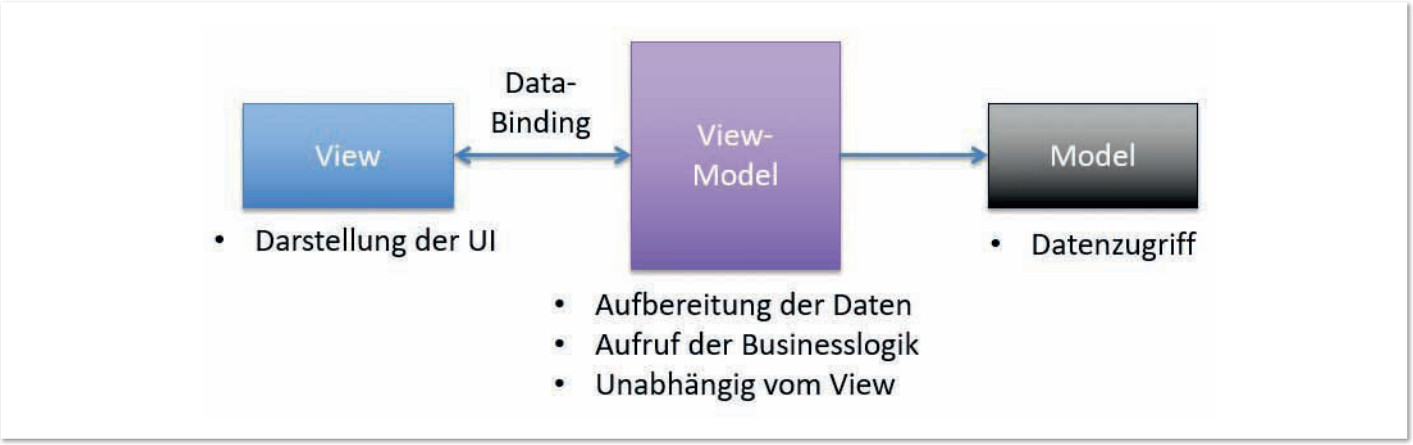


Abbildung 2: FXML-Dateien für eine UI-Beschreibung

FXML-Datei. Die Szene erstellen wir jedoch im Application-Starter und fügen ihr als Root-Element die geladene „StackPane" hinzu. Schlussendlich wird die Szene der übergebenen „primaryStage" übergeben.

Über das Attribut „fx:controller" wurde der StackPane in der FXML-Datei ein Controller zugewiesen. Controller kontrollieren einen Teil der UI und steuern die Interaktion damit. Wird einem Node über das „fx:id"-Attribut ein Identifier zugewiesen, lässt sich eine Referenz darauf über die „@FXML"-Annotation in der Controllerklasse herstellen. Das kennt man ähnlich aus der XAML-Welt.

MVVM

Bei MVVM handelt es sich um ein Entwurfsmuster. Die Buchstaben dieser Abkürzung stehen für „Model", „View" und „ViewModel"; es handelt sich dabei strenggenommen um eine Variante des MVC-Patterns *(siehe Abbildung 2)*.

Die Model-Schicht übernimmt bei MVVM die gleiche Aufgabe, die sie auch bei MVC innehat. Sie repräsentiert die Daten und ermöglicht gegebenenfalls den Datenzugriff in eine Storage-Komponente. Die View ist für die Darstellung der UI verantwortlich. Das ViewModel hingegen ist von der View unabhängig und steht zwischen View und Model. Es kann so entwickelt sein, dass es für mehrere Views verwendet werden kann – etwa für eine Mobile- und eine Desktop-View.

Das ViewModel ist für die Aufbereitung der Daten für eine oder mehrere Views zuständig und hat auch die Aufgabe, die Geschäftslogik anzusprechen. Anders als bei früheren MVC-Frameworks erfolgt die Bindung zwischen der View und der ViewModel-Ebene (die in vielen Teilen der Controller-Schicht gleicht) über sogenanntes „Data-Binding". Dabei werden Daten aus der View (etwa der Text in einem Textfeld) durch Properties in den ViewModels repräsentiert, die sich automatisch ändern, wenn sich die Daten in der View ändern, und umgekehrt. Auf dieselbe Art und Weise können auch Commands an UI-Elemente – etwa Buttons – gebunden werden, um Aktionen auszulösen.

MVVM in JavaFX und mvvmFX

Wie schon erwähnt, ist vieles, was Teil von MVVM ist, schon out of the box in JavaFX enthalten. So verfügen alle UI-Elemente über Properties, an die sich gebunden werden kann. Das Code-Beispiel in

```
Label copyLabel = new Label();
TextField textField = new TextField("Hallo JavaFX");

copyLabel.textProperty().bind(textField.textProperty());

root.getChildren().addAll(copyLabel, textField);
```

Listing 4

Listing 4 illustriert dies anhand eines Labels, das immer den Text, der in einem Texteingabefeld steht, anzeigt.

Hier werden per Code zwei neue Nodes erstellt: ein Label mit dem Namen „copyLabel", das den Text des Text-Eingabefelds („TextField") mit dem Namen „textField" darstellen soll. Durch den Aufruf der Methode „bind(...)" auf der „textProperty" des Labels und die Übergabe der „textProperty" des Textfelds werden beide Properties aneinandergebunden. Das bedeutet, dass sich die eine Property automatisch ändert, wenn sich die andere ändert. Dieses Binding ist unidirektional. Das heißt in diesem Fall, dass sich nur die „textProperty" des Labels ändert, wenn im Textfeld Text eingeben wurde, nicht aber umgekehrt. Sollen auch Änderungen von der „textProperty" an das Textfeld weitergereicht werden, ist die Methode „bindBidirectional(...)" erforderlich.

MvvmFX kann über Maven einfach in existierende oder neue Projekte eingebunden werden und hat fünf entscheidende Vorteile im Vergleich zu reiner JavaFX-Entwicklung: Es bietet eine implizite Verbindung von FXML-Markup-Dateien zur UI-Beschreibung und zugehörigen View-Klassen sowie eine explizit anzugebende Verbindung zwischen View- und ViewModel-Klassen. Hinzu kommt der Einsatz von Dependency Injection zur Entkopplung der Anwendungs-Abhängigkeiten. Außerdem ermöglicht es die Kapselung von Event-Handling in Commands und Scopes. Am einfachsten lassen sich diese Prinzipien anhand eines praktischen Beispiels erklären. Es soll ein Fenster zur Bearbeitung von Personendaten dargestellt werden, das ein Textfeld für Vor- und Nachname und einen Button zum Speichern enthält *(siehe Abbildung 3)*.

In der FXML-Datei gibt es im Vergleich zu dem, was wir bisher gesehen haben, keine Änderungen. Allein die explizite Angabe einer Action-Methode zum Click-Handling des Buttons über das „onAction"-Attribut wurde bisher nicht eingeführt, ist allerdings Bestandteil von JavaFX und nicht von mvvmFX *(siehe Listing 5)*.

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

<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.control.Button?>

<VBox xmlns="http://javafx.com/javafx"
      xmlns:fx="http://javafx.com/fxml"
      fx:controller="example5.PersonView"
      prefHeight="200.0" prefWidth="200.0">

    <TextField fx:id="firstNameField" />
    <TextField fx:id="lastNameField" />

    <Button fx:id="saveButton" text="Speichern"
onAction="#onSave"/>

</VBox>
```

Listing 5

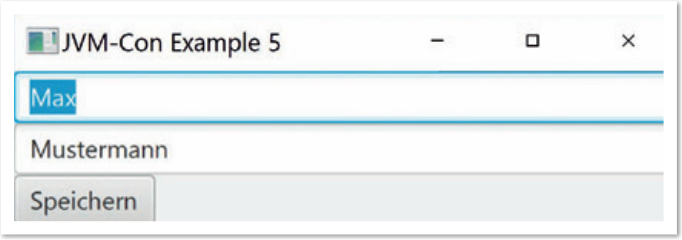


Abbildung 3: mvvmFX am Beispiel

Anders sieht es im Application-Starter aus, der hier für das Laden unserer View verantwortlich sein soll. Der FluentViewLoader aus mvvmFX lädt ein ViewTuple. Es besteht aus einer View – also dem Baum der UI-Elemente, hier mit einer VBox als Root-Element –, dem CodeBehind – hier einer Instanz der PersonView-Klasse – und einer Instanz der zur View gehörigen ViewModel-Klasse. Die Referenz auf die View wird dann, wie gewohnt, an die Szene weitergegeben und das Fenster wird aufgebaut. Man beachte, dass die FXML-Datei nirgends über den Dateinamen referenziert wird. Es gibt eine implizite Verbindung zwischen der im „res/"-Verzeichnis oder im selben Verzeichnis liegenden FXML-Datei und der View-Klasse *(siehe Listing 6)*. Klassenname und Dateiname der zugehörigen FXML-Datei müssen hier gleich sein.

Ein Blick auf die Controller-Klasse, die hier als View-Klasse fungiert, zeigt weitere Unterschiede. Am auffälligsten ist die Implementierung der Interfaces „FxmlView<T>" und „Initializable"; „T" ist dabei ein ViewModel. Auf diese Art und Weise wird eine View explizit an ein ViewModel gekoppelt. Umgekehrt kann ein ViewModel aber an mehrere Views gekoppelt sein. Das ViewModel wird dann über die Annotation „@InjectViewModel" in die View injiziert.

Das ViewModel stellt die Properties und Commands bereit, die die View benötigt. Namentlich sind das „StringProperties" für Vor- und Nachname („firstnameProperty" und „lastNameProperty"). Diese sind in diesem Fall bidirektional gebunden, um eventuelle Änderungen im Datenmodell von anderer Stelle direkt in der View darzustellen. Die „onSave"-Methode wurde aus der FXML-Datei referenziert und ruft selbst wiederum nur das vom ViewModel bereitgestellte Save-Command auf *(siehe Listing 7)*.

Das ViewModel enthält einen Verweis auf das Model, Properties zum Binden an die View und Methoden, um die Properties mit dem Model synchron zu halten. Zudem bietet es ein Command an, mit dem das Speichern des Fachobjekts ausgelöst wird *(siehe Listing 8)*. Da sich die Implementierung einer Benutzer-Schnittstelle nicht allein auf Detailfragen des eingesetzten Frameworks beschränken lässt, wird nachfolgend auf in diesem Projekt eingesetzte, neuartige Eingabemöglichkeiten eingegangen.

Natural User Interfaces

Die Interaktion über Natural User Interfaces (Touch, Leap Motion und Thalmic Myo) soll zu einer optimierten und angenehmeren Bedienung führen. Die Benutzer sollen situativ durch die Arbeitsaufgabe an den Leitständen geführt werden. Das Selektieren von Objekten auf der Bedienoberfläche wird beispielsweise in 83 Prozent der Durchführungen durch die direkte Touch-Interaktion, gegenüber der Durchführung mit der Maus, verkürzt *[1, Seiten 1 bis 4]*.

Die Touch-Bedienung an einem Natural User Interface ist spätestens durch die Einführung der Smartphones bekannt und unterliegt daher auch bestimmten Vorstellungen zur Interaktion *(siehe Abbildung 4)*. Ein Natural User Interface setzt voraus, dass es möglichst leicht ist, daran zu erinnern, wie bestimmte Funktionen bedient werden *[2, Seite 6]*.

Leap Motion ist ein kamerabasiertes Analysetool, das die Position der Hände und Finger des Nutzers im Raum analysiert *(siehe Abbildung 5, unten Mitte)*. Neben einer Handbewegung oder dem

```
public class PersonApplication extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        ViewTuple<PersonView, PersonViewModel> viewTuple = FluentViewLoader.fxmlView(PersonView.class).load();

        Scene scene = new Scene(viewTuple.getView(), 200, 200);

        primaryStage.setTitle("JavaFX Example 5");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 6

```
public class PersonView implements FxmlView<PersonViewModel>, Initializable {

    @FXML
    private TextField firstNameField;

    @FXML
    private TextField lastNameField;

    @FXML
    private Button saveButton;

    @InjectViewModel
    private PersonViewModel personViewModel;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        firstNameField.textProperty().bindBidirectional(personViewModel.firstNameProperty());
        lastNameField.textProperty().bindBidirectional(personViewModel.lastNameProperty());
    }

    @FXML
    public void onSave() {
        personViewModel.getSaveCommand().execute();
    }
}
```

Listing 7

```
public class PersonViewModel implements ViewModel {
    private SimpleStringProperty firstNameProperty;
    private SimpleStringProperty lastNameProperty;

    private Person person;

    public PersonViewModel() {
        setPerson(new Person("Max", "Mustermann"));
    }

    public SimpleStringProperty firstNameProperty() {
        return firstNameProperty;
    }


    public SimpleStringProperty lastNameProperty() {
        return lastNameProperty;
    }



    public Command getSaveCommand() {
        return new DelegateCommand(() -> new Action() {
            @Override
            protected void action() throws Exception {
                person.save();
            }
        });
    }

    // ...

    private void setPerson(Person person) {
        this.person = person;
        firstNameProperty = new SimpleStringProperty(person.getFirstName());
        lastNameProperty = new SimpleStringProperty(person.getLastName());
        setupListeners();
    }

    private void setupListeners() {
        firstNameProperty.addListener((observable, oldValue, newValue) -> updatePerson());
        lastNameProperty.addListener((observable, oldValue, newValue) -> updatePerson());
    }

    private void updatePerson() {
        person.setFirstName(firstNameProperty.get());
        person.setLastName(lastNameProperty.get());
    }
}
```

Listing 8

Abbildung 4: Varianten der Gesten-Interaktion über Touch, berührungslose Gestenbedienung mit Leap Motion und Muskelkontraktions-Analyse mit Thalmic Myo

Zusammenbewegen zweier Finger (Pinch-Gesture) bietet die Leap Motion auch die Möglichkeit, die Position der Hand vor, zwischen oder hinter dem Gerät auszuwerten, sodass ein Klicken ohne explizierte Geste möglich ist *(siehe Abbildung 5)*.

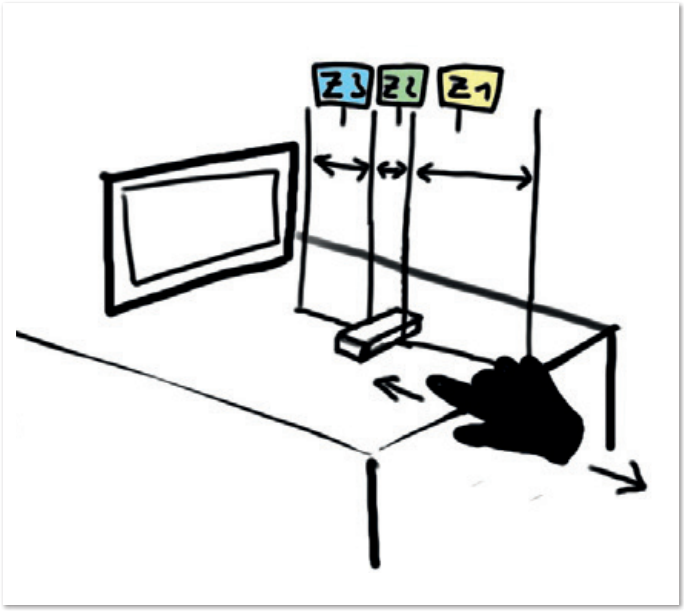


Abbildung 5: Leap Motion – Analyse der Hand (inklusive aller Finger) und die Positionen im Raum über der Leap Motion

Im Codebeispiel in *Listing 9* werden die Position der Handfläche („Palm") über der Leap Motion ausgelesen und unter anderem abgefragt, ob der Zeigefinger als einziger ausgestreckter Finger als Geste auftritt.

Das Myo-Armband hingegen wird am oberen Unterarm angelegt und analysiert die Muskelkontraktion, um eine Handbewegung zu erkennen *(siehe Abbildung 6*, links am Unterarm*)*. Entgegengesetzt zu der auf Basis von Live-Bildanalyse arbeitenden Leap Motion hat das Myo-Armband kein begrenztes Analysefeld (Kamera-Sichtfeld). Das Myo-Armband verwendet dabei das Gyroskop, das die Lagebestimmung durch Rotationskräfte bestimmt und dem „Mouse-Cursor" seine Position gibt.

Mit Paper-Prototypings zur intuitiven Leitstand-Software

Paper-Prototypings haben bei der Konzeptionierung geholfen, die Bedienfolgen zu bestimmen, bevor die MVVM-Komponenten entwickelt werden. Vor einer Auftragszuordnung zu einer Tour beispielsweise können in einem Benutzerdialog geeignete Tourenvorschläge *(siehe Abbildung 6*, Nr. 2 und 3*)* durch speziell entwickelte Services abgerufen und damit zusätzliche Informationen geboten werden. Gleichzeitig sind situativ nur die Informationen angezeigt, die an jener Stelle im Workflow notwendig sind. Bei einigen Interaktionsformen beansprucht die reine Klick-Abfolge mehr Zeit als bei anderen. Die Klicks in Benutzerdialogen sind deshalb auf ein Minimum redu-

Abbildung 6: Workflow im Benutzerdialog „Auftragszuordnungen" (Paperprototyping + Prototyp), 1 – Auftragsobjekt auswählen, 2 – Funktionswahl Tour-Zuordnung, 3 – Tourenauswahl betrachten, 4 – Position in Tour wählen

```
private void detectState(Frame frame) {
    FingerList fingerlist = frame.fingers();
    FingerList indexFingers = fingerlist.fingerType(Finger.Type.TYPE_INDEX);
    Finger indexFinger = indexFingers.get(0);

    Hand hand = frame.hands().frontmost();
    Vector palmPosition = hand.palmPosition();

    POSITIONSTATE positionState;
    int hoverStateArea = 20;
    if (palmPosition.getZ() >= hoverStateArea)
        positionState = POSITIONSTATE.NORMAL;
    else if (palmPosition.getZ() < -hoverStateArea)
        positionState = POSITIONSTATE.ACTIV;
    else
        positionState = POSITIONSTATE.HOVER;

    boolean indexFingerExtended = indexFinger.isExtended();
    int extendedCount = hand.fingers().extended().count();
    boolean handState = frame.hands().count() >= 1;

    Platform.runLater(() -> LeapMotionController.getInstance().updateLeapPointState(positionState,
indexFingerExtended, extendedCount, handState));
}
```

Listing 9

```
root.setOnMouseClicked(event -> {
    root.toFront();
});
```

Listing 10

```
private void fireEvent(EventType<? extends MouseEvent> eventType, Node node, float xValue, float yValue) {
    if(node == null) return;
    MouseEvent m = new MouseEvent(node, node, eventType, xValue, yValue, xValue, yValue,
            MouseButton.PRIMARY, 1, false, false, false, false, true, false, false, false, false, false,
            null);
    Event.fireEvent(node, m);
}
```

Listing 11

```
BeuthGestureEvent event = new BeuthGestureEvent(BeuthGestureEvent.DRAG_A_NODE, xValue, yValue);

notifyObservers(event);
setChanged();
```

Listing 12

ziert. Während der Gestendurchführung wird mit Rückmeldungen an die nutzenden Prozesse die Gestenanalyse visualisiert, was zu einer einfacheren Wahrnehmung führen soll.

JavaFX mit Gesten-Events

Das klassische Event-Handling von JavaFX für Mouse und Touch lässt sich mit Event-Handlern auf bestimmten Komponenten umsetzen *(siehe Listing 10)*. Beim Einsatz von individuellen Gesten-Erkennungstools und -geräten ist es sinnvoll, ein Event-Handling auf Basis simulierter Mouse-Events einzusetzen. Hier kann beispielsweise eine „Grab"-Geste zu einem klassischen Mouse-Event konvertiert werden, das wiederum den klassischen Event-Handler aktiviert. Alle Nodes lassen sich über ihre Position erfassen und mit der Handposition oder dem Mouse-Cursor vergleichen. Per „Event.fireEvent(node, event)" lässt sich das gewünschte Event bei diesem Node simulieren *(siehe Listing 11)*.

Falls dies nicht ausreicht und eine individuelle Geste mit bestimmten Parametern ein Ereignis auslösen soll, wird das Event-Handling mit einem Observer-Design-Pattern und eigenen Events eingesetzt. Ein entsprechendes Beispiel wären Bewegungsgesten, deren Parameter (Geschwindigkeit, Richtung) entscheidend sind und bei denen die Komponenten sich folglich auf Basis der erfassten Parameter animieren. Bei einer erkannten „Drag-Geste" können mehrere Komponenten reagieren sowie das „BeuthGestureEvent" auswerten und verarbeiten *(siehe Listing 12)*.

Fazit

Natural User Interfaces werden in Kombination mit workflowbasierten Benutzerdialogen die Bearbeitung von Aufgaben erleichtern und bieten damit mehr Raum für die inhaltliche Auseinandersetzung mit den Aufgaben. Moderne Design-Patterns wie MVVM bieten die Möglichkeit, hochwertige Komponenten für Unternehmensanwendungen zu entwickeln. Wie schon Swing und AWT ist auch JavaFX ein plattformübergreifendes UI-Framework. Seit Java 7 (JRE 7u6) ist es Teil jeder Java-SE-Installation.

Seit dem Jahr 2011 ist JavaFX über das OpenJFX-Projekt als quelloffene Software verfügbar. Im März 2018 gab Oracle im Rahmen seiner Java Client Road Map bekannt, dass JavaFX ab Java 11 nicht mehr Teil der Java Standard Edition sein wird. Oracle wird noch bis zum Jahr 2022 weiter Support anbieten und unterstützt die Weiterentwicklung im Rahmen des OpenJFX-Projekts.

Hinweis: Dieses Projekt wird im Netzwerk „Assistenz in der Logistik" unter dem Titel „Kopplung verteilter Prozesse an Gestengesteuerte Oberflächen (KoPGeO)" durch das Zentrale Innovationsprogramm Mittelstand (ZIM) gefördert.

Literatur

[1] Kin, Agrawala und DeRose (2009), Determining the benefits of direct-touch, bimanual, and multifinger input on a multitouch workstation, in A. Gooch & M. Tory (Eds.), Graphics Interface 2009, Proceedings, Kelowna, British Columbia, Canada, 25-27 May 2009 (pp. 119–124), Missisauga, Ont., Canadian Information Processing Society

[2] Norman (2010), The way I see it: Natural user interfaces are not natural, interactions, 17(3)

Mark Gebler
mgebler@beuth-hochschule.de

Mark Gebler M.Sc. hat Medieninformatik an der Beuth Hochschule für Technik Berlin studiert und entwickelte in seiner Masterarbeit eine gestengesteuerte Besucher-Informationsanwendung für das Deutsche Technikmuseum Berlin. Im Zentrum der Forschung und der Dissertation steht die Entwicklung von Koordinations- und Leitsystemen.

Hannes Walz
hwalz@beuth-hochschule.de

Hannes Walz B.Sc. hat Informatik an der Berufsakademie Berlin studiert und entwickelt seit fünf Jahren Java-Anwendungen für verschiedene Plattformen von Mobil bis Desktop. Aktuell entwickelt er auf Basis von JavaFX eine gestengesteuerte MVVM-Anwendung zur Disposition in der Abfallwirtschaft.

Prof. Dr. Gudrun Görlitz
goerlitz@beuth-hochschule.de

Prof. Dr. Gudrun Görlitz ist Professorin an der Beuth Hochschule für Technik Berlin im Fachbereich Informatik und Medien. Sie ist in der Lehre zur Programmierausbildung der Studierenden tätig, schwerpunktmäßig mit der Programmiersprache Java. Sie leitet zahlreiche anwendungsorientierte, drittmittelgeförderte F&E-Projekte, darunter das F&E-Projekt „KoPGeO – Kopplung verteilter Prozesse an gestengesteuerten Oberflächen".

Hilfe, ich muss JavaScript programmieren!

Nils Hartmann

Wer professionelle Web-Anwendungen bauen will, kommt an JavaScript nicht vorbei. Für viele Java-Entwickler ist die Vorstellung, mit JavaScript programmieren zu müssen, immer noch ein Graus. Zum einen bringt die Sprache selber einige Fallstricke und Kuriositäten mit; darüber hinaus erscheint auch das Ökosystem, das um die Sprache herum existiert, also Frameworks, Libraries und Tools, unüberschaubar groß und wenig nachhaltig. Der richtige Einstieg ist damit meist sehr schwierig. Dieser Artikel gibt eine Orientierung, indem die wichtigsten Werkzeuge vorgestellt und eingeordnet werden.

Das JavaScript-Ökosystem ist sehr groß und schnelllebig. Das macht es für Neulinge nicht einfach; selbst erfahrene JavaScript-Entwickler haben Mühe, darin die Orientierung zu behalten und mit der sehr schnellen Weiterentwicklung Schritt zu halten. Das hat sich in den vergangenen Jahren in dem Ausdruck „JavaScript Fatique", also „JavaScript-Erschöpfung", manifestiert.

Bevor wir uns auf den Weg durch das Ökosystem machen und ansehen, welche Tools wir wofür brauchen, zunächst die Frage, warum es sich überhaupt lohnt, sich mit JavaScript zu beschäftigen. Können wir nicht einfach unsere bekannten Verfahren wie Spring MVC oder JEE einsetzen und Web-Anwendungen bauen, die mithilfe einer Template-Sprache auf dem Server gerendert werden? Hier kennen wir immerhin die Sprache, die Frameworks sowie die Tools samt ihren Stärken und Schwächen.

Die Antwort ist mehrschichtig. Wir möchten dem Anwender den bestmöglichen Nutzungskomfort bieten, im Optimalfall sollte sich eine Web-Anwendung nicht von einer Desktopanwendung unterscheiden. Insbesondere soll die Anwendung sehr schnell auf Benutzer-Interaktionen reagieren, auf diversen Geräten mit unterschiedlichen Betriebssystemen gut aussehen und am besten auch dann noch funktionieren, wenn die Internet-Verbindung gerade nicht besteht. Moderne Web-Anwendungen wie zum Beispiel Spotify *(siehe „https://open.spotify.com")*, Outlook *(siehe „http://www.outlook.com")* oder das Prototype-Tool Figma *(siehe „https://figma.com")* erfüllen diese Ansprüche. Solche Anwendungen sind jedoch nur denkbar, wenn sie direkt auf dem Client, also im Browser, ausgeführt werden *(siehe Abbildung 1)*.

Die Sprache im Browser ist JavaScript. Wenn wir also Anwendungen bauen wollen, die im Browser laufen, müssen wir uns wohl oder übel damit beschäftigen. Im Übrigen wird der Browser zunehmend auch für Anwendungen interessant, die früher klassischerweise auf dem Desktop liefen. Die genannten Anwendungen sind ein Beispiel dafür, aber auch In-House-Anwendungen werden immer häufiger als Web-Anwendung implementiert. Auch hier ist es sinnvoll, sich mit JavaScript als der Sprache im Browser zu beschäftigen, da auch Java-basierte Ansätze wie JSF oder Vaadin nicht ohne JavaScript auskommen. Die direkte Verwendung von JavaScript kann sogar einfacher sein, als eine Abstraktion zu benutzen.

Das JavaScript-Ökosystem

Bei Java gibt es ein komplettes, offizielles Entwickler-Kit, das JDK. Es bringt eine ganze Reihe von Bibliotheken und Tools mit, die alle aufeinander abgestimmt sind. Dazu gehören der Compiler, die Laufzeitumgebung (JRE), sehr umfangreiche Standard-Bibliotheken und Tools wie „javadoc" und „apt", sogar eine SQL-Datenbank. Das alles existiert bei JavaScript zunächst nicht. JavaScript ist vorerst nur eine Sprache; alles andere – inklusive der Laufzeit-Umgebung, also in erster Linie der Browser – müssen wir uns selbst zusammenstellen oder sogar selbst entwickeln.

Noch vor wenigen Jahren reichte für die JavaScript-Entwicklung ein einfacher Editor aus. Man schrieb seinen Code in eine Datei, bettete diese in eine HTML-Seite ein und ließ sie vom Browser ausführen. Dieses Vorgehen eignete sich jedoch nur für eher kleinere Features, etwa das Validieren von Eingaben auf einer Webseite. Größere Anwendungen waren damit nicht oder nur sehr schwer umsetzbar. Aus dieser Situation heraus entwickelten sich unter anderem Libraries wie jQuery, die bestimmte Probleme adressierten und lösten.

Im Laufe der Zeit wurden die Anforderungen an Webseiten immer komplexer, aus ehemals statischen Seiten wurden dynamische Seiten mit immer mehr Benutzer-Interaktionen: Echte Anwendungen im Browser entstanden. Dafür reichte jQuery irgendwann nicht mehr aus; es entstanden neue Lösungen, die wiederum neue Innovationen hervorriefen, diese abermals neue Lösungen, und so weiter. Die entstandenen Lösungen wurden aber, von der Sprache JavaScript abgesehen, nicht – wie bei Java – von einem Konsortium (mit)entwickelt, spezifiziert und standardisiert, sondern entstanden und entstehen weiterhin durch die Community. So sind auch alle im Folgenden vorgestellten Tools und Frameworks Open-Source-Lösungen.

Die Sprache JavaScript

Ähnlich wie in Java gibt es mehrere Versionen der Sprach-Spezifikation, die „ECMAScript" heißt. Von den meisten Browsern *(siehe „https://caniuse.com/#search=es5")* ist die Version 5 implementiert; wenn man eine Anwendung also nach dieser Spezifikation implementiert, ist die Chance sehr hoch, dass sie in allen Browsern auch funktioniert. ECMAScript 5 wurde im Jahre 2009 veröffentlicht, danach gab es erst im Jahr 2015 die nächste Version, die aber immer noch nicht von allen Browsern vollständig implementiert ist.

Allerdings enthält Version „ECMAScript 2015" sehr viele, teilweise auch gravierende Änderungen und Neuerungen, die für die Entwicklung von Anwendungen sehr vorteilhaft sein können; so haben beispielsweise Klassen und Module Einzug in die Sprache gehalten und mit der Einführung von Block-Scoping sowie Arrow-Funktionen wurde auf einige der gravierendsten Unzulänglichkeiten in der Sprache reagiert.

Darüber hinaus wurde mit der Version auch der Release-Zyklus für die Sprache verändert: Seit dem Jahr 2015 gibt es nun jährliche Releases, die jeweils nach dem Erscheinungsjahr benannt und nicht mehr durchnummeriert sind, also „ECMAScript 2015", „ECMAScript 2016", „ECMAScript 2017" etc.

Wie erwähnt, betreffen diese Releases nur die Sprach-Spezifikation, nicht deren Umsetzungen in den Browsern. Zwar ist zu beobachten, dass die Browserhersteller bemüht sind, die neuesten Spezifikationen schnell umzusetzen, in der Praxis dauert das jedoch immer einige Zeit. Außerdem lässt sich eine Anwendung mit neuem JavaScript nur dann sicher betreiben, wenn wirklich alle für die Zielgruppe relevanten Browser die verwendete ECMAScript-Version unterstützen.

Um dieses Problem zu lösen, haben sich in der JavaScript-Entwicklung Compiler etabliert. Damit wird allerdings nicht wie in Java Source-Code zu Byte-Code übersetzt, sondern JavaScript-Code aus einer Version in eine ältere zurückkompiliert. So lassen sich neue Features zeitnah nutzen, indem diese in beispielsweise ECMAScript 5 zurückkompiliert werden, das die allermeisten Browser unterstützen.

Compiler und Polyfills

Der wohl am meisten eingesetzte Compiler ist Babel *(siehe „https://babeljs.io")*. Er kann nicht nur aktuellen JavaScript-Code in ältere Ver-

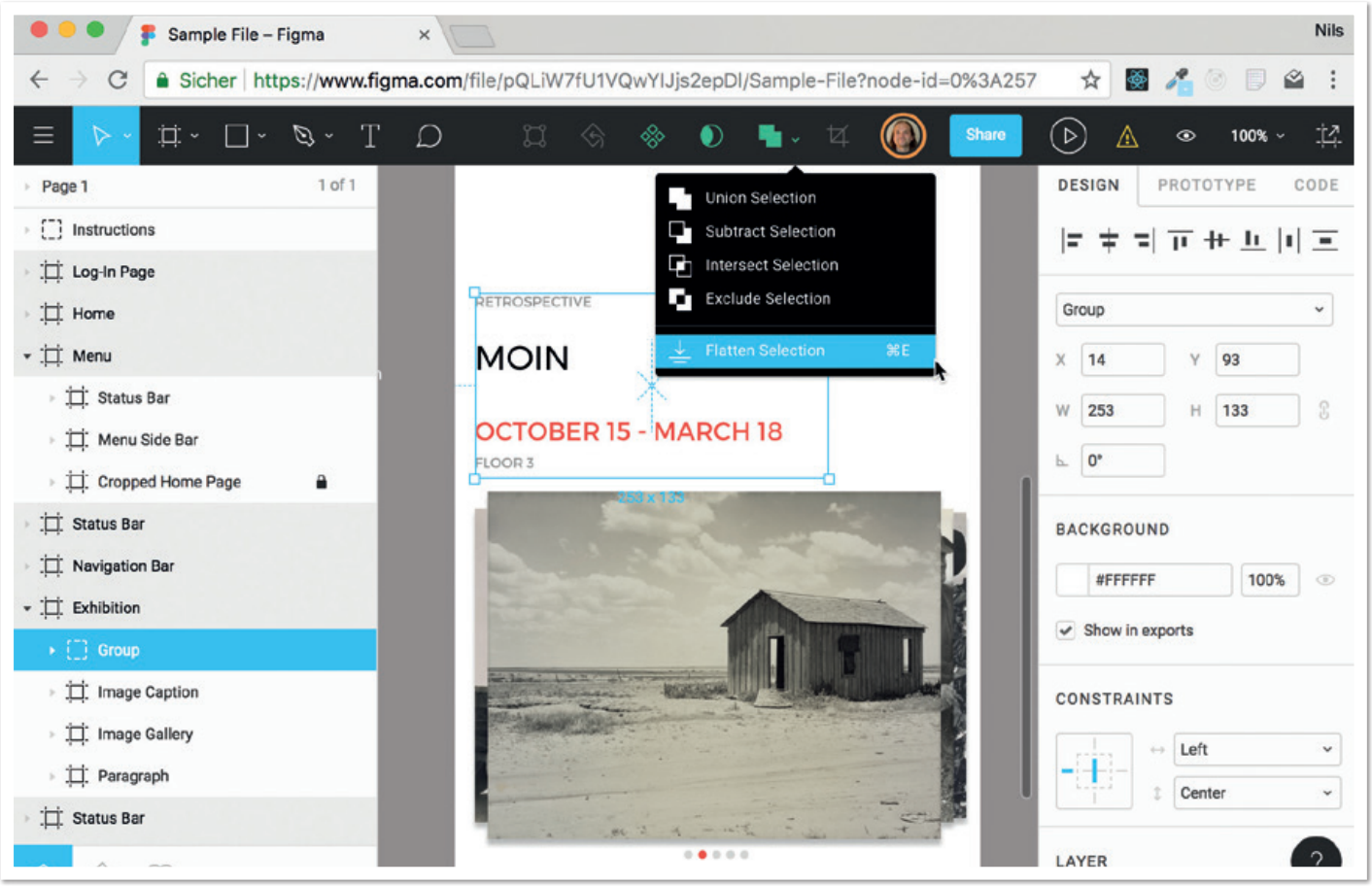


Abbildung 1: Figma, eine echte Anwendung im Browser

sionen zurückübersetzen, sondern dank eines modularen Aufbaus mit Plug-ins um weitere Features ergänzt und für den individuellen Bedarf konfiguriert werden. Oftmals stehen beispielsweise Plug-ins schon für zukünftige JavaScript-Features zur Verfügung, die noch gar nicht endgültig spezifiziert worden sind. Mittels Babel lassen sie sich jedoch bereits vorab in eigenen Projekten testen. Eine Alternative zu Babel ist der TypeScript-Compiler, der von Microsoft entwickelt wird. Er wird später noch näher betrachtet.

Ein Compiler übersetzt nur die Sprache selbst von einer Version zu einer anderen Version zurück. Daneben gibt es allerdings noch einige Standard-APIs, die ebenfalls zur Spezifikation gehören und von der Laufzeit-Umgebung zur Verfügung gestellt werden müssen. Dazu gehören beispielsweise die Funktionen, die auf „Object", „String" und „Number" definiert sind, die (neuen) Maps und Sets oder auch das DOM-API. Hier ergibt sich dasselbe Problem wie bei der Sprache: Fehlen die APIs in einem der gewünschten Ziel-Browser, können sie in der eigenen Anwendung nicht verwendet werden, da es ansonsten zur Laufzeitfehlern kommt.

Abhilfe an dieser Stelle schaffen Polyfills, also Bibliotheken, um eine oder mehrere der Standard-APIs zu implementieren und in die eigene Anwendung einzubinden. Die meisten Polyfills sind so intelligent, dass sie sich selbst nur dann aktivieren, wenn die von ihnen implementierten APIs zur Laufzeit wirklich nicht vorhanden sind. Wenn das API in einem Browser bereits verfügbar ist, wird dann von der Anwendung automatisch die nativ implementierte Variante verwendet. Polyfills für diverse APIs stellt beispielsweise „corejs" *(siehe „https://github.com/zloirock/core-js")* zur Verfügung. Dabei lässt sich auch sehr feingranular auswählen, für welche konkreten APIs ein Polyfill benötigt wird, um den Anwendungscode nicht unnötig aufzublähen. Auch Babel bietet eine Polyfill-Bibliothek *(siehe „https://babeljs.io/docs/usage/polyfill")*.

Laufzeit-Umgebungen

Bislang wurden als Laufzeit-Umgebung nur Browser betrachtet. Mittlerweile werden mit JavaScript auch Applikationen gebaut, die außerhalb eines Browsers laufen. Hier kommt dann in der Regel NodeJS zum Einsatz. Es verwendet die JavaScript-Implementierung V8 von Google, die auch die JavaScript-Engine von Chrome ist. Seit Anfang 2016 gibt es daneben eine NodeJS-Variante, die mit Chakra Core, der JavaScript-Engine von Microsoft Edge, arbeitet. Auf Basis von NodeJS werden einerseits Serverprozesse (zum Beispiel Webserver oder für ein REST-API) implementiert. Andererseits dient NodeJS auch als Ausführungsumgebung für eine ganze Reihe von Tools wie Compiler (die genannten Compiler Babel und TypeScript sind in JavaScript implementiert), Package-Manager oder Test-Tools. Das ist in gewisser Weise mit Java vergleichbar, wo die JRE auch nicht nur zum Ausführen von Server-Prozessen verwendet wird.

Modul-Systeme

Unabhängig von der späteren Laufzeit-Umgebung müssen gerade größere Anwendungen gut strukturiert sein, damit der Code verständlich und wartbar bleibt. Eine Möglichkeit dafür bieten Module. Ein Modul in JavaScript ist technisch betrachtet eine Datei und damit eine sehr viel kleinere Einheit als die meisten Java-Module, die in der Regel aus mehreren Klassen und Packages bestehen. In

JavaScript ist es nicht ungewöhnlich, dass Module nur aus einer oder zwei Funktionen bestehen.

Im Laufe der Zeit haben sich insgesamt drei Modul-Systeme etabliert: CommonJS, das auch Grundlage für das in NodeJS verwendete Modul-System und für den Einsatz außerhalb eines Browser konzipiert ist; außerdem das AMD-Modul-System, das für den Einsatz im Browser ausgelegt und unter anderen in der Lage ist, Module asynchron nachzuladen, um beispielsweise den Start einer Anwendung zu beschleunigen. Mit ECMAScript 2015 wurde nun auch ein natives Modul-System spezifiziert und in den Sprachstandard aufgenommen. Es wird zunehmend von den Browsern unterstützt und auch von NodeJS implementiert werden. Bis das Modul-System überall unterstützt ist, können Babel oder TypeScript verwendet werden, um Code, der das native System verwendet, nach CommonJS oder AMD zu übersetzen.

Mit dem nativen Modul-System lassen sich aus Modulen (also Dateien) einzelne Bestandteile exportieren und diese damit für andere Module sichtbar machen. Alle Bestandteile eines Moduls (Funktionen, Klassen, Konstanten etc.), die nicht explizit exportiert werden, bleiben für andere Module unsichtbar. Um etwas aus einem anderen Modul zu importieren, ist es explizit zu importieren. *Listing 1* zeigt ein Beispiel: Das „UserService"-Modul exportiert eine Klasse „UserService", die vom App-Modul importiert wird. Alle anderen Teile aus dem Modul (die Konstante „database" und die Funktion „initDatabase") sind für andere Module nicht sichtbar und können auch nicht importiert werden.

```
// UserService.js
export default class UserService {
  constructor() { ... }
  loadUser(id) { ... }
}
// Nur Modul-intern sichtbar
const database = ...;
function initDatabase() { ... }

// App.js
import UserService from "./UserService";
const userService = new UserService();
userService.loadUser(1);
```

Listing 1

Package-Manager

Module lassen sich (ähnlich wie mit Maven in Java) in einem zentralen Repository veröffentlichen und von dort auch installieren. Das dafür zuständige Tool ist der „Node Package Manager" (npm, siehe *„https://www.npmjs.com"),* der Bestandteil der NodeJS-Distribution ist, aber auch einzeln installiert werden kann. Das Repository ist die „npm"-Registry. Die dort veröffentlichten Artefakte heißen „Packages" und können aus mehreren Modulen und weiteren Source-Artefakten, etwa CSS-Dateien, bestehen. Packages, von denen das eigene Projekt abhängig ist, werden im Projekt in einer Beschreibungsdatei („package.json") hinterlegt, vergleichbar mit den Abhängigkeiten in der „pom.xml" von Maven.

In JavaScript-Projekten ist es üblich, über die Abhängigkeitsverwaltung die für die Entwicklung benötigten Tools zu installieren. So stehen beispielsweise Babel und TypeScript, aber auch andere Build-Tools, als „npm"-Pakete zur Verfügung. Auf diese Weise kann jedes Projekt individuell festlegen, welche Tools in genau welcher Version erforderlich sind. Alle Mitglieder des Entwicklungsteams erhalten dann automatisch die benötigten Tools in den korrekten Versionen. Außerdem entfällt weitgehend die Notwendigkeit, Tools überhaupt global zu installieren und dadurch in Versionskonflikte zu geraten. *Listing 2* zeigt einen Ausschnitt aus einer „package.json"-Datei, in der sowohl Abhängigkeiten definiert sind, die von der Anwendung selbst benötigt werden („dependencies"), als auch Abhängigkeiten, die nur für die Entwicklung notwendig sind („devDependencies").

Für (In-House-)Module, die nicht in einem öffentlichen Repository veröffentlicht werden sollen, kann man neben der offiziellen „npm"-Registry auch eigene Repositories betreiben. Sowohl Nexus als auch

```
{
  "name": "my-javascript-app",
  "version": "1.0.0",
  "devDependencies": {
    "node-sass": "^4.7.2",
    "prettier": "^1.11.1",
    "tslint": "5.8.0",
    "typescript": "^2.7.2"
  },
  "dependencies": {
    "react": "^16.3.1",
    "react-dom": "^16.3.1",
    "react-router": "^4.2.0",
    "react-router-dom": "^4.2.2"
  }
}
```

Listing 2

```
// Type Inference: age ist eine Nummer
const age = 32;

// Explizite Typ Angabe
const year:number = 2018;

// Funktionsparameter
function greet(phrase: string) {
  ...
}

// Objekte
interface Person {
  name: string,
  age: number
}

const klaus:Person = {
  name: "Klaus",
  age: 32
}
```

Listing 3: Typ-Angaben in TypeScript

Artifactory beispielsweise unterstützen neben diversen anderen Formaten auch das „npm"-Format.

Neben „npm" zum Installieren und Deployen von Packages gibt es mit dem von Facebook entwickelten „yarn" *(siehe „https://yarnpkg.com")* eine Alternative. Es nutzt dieselbe Abhängigkeitsbeschreibung in der „package.json" wie „npm" und kann außerdem auch das gleiche Registry verwenden. Vom Funktionsumfang her unterscheiden sich die beiden Tools mittlerweile nicht mehr wesentlich, allerdings sind sie hinsichtlich der Bedienung verschieden.

Der Vollständigkeit halber sei erwähnt, dass aus früheren Frontend-Projekten noch „bower" *(siehe „https://bower.io")* bekannt ist, mit dem sich ebenfalls Abhängigkeiten verwalten lassen. Dieses Tool wird zwar noch gepflegt, die Entwickler raten jedoch selber dazu, stattdessen auf „npm"/„yarn" und Webpack (siehe unten) zu setzen.

Module-Bundler

Wir haben nun einen Eindruck davon bekommen, wie sich mit JavaScript Module erstellen, in Form von Packages veröffentlichen und in ein eigenes Projekt einbinden lassen. Zum Ausführen einer Anwendung im Browser reicht das allerdings noch nicht aus, denn die Browser unterstützen beispielsweise das NodeJS-Modulsystem nicht und auch das native Modulsystem wird noch nicht von allen Browsern unterstützt.

Wenn eine Anwendung also selber das NodeJS-Modulsystem oder mittels „npm" ein externes Package verwendet, das mit dem NodeJS-Modulsystem gebaut ist, brauchen wir dafür eine Lösung. An dieser Stelle kommt ein Module-Bundler ins Spiel, ein Tool, das die Abhängigkeiten einer Anwendung zur Laufzeit analysiert und aus allen referenzierten Modulen eine vom Browser verwendbare JavaScript-Datei erstellt. Der Bundler ist dabei so schlau, dass er die Sichtbarkeiten und Scopings der Original-Module berücksichtigt, sodass es in der erzeugten Ausgabe-Datei nicht zu Namenskollisionen kommen kann.

Der wohl prominenteste Module-Bunder ist Webpack *(siehe „https://webpack.js.org")*. Er kann mit allen drei Modul-Systemen umgehen und bietet diverse Optimierungsmöglichkeiten für den erzeugten Code. Webpack selber ist modular aufgebaut und lässt sich für das eigene Projekt sehr gut anpassen. So kann Webpack beispielsweise vor dem Erzeugen des „Bundle", also der fertigen Ausgabedatei, die eingelesenen Module zuvor noch mit Babel oder TypeScript kompilieren. Dabei ist Webpack nicht auf JavaScript beschränkt, sondern kann zum Beispiel auch mit CSS-Dateien umgehen. Wenn diese im Code referenziert werden, findet Webpack die Referenzen und kann die Dateien je nach Konfiguration behandeln, zum Beispiel den CSS-Code komprimieren.

Auf Wunsch erzeugt Webpack auch immer SourceMaps. Dabei handelt es sich um Informationen für den Browser, die den kompilierten Code zurück auf den Source-Code mappen, sodass der Browser im Debugger den originalen Code anzeigen kann, den wir selber programmiert haben. Auch dieses Verhalten ist ähnlich wie in Java, wo im Debugger zum Beispiel in der IDE ja ebenfalls der Source-Code und nicht der erzeugte Byte-Code angezeigt wird. Andere bekannte Module-Bundler neben Webpack sind Browserify *(siehe „http://browserify.org")* und Rollup *(siehe „https://rollupjs.org")*.

Testen und Qualitätssicherung

Mit den bis hierher gezeigten Werkzeugen lassen sich Anwendungen bauen, die entweder im Browser oder mit NodeJS auch außerhalb des Browsers laufen können. Genau wie in Java greift man dabei auf externe Bibliotheken („npm"-Packages) zu und legt auch eigene Packages in einer Registry ab.

Eine erste Möglichkeit, die Anwendung mit Tests und anderen Maßnahmen gegen Fehler abzusichern, besteht darin, einen Type-Checker zu verwenden, der dabei helfen kann, eine ganze Reihe typischer JavaScript-Fehler zu vermeiden, so wie wir das auch aus Java gewohnt sind (eine Zahl keinem String zuweisen, falsche Anzahl und/oder Typen von Parametern einer Funktion übergeben, unsichere Zugriffe auf potentielle Null-Werte etc.). Prominente Type-Checker sind Flow *(siehe „https://flow.org")* von Facebook oder TypeScript *(siehe „http://typescriptlang.org")* von Microsoft. Beiden gemeinsam ist, dass die Angabe von Typen optional ist, man muss also Typ-Angaben nur dort hinschreiben, wo man sie auch wirklich benötigt. An vielen Stellen leiten die Type-Checker die korrekten Typen automatisch ab.

Während Flow ausschließlich ein Type-Checker ist, handelt es sich bei TypeScript um eine eigene Sprache, die allerdings auf JavaScript aufbaut. So ist jeder gültige JavaScript-Code zunächst auch gültiger TypeScript-Code. Mit TypeScript lässt sich der Code um Typ-Angaben ergänzen, zudem fügt TypeScript der Sprache auch noch einige neue Features hinzu, wie Sichtbarkeiten („protected" und „private" in Klassen) oder Aufzählungstypen („enum").

Insgesamt ist TypeScript sehr weit verbreitet – unter anderem wurden Angular und auch die neue Outlook-Webanwendung damit entwickelt – und es gibt mittlerweile Typ-Beschreibungen für fast alle bestehenden JavaScript-Bibliotheken. Auch der DIE-Support ist insbesondere mit der IntelliJ-Produktfamilie und Visual Studio Code sehr gut. Aus diesem Grunde setzt der Autor in Projekten eher TypeScript als Flow ein. *Listing 3* zeigt beispielhaft, wie Typ-Informationen in TypeScript geschrieben werden (die Syntax sieht in Flow übrigens sehr ähnlich aus), und in *Abbildung 2* ist exemplarisch zu sehen, wie in Visual Studio Code von TypeScript gefundene Fehler angezeigt werden.

Alternativ oder zusätzlich zu einem Type-Checker lassen sich sogenannte „Linter" zur Sicherstellung der Code-Qualität einsetzen. Dabei handelt es sich um Tools, die mit statischer Code-Analyse typische Probleme aufdecken. Außerdem lassen sich darüber selbst definierte Code-Konventionen überprüfen. Der am häufigsten eingesetzte Linter ist „ESLint" *(siehe „https://eslint.org")* beziehungsweise „TSLint" *(siehe „https://palantir.github.io/tslint")* für TypeScript-Code. Beide Linter lassen sich auch im Rahmen des Webpack-Builds ausführen, sodass Webpack abbricht, wenn eines der Tools ein Problem meldet *(siehe Abbildung 3)*. Zum Einhalten identischer Code-Konventionen eignet sich übrigens das Formatierungstool „Prettier" *(siehe „https://prettier.io")* hervorragend.

Type-Checker und Linter ersetzen allerdings keine Tests für die eigene Anwendung und genau wie in Java gibt es auch für JavaScript-Anwendungen diverse Test-Tools und -Frameworks für Tests auf allen möglichen Ebenen. Leider existiert aus Sicht des Autors zurzeit kein Standard, was die Auswahl der Test-Frameworks angeht. Im Wesentlichen gibt es drei Optionen: Mocha *(siehe „https://mochajs.org")*, Jasmine *(siehe „https://jasmine.github.io")* und Jest *(siehe „https://facebook.github.io/jest")*.

Mocha ist nur ein Test-Runner, der es erlaubt, einzelne Tests und Testsuites zu definieren und auszuführen. Für Dinge wie Assertions oder Mocks sind allerdings zusätzliche, nach eigenen Vorlieben frei auswählbare Module zu installieren. Jasmine und Jest hingegen bringen diese Dinge von Haus aus schon mit („Batteries included"), Jest bietet sogar Code Coverage und eine „headless"-Implementie-

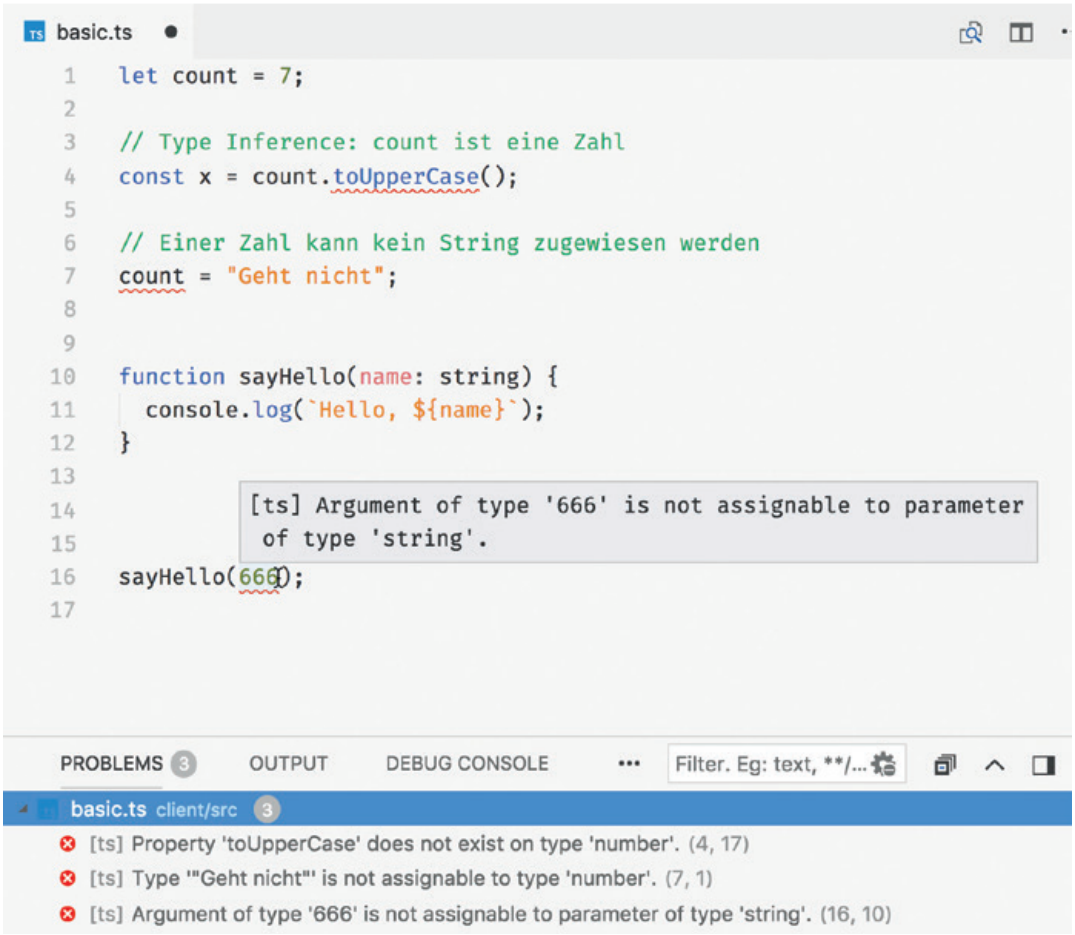


Abbildung 2: TypeScript in der IDE

Abbildung 3: ESLint findet mögliche Probleme im Source-Code

rung eines DOM, sodass man dort auch Code testen kann, der zur Laufzeit das DOM-API benötigt. Mit dem „Snapshot Testing" bringt Jest außerdem eine interessante Möglichkeit mit, JSON-basierte APIs (und auch React-Komponenten) zu testen. *Listing 4* zeigt einen einfachen Unit-Test mit Jest.

Automatisierung

Mit dem bis hier gezeigten Technologie-Stack lassen sich echte Anwendungen entwickeln, testen und betreiben. Gerade zur Entwicklungszeit sind allerdings einige Dinge immer wieder durchzuführen, insbesondere muss der Compiler und/oder Bundler ausgeführt werden, die Tests müssen laufen und eventuell auch der Linter. Das möchten wir natürlich nicht immer und immer wieder von Hand machen, sondern Tools zur Automatisierung dieser Vorgänge nutzen. Zum einen gibt es mit Gulp *(siehe „https://gulpjs.com")* und Grunt *(siehe „https://gruntjs.com")* zwei Tools, die wiederkehrende Tasks ausführen können. Dazu werden Tasks definiert, die bestimmte Aufgaben

```
// sum.js
export function sum(a,b) {
  return a+b
}

// sum.test.js
import {sum} from '../sum.js';

test('sum of 2 and 2 is 4', function() {
  expect(sum(2, 2)).toBe(4);
});

test('sum of 2 and 2 is not 3', function() {
  expect(sum(2, 2)).not.toBe(3);
});
```

Listing 4: Unit-Tests mit Jest

übernehmen (zum Beispiel das Ausführen des Compilers) sowie deren Auslöser (Änderung im Source-Verzeichnis) und Abhängigkeiten untereinander (nach dem Kompilieren den Linter ausführen). Damit lassen sich ähnlich wie mit Maven oder Gradle auch komplexe Build-Prozesse umsetzen.

Eine andere, leichtgewichtigere, meist aber ausreichende Alternative sind die „npm scripts". In der „package.json"-Datei eines Projekts lassen sich neben den Abhängigkeiten auch Skripte hinterlegen, die dann mit „npm" über die Betriebssystem-spezifische Kommandozeile (wie Bash) ausgeführt werden können. Das Besondere daran ist, dass dabei sämtliche lokal von „npm" installierten Tools (wie TypeScript oder Webpack) automatisch im Path vorhanden sind und somit einfach aufgerufen werden können. In der Praxis spielt das Problem, dass die Skripte Betriebssystem-abhängig sind, selten eine Rolle und für häufige Probleme, wie das Setzen von Umgebungsvariablen, gibt es bereits auch fertige Lösungen. Bevor man in seinem JavaScript-Projekt mit Gulp oder Grunt arbeitet, sollte man auf jeden Fall prüfen, ob „npm scripts" ausreichend sind.

Zusammenfassung

Damit sind wir nun am Ende unseres Wegs angekommen. Wir haben gesehen, dass es einen funktionierenden Technologie-Stack für die JavaScript-Entwicklung gibt, mit dem sich auch ernsthafte Anwendungen professionell entwickeln lassen:

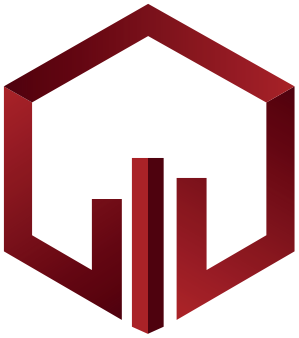
- Die Sprache JavaScript wurde im Jahr 2015 um viele wichtige Features, insbesondere ein Modul-System, erweitert.
- Module beziehungsweise Packages können mit „npm" veröffentlicht und eingebunden werden.
- Um Anwendungen, die mit neueren Sprachversionen geschrieben sind, in älteren Browsern ausführbar zu machen, verwenden wir einen Compiler (Babel oder TypeScript) und Polyfills.
- Für unterschiedliche Modul-Systeme gibt es Bundles (Webpack), die aus einer modularisierten Code-Basis eine für den Browser ausführbare JavaScript-Datei erstellen.
- Zur Erfüllung von Qualitäts-Anforderungen, Wartbarkeit und Tests gibt es Type-Checker (Flow und TypeScript), statische Code-Analyse-Tools (Linter) und natürlich Test-Frameworks (Mocha, Jasmine, Jest).
- Um die während der Entwicklung anfallenden Aufgaben zu automatisieren, lassen sich Task-Runner einsetzen, deren einfachste Form die „npm scripts" sind.

Im Gegensatz zum JDK gibt es allerdings keine zentrale Instanz, die Tools und Frameworks für uns auswählt und als fertiges Development Kit bereitstellt. Stattdessen werden die meisten Lösungen auf Eigeninitiative entwickelt und bereitgestellt – und dann von der Community angenommen, verworfen oder weiterentwickelt. Das bedeutet, dass wir einerseits eine sehr hohe Innovationsgeschwindigkeit haben, die das Arbeiten mit JavaScript sehr spannend macht, weil wir immer mehr Dinge mit JavaScript umsetzen können. Auf der anderen Seite kann das Tempo jedoch auch nervenaufreibend sein, wenn man versuchen will, Schritt zu halten.

Aus Sicht des Autors sollte man nicht jedem Trend hinterherrennen. Es ist völlig legitim, beim Erscheinen neuer Frameworks, Tools und Bibliotheken erst einmal abzuwarten, wie diese sich entwickeln, und vor allem zu verstehen, welche konkreten Probleme sie lösen. Denn nur wenn das Problem verstanden wurde, kann geprüft werden, ob es für das eigene Projekt überhaupt relevant ist.

Nils Hartmann
nils@nilshartmann.net

Nils Hartmann ist Software-Entwickler aus Hamburg. Er programmiert sowohl in Java als auch in JavaScript/TypeScript und beschäftigt sich zurzeit hauptsächlich mit der Entwicklung von React-Anwendungen. Nils Hartmann gibt seine Erfahrungen, Ideen und Fragen gern auf Konferenzen, Trainings und Workshops wieder.



INSPECTIT

InspectIT – die Open-Source-Application-Performance-Management-Lösung

Alicia Bondanza, NovaTec Consulting GmbH

Software-Systeme werden in der heutigen Zeit immer komplexer, während Endnutzer gleichzeitig höhere Anforderungen an die Reaktionszeit einer aktiven Internetseite stellen. Reagiert die Seite nicht schnell genug oder ist sie von vornherein nicht verfügbar, geschieht es in Sekundenschnelle, dass das Unternehmen einen potenziellen Kunden verliert und dieser sich einer anderen Seite bedient. InspectIT hilft dabei, bei Komplexität Transparenz zu schaffen und frühzeitig drohende Probleme zu identifizieren, sodass proaktiv Maßnahmen möglich sind.

Systemausfälle oder schlechte Performance beeinflussen bekanntermaßen den wirtschaftlichen Erfolg und das Ansehen eines Unternehmens negativ. Das Application Performance Management (APM) nimmt dieses Risiko vorweg, indem es Methoden und Tools zur Gewährleistung einer hohen Servicequalität bereitstellt und somit den reibungslosen Betrieb sicherstellt. APM-Tools bieten die Möglichkeit, den Zustand von Software-Systemen zu überwachen und auftretende Performance-Anomalien zu erkennen. Sie reagieren darauf und diagnostizieren die Ursachen von Leistungsproblemen.

InspectIT stellt eine ausgereifte Open-Source-Alternative für APM dar. Das Tool bietet alle Kernfunktionen, die zum Verwalten der Systemleistung erforderlich sind. Das Besondere an InspectIT ist, dass es die Überwachung und Analyse des Systemverhaltens zur Laufzeit ermöglicht und den Betreiber bei Problemen benachrichtigt. Es agiert sozusagen als ein ausgereiftes Mittel zur Problem-Diagnose und Prävention. Ein besonderes Merkmal ist die Fähigkeit, Ausführungsverfolgungen über mehrere Services hinweg zu korrelieren, was die Systemeinsicht erhöht und die Verfolgung des Datenflusses durch das gesamte System ermöglicht *(siehe Abbildung 1)*. Mit diesen Funktionen kann eine End-to-End-Ansicht des überwachten Systems angezeigt werden.

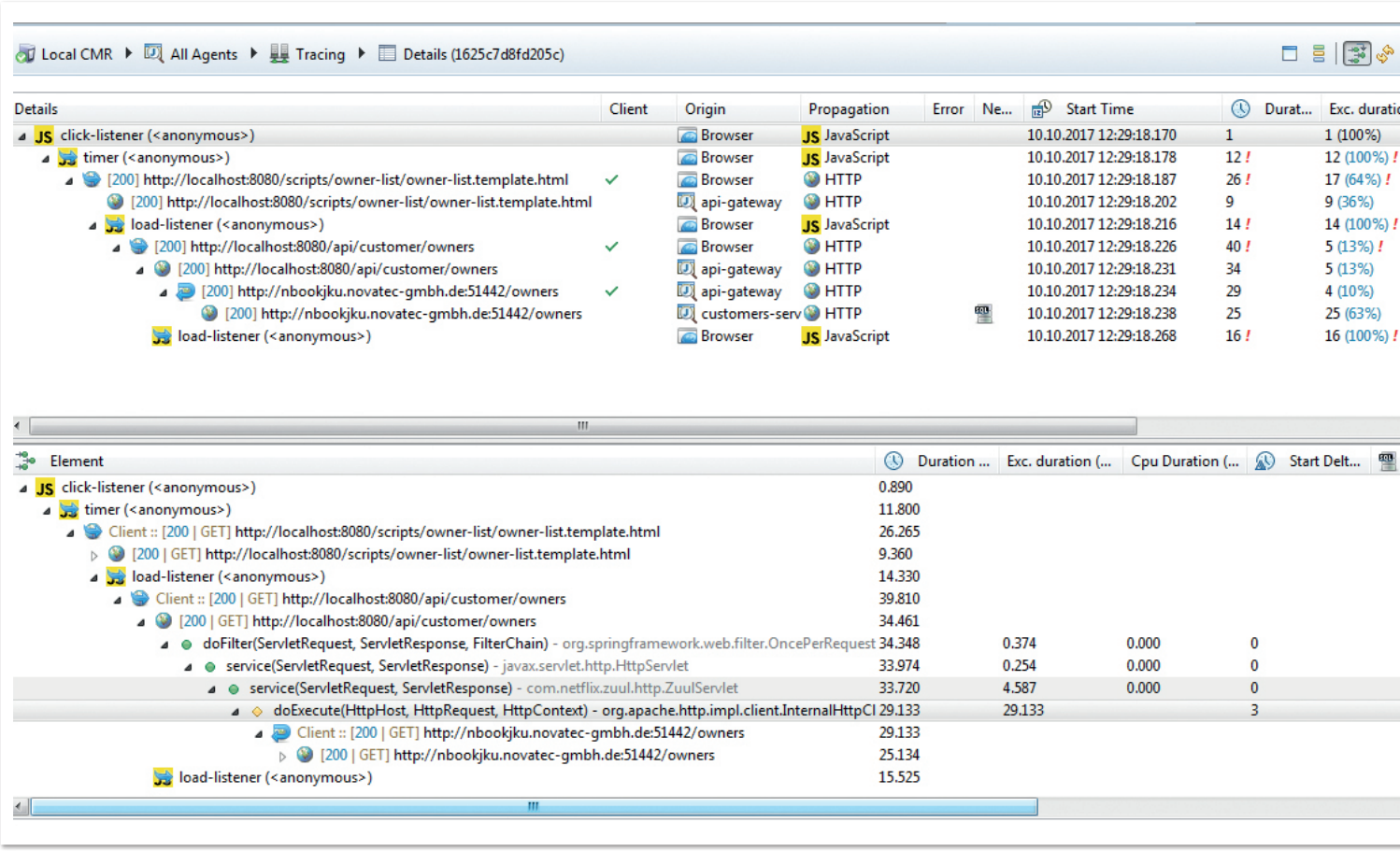


Abbildung 1: Ansicht eines Ausführungstrace über mehrere Services

Warum Performance wichtig ist

Die Studie „Power of 10: Time Scales in User Experience, 2009" von Jacob Nielsen zeigt, dass Endnutzer innerhalb von 0,1 Sekunden die Wahrnehmung einer Reaktion auf ihre Aktion haben müssen. Innerhalb einer Sekunde muss für sie das Gefühl entstehen, dass sie frei navigieren und auf ihrem momentanen Gedankengang fokussiert bleiben. Nach zehn Sekunden verspüren die Endnutzer einen Abriss im Ablauf, werden ungeduldig und verlassen die Seite. Die Konsequenz: Ladezeiten haben Auswirkungen auf die Wahrnehmung einer Marke, Konversion, Umsatz, Aufgabe von Shoppingkarten, Seitenbesuche und Search-Engine-Rangfolge.

InspectIT auf einem Blick

Die Architektur hinter InspectIT fungiert als Plattform, auf die aufgebaut, die erweitert und mit der die bereitgestellte Funktionalität an Anforderungen und Bedürfnisse angepasst werden kann. Sie ist in drei Haupttypen von Komponenten aufgeteilt: Agenten, Central Measurement Repository (CMR) und Benutzer-Schnittstellen. Die Agenten sind in das System integriert, das überwacht oder analysiert werden soll, und für das Sammeln von Daten verantwortlich. Es besteht die Möglichkeit, Statistiken der zugrunde liegenden Infrastruktur (CPU-Auslastung, verwendeter Speicher etc.) sowie detaillierte Laufzeitdaten zu erfassen. In Java-Anwendungen können dies beispielsweise Ausführungstraces, die Dauer eines Methoden-Aufrufs, JMX-Beans oder ausgeführte SQL-Anweisungen sein.

InspectIT stellt einen umfassenden Agenten für Java bereit, der Byte-Code-Instrumentierung zum Sammeln von Messdaten verwendet. Neben dem Java-Agenten erstellen Entwickler mehrere Agenten, um weitere Plattformen und Programmiersprachen zu unterstützen, so wurden zum Beispiel experimentelle Agenten für .NET, Android und Node.js erstellt.

Der Browser-Agent sammelt Informationen über die User Experience. Dabei werden Informationen über den Seitenaufbau im Browser und die Dauer von Anfragen gesammelt. Dadurch erhält man Rückschlüsse darauf, ob die User Experience den Erwartungen entsprochen hat. Die Java- und Browser-Agenten können auch in das ElasticSearch Ecosystem integriert sein. So existiert bereits ein eigener Beat, um die Daten des Browser-Agenten direkt an das ElasticSearch zu senden. Der Java-Agent kann seine Daten direkt an die Elastic-APM-Erweiterung senden.

Central Measurement Repository

Das InspectIT CRM ist die zentrale Komponente, die die von den Agenten gesammelten Daten empfängt. Es verwaltet die Daten und stellt Schnittstellen für Abfragen bereit. Neben der Datenverwaltung ist das CMR auch dafür verantwortlich, den Agenten mitzuteilen, welche Daten gesammelt werden sollen. Es unterscheidet zwischen zwei Arten von Daten: Langzeit-Daten (CPU-Auslastung, Antwortzeiten, laufende Threads etc.) und Detail-Daten (Ausführungsverfolgungen, Methodenaufruf-Hierarchien etc.).

Die detaillierten Daten werden In-Memory gehalten, sodass der InspectIT-Rich-Client sie analysieren kann. Auf Wunsch lassen sich die In-Memory-Daten auch speichern. Die Langzeit-Daten sind in einer Zeitreihen-Datenbank gespeichert, die als Grundlage für webbasierte Dashboards dient.

Abbildung 2: Überblick über das aktuelle Systemverhalten

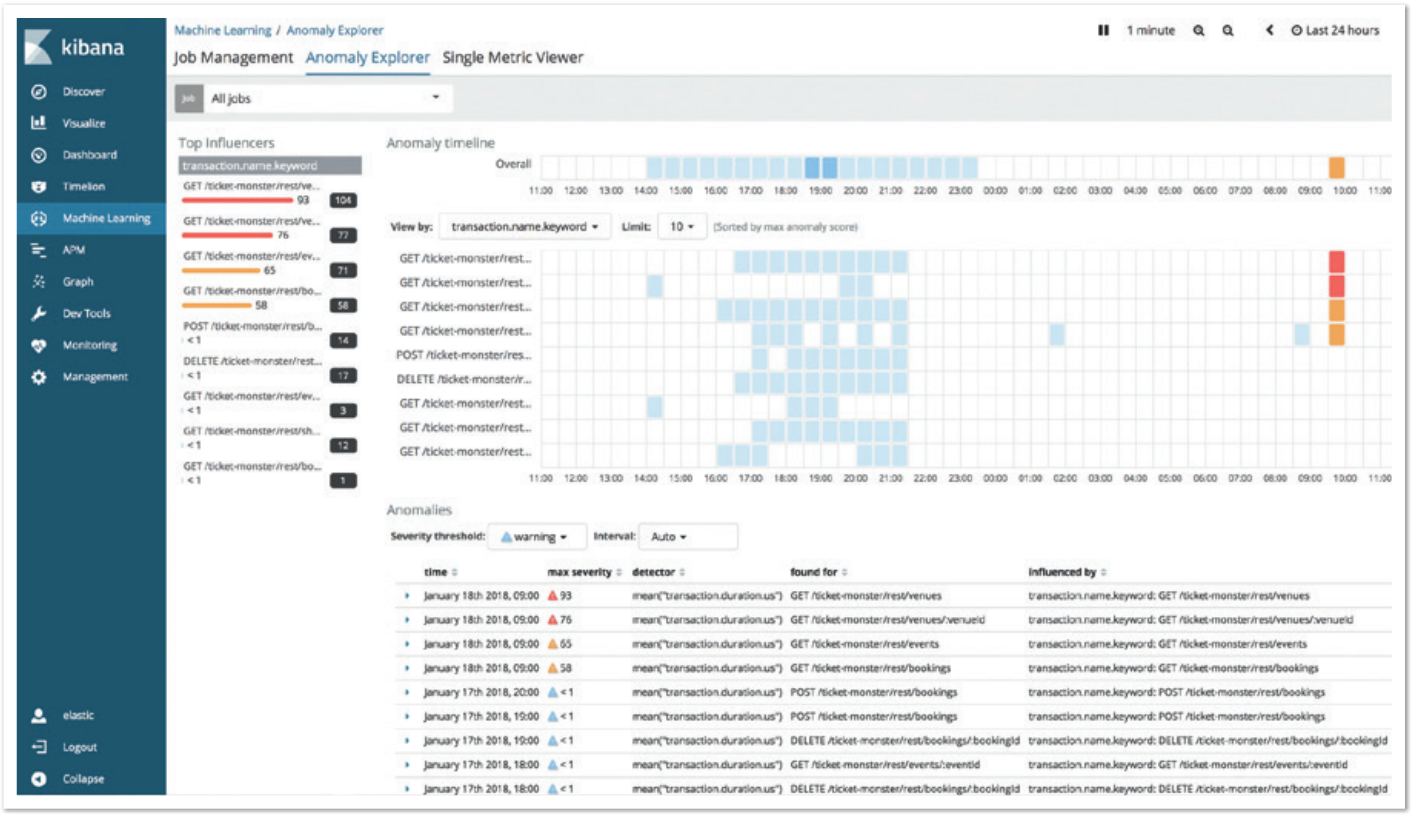


Abbildung 3: Anomalie-Übersicht in Kibana

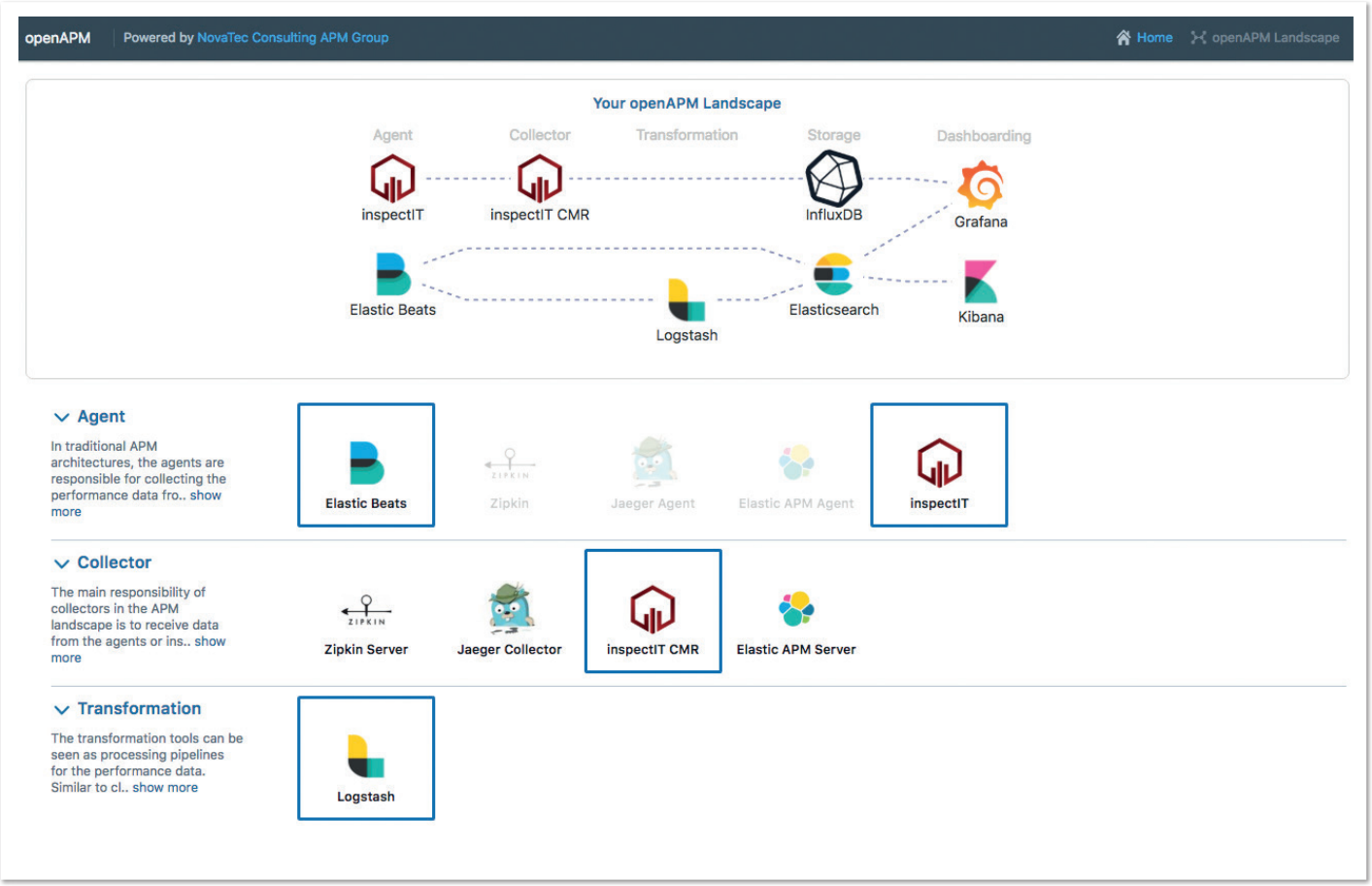


Abbildung 4: openAPM.io-Konfigurator

Benutzer-Oberflächen

Da die Überwachung der Systemzustands-Diagnose und die der Leistungsproblem-Diagnose zwei unterschiedliche Probleme mit unterschiedlichen Zielbenutzern und verschiedenen Anforderungen sind, bietet InspectIT zwei entsprechende Benutzeroberflächen. Mit dem Web-UI, das auf Grafana basiert, ist es möglich, auf einen Blick einen Überblick über das aktuelle Systemverhalten zu bekommen und den Rich Client zur weiteren Analyse der gesammelten Detaildaten zu verwenden *(siehe Abbildung 2)*. Bei der Integration in das ElasticSearch Ecosystem kann beispielsweise auch Kibana verwendet werden, um Analysen durchzuführen oder Anomalien zu erkennen *(siehe Abbildung 3)*.

Community

Das Open-Source-Projekt wurde von der Stuttgarter IT-Consulting-Firma NovaTec Consulting GmbH gestartet und wird bis heute maßgeblich unterstützt. Die Erkenntnisse aus den APM-Beratungsprojekten und die Anforderungen von Unternehmen leisten dadurch einen wesentlichen Beitrag für das Projekt.

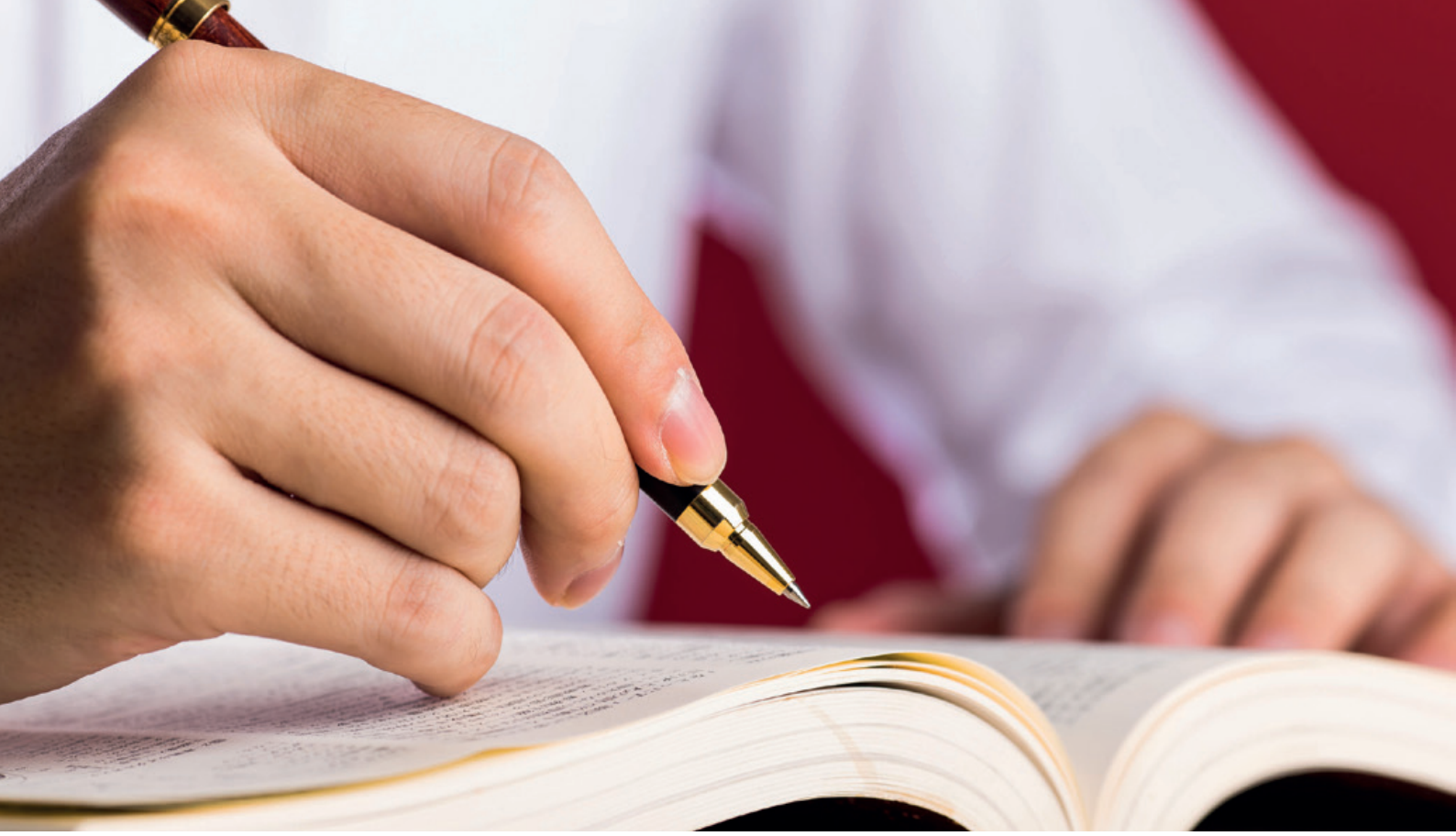
Abschließende Gedanken

Mit Open-Source-APM-Software können die Qualität eines Systems sichergestellt und Maßnahmen ergriffen werden, um weitere Verbesserungen voranzutreiben. InspectIT ist inzwischen eines von vielen Open-Source-Projekten zur Überwachung und Analyse des Systemverhaltens von Software-Systemen.

Die Integration verschiedener Open-Source-Projekte ermöglicht die Umsetzung einer umfassenden APM-Lösung. Die an eine umfassende APM-Lösung gestellten Anforderungen sind dabei so individuell wie die Systeme, die damit überwacht werden. Der Konfigurator unter *„http://www.openAPM.io“* unterstützt bei der Zusammenstellung einer individuellen APM-Lösung aus einer Vielzahl von Open-Source-Projekten *(siehe Abbildung 4)*; ganz nach den Bedürfnissen und Anforderungen des Einzelnen. Weitere Informationen zu InspectIT stehen unter *„http://www.inspectIT.rocks“*.

Alicia Bondanza
alicia.bondanza@novatec-gmbh.de

Alicia Bondanza ist Digital Content Creator im Bereich Application Performance Management der NovaTec Consulting GmbH. Alicia studiert Sprach- und Textwissenschaften an der Universität Passau und durch ihre Beiträge, bereitet sie einem breiten Publikum das Fachwissen der IT-Berater der NovaTec Consulting GmbH auf.



Continuous Documentation

Daniel Kocot, codecentric AG

Wir leben in einem Software-Entwicklungszeitalter der kontinuierlichen Prozesse. Funktionierende Software wird im Regelfall aus einem kontinuierlichen Prozess gewonnen. Der Artikel zeigt, wie man eine solche funktionierende Software dokumentiert.

Ein kontinuierlicher Entwicklungsprozess basiert auf den Phasen „Develop", „Build", „Test", „Deploy" und „Release". Sie stellen das Grundmodell von Continuous Delivery (CD) dar *(siehe Abbildung 1)*. Im Rahmen von CD findet auch immer der reine Development-Prozess nach agiler Art und Weise statt. Somit kommt als Startpunkt das Agile Manifest (AM) zum Tragen. Wenn man nun das AM bezogen auf die Dokumentation von Software betrachtet, findet sich dort der Satz „Working software over comprehensive documentation." Je nach Lesart kann der Eindruck entstehen, dass Dokumentation nur ein notwendiges Übel ist und daher vernachlässigt werden kann. Es geht jedoch vielmehr um den Umfang und den jeweiligen Adressatenkreis der Dokumentation.

Dokumentation im agilen Produktkonzept

Wenn man Dokumentation als Teil des agilen Entwicklungsprozesses betrachtet, ist es sinnvoll, Dokumente inkrementell zu erstellen. Mit jedem Update können Inhalte ergänzt oder aktualisiert werden. Idealerweise sind dabei die Zeiträume für einzelne Inkremente allerdings länger als bei der Software-Entwicklung.

Die Projektplanung gewinnt an Transparenz, wenn die Erstellung von Dokumenten darin als Arbeitspaket einfließt wie andere Tätigkeiten im Projekt auch. Damit lassen sich für Dokumentationsaufgaben auch Ressourcen einplanen und Deadlines definieren. Sinnvollerweise definiert ein Projekt einen Dokumentations-Verantwortlichen, der proaktiv dokumentationsrelevante Themen aufgreift und als Ansprechpartner für alle Frage der Dokumentation zur Verfügung steht.

Bei der Entwicklung von Software ist Anforderungsdokumentation nötig und wünschenswert, indem sie den Entwicklern bei der Umsetzung der Anforderungen tatsächlich hilft. Dabei müssen Anforderungen erst zu dem Zeitpunkt beschrieben sein, zu dem mit der Implementierung der gewünschten Funktionalität begonnen wird. Die Anforderungsdokumentation wird im Laufe des Projekts sukzessive weiterentwickelt. Neben der reinen Dokumentation für Entwickler wird auch Dokumentation nötig, die einen Überblick über ein Projekt in all seinen Facetten gibt; dies ist für viele Projekt-Beteiligte über einen langen Zeitraum hinweg nützlich.

Die Dokumentation gewinnt immens an Nutzen, wenn sie nicht nur die gewählten Konzepte beschreibt, sondern auch die Alternativen,

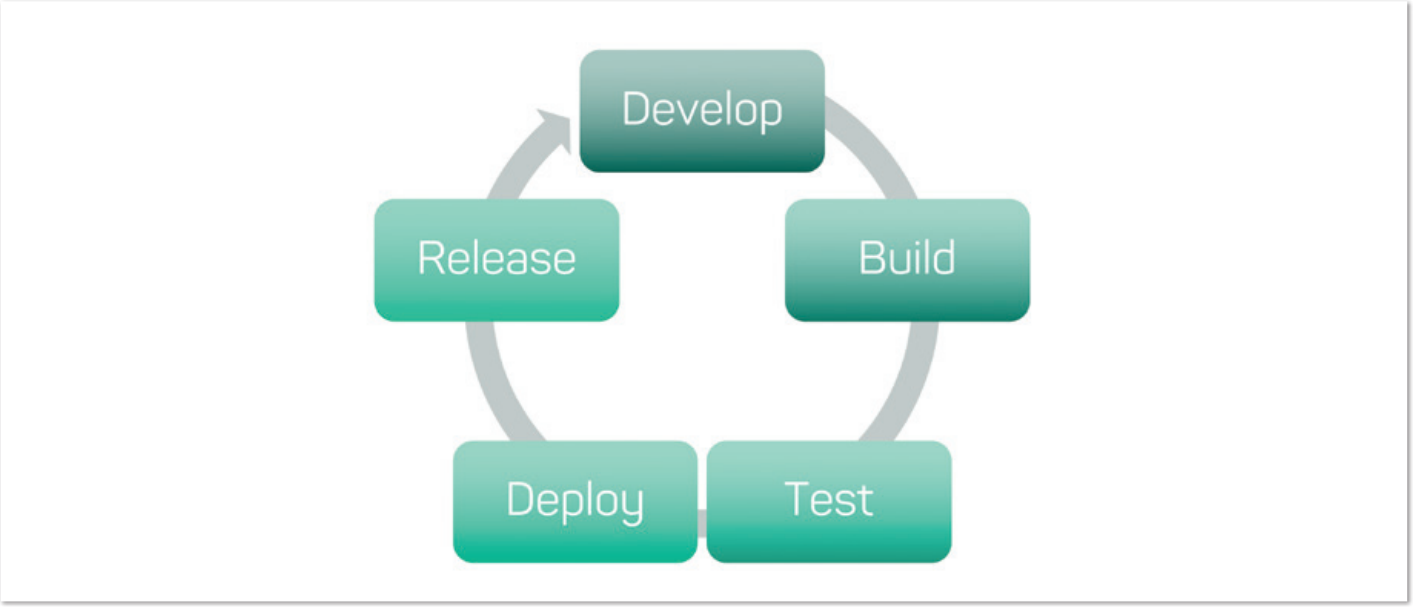


Abbildung 1: Der Continuous-Delivery-Lifecycle

die betrachtet worden sind, sowie die Gründe, die für die Entscheidungen ausschlaggebend waren. Nutzungsanleitungen sind am besten verständlich, wenn sie in Form eines Drehbuchs bereitgestellt werden. Dokumente sind sehr viel zugänglicher für den Leser, wenn sie mit konkreten Beispielen arbeiten. Jedes einzelne Dokument profitiert von einer klaren und übersichtlichen Struktur, die auch ein gewisses Maß an Querlesen erlaubt. Jedes Projekt-Dokument sollte mit Richtlinien für die Leser beginnen, die in kurzer und prägnanter Form klarmachen, für wen und in welcher Situation das Dokument geeignet ist.

Der punktuelle Einsatz von Diagrammen trägt spürbar zur Verständlichkeit der Materie bei. Ein häufiger Einsatz von Tabellen ermöglicht die systematische und übersichtliche Präsentation von Informationen. In Online-Dokumenten sollte von Hyperlinks Gebrauch gemacht werden, die auf andere Dokumente zu verwandten Themen verweisen beziehungsweise auf Abschnitte, Textpassagen oder Abbildungen darin. Auch die Dokumentation in IT-Projekten profitiert von einem ansprechenden und leserfreundlichen Layout. Idealerweise stehen für unterschiedliche Dokumenttypen geeignete Templates zur Verfügung.

Dokumente können sehr viel Nutzen entfalten, wenn die Autoren damit aktiv auf interessierte Leser zugehen. Im besten Fall entsteht im Projekt eine Dokumentationslandschaft, also ein navigierbarer Raum, in dem die unterschiedlichen Projekt-Dokumente in übersichtlicher Form abgelegt sind. Es ist sinnvoll, Retrospektiven auch dafür zu nutzen, Erkenntnisse über die Dokumentation zu gewinnen und diese Erkenntnisse in Best Practices einfließen zu lassen, die für zukünftige Projekte genutzt werden. Ein organisationsweites Wissensmanagement profitiert davon, wenn Projekt-Retrospektiven auch zu dem Zweck genutzt werden, projektübergreifende Erkenntnisse zu sammeln sowie Ideen und Konzepte zu identifizieren, die über den aktuellen Projekt-Kontext hinaus von Bedeutung sind.

Tools

Nachdem das Thema „Continuous Documentation" zuerst von der theoretischen Seite betrachtet wurde, nun der Blick auf Tools, die die Philosophie von Documentation as Code (Doc-as-Code) entsprechend unterstützen können. Doc-as-Code geht grundsätzlich davon aus, dass Dokumentation nahezu mit denselben Tools und Workflows erstellt wird wie der Sourcecode der Entwicklung. Der Einsatz von Git als Versionskontrolle und Gradle als Build-Management wird als gegeben vorausgesetzt.

Um Dokumentation nah an der Software zu schreiben, wird Plain Text Markup verwendet. Im Folgenden wird hierfür Asciidoc(tor) eingesetzt, wobei Asciidoc die reine Markup-Sprache darstellt und Asciidoctor für die Konvertierung in verschiedene Endformate benutzt wird. Ein großer Vorteil von Asciidoc und Plain Text Markup im Allgemeinen ist, dass die Dokumente schon mit einfachem Editor vor jeglicher Konvertierung les- und durchsuchbar sind. In Verbindung mit einer Versionskontrolle ist die Dokumentation genauso versionierbar wie der reine Entwicklungscode *(siehe Abbildung 2)*.

Die Dokumentation von Software beinhaltet im Regelfall auch eine Vielzahl von Grafiken und Diagrammen. Um diese nun in einen Workflow mit Asciidoc(tor) zu integrieren, soll PlantUML verwendet werden. Es bietet unterstützt durch Plain Text Markup die Möglichkeit, UML-Diagramme und weitere zu erstellen. Für Asciidoctor ist eine Erweiterung (Asciidoctor Diagram) verfügbar, die es dem Anwender ermöglicht, PlantUML-Notation direkt in das Asciidoc-Dokument zu integrieren *(siehe Abbildung 3)*.

Wie bereits zu Beginn erwähnt, ist es für die Entwicklung wichtig, dass die Anforderungen an die Software entsprechend dokumentiert werden. Wie kann der Entwickler auf direkte Änderungen der Anforderung innerhalb eines Sprints reagieren? Lassen sich Anforderungen auch durch Product Owner ohne Entwicklungshintergrund als „as Code" abbilden? Um das beschriebene Ziel zu erreichen, kommt Gherkin, eine sogenannte „Business Readable DSL", zum Einsatz. Gherkin ist eine zeilenorientierte Sprache, die Einrückungen für die Strukturierung benutzt *(siehe Listing 1)*.

Für das Testen der Anforderungen soll Spock verwendet werden. Spock ist in der JVM-Sprache Groovy geschrieben und nutzt eine

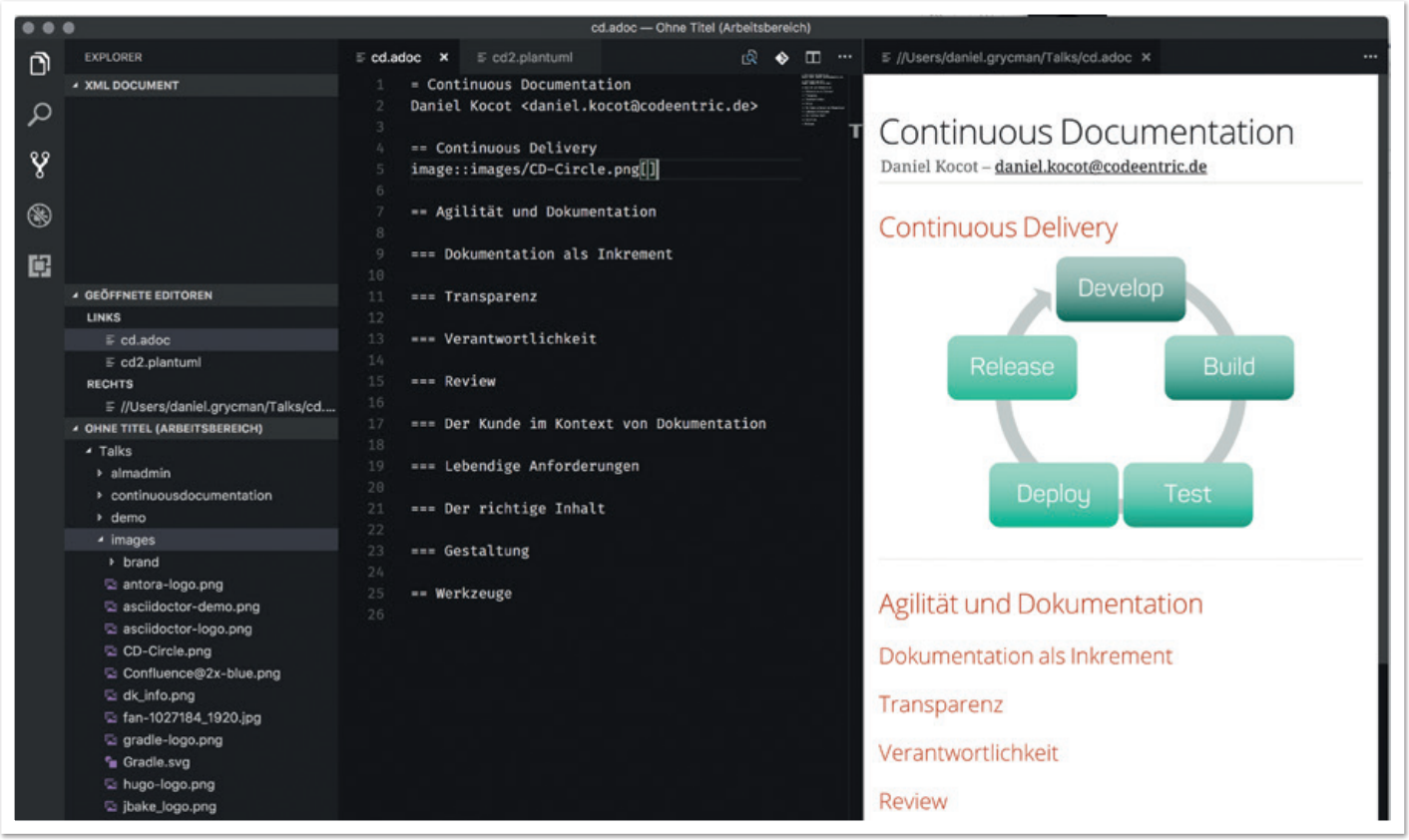


Abbildung 2: Asciidoc mit Preview im Visual Studio Code Editor

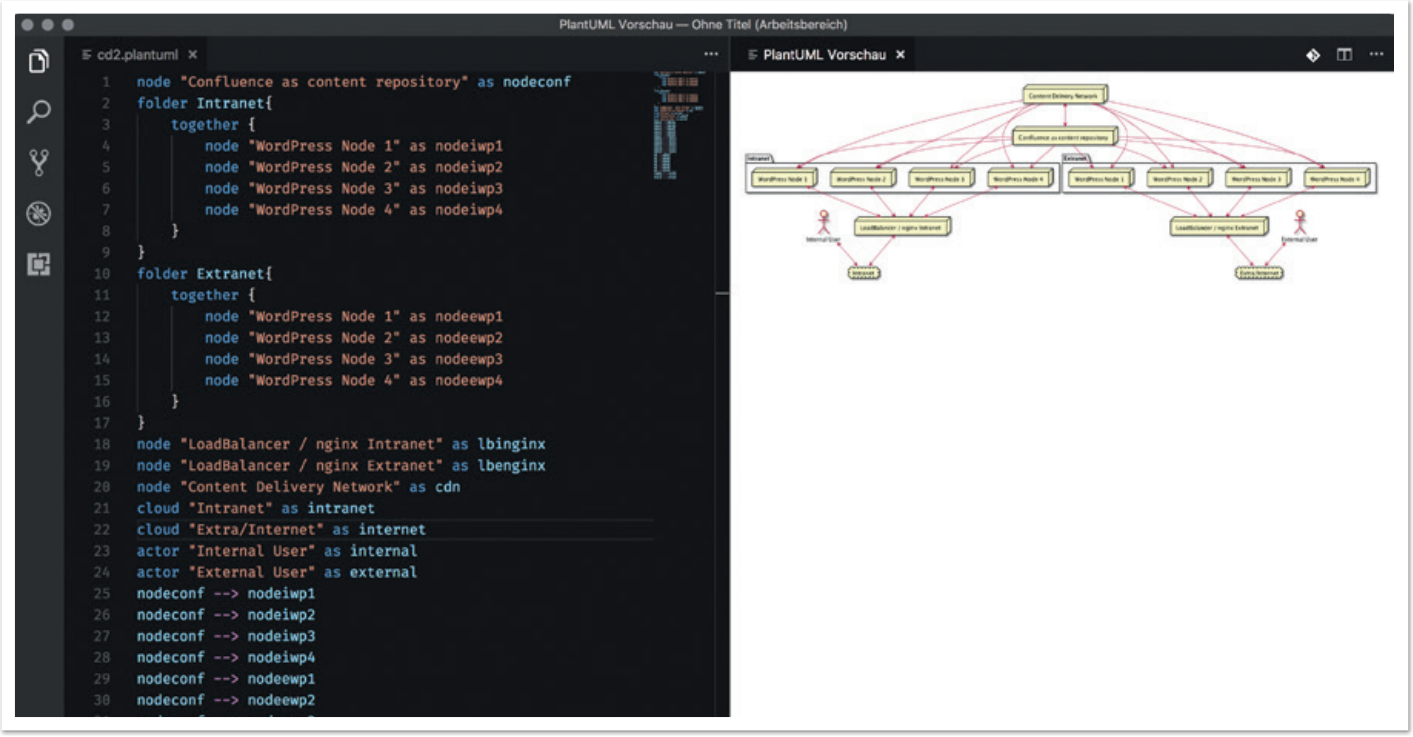


Abbildung 3: PlantUML mit Preview im Visual Studio Code Editor

```
Feature: Search
  Scenario: Simple Search
  Given a web browser is on the Google page
  When the search phrase "codecentric" is entered
  Then results for "codecentric" are shown
```

Listing 1

eigene Domain Specific Language (DSL) für die Beschreibung der Tests. Genau diese DSL fördert die Übersichtlichkeit und Lesbarkeit der Tests. Natürlich lässt sich auch reiner Java-Code mit Spock ausführen.

Eine weitere Besonderheit von Spock ist die Darstellung von fehlgeschlagenen Tests. Hiermit wird die fehlgeschlagene Bedingung bis in ihre Einzelkomponenten visualisiert. Mit einem selbst geschriebenen Gradle-Task können nun die einzelnen Gherkin-Specifications in Spock-Specifications transformiert werden. Für den Artikel umfasst der Gradle-Task nur die Transformation der einfachsten Gherkin-Struktur *(siehe Listing 2)*.

Das Groovy-Skript stellt eine erste Basisumsetzung der Gherkin-Spezifikation dar. Durch Gherkin können wir nun einen Workflow zwischen dem Produkt-Owner und den Entwicklern umsetzen, der

```
task createSpockSpecs {
  group = "Generating Spock Specs from gherkin"
  doLast {
     def gherkinFiles = fileTree("src/test/resources/Features").files
     def specBlock
     gherkinFiles.each {

        File transformFile = new File("$projectDir/src/test/groovy/$it.name".replaceFirst(~/\.[^\.]+$/,'Spec.groovy'))

        if (!transformFile.exists()) {
           transformFile.createNewFile()

           def completeScenarioString
           def completeGivenString
           def fileScenarioString
           def completeWhenString
           def completeThenString

           println "Generate Spock Spec"
           def multiline = it.text
           def list = multiline.readLines()
           list.removeAll { it.startsWith("Feature:") }
           list.find {
              if (it.contains("Scenario")) {
                 def scenarioString = it.toString().replace("Scenario: ", "def ")
                 fileScenarioString = scenarioString.substring(4)
                 completeScenarioString = scenarioString + "() {"
              }
              if (it.contains("Given")) {
                 String givenString = it.toString().replace("Given", "given: \"")
                 completeGivenString = givenString + "\""
              }
              if (it.contains("When")) {
                 String whenString = it.toString().replace("When", "when: \"")
                 completeWhenString = whenString + "\""
              }
              if (it.contains("Then")) {
                 def thenString = it.toString().replace("Then", "then: \"")
                 completeThenString = thenString + "\"\n}"
              }
           }

           def minusContent = new StringBuilder(it.text)
           def fileContentSb = new StringBuilder(it.text)

           specBlock = fileContentSb.insert(0, "import spock.lang.Specification\n" +
                  "\n" +
                  "\n" +
                  "class " + it.name.take(it.name.lastIndexOf('.')) + "Spec extends Specification {\n"
                  + completeScenarioString + "\n"
                  + completeGivenString + "\n"
                  + completeWhenString + "\n"
                  + completeThenString + "\n}").minus(minusContent)

           def spockFiles = fileTree("src/test/groovy").files

           spockFiles.each {
              println it.name
              println it.name.take(it.name.lastIndexOf('.'))
              if (specBlock.contains(it.name.take(it.name.lastIndexOf('.')))) {
                 it.setText(specBlock)
              }
           }
        }
     }
  }
}
```

Listing 2

```
com.athaydes.spockframework.report.IReportCreator=com.athaydes.spockframework.report.template.TemplateReportCreator
com.athaydes.spockframework.report.template.TemplateReportCreator.specTemplateFile=/template/spec-template.ad
com.athaydes.spockframework.report.template.TemplateReportCreator.reportFileExtension=ad
com.athaydes.spockframework.report.template.TemplateReportCreator.summaryTemplateFile=/template/summary-template.ad
com.athaydes.spockframework.report.template.TemplateReportCreator.summaryFileName=index.ad
com.athaydes.spockframework.report.outputDir=src/documentation/content/test-reports
com.athaydes.spockframework.report.hideEmptyBlocks=false
```

Listing 3

```
plugins {
   id 'org.jbake.site' version '1.0.0'
}
```

Listing 4

```
jbake {
   srcDirName = 'src/documentation'
   destDirName = 'documentation'
}
```

Listing 5

```
= Project documentation
Daniel Kocot
2017-10-03
:jbake-type: page
:jbake-tags: documentation, manual
:jbake-status: published
```

Listing 6

zu weniger Iterationen zwischen einer Anforderung und deren Umsetzung innerhalb eines Sprints beitragen kann.

Für die Darstellung der transformierten Anforderungen wird Spock Reports verwenden. Damit haben wir die Möglichkeit, die Ergebnisse des Spock-Tests mithilfe von Templates in Asciidoc darzustellen. Um diese Templates anpassen zu können, müssen wir eine Properties-Datei unter „test/resources" anlegen *(siehe Listing 3)*. Nun wird bei jedem Build der Software die Dokumentation um die getesteten Anforderungen erweitert.

Mit dem wachsenden Bedarf von APIs stellt sich auch die Frage, wie man deren Dokumentation in einen kontinuierlichen Prozess einbeziehen kann. Hier zeichnen sich nun zwei alternative Wege ab, „contract-first" und „code-first". Bei „contract-first" wird die Dokumentation des Rest-API aus der Swagger-YAML erstellt. „Code-first" hingegen erzeugt die entsprechende Dokumentation Test-getrieben. Die bei beiden Vorgehen erstellten Asciidoc-Dateien lassen sich dann auch wieder in den schon vorhandenen kontinuierlichen Prozess einbinden.

Bisher haben wir die Dokumentation als lokale Generierung betrachtet. Im Continuous Delivery kommen jedoch grundsätzlich sogenannte „Build-Server" wie Jenkins zum Einsatz. Hier werden die Software-Artefakte entsprechend den jeweiligen Quellen gebaut und im Anschluss verteilt. Neben der reinen Verteilung der Artefakte kann über den Build-Server eine separate Bereitstellung der Dokumentation gewährleistet werden.

Bislang haben wir die mögliche Generierung von Inhalten und deren Bereitstellung mithilfe von Tools angeschaut. Die Dokumentation sollte Multi-Channel-basiert aufbereitet sein, um diese dann gemäß dem Single-Source-Publishing-Prinzip zu verbreiten. Genau diese Vorgehensweise haben wir durch den Einsatz der weiter oben vorgestellten Werkzeuge erreicht. Nun gilt es einen Blick auf die verschiedenen Ausgabekanäle zu werfen.

Den wohl wichtigsten Kanal zur Bereitstellung von Informationen innerhalb von Software-Entwicklungsprozessen stellen Wikis dar. Beispielhaft seien hier Confluence und XWiki genannt, die beide über ein Rest-API verfügen. Darüber lassen sich innerhalb des Deployment-Prozesses Elemente zu einer Dokumentation hinzufügen oder aktualisieren.

Die Static Site Generators (SSGs) sind der zweite Kanal, den wir uns etwas genauer anschauen wollen. Zuerst wollen wir allerdings klären, was SSGs eigentlich sind. Damit lassen sich statische Webseiten auf der Grundlage von Plain Text Markup erstellen, wie in unserem Falle Asciidoc. Die SSGs werden für verschiedenste Programmiersprachen angeboten. Da wir uns im Java-Umfeld bewegen, soll unser Schwerpunkt auf JBake liegen. JBake lässt sich zum einem über die Kommandozeile und zum anderem auch über Build-Management steuern und ausführen. Durch die Benutzung von Gradle als Build Management Tool muss die „gradle.build"-Datei um das „jbake gradle"-Plug-in erweitert werden *(siehe Listing 4)*.

Standardmäßig erwartet das Gradle-Plug-in die Dokumentationsquellen unter „src/jbake". Mit Parametern im Build-File lässt sich der Quell- und Zielort jedoch anpassen, wobei das Ziel immer im Build-Ordner liegt. In unserem Fall mit den Werten in *Listing 5*.

Innerhalb des Asciidoctor-Dokuments können JBake-Metainformationen hinterlegt sein wie Seitentyp, Tags und der Status der jeweiligen Seite. Letzterer regelt die Sichtbarkeit der Seiten nach dem „Backprozess". Wem das Standard-Template nicht gefällt, kann dieses mithilfe der Template-Sprachen Freemaker, Groovy Simple, Groovy Markup, Thymeleaf und Jade seinen Wünschen nach anpassen *(siehe Listing 6)*.

Somit sind die Grundlagen für die Benutzung von JBake gelegt. Es soll nicht unerwähnt bleiben, dass Dan Allen, Asciidoctor-Lead, mit Antora einen neuen Ansatz in dem Bereich „Static Site Generation" veröffentlicht hat. Damit ist es möglich, anhand von sogenannten „Playbooks" Dokumentation auf Basis von verschiedensten Git-Repositories zu erstellen und somit ein noch umfassenderes Verständnis von Projektdokumentation und deren Erstellung zu schaffen.

Wir wollen uns noch einen weiteren und zugleich letzten Ausgabekanal anschauen, das Portable Document Format (PDF). Dieses

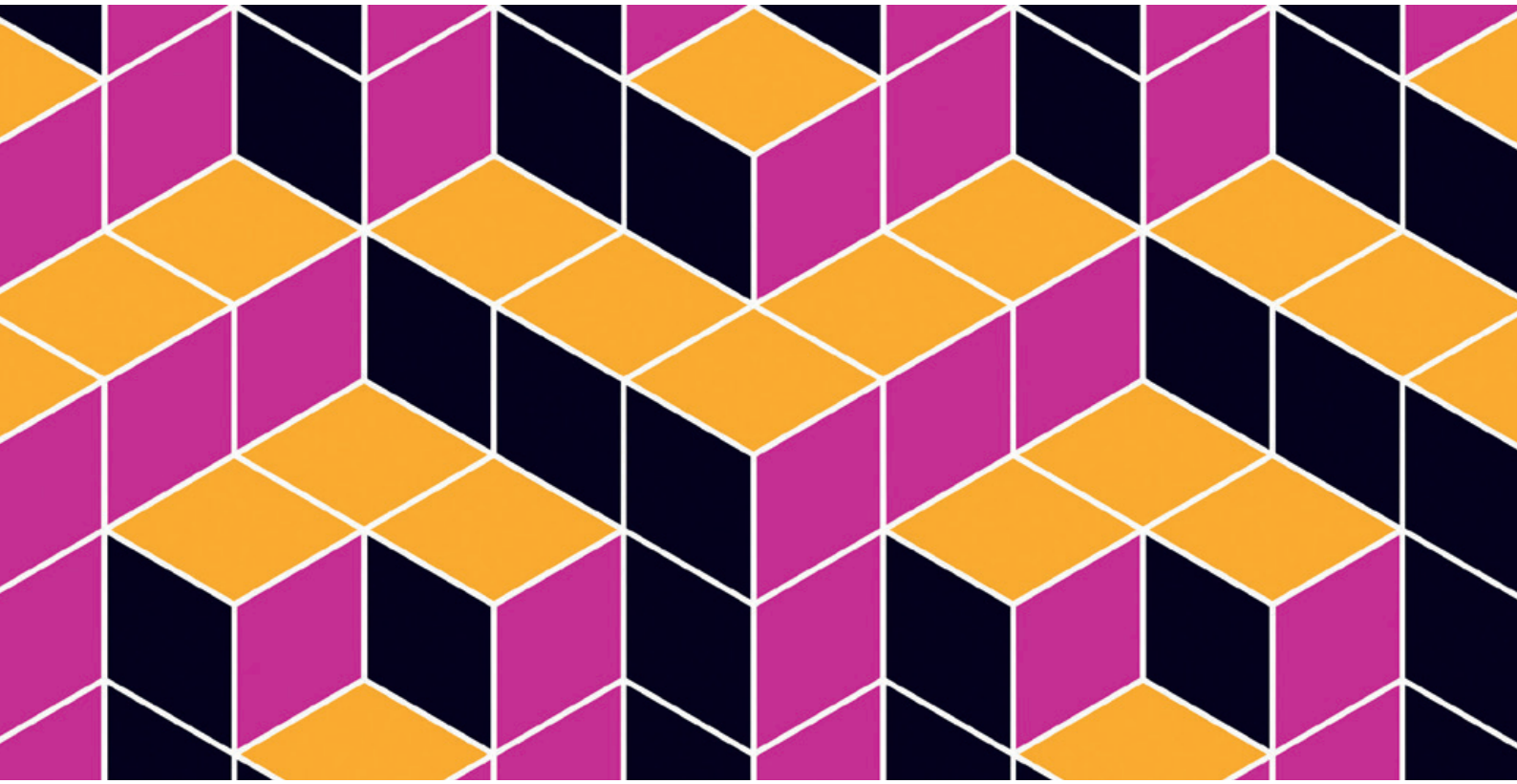
Format begleitet uns schon seit Anfang der 1990er Jahre und ist ein sogenanntes „plattformunabhängiges Dateiformat". Es stellt sich nun die Frage, wie wir aus der erstellten Asciidoc-Datei zu einem PDF-Dokument kommen. Genau für dieses Vorhaben existiert eine Asciidoctor-Erweiterung (Asciidoctor-PDF). Über diese lässt sich im gesamten Prozess, sowohl lokal als auch über den Build-Server, ein PDF-Dokument erstellen. Wichtig ist es, hierbei anzumerken, dass es sich bei dem PDF dann um einen nicht veränderbaren Snapshot der jeweiligen Dokumentation handelt.

Fazit

Dieser Artikel zeigt, dass es möglich ist, eine Dokumentation von kontinuierlichen Prozessen umsetzen. Hierzu wurde mit verschiedensten Tools eine Continuous-Delivery-Pipeline erstellt. Die aufgezeigten Möglichkeiten sollen aber nicht als alleinige Lösung für die Herausforderung der Dokumentation von Software verstanden werden. Es sollen vielmehr Denkanstöße geliefert werden, sich mit dem Thema auseinanderzusetzen.

Daniel Kocot
daniel.kocot@codecentric.de

Daniel Kocot ist seit Oktober 2016 Mitglied des Teams der codecentric am Standort in Solingen. Seit mehr als fünfzehn Jahre setzt er sich mit IT-Herausforderungen und deren Lösungen auseinander. Schwerpunktmäßig widmet er sich gerade den Themen „API Thinking" und „Legacy Modernization".



Modulare Multi-Release-JAR-Dateien

Guido Oelmann, Freelancer

Mit Java 9 wurde das Konzept der Multi-Release-JARs, auch als „MRJAR" bekannt, eingeführt. Mit diesem Feature ist es möglich, mehrere Versionen gleicher Klassen für unterschiedliche Java-Laufzeitumgebungen in ein einzelnes JAR zu verpacken. Der Artikel zeigt die Verwendung dieser Möglichkeit unter Berücksichtigung der Java-Versionen 8, 9 und 10 und von Java-Modulen.

Viele Frameworks und Bibliotheken unterstützen unterschiedliche Java-Versionen; beispielsweise unterstützt das Spring-Framework in der Version 4 die Versionen 6, 7 und 8 der Java-Plattform. Die Unterstützung verschiedener Java-Releases führt häufig dazu, dass Sprach-Features der neuesten Java-Version in den einzelnen Frameworks oder Bibliotheken nicht verwendet werden, um eine Abwärtskompatibilität einfacher zu gewährleisten. Die eigentlich notwendigen, bedingten Plattform-Abhängigkeiten im Programmcode auszudrücken oder die Verteilung verschiedener Artefakte für verschiedene Java-Versionen stellt sich dabei als schwierig dar.

Bei der Realisierung von bedingten Plattform-Abhängigkeiten im Programmcode greift man oft auf die Möglichkeiten von Reflection-Zugriffen zurück. Dabei wird dem Service-Provider-Mechanismus folgend eine allgemeine Provider-Schnittstelle definiert, die für alle Java-Versionen Gültigkeit hat und die für die unterschiedlichen Java-Versionen jeweils Implementierungen der Schnittstelle erstellt, die dann als Provider fungieren und zur Laufzeit geladen werden. Eine alternative Möglichkeit ist die Verwendung einer einzigen Klasse mit unterschiedlichen Methoden für unterschiedliche Versionen und der Zugriff auf diese mit Reflection. Dass dies nicht der optimale Lösungsansatz ist, sollte offensichtlich sein.

Die Verteilung verschiedener Artefakte ist eine weitere Möglichkeit, die im Grunde nur die Bereitstellung verschiedener JARs für die einzelnen Java-Versionen bedeutet. Hier würden die verschiedenen Implementierungen der gleichen Klasse gleichzeitig vorgehalten und es obliegt dem Build-Tool, diese in zwei verschiedene Artefakte zu kompilieren, zu testen und zu verpacken. Die Unterstützung für ein solches Vorgehen ist in den Build-Tools der Java-Welt unterschiedlich stark ausgeprägt.

Es zeigt sich also ein echtes Problem insbesondere für Drittanbieter von Bibliotheken und Frameworks, Sprach-Features neuerer Java-Versionen zu nutzen, die zum Teil einen Performance-Gewinn bedeuten würden, bei gleichzeitiger Berücksichtigung der Abwärtskompatibilität. Multi-Release-JAR-Dateien adressieren genau dieses Problem, indem das JAR-Dateiformat in der Form erweitert wurde, dass auch mehrere Java-Release-spezifische Versionen von Klassen und Ressourcen-Dateien gleichzeitig im selben Artefakt ihren Platz finden. Dadurch können insbesondere Drittanbieter von Bibliotheken und Frameworks die mit höheren Java-Versionen eingeführten Sprach-Features einfacher nutzen. Im folgenden Beispiel wird ein Multi-Release-JAR für die Java-Versionen 8, 9 und 10 unter Verwendung von Java-Modulen erstellt.

Multi-Release-JAR-Architektur

Zunächst ein Blick auf den Aufbau eines Multi-Release-JAR. *Listing 1* zeigt den grundsätzlichen Aufbau anhand eines Beispiels. Zu sehen ist der Inhalt eines Multi-Release-JAR, das Klassen für die Java-Versionen 8, 9 und 10 enthält. Bei den Dateien „Main.class" und „Example.class" im Hauptverzeichnis handelt es sich um die für Java 8 kompilierten Klassen. Von der Datei „Example.class" gibt es noch zwei weitere Versionen für Java 9 und 10, die unter „META-INF/versions" zu finden sind. Diese Dateien werden dann von einer Java-9- oder Java-10-Laufzeitumgebung anstatt der Datei im Hauptverzeichnis verwendet.

Im Falle einer Java-8-Laufzeitumgebung wird der Ordner „/versions" einfach ignoriert. Damit die JVM überhaupt weiß, dass es sich nicht um eine gewöhnliche JAR handelt, ist ein Eintrag in der „Manifest"-Datei erforderlich. Hierzu wird der Inhalt der „MANIFEST.MF"-Datei um das Attribut „Multi-Release: true" ergänzt. Bei einer JVM der Version 8 oder niedriger wird dieser Eintrag einfach ignoriert.

Ein Multi-Release-JAR erstellen

Zunächst wird eine geeignete Dateistruktur angelegt *(siehe Listing 2)*. Unterhalb des Projektverzeichnisses „multi-release-jar-example" liegen die drei Ordner „lib8", „lib9" und „lib10" mit den jeweiligen Sourcen für die verschiedenen Java-Versionen. Bei den Java-9- und Java-10-Versionen handelt es sich um Java-Module, weshalb sich oberhalb der Paketstruktur der Modul-Deskriptor „module-info.java" findet. Der Programmcode für die Java-8-Version steht im *Listing 3*.

Das Hauptprogramm „Main" erzeugt eine Instanz der Example-Klasse und führt dort die Methode „showVersion" aus. Die „Main"-Klasse soll für alle Java-Versionen gleich sein, nur die „Example"-Klassen sind unterschiedlich. *Listing 4* zeigt den Modul-Deskriptor und den Code für die Java-9-Version. Der Modul-Deskriptor der Java-10-Version sieht genauso aus und die Klasse „Example" unterscheidet sich nur darin, dass dort als Version „Java 10" ausgegeben wird. Der „ModuleLayer" wird angesprochen, um bei der Ausführung zeigen zu können, ob das JAR-Artefakt auf dem Klassen- oder auf dem Modulpfad liegt.

Bevor das eigentliche JAR erstellt wird, müssen zunächst alle Dateien für die jeweiligen Versionen kompiliert werden. Seit Java 9 gibt es dafür das Compiler-Flag „--release". Zum Kompilieren reicht also der Java-10-Compiler, um auch den Byte-Code für die niedrigeren Versionen zu erzeugen. Im Projektverzeichnis „multi-release-jar-example" sind die Befehle auszuführen, um die einzelnen Sourcen zu

```
multirelease.jar/
├── META-INF/
│   ├── versions
│   │   ├── 9
│   │   │   └── Example.class
│   │   └── 10
│   │       └── Example.class
│   └── MANIFEST.MF
├── Main.class
└── Example.class
```

Listing 1

```
multi-release-jar-example /
├── lib8/
│   ├── src/
│       ├── de/javaakademie/multirelease
│                             ├── Main.java
│                             └── Example.java
├── lib9/
│   ├── src/
│       ├── de/javaakademie/multirelease
│       │                     └── Example.java
│       └ module-info.java
└── lib10/
    └── src/
    └── de/javaakademie/multirelease
        │                     └── Example.java
        └ module-info.java
```

Listing 2

```
package de.javaakademie.multirelease;

import de.javaakademie.multirelease.Example;

public class Main {
    public static void main(String[] args) {
        new Example().showVersion();
    }
}

package de.javaakademie.multirelease;

public class Example {
    public void showVersion() {
        System.out.println("Version: Java 8");
    }
}
```

Listing 3

kompilieren *(siehe Listing 5)*. Danach werden die kompilierten Dateien in ein gemeinsames Multi-Release-JAR verpackt *(siehe Listing 6)*.

Beim Verpacken werden als Erstes die Dateien „Main.class" und „Example.class" als Default-Klassen in die Datei „multi-release-jar-example.jar" gepackt. Danach werden mit den Flags „--release 9" und „--release 10" die weiteren versionsbezogenen Klassen hinzugefügt. Durch die Verwendung des Flag werden die entsprechenden Strukturen unter „META-INF/versions" angelegt. Als Default-Klassen sollten immer die Klassen für die kleinste Java-Version gewählt werden.

Die Anwendung im resultierenden Artefakt kann dann mit „java -jar multi-release-jar-example.jar" auf den verschiedenen JVM-Versionen zur Ausführung gebracht werden. Allerdings würde die JAR-

```
module de.javaakademie.multirelease {
  exports de.javaakademie.multirelease;
}

package de.javaakademie.multirelease;

import java.lang.ModuleLayer;

public class Example {
    public void showVersion() {

        System.out.println("Version: Java 9");

        ModuleLayer ml = Example.class.getModule().getLayer();
        if( ml != null ) {
            System.out.println("Layer.Modules: " + ml.modules());
        } else {
            System.out.println("ModuleLayer ist null");
        }
    }
}
```

Listing 4

```
javac --release 8 -d lib8/classes lib8/src/de/javaakademie/multirelease/*.java

javac --release 9 -d lib9/classes lib9/src/module-info.java lib9/src/de/javaakademie/multirelease/Example.java

javac --release 10 -d lib10/classes lib10/src/module-info.java lib10/src/de/javaakademie/multirelease/Example.java
```

Listing 5

```
jar --create --file multi-release-jar-example.jar --main-class=de.javaakademie.multirelease.Main -C lib8/classes .
--release 9 -C lib9/classes .  --release 10 -C lib10/classes .
```

Listing 6

Datei dann auf dem Klassen- und nicht auf dem Modulpfad liegen. Bei Verwendung einer JVM ab Version 9 lässt sich die Anwendung außerdem mit „java -p . -m de.javaakademie.multirelease" starten, um den Modulpfad zu verwenden. Zum Starten wird lediglich der Modul-Name angegeben und mit den Informationen in „MANIFEST.MF" weiß die JVM zudem, wo die „Main"-Klasse zu finden ist.

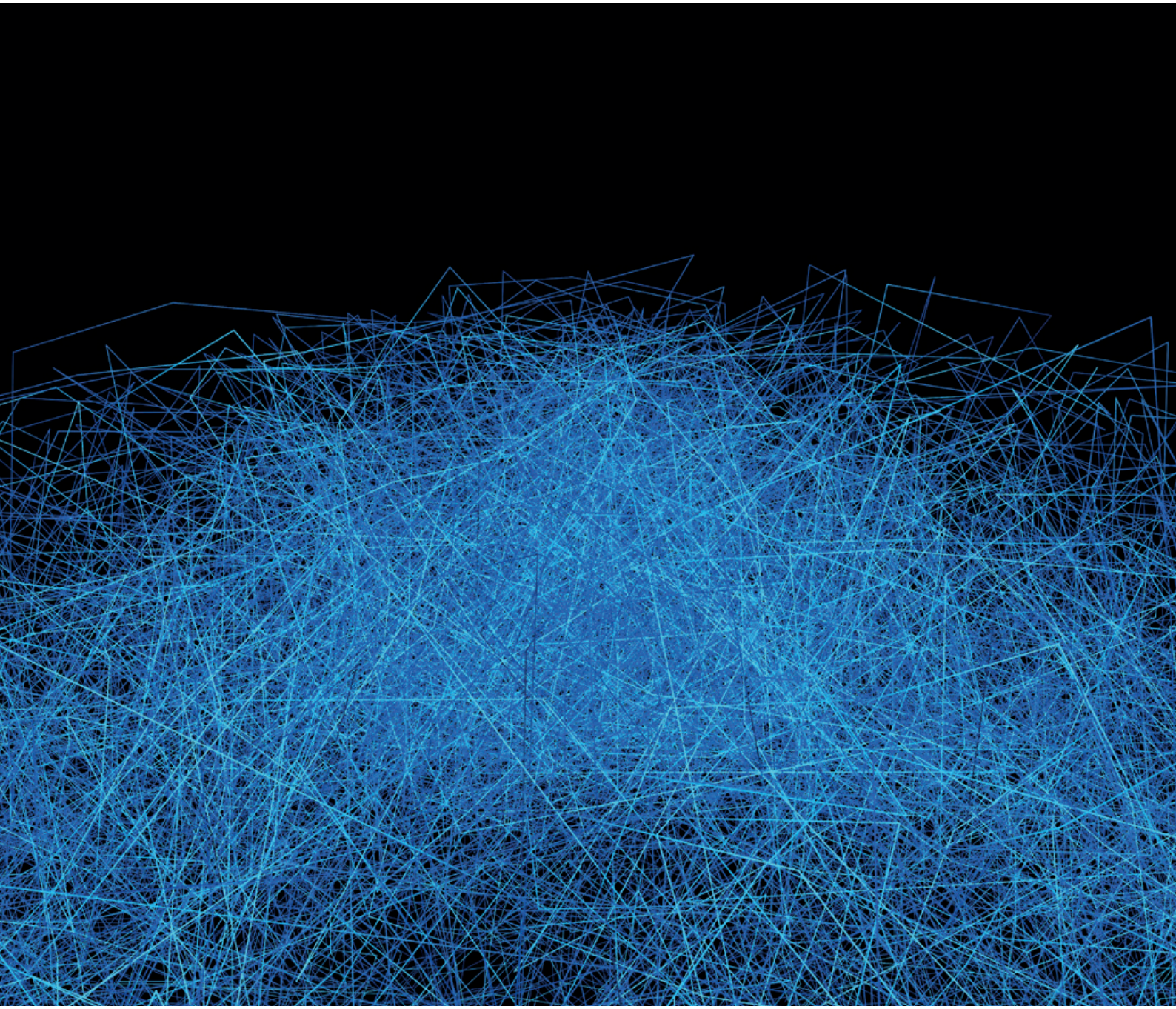
Angemerkt sei an dieser Stelle, dass sich das Nachladen von Klassen oder Ressourcen zur Laufzeit etwas anders darstellt als bei gewöhnlichen JARs. Anstatt den Zugriff auf eine Klasse über „jar:file:/multi-release-jar-example.jar!/Example.class" durchzuführen, wird als Ressourcen-Pfad im „UrlClassLoader", wenn beispielsweise die Java-10-Version verwendet werden soll, „jar:file:/multi-release-jar-example.jar!/META-INF/versions/10/Example.class" angegeben. Das komplette Beispiel ist unter *„https://github.com/javaakademie/Modulare-Multi-Release-JAR"* abgelegt.

Fazit

Multi-Release-JARs bieten eine leichte und komfortable Möglichkeit, Klassen auszuliefern, die gleich mehrere Java-Plattform-Versionen unterstützen. Insbesondere Drittanbieter von Frameworks und Bibliotheken können dadurch neue Sprach-Features in ihren Programmcode einbringen, ohne direkt den gesamten Code des Projekts auf eine neue Version migrieren zu müssen. Anzumerken sei allerdings, dass sich in vielen Fällen die Berücksichtigung von Plattform-Abhängigkeiten durch Build-Tools durch das Erzeugen unterschiedlicher JARs eher anbietet. Abzuraten ist meistens von einer programmtechnischen Lösung wie dem Zugriff über Reflections. Der Artikel hat gezeigt, wie Multi-Release-JARs gebaut werden und wie die mit Java 9 eingeführten Java-Module dabei berücksichtigt werden können.

Guido Oelmann
guido.oelmann@javaakademie.de

Guido Oelmann arbeitet als freiberuflicher Software-Architekt, Berater und Trainer. Zu seinen Schwerpunkten gehört neben agilen Entwicklungsmethoden und Software-Architekturen der Einsatz von Java-/Java-EE-Technologien in verteilten Systemen. Er unterstützt Unternehmen durch die Mitarbeit in Entwicklungsprojekten und verfügt über viele Jahre Erfahrung beim Entwurf und bei der Entwicklung großer IT-Systeme in unterschiedlichen Branchen. Darüber hinaus ist er Autor des Buches „Modularisierung mit Java 9", das im dpunkt.verlag erschienen ist.



Generierung von Regressionstests für Legacy-Code

Felix Schumacher, INNOQ

In diesem Artikel geht es um die Möglichkeit, bei Legacy-Anwendungen Regressionstests anhand des Quellcodes zu generieren, um vor einem möglichen Refactoring Tests erzeugt zu haben. Diese sollen sicherstellen, dass die Anwendung nach dem Refactoring noch genauso funktioniert wie vorher. Hier gibt es einige interessante Ansätze und auch einige Tools, die diese implementieren. Der Artikel zeigt zwei mögliche Ansätze.

```
public class TestValue{
  public static boolean is23(int parameter){
     if(parameter == 23){
        return true;
     }
     return false;
  }
}
```

Listing 1

```
public class TestValueMultiplication{
  public static boolean is23(int parameter){
     if(parameter * 2 == 46){
        return true;
     }
     return false;
  }
}
```

Listing 2

Im Projektalltag hat man es häufig mit Legacy-Anwendungen zu tun, die weiterentwickelt und gewartet werden müssen. Meist sind diese Anwendungen historisch gewachsen und das Finden von Bugs oder das Einbauen neuer Features ist schwierig und langwierig. Ein Refactoring ist bei einem System in der Wartungs- und Betriebsphase meistens nicht leicht unterzubringen, vor allem, wenn der anschließende Testaufwand groß ist. Ein Anfang wären Unit-Tests, um die technische Funktionalität des Systems sicherzustellen.

Leider sind gerade in den historisch gewachsenen Anwendungen Unit-Tests eine Seltenheit. Umso praktischer wäre es, wenn man in dieser Situation ohne großen Zeitaufwand automatisch Unit-Tests erzeugen könnte. Diese Tests müssten keine gut lesbaren Unit-Tests sein, wie sie ein Entwickler schreiben würde. Es würde schon reichen, vor einem Refactoring per Knopfdruck Unit-Tests zu erzeugen, die man nach erfolgtem Refactoring laufen lassen könnte, um die vorherige Funktionalität sicherzustellen – mit anderen Worten die Generierung von Regressionstests.

Das Prinzip, nach dem man dabei vorgehen kann, ist folgendes: Man hat eine Klasse, die man testen möchte, das System unter Test. Diese ruft man nun im Testfall-Generator auf und merkt sich die Ausgabewerte. Dann erzeugt man mit den gleichen Aufruf-Parametern und den gemerkten Ausgabewerten einen Testfall. Das Ziel ist es, für alle möglichen Pfade des Systems unter Test einen Testfall zu erzeugen; die Schwierigkeit besteht dabei darin, die richtigen Aufruf-Parameter zu finden.

Der Ansatz „Zufallsdaten“

Der erste Ansatz ist ein simpler Code-Generator, der eine Testklasse generiert, in der Methode für Methode des Systems unter Test Testfälle erzeugt werden, in denen die Methoden jeweils einfach mit Zufallsdaten aufgerufen werden, bis die Test-Coverage 100 Prozent beträgt. Leider ist dieses Prinzip sehr aufwendig, was sehr schnell klar wird, wenn man lediglich über eine Methode nachdenkt, die nur einen „int“-Parameter hat und zum Beispiel eine „if“-Abfrage danach, ob dieser „int“-Parameter „23“ ist *(siehe Listing 1)*.

Im Falle „23“ soll etwas anderes getan werden als in allen anderen Fällen. Allein für eine solche Methode müsste der Zufallsansatz im schlechtesten Fall alle möglichen Integerwerte (232 = 4.294.967.296 Möglichkeiten) durchprobieren. Dieser Ansatz lässt sich allerdings durch einfache Maßnahmen verbessern. Eine gute Möglichkeit ist, etwa den Source- oder Byte-Code des Systems unter Test nach Konstanten zu durchsuchen und für diese Konstanten auch Tests zu generieren. Der beschriebene Fall lässt sich so leicht umgehen und auch andere Pfade können so einfach durchlaufen werden. Ein Tool, das tatsächlich JUnit-Tests für eine Klasse generiert und dabei intelligent gewählte Zufallswerte als Basis nimmt, ist Randoop *(siehe „https://randoop.github.io/randoop“).*

Der Ansatz „genetische Algorithmen“

Ein weiterer möglicher Ansatz zum Erzeugen einer möglichst hohen Test-Coverage ist die Betrachtung der Thematik als Suchproblem. Als Beispiel dient wieder eine „if“-Abfrage auf einen „int“-Parameter, allerdings nicht mehr nur danach, ob dieser den Wert „23“ hat, sondern auch, ob er mal 2 genommen den Wert „46“ ergibt *(siehe Listing 2)*.

Dieses Problem lässt sich nicht mehr durch clevere Wahl der Integer-Werte aus dem Bytecode lösen, da weder „2“ noch „46“ als Werte mal 2 genommen „46“ ergeben. Als Suchproblem gilt es also, einen

```
private static Function<Integer,Integer> distance
= x-> Math.abs(x*2-46);

public static int simpleIntegerHillClimbing(){
    int bestValue=Integer.MAX_VALUE;
    int currentValue =0;

    while(distance.apply(currentValue)!=0){
        if(distance.apply(currentValue)<bestValue){
            bestValue=distance.apply(currentValue);
        }
        if(distance.apply(currentValue+1)<bestValue)          {currentValue+=1;}
        if(distance.apply(currentValue-1)<bestValue)          {currentValue-=1;}
    }

    return currentValue;
}
```

Listing 3

```
public class TestTwoValues {
    private static boolean twoParameters(int x, int y){
        if(x==2 && x*y == 46){
            return true;
        }
        return false;
    }
}
```

Listing 4

Parameter zu finden, der die Bedingung „hat multipliziert mit 2 den Wert 23'" erfüllt. Dafür definiert man sich eine Distanzfunktion, die einem den Abstand zu dem Wert liefert, in diesem Fall „!Distanz = |x*2-46|!". Ein einfacher Ansatz wäre, auf diese Funktion „Hillclimbing" (den Bergsteiger-Algorithmus) anzuwenden, um nach einem Wert „X" zu suchen, der die Distanz „0" ergibt, wenn man ihn in die Abstandsfunktion einsetzt *(siehe Listing 3)*.

Im Beispiel handelt es sich um eine sehr einfache Implementierung für das Finden von Integern, die nur funktioniert, wenn es eine Lösung gibt. Für ein Problem wie das Durchlaufen der Abfrage auf den Wert „23" wird man auf diesem Weg relativ schnell eine Lösung finden. Allerdings stößt auch Hillclimbing bei komplizierteren Problemen schnell an seine Grenzen. Ein Beispiel dafür wäre, wenn wir als Bedingung der „if"-Abfrage nicht mehr nur einen Parameter hätten, sondern zwei *(siehe Listing 4)*. In diesem Fall wäre die Distanzfunktion für unser Suchproblem „| x-2 | + | x*y -46 |". Die angepasste Hillclimbing-Funktion für zwei Parameter sieht schon deutlich komplizierter aus *(siehe Listing 5)*. Hinzu kommt, dass diese Funktion in eine Endlosschleife läuft, da Hillclimbing hier an seine Grenzen stößt. Von „[6,6]" aus angefangen sehen die Schritte wie in *Listing 6* aus.

Wie man sieht, wird an der Stelle [[6,7]] ein lokales Optimum erreicht, das bei der Schrittweite im Bereich seiner Nachbarn keinen

```
private static Function<int[], Integer> distance = (int[] parameters) -> Math.abs(parameters[0] - 2) + Math.
abs(parameters[0] * parameters[1] - 46);

public static int[] twoIntegerHillClimbing() {
    int lowestDistance = Integer.MAX_VALUE;
    int[] current ={0,0};

    while (distance.apply(current) != 0) {
        if (distance.apply(current) < lowestDistance) {
            lowestDistance = distance.apply(current);
        }
        int[] bestNeighbor=
     bestNeighbor(getNeighbors(current));
        if (distance.apply(bestNeighbor) < lowestDistance) {
            current=bestNeighbor;
        }
    }

    return current;
}

private static int[][] getNeighbors(int[] current) {
    int x=current[0], y=current[1];
    return new int[][]{{x + 1, y}, {x, y + 1}, {x + 1, y + 1},        {x - 1, y}, {x, y - 1}, {x - 1, y - 1}};
}

private static int[] bestNeighbor(int[][] neighbors) {
    return Arrays.stream(neighbors)
            .min(Comparator.comparing(neighbor ->                    distance.apply(neighbor)))
            .get();
}
```

Listing 5

```
neighbors = [[6, 5], [5, 6], [6, 6], [4, 5], [5, 4], [4, 4]]
bestneighbor = [[6, 6]]
distance = 14
neighbors = [[7, 6], [6, 7], [7, 7], [5, 6], [6, 5], [5, 5]]
bestneighbor = [[6, 7]]
distance = 8
neighbors = [[7, 7], [6, 8], [7, 8], [5, 7], [6, 6], [5, 6]]
bestneighbor = [[6, 8]]
distance = 6
neighbors = [[7, 8], [6, 9], [7, 9], [5, 8], [6, 7], [5, 7]]
bestneighbor = [[6, 7]]
distance = 8
neighbors = [[7, 8], [6, 9], [7, 9], [5, 8], [6, 7], [5, 7]]
bestneighbor = [[6, 7]]
distance = 8 […]
```

Listing 6

```
public static int[] hillClimbing(int[] current) {
    int lowestDistance = Integer.MAX_VALUE;
    int[] bestNeighbor;
    int circles = 0;

    while (circles <= 2) {
        if (distance.apply(current) < lowestDistance) {
            lowestDistance = distance.apply(current);
        }
        if (distance.apply(current) == lowestDistance) {
            circles++;
        }
        bestNeighbor = bestNeighbor(getNeighbors(current));
        if (distance.apply(bestNeighbor) < lowestDistance) {
            current = bestNeighbor;
        }
    }
    return current;
}
```

Listing 7

```
public static void main(String[] args) {
    Random random = new Random();
    int[] bestResult = IntStream.range(0, 100000)
            .boxed()
            .map(i -> new int[]{random.nextInt(1000),
                        random.nextInt(1000)})
            .map(HillClimbing::hillClimbing)
            .min(Comparator.comparing(result ->
                        distance.apply(result)))
            .get();
    System.out.println("bestResult = " +
            Arrays.deepToString(new int[][]{bestResult}));
    System.out.println("distance = " +
            distance.apply(bestResult));
}
```

Listing 8

besseren Wert mehr erreichen kann. Genau an dieser Stelle setzen genetische Algorithmen an. Um den Ansatz zu verstehen, lässt sich Hillclimbing einfach weiterverwenden, jedoch nimmt man für das Problem nicht mehr nur einen Ausgangswert pro Parameter, sondern gleich mehrere, man spricht von einer „Generation". Es gibt also mehrere Einstiegswerte, von denen ausgehend man versucht, eine Lösung zu finden. Ziel der mehreren Einstiegswerte ist es, lokale Optima zu vermeiden, da ja durch das Loslaufen von mehreren Punkten mit hoher Wahrscheinlichkeit von einem der Einstiegspunkte auch das globale Optimum erreicht wird.

Um von vornherein eine Endlosschleife zu vermeiden, zählt man das Wiederauftauchen der gleichen Nachbarn nach einem Optimierungsschritt. Wenn mehr als zwei Mal die gleichen Nachbarn hintereinander auftauchen, befindet man sich entweder an einem lokalen oder aber am globalen Optimum, daher kann man die Bedingung der „while"-Schleife entsprechend anpassen. Danach macht man die Hillclimbing-Funktion parametrisierbar *(siehe Listing 7)*. Aufrufen kann man das Ganze dann beispielsweise wie in *Listing 8*.

In diesem Aufrufbeispiel wird mit 1.000 verschiedenen zufälligen Ausgangswertepaaren nach dem globalen Optimum gesucht und am Ende der beste gefundene Wert ausgegeben. Dieses Beispiel ist ein stark vereinfachter genetischer Algorithmus, der auf ein bestehendes Suchproblem hin optimiert. Am Ende kann man auf diesem Weg anhand einer Distanz-Funktion Werte finden, die einem dabei helfen, in Bedingungen zu laufen, um bei der Erzeugung eine möglichst hohe Coverage zu erhalten. Ein Tool, das diesen Ansatz für die Generierung von JUnit-Tests verwendet, ist EvoSuite *(„siehe „http://www.evosuite.org")*.

Ausprobieren

Nachdem zwei verschiedene Ansätze erläutert sind, stellt man beide auf die Probe. Da das Ziel sein soll, für Legacy-Code echte Regressionstests zu erzeugen, ist als Erstes Legacy-Code erforderlich. Dafür gibt es eine Benchmark-Anwendung *(siehe „https://github.com/ch4inl3ss/legacyExample")*, die eine Klasse enthält, die stark verschachtelt ist und viele verschiedene Dinge in einer einzigen Methode ausführt, also etwas, das den Reflex, Refactoring betreiben zu wollen, direkt auslöst.

Die Methode in der Klasse erledigt drei Dinge, je nachdem, was für ein String ihr in einem Parameter namens „state" übergeben wurde. Diese drei Dinge sind das (ineffiziente) Ausrechnen von Primzahlen, das Aufrufen eines Remote-Service für Wetterdaten und das Speichern von Programmiererinnen und Programmierern in einer Datenbank. Natürlich ließe sich dieser Code gut in drei Teile aufteilen, die jeweils eine der drei Aufgaben übernehmen. Genau für ein solches Refactoring wäre es praktisch, wenn man Regressionstests für den bestehenden Code generieren könnte, um sicherzustellen, dass nach dem Refactoring die Funktionalität erhalten geblieben ist.

	Randoop	evosuite
Laufzeit	112 Sekunden	240 Sekunden
LineCoverage	10 %	66 %
Mutation-Testing (gemessen mit PIT)	31 mutations, Killed 4 (13%)	31 mutations, Killed 24 (77%)
erzeugte Testfälle	4402	7

Abbildung 1: Vergleich zwischen Randoop und EvoSuite

Im nächsten Schritt werden die beiden Tools Randoop und EvoSuite auf diese Methode als Benchmark losgelassen, um genau das zu tun und dann anschließend die Ergebnisse zu vergleichen. Bei Randoop werden für die eine Methode um die 4.000 Testfälle erzeugt, die Anzahl variiert etwas. Die erreichte Line-Coverage liegt bei etwa zehn Prozent. Dafür braucht Randoop etwa zwei Minuten Laufzeit; EvoSuite erzeugt für die gleiche Methode sieben Testfälle. Diese erreichen eine Line-Coverage von 66 Prozent. Die von EvoSuite benannte Dauer für die Generierung dieser Tests liegt bei etwa vier Minuten.

Die Line-Coverage der erzeugten Testfälle sagt allerdings noch nichts über deren Qualität aus. Selbst wenn die Testfälle alle Pfade des Codes durchlaufen, ist nicht sichergestellt, dass die Prüfungen in den Testfällen auch eine echte Aussage haben. Trotz 100 Prozent Line-Coverage sind ohne sinnvolle Asserts die Testfälle im Grunde wertlos.

Um herauszufinden, ob die Assertions in den Testfällen eine sinnvolle Aussage haben, kann man Mutation-Testing verwenden. Dabei werden vom eingesetzten Tool Mutationen im System unter Test erzeugt und anschließend gemessen, wie viele dieser Mutationen dazu führen, dass Testfälle tatsächlich fehlschlagen. Nur wenn ein Testfall fehlschlägt, gilt die Mutation als eliminiert. Je mehr Mutationen die Testfälle eliminieren, desto aussagekräftiger sind die Assertions in den Tests. Als Tool für Mutation-Testing kam Pit *(siehe „http://pitest.org")* zum Einsatz, um die anhand des Benchmarks generierten Testfälle zu messen. Dabei hat Pit jeweils 31 Mutationen im Ausgangscode erzeugt. Davon konnten die von Randoop generierten Tests vier Mutationen eliminieren (13 Prozent), die Tests von EvoSuite eliminierten 24 Mutationen (77 Prozent, siehe *Abbildung 1)*.

Fazit

Für die Generierung von Regressionstests ist Randoop derzeit nicht besonders gut; es eignet sich deutlich besser, um beispielsweise Fehler im eigenen Code zu finden. Auch ist der Praxiseinsatz von Randoop eher schwierig. Es gibt zwar ein Maven-Plug-in, dieses scheint aber nicht mehr zu funktionieren. EvoSuite eignet sich tatsächlich als Tool für den praktischen Einsatz. Es gibt ein funktionierendes Maven-Plug-in und auch die erzeugten Testfälle haben eine relativ hohe Abdeckung und Aussagekraft. Vor einem möglichen Refactoring, bei dem keine Tests zur Verfügung stehen, sollte man in jedem Fall über die Generierung von Tests mit EvoSuite nachdenken. Neben den beiden im Artikel betrachteten Ansätzen und Tools gibt es durchaus noch weitere, etwa den Ansatz, einen Recorder während der Benutzung der Anwendung laufen zu lassen, um Daten für die Generierung von Testfällen zu erzeugen. Die Code-Ausschnitte für Hillclimbing sind unter *„https://github.com/ch4inl3ss/hillclimbing-Examples"* zu finden.

Felix Schumacher
felix.schumacher@innoq.com

Felix Schumacher ist Consultant bei der innoQ Deutschland GmbH. Seine Schwerpunkte liegen in den Bereichen „Clean Code", „Tests" und „Umgang mit Legacy-Code". Vor seiner jetzigen Tätigkeit hat er in großen Projekten für eine Versicherung gearbeitet und dort viel Erfahrung mit Legacy-Software gesammelt.

Java-Performance-Analyse mit YourKit

Karsten Thoms, itemis AG

Wer die Performance von Anwendungen genauer untersuchen und optimieren muss, kommt um ein Profiling der Anwendung nicht herum. Es nützt nichts, Mutmaßungen über vermeintliche Schwachstellen anzustellen oder Code vorweg zu optimieren, denn oft liegen Potenziale zur Optimierung an ganz anderer Stelle.

Zum Profiling sind geeignete Werkzeuge erforderlich. Mit dem JDK wird zwar mit JVisualVM *[1]* ein kostenfreies Werkzeug mitgeliefert, doch es lohnt sich an dieser Stelle, den Blick auch auf kommerzielle Alternativen zu werfen. Dieser Artikel nimmt das Werkzeug YourKit Java Profiler *[2]* genauer unter die Lupe und zeigt, wie man es zur Analyse von Performance-Problemen einsetzt.

YourKit im Überblick

Das seit dem Jahr 2003 am Markt erhältliche Profiling-Werkzeug YourKit liegt aktuell in der Version 2018.04 vor. Neben JProfiler *[3]* und JVisualVM ist es das am meisten verwendete Werkzeug in dieser Kategorie. Auf der Startmaske besteht zunächst einmal die Möglichkeit, sich mit laufenden Java-Prozessen zu verbinden. Im Bereich „Monitor Applications" sind die lokal laufenden Prozesse aufgelistet. Meist erfolgt ein Profiling für lokale Programme, doch bei Bedarf kann YourKit auch mit Remote-Prozessen verbunden werden, dazu gehören auch Prozesse in Application-Servern oder innerhalb von Docker-Containern *(siehe Abbildung 1)*.

Obwohl nicht zwingend notwendig, wird empfohlen, zu untersuchende Prozesse mit einem speziellen Java-Agent zu starten, der mit YourKit geliefert wird. Wenn YourKit Bytecode instrumentalisieren muss, werden damit Wartezeiten mit dem Agent drastisch reduziert, da dieser die Instrumentalisierung bereits beim Laden der Klassen vornimmt. In der Übersicht „Monitor Applications" ist zu sehen, welche Prozesse mit dem Agent gestartet wurden (grün) und welche nicht (orange).

CPU-Profiling

Hat man sich mit einem laufenden Prozess verbunden, kann das Profiling beginnen. Die meistverwendete Methode für die Untersuchung der Performance ist das CPU-Profiling. Für diese Methode stehen drei unterschiedliche Modi zur Verfügung *(siehe Abbildung 2)*, die sich in ihrem Informationsgehalt und Overhead unterscheiden:

- **Sampling**
  Die Ausführungszeit von Methoden wird mit geringem Overhead abgeschätzt und zusätzlich werden Stichproben von Stack-Frames erhoben.
- **Tracing**
  Die Ausführungszeit von Methoden wird exakt gemessen und zusätzlich die Anzahl der Methodenaufrufe gezählt. Dieser Mehrwert geht mit einem höheren Overhead einher. Aus diesem Grund ist es empfehlenswert, Tracing nur mit Prozessen zu nutzen, die mit dem Java-Agent gestartet wurden.
- **Call Counting**
  In diesem Modus wird die Anzahl von Methoden-Aufrufen ohne Ausführungszeiten erfasst.

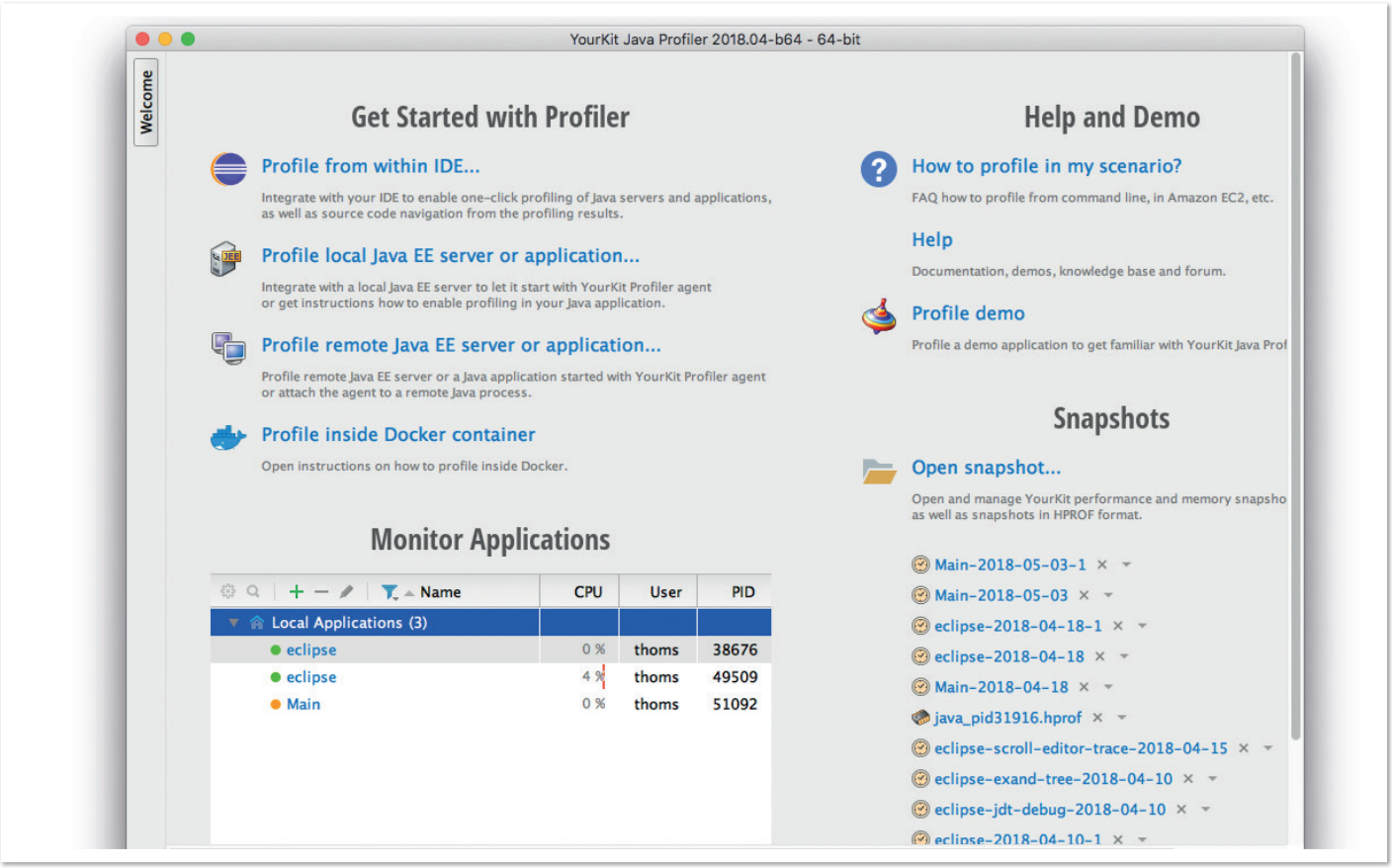


Abbildung 1: Die YourKit-2018.04-Startmaske

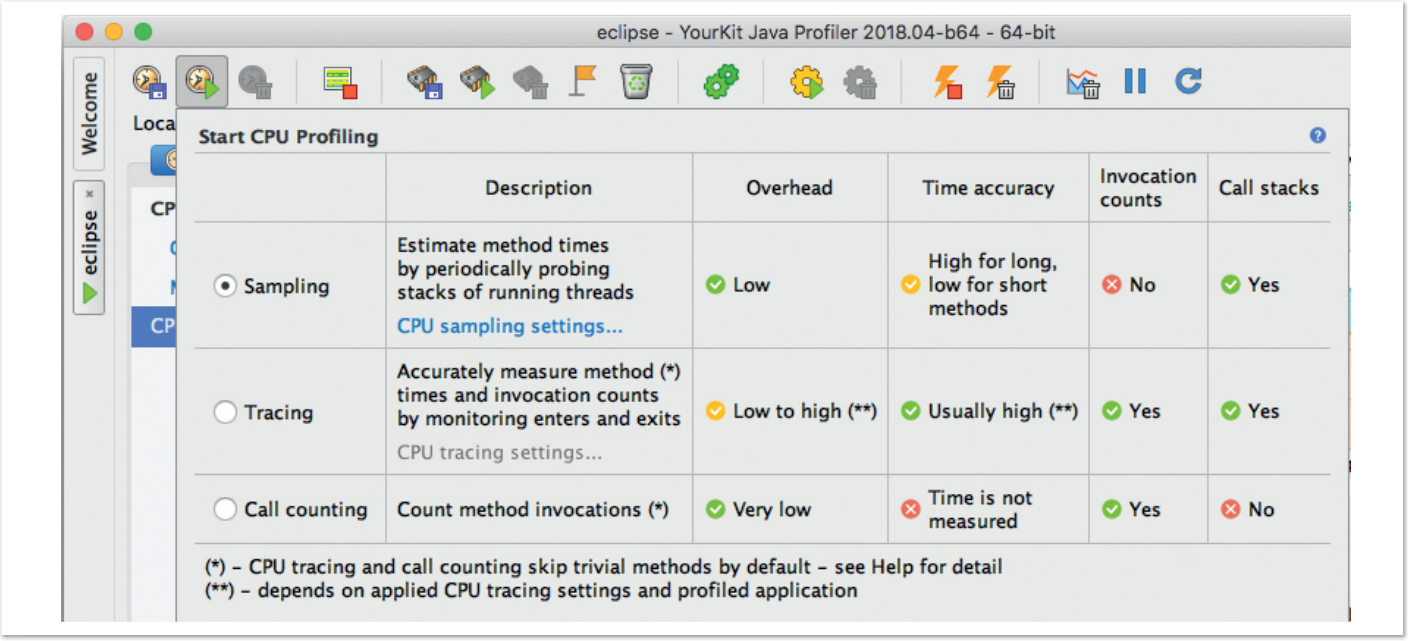


Abbildung 2: Das CPU-Profiling starten

In der Praxis bewährt es sich, mit dem Sampling zu beginnen. Die dadurch gewonnenen Erkenntnisse helfen, die möglichen Hotspots und die Auswirkung von Änderungen zu untersuchen. Das Starten des Profilings erfolgt ohne Verzögerung und die zu untersuchende Anwendung wird während des Profilings kaum spürbar ausgebremst. Hier fehlt jedoch die Information darüber, wie häufig Methoden aufgerufen werden. Eine Methode kann insgesamt viel Zeit benötigen, wenn sie sehr häufig aufgerufen wird oder weil sie selbst sehr viel Zeit für die Berechnung braucht. Für die Performance-Analyse einer Anwendung zählen die Dauer und die Häufigkeit für die Ausführung einzelner Methoden.

Für die erweiterte Analyse bietet sich das Tracing an. Die Ausführung der Anwendung wird damit deutlich ausgebremst, dafür werden wertvolle zusätzliche und genauere Informationen gesammelt. Der „Call counting"-Modus kommt eher selten zum Einsatz, da die

Anzahl von Methoden-Aufrufen allein wenig Rückschlüsse darauf bietet, ob dies zu einer Beeinträchtigung der Performance führt.

Das CPU-Profiling sollte kontrolliert während eines konkreten Anwendungsszenarios eingesetzt werden, damit die Messungen möglichst wenig durch andere Aktionen beeinflusst sind. YourKit bietet Live-Daten, während die Anwendung mit dem Profiling ausgeführt wird. Diese besitzen jedoch eingeschränkte Aussagekraft und lassen sich lediglich als Orientierungshilfe einsetzen, um einen ersten Eindruck darüber zu gewinnen, welche Funktionen in der Anwendung gerade ausgeführt werden. Wenn das Szenario abgeschlossen ist, wird über den Speichern-Button ein Snapshot gespeichert, der anschließend zur Analyse geöffnet wird.

Analyse von Hot Spots und Methoden

Die Ansicht „Hot spots" gibt einen Anhaltspunkt, welche Methoden während des Profilings besonders auffällig sind *(siehe Abbildung 3)*. Die darin aufgelisteten Methoden müssen nicht zwingend ein Problem aufzeigen, bieten aber Grund für eine nähere Inspektion. Die vollständige Ansicht ist wiederum in der Ansicht „Method list" zu sehen *(siehe Abbildung 4)*.

Während die Ansicht „Hot spots" nur die Zeiten anzeigt, die für die Ausführung der Methoden insgesamt verbraucht wurden, ist in der Ansicht „Method list" auch die Zeit zu sehen, die in der Methode selbst verbraucht wurde, unabhängig von den aufgerufenen Methoden. Das kann nützlich sein, um Potenziale zur Optimierung innerhalb von Methoden zu erkennen. Es bietet sich an, diese dann nach Zeiten zu sortieren und top down nach möglichen Kandidaten zu suchen, bei denen sich eine genauere Analyse lohnt.

Sehr hilfreich ist der untere Bereich der Anzeige, in dem zu sehen ist, von wo aus eine Methode aufgerufen wird („Back Traces") oder welche Methoden von ihr aufgerufen werden („Callees") und wie viel Zeit an jedem dieser Knoten jeweils verbraucht wurde. Diese Aufschlüsselung hilft bei der Analyse auffälliger Methoden.

Es ist auch nützlich, nach Methoden-Namen zu filtern. Dazu gibt man nach Anklicken des Such-Icons einen Teil des gesuchten Methoden-Namens ein und bekommt daraufhin eine gefilterte Ansicht angezeigt.

Bei der Betrachtung von Ausführungszeiten kommt es häufig vor, dass Methoden, auf die man keinen Einfluss hat oder deren Zeiten plausibel und nicht optimierbar sind, sehr prominent in den Messwerten erscheinen. Andere Methoden treten dadurch in den Hintergrund, obwohl sie vielleicht mehr Potenzial für Optimierungen bieten.

Über das Kontextmenü lassen sich einzelne Methoden aus der Analyse ausklammern („Exclude Method") oder genauer untersuchen („Focus on Method"). Es öffnet sich dann ein weiterer Reiter mit dem Titel „CPU (What-if)". Damit lassen sich Methoden isoliert betrachten, um weitere Erkenntnisse für die Performance-Analyse zu sammeln.

Threads betrachten

Die „Threads"-Ansicht zeigt die Threads der Anwendung in dem untersuchten Zeitabschnitt an. Wird ein Thread angeklickt, so stehen im unteren Teil der Ansicht der Stack-Trace des selektierten Thread (oder aller Threads) zu dem Zeitpunkt *(siehe Abbildung 5)*.

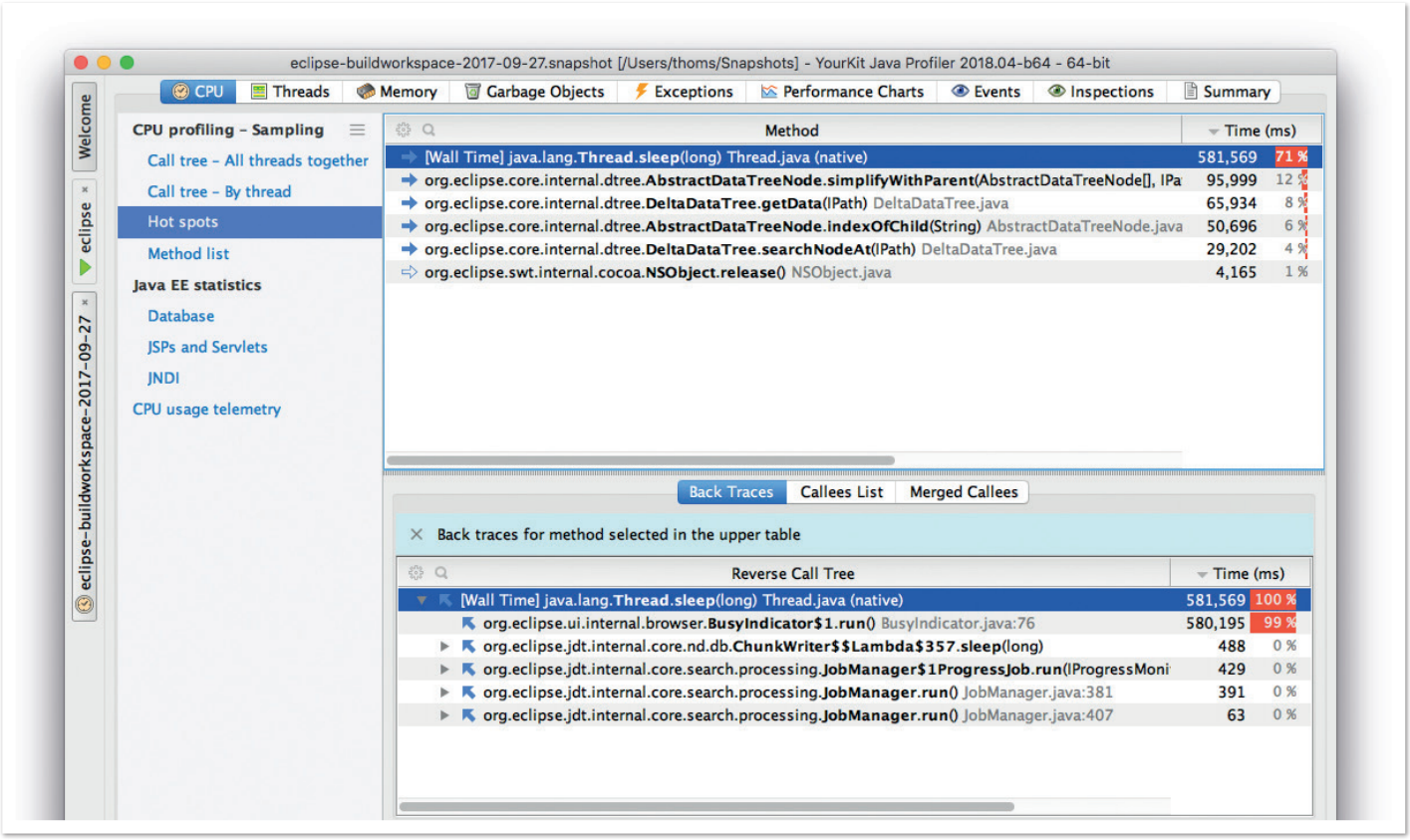


Abbildung 3: Die „Hot spots"-Ansicht

Die farbliche Kennzeichnung stellt den Status der einzelnen Threads dar; Grün sind Zeiten angezeigt, in denen ein Thread Code ausführt; Gelb bedeutet, dass ein Thread wartet; Rot zeigt, zu welchen Zeiten ein Thread blockiert wird. Ausschließlich anhand der Farben ist die Bewertung der Analyse allerdings nicht möglich. Auffällig vermehrt blockierte Threads wie in *Abbildung 5* sind ein Indiz dafür, dass eine weitere Untersuchung erforderlich ist. Die Abbildung zeigt ein Szenario, bei dem es zu stark parallelem Aufruf von synchronisierten Methoden kommt. Durch Anklicken eines Thread auf einen blockierendem Abschnitt kann im Stack-Trace darunter die kritische Methode identifiziert werden.

Über die Optionen auf der linken Seite lassen sich nützliche Zusatzinformationen einblenden, etwa Zeiten, zu denen Dateizugriffe stattfinden inklusive Hover-Informationen darüber, um welche Dateien es sich handelt. Außerdem lässt sich über den Methodenfilter die Ansicht eingrenzen, wann auf welchen Threads bestimmte Methoden ausgeführt werden. Ähnliches gilt für andere Event-Arten wie zum Beispiel die Ausführung von SQL-Statements.

Performance Charts

Die Ansicht „Performance Charts" bietet einen Überblick der Systemzustände über den jeweiligen Zeitverlauf. Für die Analyse sind dabei vor allem die CPU-Auslastung, Speicherverbrauch und Garbage Collection von Interesse. I/O-Operationen und Datenbank-Zugriffe runden die Möglichkeiten der Analyse ab. Wie auch in anderen Ansichten lassen sich die Stack-Traces zu den gewählten Zeitpunkten im unteren Bereich anzeigen *(siehe Abbildung 6)*.

Über das Kontextmenü können ein Maßstab für die Zeitachse festgelegt (Sekunden-, 1-, 5-, 10-, 60-Minuten-Abschnitte) und die Methoden oder Threads auf der horizontalen Achse nach Uptime oder Uhrzeit beschriftet werden. Auch über diese Optionen lassen sich andere Zusammenhänge erkennen.

Events anzeigen

In der Ansicht „Events" werden System-Ereignisse wie I/Os oder Datenbank-Zugriffe angezeigt. Die Events lassen sich auf unterschiedliche Weise sortieren:

- Die „Events by Table"-Ansicht zeigt je nach Sortierung der Tabelle die zeitliche Reihenfolge der Ereignisse oder die verbrauchte Zeit an.
- Die „Event Timeline"-Ansicht (siehe Abbildung 7) ermöglicht beispielsweise die Anzeige, auf welche Dateien wann zugegriffen wird. Mit entsprechender Sortierung lässt sich auch ermitteln, auf welche Dateien häufig zugegriffen wird.
- Über den „Event Call Tree" kann nachvollzogen werden, welche Methoden (nach Threads gruppiert) wie häufig bestimmte Arten von Events auslösen.
- Die „CPU Usage Estimation" wird über das Kontextmenü der anderen drei Ansichten geöffnet und zeigt die benötigte Zeit für selektierte Events an.

Wie auch in den anderen Ansichten kann man die Einträge filtern oder Stack-Traces zu den ausgewählten Ereignissen einsehen.

Memory Profiling

Neben der Analyse der Ausführungsgeschwindigkeit lässt sich das Verhalten der Anwendung hinsichtlich der Speichernutzung untersuchen – einerseits im Hinblick auf den Speicherbedarf der Objekte, andererseits kann das Verhalten der Garbage Collection und Allo-

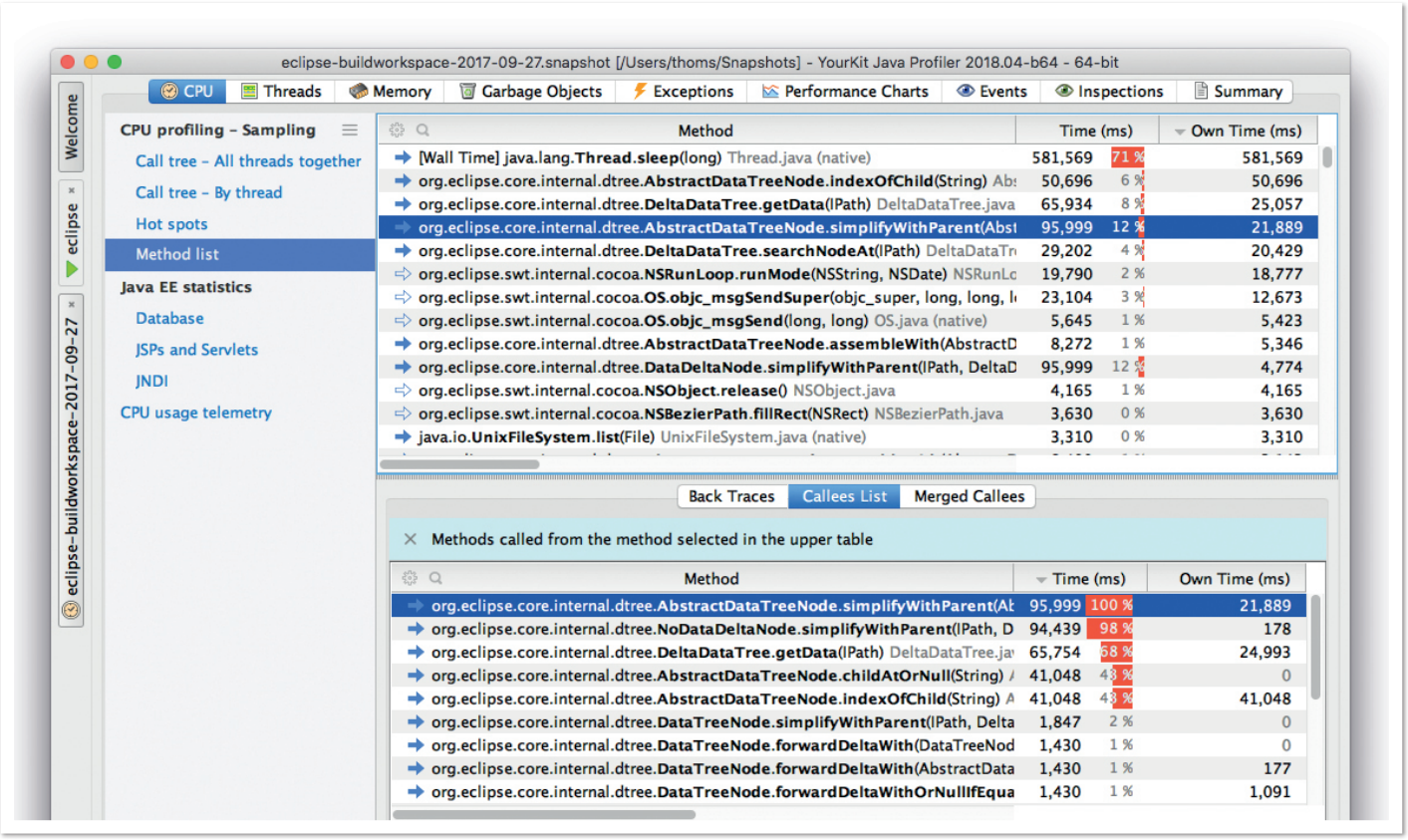


Abbildung 4: Die „Method list"-Ansicht

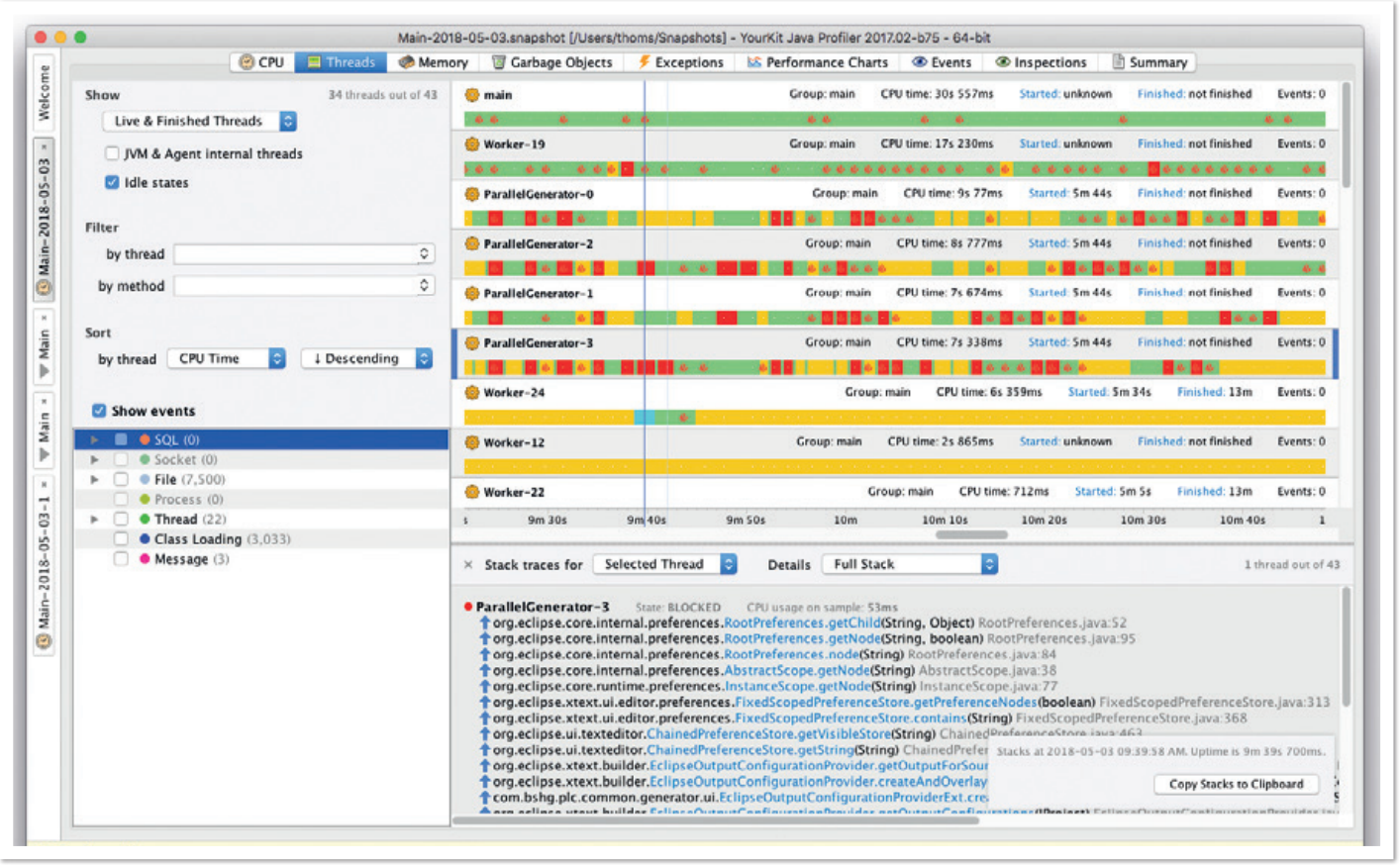


Abbildung 5: „Threads"-Ansicht

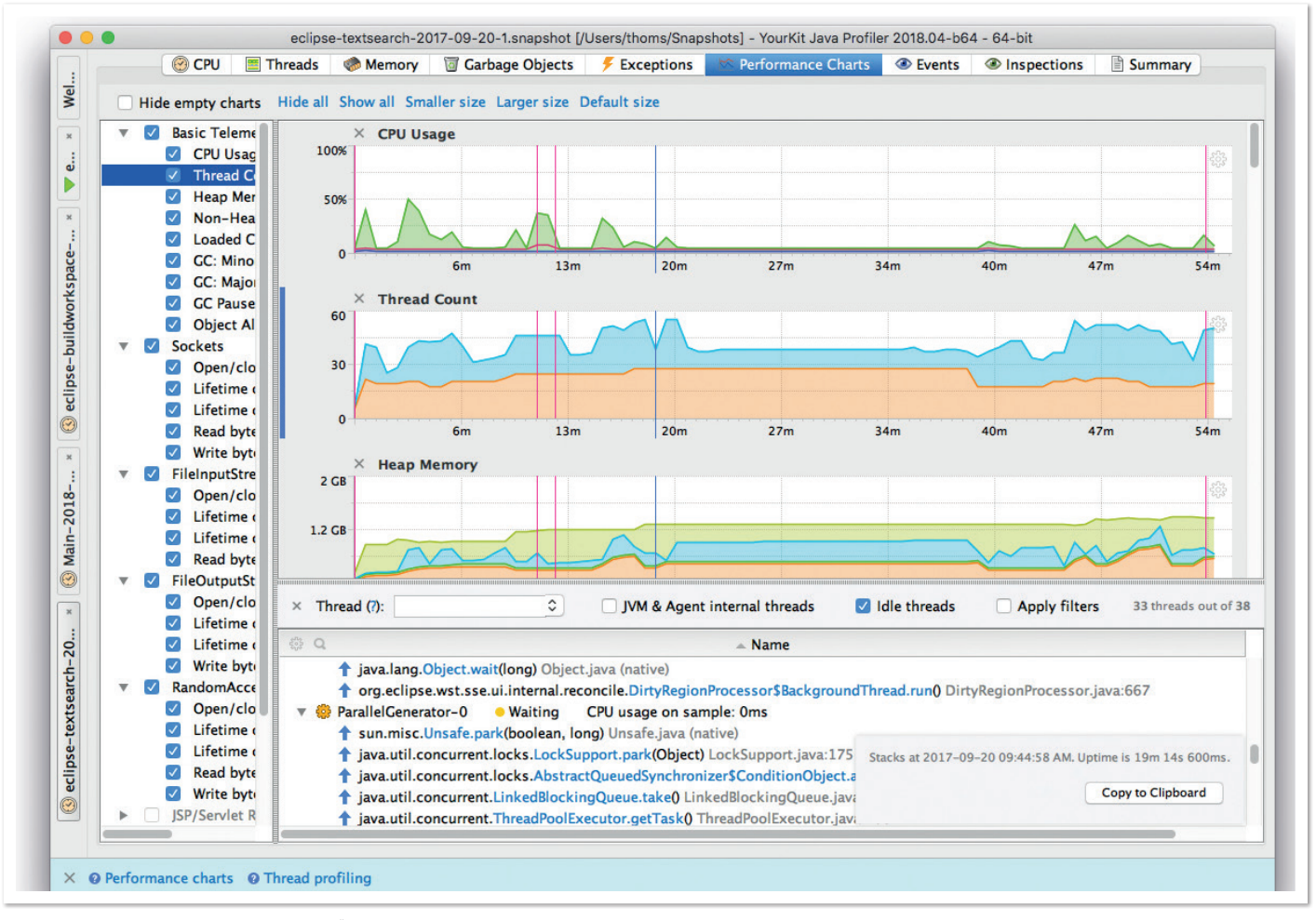


Abbildung 6: Die „Performance Charts"-Ansicht

kation der Objekte zur Laufzeit von Interesse sein. Letzteres wirkt sich auch auf die Ausführungsgeschwindigkeit aus, denn auch die exzessive Erzeugung von kurzlebigen Objekten kann zu unnötig verbrauchter Zeit für den Garbage Collector führen. Zunächst wird allerdings nur die statische Speicher-Analyse betrachtet.

In einem Memory-Snapshot lässt sich ablesen, wie viele Instanzen einer Klasse sich zum Zeitpunkt eines Snapshots im Speicher befinden und wie hoch ihr Speicherbedarf ist. Ein solcher Memory-Snapshot lässt sich über das Toolbar-Icon „Create Memory Snapshot" anfertigen. Alternativ kann man Memory Dumps der JVM auch mit dem Dateiformat „.hprof" öffnen.

Memory-Snapshots sind ungleich größer als CPU-Profiling-Snapshots und können leicht einige Hundert MB erreichen. Es bietet sich oft an, vor der Erhebung eines solchen Snapshots eine Garbage Collection zu erzwingen, was über ein Toolbar-Icon ausgeführt wird, denn oft ist man eher an Objekten interessiert, die fest im Heap-Speicher gehalten werden, als an solchen, die bei der nächsten Garbage Collection wieder verschwinden würden.

Shallow & Retained Size

In den verschiedenen Ansichten zur Analyse von Objekten sind jeweils zwei Werte angezeigt: Bei der „Shallow Size" handelt es sich um die Größe eines Objekts ohne referenzierte Objekte; die „Retained Size" gibt die Größe eines Objekts inklusive der transitiven Hülle aller referenzierte Objekte an, die ausschließlich von dem Objekt erreicht werden und damit existenzabhängig sind.

Auch hier bietet es sich an, zunächst einen Blick auf die Klassen zu werfen, die den meisten Speicher verbrauchen. Auffälligkeiten bei der „Shallow Size" geben zunächst einen Hinweis darauf, ob es Möglichkeiten gibt, die Anzahl der Objekte zu reduzieren, oder ob die Klassen anders geschnitten werden können, um weniger Speicher zu verbrauchen. In Einzelfällen kann es sein, dass eine Vielzahl der Objekte nur wenige Felder füllt. Speicher wird auch für die ungenutzten Felder benötigt. Durch eine geschickte Wahl von Entwurfsmustern (wie Fliegengewicht) können häufig verwandte Felder gebündelt und nur bei Bedarf die Variante mit mehr Feldern verwendet werden.

Deutlich häufiger wird man aber einen Blick auf Klassen werfen, die viel Speicherverbrauch in der „Retained Size" aufweisen. Hier stellt sich die Frage, warum die Objekte so groß sind und wo sie gehalten werden. Mitunter kommt man durch diese Vorgehensweise auch Speicherlecks auf die Spur.

Object Explorer

Ein wichtiges Werkzeug für die Analyse des Speichers ist der „Object Explorer", in dem sich Objekte inhaltlich untersuchen lassen. Neben den Feldwerten wird bei Collections auch deren Größe direkt angezeigt. Die Möglichkeit, die Größen zu sehen und entsprechend zu sortieren, ist für die Bewertung relevant *(siehe Abbildung 8)*.

Zudem lässt sich im unteren Bereich der Ansicht sehen, durch welchen Pfad ein Objekt vom GC Root erreichbar ist (Reiter „Paths from GC Roots"). Aus der Erreichbarkeit der Objekte kann nachvollzogen werden, welche Referenzen auf das Objekt verhindern, durch den Garbage Collector eingesammelt zu werden.

In der Ansicht „Inspections" *(siehe Abbildung 9)* lassen sich Auswertungen anstoßen, die anschließend Hinweise auf mögliche Speicherverschwendung geben, beispielsweise durch redundante Strings, inhaltlich gleiche Objekte, leere oder spärlich gefüllte Arrays, Maps etc. Inwiefern es sich lohnt, hier Optimierungen durchzuführen, hängt von den Anforderungen an die Anwendung und dem ausgewiesenen Einsparpotenzial („Waste") ab. Um die aufgezeigten Objekte weiter bewerten zu können, kann von den Ansichten über das Kontextmenü wieder zum „Object Explorer" navigiert werden.

Nur ein Werkzeug

YourKit kann ungemein nützlich sein, um Performance-Engpässe und Speicherverbrauch darzustellen; die Probleme einer Anwendung analysieren kann es allerdings nicht. Die Interpretation der Daten bleibt dem Benutzer überlassen und es bedarf viel Kenntnis über die untersuchte Anwendung, um Potenziale erkennen und ausschöpfen zu können.

Die Untersuchung und Verbesserung der Performance mit Profilern (an dieser Stelle ist es irrelevant, welches Werkzeug verwendet wird) kann man mit Eisbergen vergleichen: Es ist immer nur die Spitze zu sehen. Wird diese gekappt, kommen weitere Engpässe als neue Spitzen ans Licht. Bei der Verbesserung der Anwendung ist auch wichtig, nicht zu viel auf einmal zu ändern, sondern ganz gezielt zunächst ein Finding herauszupicken, dann dieses zu optimieren und wieder zu messen. Nur so lässt sich der Effekt einzelner Veränderungen quantifizieren.

In der Praxis

Profiling-Werkzeuge wie YourKit sind nicht ständig im Einsatz und nicht jeder Entwickler muss gleich zum Performance-Engineer werden. Wann es sich lohnt, ein Augenmerk darauf zu werfen, ob ein Tuning sinnvoll ist, hängt von der Art und Nutzung der zu untersuchenden Anwendung ab.

Eine Art von Anwendung, die sich für eine kontinuierliche Optimierung der Performance lohnt, sind etwa Developer Workbenches beziehungsweise IDEs. Hier bestehen viele Potenziale für Verbesserungen, denn solche Anwendungen werden millionenfach benutzt und jede eingesparte Millisekunde trägt zur Produktivität, Zufriedenheit und Kostensenkung bei. So wurden etwa für das aktuelle Release von Eclipse Photon Untersuchungen mit YourKit durchgeführt, um aufwendige Prozesse zu optimieren und den Speicherbedarf zu reduzieren (die im Artikel gezeigten Abbildungen stammen aus diesen Untersuchungen). Als Ergebnis konnte so die Eclipse IDE schneller und speicherschonender als zuvor gemacht werden.

Auch für kleine Anwendungen lässt sich YourKit gut einsetzen, um etwa IoT-Anwendungen zu optimieren. Diese müssen mit sehr limitierten Ressourcen hinsichtlich Prozessorleistung und zur Verfügung stehendem Arbeitsspeicher auskommen und auch hier zahlen sich bereits kleine Verbesserungen aus.

Ganz anderer Natur sind wiederum klassische Enterprise-Anwendungen. Hier kann es einerseits nötig sein, komplexe Abläufe zeitlich zu optimieren, andererseits sind diese oft sehr speicherhungrig. Das allein ist nicht einmal ein größeres Problem, da Speicher relativ günstig ist. Aber es gibt durchaus Szenarien, gerade im Umgang mit Big Data, in denen die Verarbeitung der Daten bei

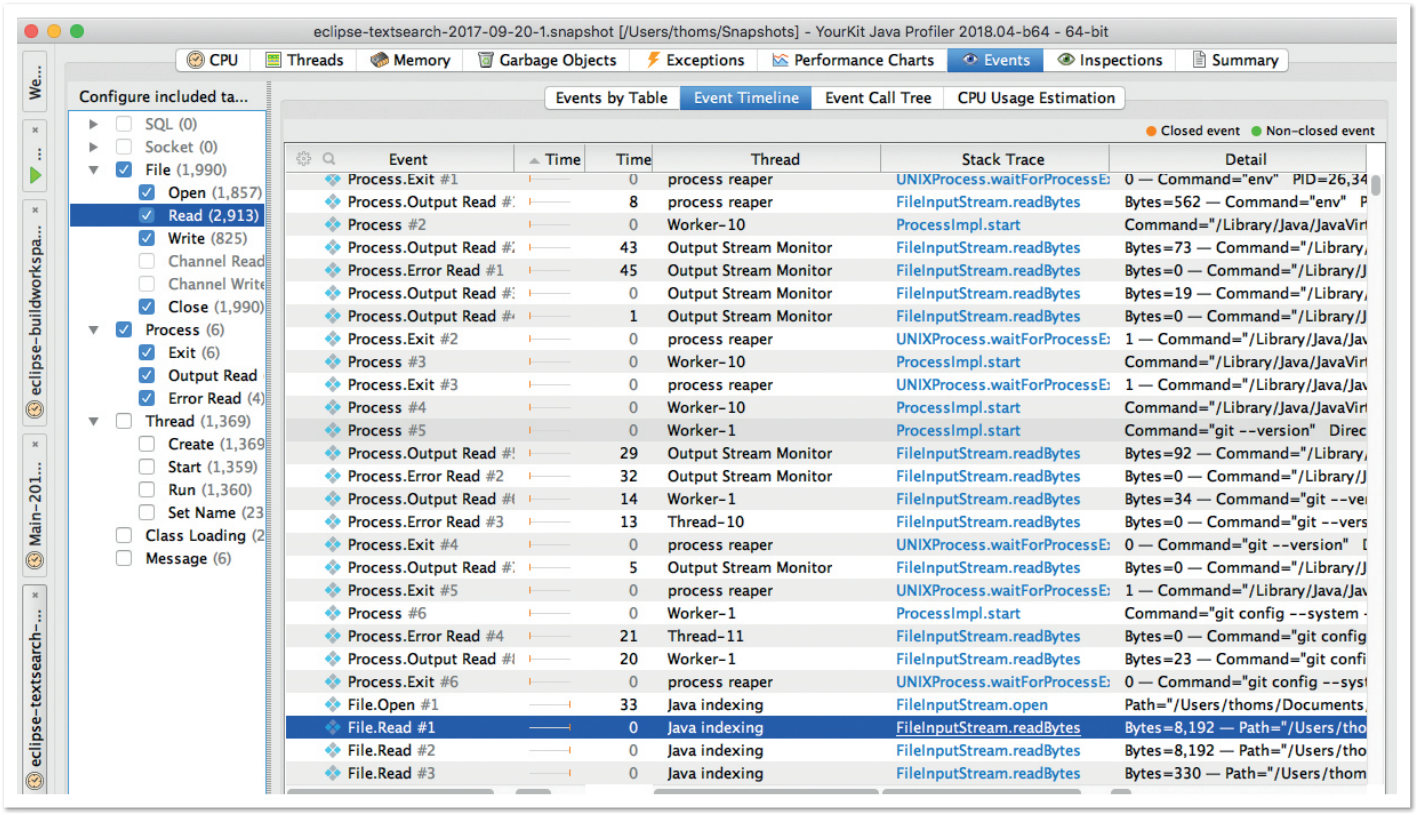


Abbildung 7: Die „Events"-Ansicht

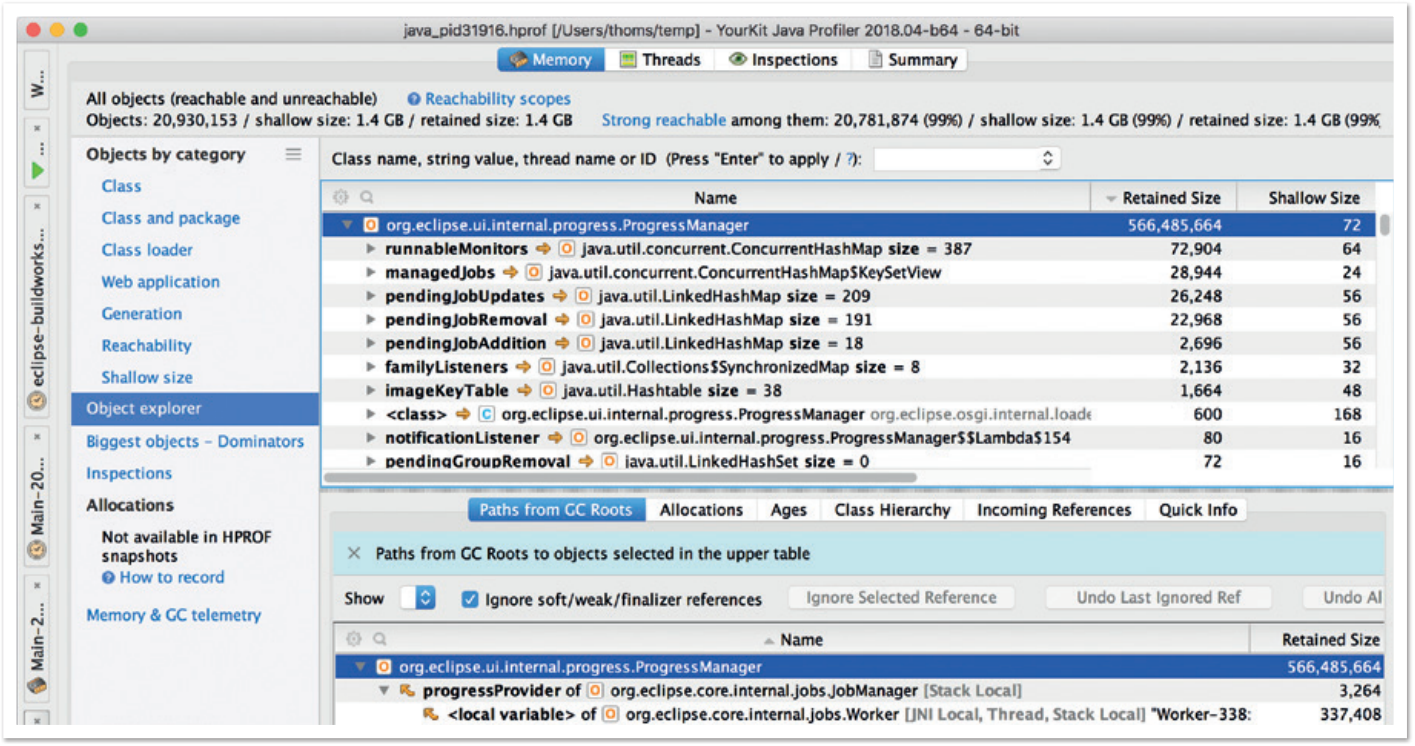


Abbildung 8: Der Object Explorer

unsauberem Umgang zu größeren Einflüssen der Garbage Collection führt oder häufige beziehungsweise langsame SQL-Statements zur Verlangsamung der untersuchten Anwendung führen. Die Analyse-Möglichkeiten mit YourKit geben hier gute Hinweise darauf, an welcher Stelle unnötig Garbage erzeugt wird, welche Art von Datenstrukturen optimiert werden können und welche SQL-Statements viel Zeit benötigen.

Etwas schwieriger sieht es bei Microservices aus. Hier läuft eine Anwendung in einer Vielzahl einzelner Prozesse verteilt auf unterschiedlichen Knoten. Mit YourKit kann man allerdings nur einzelne Prozesse untersuchen und Potenziale lassen sich nur dann erkennen, wenn bestimmte Funktionen unter Last zu auffälligen Messwerten führen. Für diesen Fall eignen sich andere Werkzeuge zur Performance-Analyse besser.

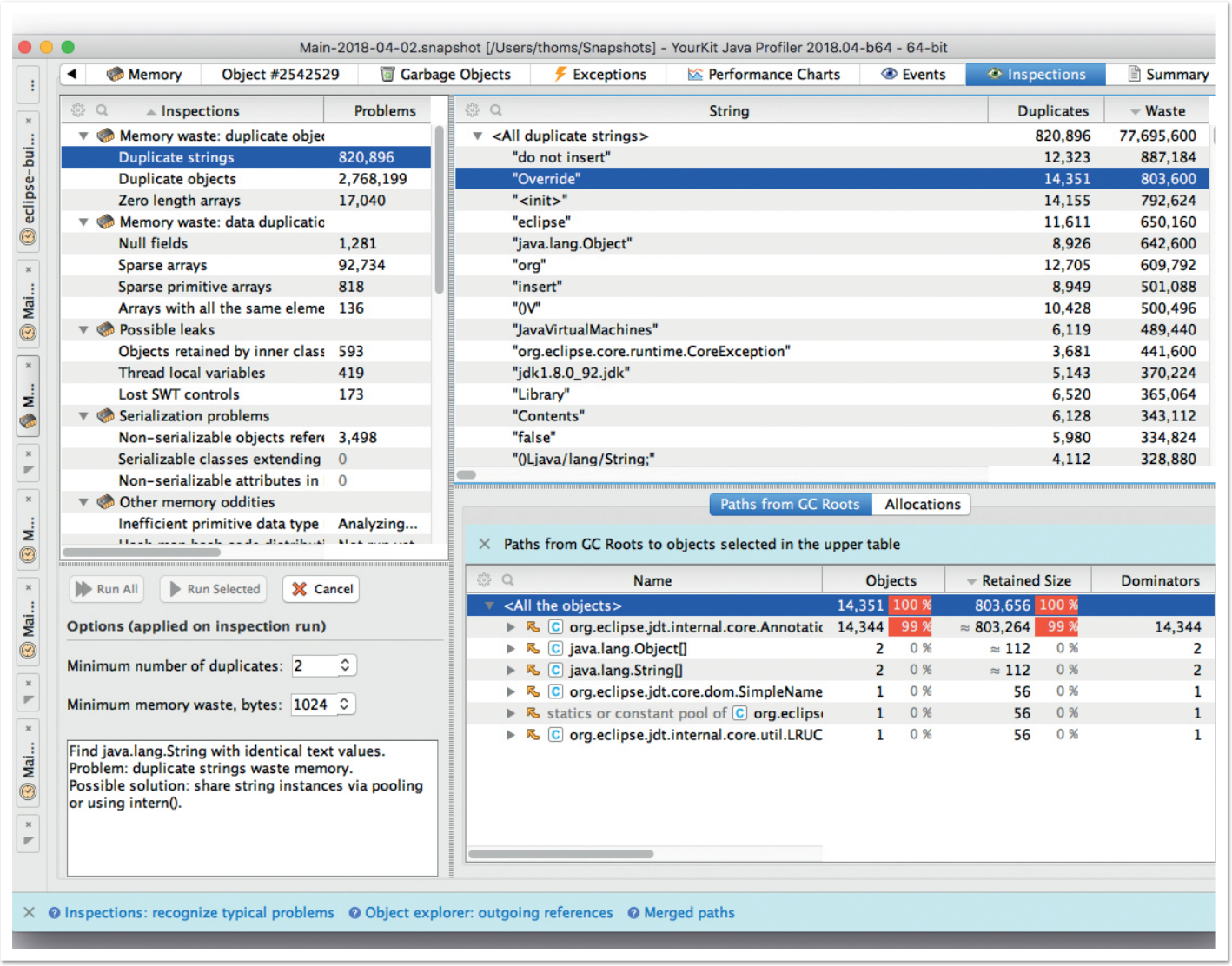


Abbildung 9: „Inspections"-Ansicht

Fazit

YourKit ist ein ausgereiftes Werkzeug zum Profiling von Anwendungen, das sehr leichtgewichtig ist und auch nebenläufig im Entwicklungsprozess eingesetzt werden kann. Die damit erhobenen Messungen lassen sich auf vielfältige Art untersuchen. Besonders nützlich sind dabei die verschiedenen Möglichkeiten zur Sortierung, Filterung und Navigation der Daten.

Das Werkzeug kann keine Probleme automatisch aufdecken. Es gibt allerdings sehr gute Hinweise, die interpretiert werden können, um mögliche Schwachstellen zu identifizieren.

YourKit ist kommerziell; preislich startet das Werkzeug bei einer Einzelplatz-Lizenz aktuell bei 459 Euro. Diese Anschaffung kann sich durchaus lohnen, denn mithilfe dieses Werkzeugs können schnell Kosten gespart werden – insbesondere wenn Performance-Optimierungen umgesetzt werden sollen und andere kostenfreie Alternativen nicht oder langsamer zum Ziel führen.

Weiterführende Links

[1] *https://visualvm.github.io*
[2] *https://www.yourkit.com*
[3] *https://www.ej-technologies.com/products/jprofiler/overview.html*

Karsten Thoms
karsten.thoms@itemis.de

Karsten Thoms arbeitet seit mehr als fünfzehn Jahren als IT-Consultant und ist Experte in den Bereichen „Domänenspezifische Sprachen" (DSL), „Code-Generatoren", „Modellgetriebene Entwicklung" und „Workbenches". Seit einigen Jahren ist er aktiver Eclipse Committer und gehört zum Kernteam des Eclipse-Xtext-Projekts sowie der Eclipse IDE. Seine Kunden berät und unterstützt er bei dem Entwurf, der Entwicklung und Integration von DSLs; darüber hinaus bietet er Professional-Support-Services an.

iJUG Verbund

Alle Mitglieder auf einen Blick

Der iJUG möchte alle Java-Usergroups unter einem Dach vereinen. So können sich alle Java-Usergroups in Deutschland, Österreich und der Schweiz, die sich für den Verbund interessieren und ihm beitreten möchten, gerne unter office@ijug.eu melden.

www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, www.ijug.eu) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Stefan Kinnen. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
Chefredakteur (ViSdP): Wolfgang Taschner
Kontakt: redaktion@doag.org

Redaktionsbeirat:
Andreas Badelt, Melanie Feldmann, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

Titel, Gestaltung und Satz:
Caroline Sengpiel,
DOAG Dienstleistungen GmbH

Fotonachweis:
Titel: Original © nearbirds / *123RF.com*
S. 8: © ahasoft2000 / *123RF.com*
S. 13: © Maxim Evseev / *123RF.com*
S. 19: © totallyout / *123RF.com*
S. 25: © i3d / *123RF.com*
S. 33: © dolgachov / *123RF.com*
S. 40: © Logo, NovaTec Consulting GmbH
S. 44: © bee32 / *123RF.com*
S. 50: © sylverarts / *123RF.com*
S. 53: © yongheng1996 / *123RF.com*
S. 58: © vectora / *123RF.com*

Anzeigen:
Simone Fischer, DOAG Dienstleistungen GmbH
Kontakt: anzeigen@doag.org

Mediadaten und Preise unter:
www.doag.org/go/mediadaten

Druck:
adame Advertising and Media GmbH,
www.adame.de

Inserentenverzeichnis


ORAWORLD
Das e-Magazine für alle Oracle-Anwender!
EOUC
EMEA ORACLE USERGROUP COMMUNITY
• Spannende Geschichten aus der Oracle-Welt
• Technologische Hintergrundartikel
• Leben und Arbeiten heute und morgen
• Einblicke in andere User Groups weltweit
• Neues (und Altes) aus der Welt der Nerds
• Comics, Fun Facts und Infografiken
Jetzt Artikel einreichen oder Thema vorschlagen!
Bis 9. August 2018
Jetzt e-Magazine herunterladen
www.oraworld.org

JavaLand

19. - 21. März 2019 in Brühl bei Köln

Ab sofort Ticket & Hotel buchen!

www.javaland.eu

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu