Inhalt

Java auf dem Weg zur nächsten LTS-Version

Sweet 16

14, 15 oder doch schon 16? Da kann man schon mal durcheinanderkommen. Durch die mittlerweile halbjährlichen Major-Releases von Java fällt es gar nicht so leicht, die aktuelle Version richtig zu benennen. Vor Kurzem hat man sich in einem Vortrag noch über die Neuerungen des JDK 14 informiert, und wenig später wurden in einem Artikel bereits die Features von Version 15 näher beleuchtet. Und da sich die Welt bekanntlich schnell weiterdreht, ist nun im März 2021 bereits das OpenJDK 16 herausgekommen.

von Falk Sippach

Um die Verwirrung komplettzumachen, kann man sich natürlich auch nur auf die sogenannten Long-Term-Support-Versionen (LTS) konzentrieren, die alle drei Jahre erscheinen. Das ist im Moment Java 11, wobei in der freien Wildbahn die Version 8 ebenfalls noch sehr weit verbreitet ist. Im September 2021 wird nun mit dem JDK 17 das nächste LTS-Release erscheinen. Dort werden die Neuerungen der vergangenen drei Jahre (Java 12 bis 16) finalisiert, um bis zur darauffolgenden LTS-Version (JDK 23) gut dazustehen. In den vergangenen „Zwischen"-Releases wurden die diversen, teilweise auch größeren Änderungen häufig als Previews veröffentlicht. Dadurch konnte frühzeitig Feedback eingesammelt und bereits im nächsten Release eingearbeitet werden.

(Nicht ganz so) neue Features

Die Liste der für das OpenJDK 16 umgesetzten JEPs (Java Enhancement Proposals) sieht auf den ersten Blick wieder relativ lang aus [1]:

- 338: Vector API (Incubator)
- 347: Enable C++14 Language Features
- 357: Migrate from Mercurial to Git
- 369: Migrate to GitHub
- 376: ZGC: Concurrent Thread-Stack Processing
- 380: Unix-Domain Socket Channels
- 386: Alpine Linux Port
- 387: Elastic Metaspace
- 388: Windows/AArch64 Port
- 389: Foreign Linker API (Incubator)
- 390: Warnings for Value-Based Classes
- 392: Packaging Tool
- 393: Foreign-Memory Access API (Third Incubator)
- 394: Pattern Matching for instanceof
- 395: Records
- 396: Strongly Encapsulate JDK Internals by Default
- 397: Sealed Classes (Second Preview)

Einige der Punkte sind für Java-Entwickler aber nicht direkt relevant. Dazu zählen beispielsweise die Migration zu Git/GitHub, die Aktivierung der C++-14-Sprachfeatures und auch das Foreign Linker API als zukünftiger Ersatz für das Java Native Interface (JNI). Wir schauen am Ende dieses Artikels trotzdem genauer darauf. Werfen wir aber zunächst einen Blick auf die Funktionen, die uns Entwickler betreffen. Dabei werden dem aufmerksamen Beobachter der vergangenen Java-Releases allerdings keine bahnbrechenden Neuerungen ins Auge springen. Das hängt vermutlich mit dem bevorstehenden LTS-Release zusammen, das im Herbst 2021 erscheinen wird. In Java 17 werden die neuen Features der vergangenen Monate stabilisiert, um für die folgenden Jahre eine gute Ausgangsbasis für die notwendigen Updates und Patches zu schaffen.

Java goes Pattern Matching

Bereits seit einiger Zeit schwebt das Thema Pattern Matching im Raum und hält nach und nach Einzug in Java. Dazu sind aber zur Vorbereitung diverse Änderungen in der Sprache selbst notwendig und deshalb erfolgt die Einführung nur schrittweise. Los ging es mit den Switch Expressions bereits im JDK 12. Seit Version 14 gab es zudem bereits zwei Previews zu „Pattern Matching for *instanceof*". Das wird nun mit Java 16 abgeschlossen.

Ein Pattern ist übrigens eine Kombination aus einem Prädikat (das auf eine Zielstruktur passt) und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen und damit extrahiert. Die Intention des Pattern Matching ist letztlich die Destrukturierung von Objekten, also das Aufspalten in die Bestandteile und Zuweisen in einzelne Variablen zur weiteren Bearbeitung. Die Spezialform des Pattern Matching beim *instanceof*-Operator spart unnötige Casts auf die zu prüfenden Zieldatentypen. Wenn *o* ein String oder eine Collection ist, dann kann direkt mit den neuen Variablen (*s* und *c*) mit den entsprechenden Datentypen weitergearbeitet werden. Das Ziel ist es, Redundanzen zu vermeiden und dadurch die Lesbarkeit zu erhöhen (Listing 1).

Der Unterschied zum zusätzlichen Cast mag marginal erscheinen. Für die Puristen unter den Java-Entwicklern spart das allerdings eine kleine, aber dennoch lästige Redundanz ein. Laut Brian Goetz soll die Sprache dadurch prägnanter und die Verwendung sicherer gemacht werden. Erzwungene Typumwandlungen werden vermieden und stattdessen implizit durchgeführt. Bereits die zweite Preview, die im JDK 15 erschienen war, hatte keine nennenswerten Änderungen mehr mit sich gebracht. Deswegen wird das Feature jetzt als JEP 394 finalisiert. In zukünftigen Java-Versionen wird es aber noch weitere Funktionen rund um das Pattern Matching geben, zum Beispiel in Zusammenarbeit mit den Switch Expressions.

Versiegelte Klassen

Erst das zweite Mal dabei sind die Sealed Classes. Sie wurden in Java 15 als Previewfeature eingeführt und verbleiben als JEP 397 auch im JDK 16 im Vorschaumodus. Es gibt ein paar kleine Ergänzungen gegenüber der letzten Version und vermutlich werden sie dann im LTS-Release des OpenJDK 17 finalisiert. Bis dahin möchten die Macher aber noch Rückmeldungen einsammeln.

Dieses Feature wurde übrigens im Rahmen von Projekt Amber entwickelt und gehört ebenfalls zu einer Reihe von vorbereitenden Maßnahmen für die Umsetzung von Pattern-Matching-Mechanismen in Java. Ganz konkret soll es bei der Analyse von Mustern unterstützen. Aber auch für Framework-Entwickler bieten die Sealed Classes einen interessanten Mehrwert. Die Idee ist, dass versiegelte Klassen und Interfaces entscheiden können, welche Subklassen oder -interfaces von ihnen abgeleitet werden dürfen. Bisher konnte man als Entwickler Ableitungen von Klassen nur durch Zugriffsmodifikatoren (*private, protected*, ...) einschränken oder durch die Deklaration der Klasse als *final* komplett durch den Compiler untersagen. Sealed Classes bieten nun einen deklarativen Weg, um gezielt bestimmten Subklassen die Ableitung zu erlauben:

```
public sealed class Vehicle
  permits Car, Bike, Bus, Train {
}
```

Vehicle darf nur von den vier genannten Klassen überschrieben werden. Damit wird auch dem Aufrufer deutlich gemacht, welche Subklassen erlaubt sind und überhaupt existieren. In Zukunft sollen Sealed Classes auch bei Switch Expressions eingesetzt werden können (im Rahmen des Pattern Matching). Wenn man dann je *case*-Zweig alle erlaubten Subklassen verwendet, kann der Einsatz des *default*-Blocks entfallen. Durch die Information aus der *permit*-Anweisung kann der Compiler sicherstellen, dass mindestens einer der Zweige aufgerufen wird (Listing 2).

Subklassen bergen immer die Gefahr, dass beim Überschreiben der Vertrag der Superklasse und damit das Liskovsche Substitutionsprinzip verletzt wird. Zum Beispiel ist es unmöglich, die Bedingungen der *equals*-Methode aus der Klasse *Object* zu erfüllen, wenn man

Listing 1

```
boolean isNullOrEmpty( Object o ) {
  return o == null ||
    o instanceof String s && s.isBlank() ||
    o instanceof Collection c && c.isEmpty();
}
```

Listing 2

```
// noch kein gültiger Code, kommt erst in späteren Java-Versionen
public BigDecimal calculateExpense(Vehicle vehicle) {
  return switch(vehicle) {
    case Car c -> calculateCarExpense(c);
    case Bike b -> calculateBikeExpense(b);
    case Bus b -> calculateBusExpense(b);
    case Train t -> calculateTrainExpense(t);
  }
}
```

Instanzen von einer Super- und einer Subklasse miteinander vergleichen will. Weitere Details dazu kann man in der API-Dokumentation [2] unter dem Stichwort Äquivalenzrelationen (konkret Symmetrie) nachlesen.

Sealed Classes funktionieren auch mit abstrakten Klassen. Es gibt aber ein paar Einschränkungen. Eine Sealed Class und alle erlaubten Subklassen müssen im selben Modul existieren. Im Falle von Unnamed Modules müssen sie sogar im gleichen Package liegen. Außerdem muss jede erlaubte Subklasse direkt von der Sealed Class ableiten. Die abgeleiteten Klassen dürfen übrigens wieder selbst entscheiden, ob sie weiterhin versiegelt, final oder komplett offen sein wollen. Die zentrale Versiegelung einer ganzen Klassenhierarchie von oben bis zur untersten Hierarchiestufe ist nicht möglich.

Records

Bereits zum dritten Mal dabei sind wieder die in Java 14 eingeführten *record*-Datentypen. Mit dem JEP 395 sollen sie nun finalisiert werden. Es gab seit der zweiten Preview (JDK 15) noch einige kleine Änderungen, die sich aus den Rückmeldungen der letzten Monate ergeben haben. Bei den Records handelt es sich um eine eingeschränkte Form der Klassendeklaration, ähnlich den Enums. Entwickelt wurden Records im Rahmen des Projektes Valhalla. Es gibt gewisse Ähnlichkeiten zu Data Classes in Kotlin und Case Classes in Scala. Auch sie sind im Umfeld der Einführung von Pattern Matching entstanden und werden in folgenden JDK-Releases noch relevanter werden. Zudem könnte die kompakte Syntax Bibliotheken wie Lombok in Zukunft zumindest zum Teil obsolet machen. Die einfache Definition einer Person mit zwei Feldern kann man nachfolgend betrachten:

```
public record Person(String name, Person partner ) {}
```

Eine erweiterte Variante mit einem zusätzlichen Konstruktor ist erlaubt. Dadurch lassen sich neben Pflichtfeldern auch optionale Felder abbilden (Listing 3).

Erzeugt wird vom Compiler eine unveränderbare (immutable) Klasse, die neben den beiden Attributen und den eigenen Methoden natürlich auch noch die Implementierungen für die Accessoren, den Konstruktor sowie *equals/hashCode* und *toString* enthält. Im Listing 4 sieht man den Pseudocode, den man dafür hätte schreiben müssen.

Verwendet werden Records dann wie normale Java-Klassen. Der Aufrufer merkt also gar nicht, dass ein spezieller Typ instanziiert wird (Listing 5).

Records sind übrigens keine klassischen Java Beans, da sie keine echten Getter enthalten. Man kann auf die Membervariablen aber über die gleichnamigen Methoden zugreifen (*name()* statt *getName()*). Sie können im Übrigen auch Annotationen oder JavaDocs enthalten. Im Body dürfen zudem statische Felder sowie Methoden, Konstruktoren oder Instanzmethoden deklariert werden. Nicht erlaubt ist die Definition von weiteren Instanzfeldern außerhalb des *record*-Headers.

Zwischen Sealed Classes und den *record*-Typen gibt es eine Integration, wie das Beispiel in Listing 6 zeigt.

Eine Familie von Records kann vom gleichen Sealed Interface ableiten. Die Kombination aus Records und versiegelten Datentypen führt uns zu algebraischen Datentypen, die vor allem in funktionalen Sprachen wie Haskell zum Einsatz kommen. Konkret können wir jetzt

Listing 3

```
public record Person(String name, Person partner ) {
public Person(String name ) {
  this( name, null );
}
  public String getNameInUppercase() {
    return name.toUpperCase();
  }
}
```

Listing 5

```
var man = new Person("Adam");
var woman = new Person("Eve", man);
woman.toString(); // ==> "Person[name=Eve,
partner=Person[name=Adam, partner=null]]"

woman.partner().name(); // ==> "Adam"
woman.getNameInUppercase(); // ==> "EVE"

// Deep equals
new Person("Eve", new Person("Adam")).equals( woman ); // ==> true
```

Listing 4

```
public final class Person extends Record {
  private final String name;
  private final Person partner;

  public Person(String name) { this(name, null); }
  public Person(String name, Person partner) {
    this.name = name; this.partner = partner;
  }

  public String getNameInUppercase() {
    return name.toUpperCase();
  }
  public String toString() { /* ... */ }
  public final int hashCode() { /* ... */ }
  public final boolean equals(Object o) { /* ... */ }
  public String name() { return name; }
  public Person partner() { return partner; }
}
```

mit Records Produkttypen und mit versiegelten Klassen Summentypen abbilden. Und auch hier schließt sich wieder der Kreis zum Pattern Matching, das ebenfalls vor allem in funktionalen Sprachen zum Einsatz kommt. Die Modernisierung von Java entwickelt sich also weiterhin in die vielversprechende Richtung der funktionalen Programmierung. Aber bitte nicht erwarten, dass Java in ein paar Jahren eine rein funktionale Sprache sein wird. Da wird es auch in Zukunft besser geeignete Kandidaten (Haskell, Clojure, …) geben. Nichtsdestotrotz wird man auch bei der Entwicklung mit Java von einigen dieser Möglichkeiten profitieren.

Weitere interessante Neuerungen

Bisher wurde die Quellen des OpenJDK in Mercurial verwaltet, einem nicht so verbreiteten Versionsverwaltungssystem. Dadurch war die Hürde für neue Entwickler relativ hoch, sich an der Entwicklung des JDK zu beteiligen. Im Rahmen des JEP 357 wurde der Sourcecode nun in ein Git-Repository migriert und sogar noch nach GitHub [3] umgezogen (JEP 369). Dabei gab es drei Hauptgründe für die Migration:

1. Größe der Metadaten des Versionsveraltungssystems
2. verfügbare Werkzeuge für die Versionsverwaltung
3. Angebote an Hostingoptionen

Bei den Metadaten kam es immerhin zu einer Reduktion von 1,2 GByte auf 300 MByte im *.git*-Ordner. Zudem wird bei vielen Tools wie IDEs oder Texteditoren Git bereits standardmäßig unterstützt oder lässt sich leicht über Plug-ins nachrüsten. Git hat den Markt für verteilte Versionsverwaltungssysteme in den letzten Jahren nahezu überrollt. Der Schritt, die Quellen des OpenJDK nach GitHub umzuziehen, ist also nachvollziehbar. Um alle relevanten Informationen wie die Historie und Tags mit zu übertragen, wurde eigens ein kleines Tool geschrieben. Dieses hat die Mercurial Commit Messsages in das Git-Format überführt.

Wer schon sehr lange in der Java-Welt unterwegs ist, wird sich vielleicht noch an das Java Native Interface (JNI) erinnern. Damit kann man nativen C-Code aus Java heraus aufrufen. Der Ansatz ist aber relativ aufwendig und fragil. Das Foreign Linker API (JEP 389) bietet nun einen statisch typisierten, rein Java-basierten Zugriff auf nativen Code. Zusammen mit dem Foreign-Memory Access API (JEP 393) kann diese Schnittstelle den bisher fehleranfälligen Prozess der Anbindung einer nativen Bibliothek beträchtlich vereinfachen. Mit Letzterer bekommen Java-Anwendungen die Möglichkeit, außerhalb des Heap zusätzlichen Speicher zu allokieren.

Wer häufig mit primitiven Wrapper-Klassen (Integer, Boolean, …) arbeitet, wird ab dem JDK 16 womöglich über neue Deprecation-for-Removal-Warnungen stolpern. Das betrifft sowohl die Konstruktoren mit dem Stringparameter als auch mit dem jeweiligen primitiven Datentyp als Argument (*int* bei Integer). Hinter dieser Maßnahme steckt auch das Projekt Valhalla. Dort strebt man die Erweiterung des Java-Programmiermodells in Form von primitiven Klassen an. Diese primitiven Klassen sollen keine Identität besitzen und dadurch auch leicht vom Compiler bzw. dem Laufzeit-Interpreter „ge-inlined“ werden können. Dadurch lassen sie sich frei zwischen Speicherorten kopieren und als Werte von Instanzfeldern kodieren.

Ebenfalls dem Vorschaufeature entwachsen ist mit dem OpenJDK 16 das jpackage-Werkzeug. Es unterstützt native Paketformate, um den Nutzern eine einfache Installation zu ermöglichen, inklusive der Angabe von Startparametern zum Zeitpunkt der Paketierung. Zu den Formaten gehören *msi* und *exe* unter Windows, *pkg* und *dmg* unter macOS, sowie *deb* und *rpm* unter Linux. Das Tool kann direkt über die Befehlszeile oder auch programmatisch aufgerufen werden.

Durch das in Java 9 eingeführte Java Platform Modul System (JPMS) können nun JDK-interne Klassen vor dem Zugriff von außen geschützt werden. Bisher gab es zum Übergang aber die eingeschränkte Kapselung als Default, d. h., dass interne APIs weiterhin verwen-

Listing 6

```
public sealed interface Expr
  permits ConstantExpr, PlusExpr, TimesExpr, NegExpr {
...
}
public record ConstantExpr(int i) implements Expr {...}
public record PlusExpr(Expr a, Expr b) implements Expr {...}
public record TimesExpr(Expr a, Expr b) implements Expr {...}
public record NegExpr(Expr e) implements Expr {...}
```

det werden konnten. Dieses Schlupfloch wird mit dem OpenJDK 16 geschlossen. Die neue Standardeinstellung ist die strikte Kapselung der JDK-Internas, außer für sehr kritische interne APIs wie *misc.Unsafe*. Das Ziel dieser Maßnahme ist die Erhöhung der Sicherheit und Wartbarkeit des JDK. Man möchte die Entwickler ermutigen, alte, auf Internas basierende Lösungen zukünftig für den Zugriff auf Standard-APIs umzubauen. Somit sollen sowohl Java-Entwickler als auch Endbenutzer viel problemloser auf zukünftige Versionen updaten können.

Neben den prominenten JEPs gibt es in jeder neuen JDK-Version noch viele kleine Änderungen, zum Beispiel an der Java-Klassenbibliothek. Mit dem Java Version Almanac [4] kann man sehr einfach und kompakt die Differenzen zwischen den Releases, aber auch bei Versionssprüngen von z. B. JDK 8 auf 16 einsehen. Aus Entwicklersicht sind besonders zwei Neuerungen an der Klasse *Stream* interessant. *Stream.toList()* bietet eine prägnantere und im Einsatz mit *parallel()* meist auch effizientere Alternative zu *Stream.collect(Collectors.toList())*. Als Ergebnis wird eine nicht veränderbare (unmodifiable) *ArrayList* zurückgegeben. Weitere Informationen kann man der API-Dokumentation [5] oder dem Artikel von Donald Raab [6] entnehmen. Die zweite Neuerung in der Klasse *Stream* ist die in diversen Ausprägungen hinzugekommene Methode *mapMulti(BiConsumer)*. Sie stellt eine imperative und schnellere Alternative zu *flatMap* dar. Nicolai Parlog hat die neue Funktion in einem Blogpost näher unter die Lupe genommen und zum Vergleich Performancemessungen durchgeführt [7].

Ausblick

Schon kurz vor der Fertigstellung des OpenJDK 16 wurden die ersten Features für das JDK 17 angekündigt [8]. Unter anderem soll es eine neue Rendering Pipeline für macOS und die Erweiterung des Pseudozufallszahlengenerators geben. Höchstwahrscheinlich werden wir allerdings keine weiteren großen, prominenten Änderungen sehen. Schließlich werden mit Java 17 die Entwicklungen der vergangenen 36 Monate abgeschlossen und es wird eine Version bereitgestellt, die wieder für die nächsten Jahre mit Updates und Sicherheitpatches versorgt werden muss. Die wirklich spannenden Neuerungen und Syntaxerweiterungen werden wir dann voraussichtlich erst wieder in der Version 18 sehen. Ab da hat Oracle dann wieder zweieinhalb Jahre Zeit, Feedback zu Previewfeatures einzuarbeiten und diese abzurunden.

Der 2018 eingeschlagene Weg – der anfangs nicht unumstritten war – wird weiterhin konsequent verfolgt. Die halbjährlichen Updates der Programmiersprache und der Plattform Java geben uns Entwicklern die einfache Möglichkeit, regelmäßig neue Funktionen ausprobieren zu können. Potenziell kann man sogar sein Produktivsystem alle sechs Monate aktualisieren und vermeidet langwierige Migrationsaufwände, die sonst irgendwann anfallen würden. Konservative Kunden können aber trotzdem LTS-Versionen einsetzen, die wie frühere Releases (vor Java 8) über mehrere Jahre mit Updates versorgt werden. Dabei hat man mittlerweile die freie Wahl zwischen verschiedenen Anbietern. Neben kommerziellen Versionen (Oracle JDK und andere) gibt es auch genügend JDKs mit freien Updates (allen voran AdoptOpenJDK). Das Java-Ökosystem ist also lebendiger denn je und weiterhin sehr innovativ. Konkurrenz wie beispielsweise Python (hauptsächlich wegen Machine/Deep Learning), Go und die C-basierten Sprachen beleben das Geschäft. Java, das besonders im Unternehmensanwendungsumfeld vertreten ist, wird aber weiterhin ein gewichtiges Wort mitreden.

Falk Sippach ist bei der embarc Software Consulting GmbH als Softwarearchitekt, Berater und Trainer stets auf der Suche nach dem Funken Leidenschaft, den er bei seinen Teilnehmern, Kunden und Kollegen entfachen kann. Bereits seit über 15 Jahren unterstützt er in meist agilen Softwareentwicklungsprojekten im Java-Umfeld. Als aktiver Bestandteil der Community (Mitorganisator der JUG Darmstadt) teilt er zudem sein Wissen gern in Artikeln, Blogbeiträgen sowie bei Vorträgen auf Konferenzen oder User-Group-Treffen und unterstützt bei der Organisation diverser Fachveranstaltungen.

@sippsack

Links & Literatur

[1] https://openjdk.java.net/projects/jdk/16/

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html

[3] https://github.blog/2020-09-30-github-welcomes-the-openjdk-project

[4] https://javaalmanac.io

[5] https://download.java.net/java/early_access/jdk16/docs/api/java.base/java/util/stream/Stream.html

[6] https://medium.com/javarevisited/stream-tolist-and-other-converter-methods-ive-wanted-since-java-2-c620500cb7ab

[7] https://nipafx.dev/java-16-stream-mapmulti/

[8] https://openjdk.java.net/projects/jdk/17/

I/O Stream Memory Overhead

Von Speicherfressern und viel leerem Nichts

Vor ein paar Wochen haben mein Kollege John Green und ich mit virtuellen Threads experimentiert (Projekt Loom). Unser Server empfing Textnachrichten, änderte ihren Case und sendete sie als Echo zurück. Unser Client simulierte jede Menge Benutzer.

von Dr. Heinz Kabutz

Unser Experiment hatten wir auf 100 000 Sockets pro JVM hochgefahren, was einer Gesamtzahl von 200 000 virtuellen Threads entsprach. Sowohl die Server- als auch die Clientkomponenten liefen gut, aber wir bemerkten, dass der Speicherverbrauch auf dem Client um ein Vielfaches höher war. Aber warum? Der Server-Task sah aus, wie in Listing 1 gezeigt. Der clientseitige Task verwendete bequem *PrintStream* und *BufferedReader* zur Kommunikation mit dem Server (Listing 2).

Nachdem wir das Histogramm von jmap auf beiden JVMs ausgeführt hatten, stellten wir fest, dass der größte Speicherfresser der *PrintStream* war, gefolgt vom *BufferedReader*. Wir änderten daher den Client Task, um stattdessen einzelne Bytes zu senden und zu empfangen. Nicht alle Clients sind ausführlich, daher erstellen wir einen *StringBuilder* nur, wenn es erforderlich ist. Darüber hinaus verwendet standardmäßig jeder *ClientTask* dasselbe statische Appendable, das einen *StringBuilder* zurückgibt, wenn es sich um einen ausführlichen Client handelt (Listing 3).

Listing 1

```
import java.io.*;
import java.net.*;

class TransmogrifyTask implements Runnable {
  private final Socket socket;

  public TransmogrifyTask(Socket socket) throws IOException {
    this.socket = socket;
  }

  public void run() {
    try (socket;
      InputStream in = socket.getInputStream();
      OutputStream out = socket.getOutputStream()
    ) {
      while (true) {
        int val = in.read();
        if (Character.isLetter(val))
          val ^= ' '; // change case of all letters
        out.write(val);
      }
    } catch (IOException e) {
      // connection closed
    }
  }
}
```

Das funktionierte viel besser und der Speicherverbrauch auf dem Server und dem Client war ungefähr gleich. Wir ließen unser Experiment etwas länger laufen und hatten schließlich zwei Millionen Sockets auf der Server-JVM geöffnet, die von zwei Millionen virtuellen Threads bedient wurden, die wiederum von nur zwölf Träger-Threads bedient wurden. Unsere Clientsimulation hatte die gleiche Anzahl von Sockets und virtuellen Threads, mit insgesamt vier Millionen Sockets und Threads. Der Speicherverbrauch von all dem lag bei unter 3 GB pro JVM. Unglaubliche Technologie: Ich kann es nicht erwarten, bis sie in Java zum Mainstream wird.

Wir haben ein weiteres Experiment durchgeführt, um festzustellen, wie viel Speicher jeder der *InputStreams* und *OutputStreams* und der Reader und Writer verbraucht. Das geschah auf unserem eigenen Rechner, sodass eure Erfahrungen variieren können:

Listing 2

```
import java.io.*;
import java.net.*;
import java.util.concurrent.*;

class ClientTaskWithIOStreams implements Runnable {
  private final Socket socket;
  private final boolean verbose;

  public ClientTaskWithIOStreams(Socket socket, boolean verbose) {
    this.socket = socket;
    this.verbose = verbose;
  }

  private static final String message = "John 3:16";

  public void run() {
    try (socket;
      BufferedReader in = new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
      PrintStream out = new PrintStream(
        socket.getOutputStream(), true)
    ) {
      while (true) {
        out.println(message);
        TimeUnit.SECONDS.sleep(2);
        String reply = in.readLine();
        if (verbose) System.out.println(reply);
        TimeUnit.SECONDS.sleep(2);
      }
    } catch (Exception consumeAndExit) {}
  }
}
```

- OutputStream
  - PrintStream 25064
  - BufferedOutputStream 8312
  - DataOutputStream 80
  - FileOutputStream 176
  - GZIPOutputStream 768
  - ObjectOutputStream 2264
- InputStream
  - BufferedInputStream 8296
  - DataInputStream 328
  - FileInputStream 176
  - GZIPInputStream 1456
  - ObjectInputStream 2256
- Writer
  - PrintWriter 80
  - BufferedWriter 16480
  - FileWriter 8608
  - OutputStreamWriter 8480
- Reader
  - BufferedReader 16496
  - FileReader 8552
  - InputStreamReader 8424

So praktisch virtuelle Threads auch sind, wir werden unsere Art zu coden ändern müssen. Wer hätte gedacht, dass wir eines Tages in der Lage sein würden, Millionen von Threads in unseren JVMs zu erstellen? Selbst der Phaser hat ein maximales Limit von 65 535 Threads. Es ist zwar möglich, Phaser zusammenzusetzen, ich kann mir jedoch vorstellen, dass die Erfinder ursprünglich dachten, dass niemand jemals mehr als 64 000 Threads

haben würde. Der *ForkJoinPool* hat eine ähnliche Begrenzung für die maximale Länge seiner Work Queues. Diese Zahlen sind vernünftig, wenn wir Tausende von Threads haben, aber nicht so sehr, wenn es Millionen Threads sind.

Mit freundlichen Grüßen aus Kreta
Heinz

PS: Die offensichtliche Frage, warum diese Objekte so viel Speicher verbrauchen, habe ich nicht beantwortet. Es liegt meist an leerem Space in Form von Puffern. Zum Beispiel hat der *BufferedReader* ein 8 k *char[]*. Da jedes char zwei Byte groß ist, ergibt das 16 kB. Der *PrintStream* enthält einen *OutputStreamWriter* (8 kB) und einen *BufferedWriter* (16 kB), sodass er etwa 25 kB groß ist. Einfach nur viel, viel leeres Nichts.

Dr. Heinz Kabutz schreibt den beliebten „The Java Specialists' Newsletter", der von zehntausenden begeisterten Fans in über 144 Ländern gelesen wird.

www.javaspecialists.eu

Listing 3

```
import java.io.*;
import java.net.*;
import java.util.concurrent.*;

class ClientTask implements Runnable {
  private final Socket socket;
  private final boolean verbose;

  public ClientTask(Socket socket, boolean verbose) {
    this.socket = socket;
    this.verbose = verbose;
  }

  private static final byte[] message = "John 3:16\n".getBytes();

  private final static Appendable INITIAL = new Appendable() {
    public Appendable append(CharSequence csq) {
      return new StringBuilder().append(csq);
    }

    public Appendable append(CharSequence csq, int start, int end) {
      return new StringBuilder().append(csq, start, end);
    }

    public Appendable append(char c) {
      return new StringBuilder().append(c);
    }
  };

  public void run() {
    Appendable appendable = INITIAL;
    try (socket;
      InputStream in = socket.getInputStream();
      OutputStream out = socket.getOutputStream()
    ) {
      while (true) {
        for (byte b : message) {
          out.write(b);
        }
        out.flush();
        TimeUnit.SECONDS.sleep(2);

        for (int i = 0; i < message.length; i++) {
          int b = in.read();
          if (verbose) {
            appendable = appendable.append((char) b);
          }
        }
        if (verbose) {
          System.out.print(appendable);
          appendable = INITIAL;
        }
        TimeUnit.SECONDS.sleep(2);
      }
    } catch (Exception consumeAndExit) {}
  }
}
```

Warum es sich lohnt, das Web-Framework auszuprobieren

Wer Kotlin mag, wird Ktor lieben

Mit der steigenden Beliebtheit von Kotlin kommt auch der Wunsch nach einem Web-Framework auf, das perfekt zu Kotlin passt und dessen Features sinnvoll sind. Ein Web-Framework wie Ktor. Achtung: Dieser Artikel bietet keine Einführung in Kotlin. Das Meiste sollte aber auch ohne fundiertes Kotlin-Know-how verständlich sein.

von Philipp Mandler

Ktor ist wohl das bekannteste in Kotlin geschriebene Framework für Webserver und -clients. Eigentlich kein Wunder, da es vom Kotlin-Schöpfer JetBrains selbst stammt. Nach über sechs Jahren Entwicklung [1] macht das Framework einen recht ausgereiften Eindruck – und wenn man nicht gerade ein spezielles Feature vermisst, sollte man es auf jeden Fall für neuen Projekte in Betracht ziehen.

Neben der Verwendung von Kotlin-nativen Features wie Koroutinen zeichnet sich Ktor dadurch aus, dass es sehr leichtgewichtig ist. In Kombination mit der auf Erweiterungen ausgelegten Architektur ist die Entwicklung mit Ktor sehr empfehlenswert.

Server

Den größten Teil von Ktor bildet das API zum Erstellen von HTTP APIs. Ich möchte zeigen, wie man mit Ktor einen simplen HTTP-Server erstellt und mit welchen Konzepten man diesen erweitern kann.

Listing 1

```
fun main() {
  embeddedServer(Netty, port = 8080) {
    routing {
      get ("/") {
        call.respondText("Hello, world!")
      }
    }
  }.start(wait = true)
}
```

Hello World

Ktor macht intensiven Gebrauch vom Typesafe-Builder-Pattern (aka Kotlin DSL). So sieht ein „Hello World“ mit Ktor etwa aus wie in Listing 1.

embeddedServer akzeptiert als letzten Parameter eine Funktion, die im Application-Kontext ausgeführt und dazu verwendet wird, den Ktor-Server zu konfigurieren. Diese Funktion bildet den Einstiegspunkt in die Konfiguration des Servers. In der Regel wird von hier aus jeglicher Ktor-bezogene Code aufgerufen.

Features

Die wohl wichtigste Funktion im Application-Kontext ist *install(…)*. Mit *install(…)* können Ktor-Features aktiviert und konfiguriert werden.

Will man beispielsweise CORS-Support (Cross-Origin Resource Sharing) aktivieren, sieht das wie in Listing 2 aus.

Auch in Listing 1 wird bereits – wenn auch etwas versteckt – ein Feature verwendet. Die Implementation von

Listing 2

```
fun main() {
  embeddedServer(Netty, port = 8080) {
    install(CORS)

    routing {
      get ("/") {
        call.respondText("Hello from another origin!")
      }
    }
  }.start(wait = true)
}
```

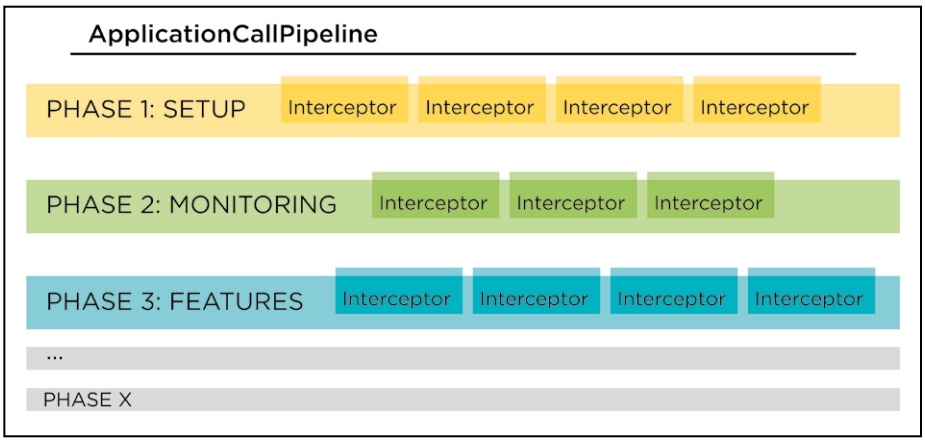


Abb. 1: Pipeline-Grafik

routing sieht hier nämlich so aus [2], wie in Listing 3 gezeigt.

routing(…) prüft demnach, ob das Routingfeature bereits installiert wurde. Wenn das der Fall ist, führt es die Konfigurationsfunktion des Features im Kontext des existierenden Features aus. Falls nicht, ruft es *install(Routing, configuration)* auf, aktiviert also das Feature und konfiguriert es dann initial mit der gegebenen Konfiguration. Dieser Mechanismus ermöglicht es, *routing* mehrmals aufzurufen, um die Implementation verschiedener Routen aufteilen und die Anwendung besser strukturieren zu können. Wichtig ist: Features dürfen nur einmal „installiert" werden.

Neben dem Routingfeature gibt es zahlreiche weitere Features, darunter die in Tabelle 1 gezeigten.

Wem die mitgelieferten Features nicht genügen, der kann weitere Features aus Bibliotheken installieren oder auch eigene Features implementieren. Oft bedarf es aber keiner vollwertigen Features, sondern es reicht, das zugrunde liegende Konzept der Interceptors zu verwenden.

Listing 3

```
@ContextDsl
fun Application.routing(configuration: Routing.() -> Unit): Routing =
  featureOrNull(Routing)?.apply(configuration) ?: install(Routing, configuration)
```

Feature	Funktionalität
Authentication	erlaubt es, Authentifizierungsmethoden für Routen zu definieren
CallLogging	Logging von Requests
Content-Negotiation	transformiert eingehende Daten basierend auf ihrem Content-Type
Compression	aktiviert Kompression für die Response
CORS	setzt CORS-Header in der Response
HSTS	setzt HSTS-Header in der Response
Locations	ergänzt Routing um typisierte Routen
Sessions	ermöglicht Sessions via Cookie oder Header (inkl. Signierung oder Verschlüsselung)

Tabelle 1: Weitere Features

Interceptors und Pipelines

Um Interceptors zu verstehen, müssen wir einen kurzen Blick darauf werfen, wie Ktor intern Requests behandelt.

Ktor nutzt zur Bearbeitung von Requests drei wichtige Konzepte: Pipelines, Phases und Interceptors. Die Verarbeitung eines Requests wird in kleine Schritte aufgeteilt – sogenannte Interceptors. Wer Spring kennt, wird diese am ehesten mit Filtern vergleichen. Wenn ein Request eintrifft, durchläuft er eine Reihe von Interceptors, die beispielsweise den Kontext des Requests bearbeiten, eine Response erstellen oder die Weiterverarbeitung abbrechen können.

Um Abhängigkeiten zwischen Interceptors abzubilden, bündelt Ktor mehrere Interceptors zu einer Phase. Im Gegensatz zu einzelnen Interceptors wird für Phases eine Reihenfolge definiert. Die Idee dahinter ist, dass

Interceptors in derselben Phase keine Abhängigkeit zueinander haben, sondern diejenigen Interceptors, von denen sie abhängen, in einer früheren Phase gebündelt sind. Beim Registrieren eines Interceptors wird angegeben, in welcher Phase sich dieser befinden soll.

Der Mechanismus, der die Phasen enthält und sie in der definierten Reihenfolge ausführt, nennt sich Pipeline.

Ktor hat standardmäßig eine vorgegebene Pipeline, die bei jedem eingehenden Request ausgeführt wird – die *ApplicationCallPipeline*. Diese hat bereits vordefinierte Phasen, es ist jedoch gängig, weitere Phasen zu ergänzen.

Neben der *ApplicationCallPipeline* gibt es auch noch zwei andere erwähnenswerte Pipelines (Tabelle 2). Die *ApplicationReceivePipeline* wird ausgeführt, wenn Daten aus einem Request empfangen werden sollen. Das passiert vor allem, wenn man *call.receive()* aufruft (vgl. später Listing 6), um Daten aus dem Body eines Requests zu entnehmen. Das *ContentNegotiation*-Feature registriert beispielsweise in dieser Pipeline einen Interceptor, um eingehende Daten zu transformieren (z. B. JSON via Jackson).

Das Pendant zu dieser Pipeline ist die *ApplicationSendPipeline*. Diese wird – wie es ihr Name schon vermuten lässt – ausgeführt, wenn eine Response gesendet werden soll (via *call.send(…)* o. Ä.).

Zu Pipelines sollte noch erwähnt werden, dass sie über einen Context und ein Subject generisch sind. Der Context bildet den Kontext, der für die Ausführung der Interceptors verfügbar ist. Im Fall der *ApplicationCallPipeline* ist das der *ApplicationCall*. Das Subject ist das Objekt, das von der Pipeline verarbeitet wird. Es wird von Interceptor zu Interceptor weitergereicht. Ein Interceptor darf dabei auch ein neues Objekt zurückgeben. Da die *ApplicationCallPipeline* kein Objekt verarbeitet, ist der Typ in diesem Fall der Einheitstyp *Unit*.

Es können auch eigene Pipelines definiert und ausgeführt werden. Ein Beispiel dafür ist das Routingfeature, das sehr starken Gebrauch von Pipelines macht.

Ein Großteil der Funktionalität von Ktor ist mit Interceptors implementiert. Wer sich Features genauer anschaut, wird feststellen, dass auch diese in der Regel „nur“ Interceptors und Phases registrieren.

Erstellen eines eigenen Interceptors

Mit dem Verständnis, wie Interceptors funktionieren, kann man nun wiederverwendbare Logik abbilden. Will

Listing 4

```
fun Route.requirePrincipal() = intercept(ApplicationCallPipeline.Call) {
  if (call.principal<Principal>() != null) {
    // additional checks
    proceed()
    return@intercept
  }

  logger.warn { "Not authenticated." }
  call.respond(HttpStatusCode.Unauthorized)
  finish()
}
```

Listing 5

```
routing {
  requirePrincipal()

  get ("/secured-route") {
    call.respond("Hello from a secured route!")
  }
}
```

Listing 6

```
class UserController(override val kodein: Kodein) : KodeinController() {

  val service: UserService by instance()

  /**
   * Registers the routes.
   */
  override fun Routing.registerRoutes() {
    get ("/login") {
      val loginData = call.receive<LoginData>()
      // handle login...
      call.respond(loginResult)
    }

    authenticated {
      get ("/currentUser") {
        // get user data...
        call.respond(userData)
      }
    }
  }
}
```

Pipeline	Subject	Context
ApplicationCallPipeline	Unit	ApplicationCall
ApplicationReceivePipeline	ApplicationReceiveRequest	ApplicationCall
ApplicationSendPipeline	Any	ApplicationCall

Tabelle 2: Ktor Pipelines

man beispielsweise den Aufruf der Routen nur erlauben, wenn eine gültige Authentifizierung besteht, kann das wie in Listing 4 aussehen.

Da die Funktion für *Route* implementiert wurde, kann sie im *routing*-Block ausgeführt werden (Listing 5).

Dependency Injection

Ktor kommt ohne eingebaute Dependency Injection. Das soll aber kein Nachteil sein, im Gegenteil – Ktor macht eine Sache und die macht es gut: Webschnittstellen. Wer Dependency Injection möchte, greift einfach auf eine vorhandene Library zurück. Wer hier wieder den Kotlin-Weg nimmt, greift dann vermutlich zu Kodein DI [3].

Ktor hat sogar ein Beispiel parat, mit dem man das bekannte Controller-Pattern (Listing 6) umsetzen kann.

Man kann Ktor-Anwendungen allerdings auch ohne Dependency Injection und mittels Kotlins Extension Functions übersichtlich strukturieren [4].

Client

Neben dem Server bietet Ktor auch einen HTTP-Client. Dieser ist nicht nur nützlich, um das erstellte HTTP API testen zu können, sondern bietet einen vollwertigen Client, um mit anderen HTTP Services zu kommunizieren.

Konfiguration

Wie der Server wird auch der Client mit dem Typesafe-Builder-Pattern konfiguriert. Ein typischer HTTP-Client, der abgerufenes JSON automatisch mittels Jackson mappt und dabei Java-8-*Time*-Datentypen (*LocalTime*, *OffsetTime* etc.) unterstützt, sieht aus wie in Listing 7.

Listing 7

```
val client = HttpClient() {
  install(JsonFeature) {
    val mapper = jacksonObjectMapper().registerModule(JavaTimeModule())
    serializer = JacksonSerializer(mapper)
  }
}
```

Feature	Funktionalität
Auth	erlaubt es, Credentials für den Aufruf zu konfigurieren
Cookies	ermöglicht es, Cookies zwischen Requests zu speichern
JSONFeauture	serialisiert ausgehende Daten als JSON und deserialisiert eingehendes JSON
Logging	loggt HTTP Requests und Reponses des Clients
Retry	ermöglicht es, fehlgeschlagene Aufrufe automatisch erneut zu versuchen

Tabelle 3: Die install(…)-Methode steht wieder zur Verfügung, um Features zu installieren

Analog zu *embeddedServer* bekommt *HttpClient* eine Konfigurationsfunktion, die im *HttpClientConfig*-Kontext ausgeführt wird (Tabelle 3). In diesem steht auch wieder die *install(…)*-Methode zur Verfügung, um Features zu installieren.

Wer mehr Kontrolle über die Client-Engine benötigt, kann diese explizit angeben und mit dem *engine*-Feld auf dieser Ebene die Konfiguration vornehmen:

```
HttpClient(Apache) {
  engine {
    socketTimeout = 20_000
  }
}
```

Ktor unterstützt verschiedene Client-Engines: Apache, OkHttp, Android, iOS, Js, Jetty, CIO und Mock. Bei den meisten Engines sollte der Name für sich selbst sprechen. Speziell erwähnen möchte ich hier jedoch:

- CIO – eine auf Koroutinen basierte Eigenimplementation von Ktor
- Mock – erlaubt es, den Client für Tests mit gemockten Antworten zu verwenden

Requests ausführen

Mit dem konfigurierten Client können nun Requests ausgeführt werden. Dafür hat die *HttpClient*-Klasse Methoden entsprechend der üblichen HTTP-Methoden *get(…)*, *post(…)*, *put(…)*, *delete(…)*, *head(…)*, *options(…)* und *patch(…)*. Die Funktionen sind mehrfach überladen, oft findet diese Form Verwendung [5]:

```
suspend fun <T> HttpClient.post(
  urlString: String,
  block: HttpRequestBuilder.() -> Unit = {}
): T
```

Dabei wird der URL für den Request als String übergeben. Optional kann ein zweiter Parameter für die Konfiguration mittels des Typesafe-Builder-Patterns angegeben werden.

```
val response: User = client.get("https://example.com/user?id=1")
```

Hier sollte beachtet werden, dass der Client auf Koroutinen aufsetzt und die Methoden somit suspending sind.

Caveats

Bei der Arbeit mit Ktor gibt es auch Fallstricke. Die Folgenden möchte ich euch nicht vorenthalten.

Einige nützliche Funktionen – beispielsweise das komplette Locations-Feature – sind bei Ktor noch als experimentell gekennzeichnet. Das ist nicht weiter schlimm, man sollte sich jedoch bewusst sein, dass sich als experimentell gekennzeichnete APIs auch ohne ein Major-Release in gewissem Maße verändern können. Mehr dazu kann in JetBrains Issue Tracker nachgelesen

werden [6]. Um die experimentellen APIs verwenden zu können, muss man das dem Kotlin-Compiler explizit mitteilen [7]. Achtung: Leider ist in Ktors API-Referenz aktuell nicht ersichtlich, ob das API experimentell ist. Hier empfiehlt es sich, vor der Verwendung eines neuen Features in die Dokumentation oder in den Code zu schauen.

Ein weiterer Fallstrick kann in den Koroutinen liegen. Ktor führt alle Requests asynchron aus, was dem Durchsatz generell sehr zugutekommt. Will man nun aber beispielsweise auf eine Datenbank zugreifen, stellt man fest, dass die meisten Bibliotheken das synchron tun und selbst Exposed [8], ein Kotlin-nativer ORM, unter der Haube aktuell synchrone Treiber verwendet. Es sollte also darauf geachtet werden, möglichst asynchrone Bibliotheken zu verwenden, sonst kann der Performancevorteil der Koroutinen schnell verflogen sein.

Fazit

Ich hoffe, ich konnte euch mit diesem Artikel einen kurzen Einblick in die Konzepte von Ktor geben.

Wie schon erwähnt, empfehle ich es jedem, Ktor einmal auszuprobieren. Für neue Kotlin-Projekte sollte man Ktor auf jeden Fall auf dem Schirm haben. Dank der Leichtgewichtigkeit macht es deutlich mehr Spaß, sich durch den Code von Ktor zu lesen, als beispielsweise durch die vielen Abstraktionsschichten von Spring Boot zu springen. Da Ktor stark auf Koroutinen setzt, rate ich jedoch davon ab, spontan eine bestehende größere Java-Codebasis zu Ktor zu migrieren.

In meinen Augen hat Ktor an den richtigen Stellen mächtige Konzepte verwendet – und auch an anderen Stellen, die vielleicht nicht so sehr im Fokus stehen.

Zudem finde ich Ktor auch im Bereich von Microservices überzeugender als beispielsweise Spring Boot. So bleibt die Komplexität einer Ktor-Anwendung in den meisten Fällen doch weit unter der einer Spring-Boot-Anwendung mit Spring Security. Hier sollte jedoch erwähnt werden, dass Ktor out of the box aktuell kein SAML unterstützt. In der Praxis hat sich hier bewährt, diesen Teil an Keycloak [9] abzugeben.

Wer jetzt neugierig geworden ist und Ktor direkt ausprobieren will, sollte sich auf jeden Fall die schicke Website von Ktor [10] anschauen. Hier findet man einige Beispiele und Anleitungen, die allgemeine Dokumentation und die API-Referenz. Die Seite wird sehr aktiv gepflegt und aktuell gehalten. Ein paar kleine Baustellen sollte man den Maintainern da nicht übelnehmen. Wer sich mit Kotlin auskennt, sollte auch mal im Source Code von Ktor stöbern [11], und wer einen ORM sucht, der gut mit Ktor zusammenspielt, sollte mal einen Blick auf Exposed von JetBrains werfen [8]. Um Ktor-Anwendungen um Dependency Injection zu ergänzen, lohnt sich ein Blick auf Kodein DI [3].

Philipp Mandler ist Softwareentwickler bei der Micromata GmbH. Mit Spaß an der Abwechslung erstreckt sich sein Aufgabenbereich von der Gestaltung von Softwarearchitekturen über die Entwicklung von Frontend- und Backend-Komponenten bis hin zum Aufbau von CI/CD-Systemen. Dabei interessiert er sich für eine Vielzahl von Entwicklungskonzepten und Technologien. Er ist nämlich überzeugt davon, dass man von Technologien grundsätzlich viel lernen kann, auch wenn man sie selbst nie produktiv verwenden wird. Deshalb guckt er gern über den Tellerrand, immer auf der Suche nach noch unbekanntem Terrain. Getreu seinem Lieblingsmotto „The right tool for the right job“ hat sich diese Neugier schon oft als sehr hilfreich erwiesen. Wenn er nicht gerade arbeitet, findet man ihn oft im lokalen Hackspace oder in der nächstgelegenen Holzwerkstatt wieder.

Links & Literatur

[1] https://github.com/ktorio/ktor/graphs/contributors

[2] https://github.com/ktorio/ktor/blob/99e490246faaf1a85d85c4c28e5fc278e34c7d0e/ktor-server/ktor-server-core/jvm/src/io/ktor/routing/Routing.kt#L127

[3] https://kodein.org/di/

[4] https://ktor.io/docs/structuring-applications.html

[5] https://api.ktor.io/1.5.0/io.ktor.client/-http-client/index.html

[6] https://youtrack.jetbrains.com/issue/KTOR-1035

[7] https://kotlinlang.org/docs/reference/opt-in-requirements.html

[8] https://github.com/JetBrains/Exposed

[9] https://www.keycloak.org

[10] https://ktor.io

[11] https://github.com/ktorio/ktor

Erkenntnisse von der anderen Seite des Tellerrands

Was ich durch Clojure über Java gelernt habe

Wir schreiben das Jahr 2012. Curiosity landet auf dem Mars, Windows 8 wird veröffentlicht, der erste Teil von „Der Hobbit“ kommt in die Kinos, die Beastie Boys lösen sich auf und Deutschland wird mal wieder nicht Fußballeuropameister. Nimmt man die Marslandung aus, also (subjektiv) ein Jahr voller Enttäuschungen. Scheinbar …

von Tim Zöller

Im selben Jahr lernte ich in meinem damaligen Job viel über Softwareentwicklung mit dem noch gar nicht so alten Java 7 und Java EE 6. Während ich mich also mit Application-Servern, EJB, JPA und JSF beschäftigte und die „Gang of Four“-Patterns in meinen Handwerkskoffer aufnahm, stolperte ich zufällig in einem Forum über Clojure – eine JVM-kompatible Sprache, die von meinen mühsam erlernten OOP- und Java-Skills sehr wenig zu halten schien: Keine statische Typisierung, keine Klassen, unveränderliche Datenstrukturen, Funktionen im Zentrum der Sprache. Nachdem der erste Kulturschock verdaut war, beschäftigte ich mich privat zunehmend mit den Konzepten der Sprache. Dabei fiel mir recht schnell auf, dass sich nicht alle, aber einige Problemstellungen in Clojure gefühlt besser lösen ließen – und dass das nicht immer an der Sprache an sich, sondern eher an der idiomatischen Herangehensweise lag. Über die letzten Jahre sind einige dieser Paradigmen in meinen Java-Code durchgesickert und haben mein Bild von gutem und wartbarem Code stark geprägt. In diesem Artikel möchte ich einige dieser Konzepte herausstellen. Dabei ist es natürlich nicht das Ziel, Java-Code zu schreiben, den nur noch Clojure-Entwickler verstehen, oder gänzlich von Javas Paradigmen abzuweichen. Um dem Artikel folgen zu können, werden keine Clojure-Kenntnisse vorausgesetzt.

Veränderlichen Zustand vermeiden

Beim Einstieg in Clojure fällt schnell auf, dass Zustand in der Applikation drastisch anders behandelt wird als in Java. Zum einen wird das durch Clojures unveränderliche Datentypen bedingt. Wird auf ihnen eine Funktion angewandt, gibt diese eine neue Datenstruktur zurück und lässt die ursprünglich übergebene unverändert. Am Beispiel in Listing 1 zeigt sich, dass die Datenstruktur hinter dem Symbol *my-vector* nicht nachträglich manipuliert werden kann. Das Inkrement wird im zweiten Statement zwar auf jedes Element des Vektors angewandt, das Ergebnis ist jedoch eine neue Datenstruktur. Daraus folgt, dass Entwickler eine Datenstruktur jederzeit bedenkenlos an weitere Funktionen übergeben und dabei sicher sein können, dass keine Operation sie manipuliert. Das Verändern dieser übergebenen Daten hätte mehrere Nachteile: Parallelisierte Ausführung von Code wird erschwert, da das Auftreten von Race Conditions nicht ausgeschlossen werden kann. Immerhin kann niemand sagen, welcher aufgerufene Code das Objekt wann modifiziert. Zum anderen kann eine angepasste Codestelle auf einmal an einer ganz anderen Stelle des Codes für Fehler sorgen, ohne dass

Listing 1

```
; Zuweisen eines Vektors zum Symbol my-vector
(def my-vector [1 2 3 4 5])

; Anwenden der Funktion inc auf jedes Element
(map inc my-vector)
-> [2 3 4 5 6]

; Ausgabe des ursprünglichen Vektors
my-vector
-> [1 2 3 4 5]
```

Entwickler das erwarten. Dieses Problem sprach der Erfinder von Clojure, Rich Hickey, auf der Java One in seinem Talk „Clojure Made Simple“ [1] ausführlich an. Ein erster Schritt auf dem Weg zu einem robusteren Datenmodell kann der Verzicht auf ein anämisches Datenmodell sein, also etwa Java Beans, in denen jedes Feld über Getter und Setter verfügt. Die Klasse *Square* in Listing 2 ist ein Beispiel für eine unveränderliche Datenstruktur in Java. Reflection außen vorgelassen, kann ein einmal erstelltes Objekt dieser Klasse nicht mehr geändert werden.

In Java lässt sich Code zwar defensiv schreiben, faktisch kann aber nicht immer verhindert werden, dass ein Objekt im weiteren Verlauf der Arbeit manipuliert wird – verschachtelte Objektstrukturen und Collections bieten viel Raum für (unabsichtliche) Zustandsmanipulationen am Objekt. Letztendlich hilft lediglich die Disziplin des Teams und die Verwendung von Bibliotheken und Frameworks festzulegen, die übergebene Objekte nicht manipulieren. Außerdem existieren immer noch Bibliotheken und Frameworks, die die Java-Bean-Struktur erwarten, etwa in Bezug auf Mapping, Persistenz oder Datenserialisierung.

Aufrufe verketten, nicht verschachteln

In Enterprise-Java-Projekten ist das Konzept der Schichtarchitektur weit verbreitet. Die Idee dahinter ist, Klassen im Projekt nach ihrer technischen Zugehörigkeit zu strukturieren. Dabei erhalten Klassen jeweils nur Zugriff auf die nächste zu verwendende Schicht, um die Abhängigkeiten möglichst minimal zu halten. Üblicherweise wird die Referenz auf das zuständige Objekt der nächsten Schicht mit Dependency Injection übergeben und dann im Programmcode verwendet. Eine sehr simple Darstellung dieses Vorgehens findet sich in Listing 3. In Clojure existieren keine vergleichbaren Konzepte, in meinen Anfängen versuchte ich, dieses Schichtenmodell jedoch zunächst zu reproduzieren (Listing 4) – und trat in dieselbe Falle, die die Schichtenarchitektur auch im Java-Bereich mit sich bringt. Weder die hier dargestellte Java-Variante noch die zugehörige Clojure-Version lassen sich als eigenständige Unit testen, ohne eine Mocking Library zur Hilfe zu nehmen. In einem Unit-Test soll ausschließlich die kleinstmögliche Funktionalität getestet werden und auf keinen Fall der Speichermechanismus – schon gar nicht, wenn dazu eine Datenbank benötigt wird. Der Zugriff auf einen anderen Namespace lässt sich in Clojure allerdings nicht wirklich mocken, er ähnelt dem Aufruf einer statischen Methode in Java. Eine Lösung könnte sein, die Speicherfunktion als Parameter im Aufruf mitzugeben. Damit könnte man die Speicherfunktion im Test tauschen/mocken und somit die Logik testen. Am sinnvollsten wäre es jedoch, diese Verkettung aufzuheben und die Abhängigkeiten eine Ebene weiter oben zusammenzuführen. Damit werden die zu testenden Units verkleinert – da sie weniger tun, lassen sie sich isolierter testen. Das klingt sehr nach

Listing 2

```
class Square {

  private final int x;
  private final int y;

  Square(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

}
```

Listing 3

```
public class PersonService {

  private PersonRepository personRepository;

  public PersonService(PersonRepository personRepository) {
    this.personRepository = personRepository;
  }

  public void saveWithUpdateDate(Person person) {
    person.setUpdateDate(LocalDateTime.now());
    personRepository.save(person);
  }

  ...
}
```

Listing 4

```
;; Defnition eines Namespaces, Referenzierung eines anderes Namespaces
;; mit dem alias 'db'
(ns myapp.service.person
  (:require [myapp.db.person :as db]))

;; Beispiel 1: Keine gute Idee
;; Definition einer Funktion mit dem Namen 'save-with-update-date' und
;; dem parameter 'person'
(defn save-with-update-date! [person]
  (db/save-person!
    (assoc person :update-date (java.time.LocalDateTime/now))))
;assoc assoziiert den Datumswert mit dem key :update-date an 'person'

;; Beispiel 2: Besser, aber nicht toll
(defn save-with-update-date! [person save-function]
  (save-function
    (assoc person :update-date (java.time.LocalDateTime/now))))


;; Beispiel 3: Trennen von Seiteneffekt und Aktion
(ns myapp.othernamespace.person
  (:require [myapp.service.person :as service]
    [myapp.db.person :as db]))

(db/save-person! (service/update-date person))
```

dem Single Responsibility Principle aus der Objektorientierung und sollte eigentlich keine große Entdeckung sein – ist aber in den oft üblichen Schichtarchitekturen in meinen Augen weitestgehend verloren gegangen. Die Units auch im Java-Code zu verkleinern, um mehr Methoden im Code zu haben, die frei von Seiteneffekten sind, macht Testing wesentlich einfacher, verringert den Einsatz von Mocks und entkoppelt den Code.

Es müssen nicht immer Objekte übergeben werden

Eine Java-Methode, die einen bestimmten Objekttyp erwartet, ist zwangsläufig an diesen Typ gekoppelt – so weit, so logisch. Möchte man verschiedene Typen an diese Methode übergeben können, könnte man sich vorstellen, dass sie ein gemeinsames Interface implementieren (Listing 5). Dieses Interface koppelt jedoch sämtliche Typen, die es implementiert, in einem gewissen Grad aneinander. Entwickler treffen damit die Entscheidung, dass diese Klassen Gemeinsamkeiten haben und in Zukunft alle Änderungen an diesem Interface mitgehen müssen. Das kann natürlich tatsächlich gewünschtes Verhalten sein, eventuell aber auch dazu führen, dass Klassen, die sich gemeinsame Felder teilen, als zusammengehörend behandelt werden, obwohl sie es gar nicht sind. Eine „Person“ kann in verschiedenen Kontexten einer Klasse vorkommen – und diese Kontexte können eventuell mit Absicht voneinander entkoppelt sein. Wäre es da tatsächlich sinnvoll, den Klassen nach der Entkopplung ein gemeinsames Interface überzustülpen?

Listing 5

```
// Funktioniert nur mit Klasse Person
public String renderAddress(Person person) {
  return String.format("%s\n %s \n %s %s", person.getName(), person.
                        getStreet(), person.getZipCode(), person.getCity());
}

// Definition eines gemeinsamen Interface
public class Person implements Address {
 ...
}

// Jeder Typ, der das Interface implementiert, kann übergeben werden
public String renderAddress(Address address) {
 ...
}

// Unbequemer, aber vielfältig einsetzbar, ohne Annahmen über den Typ
// zu treffen
public String renderAddress(String name, String street, String zipCode,
                                                        String city) {
 ...
}
```

Eine weitere Antwort auf dieses Problem lässt sich erörtern, wenn man es aus der Perspektive von Clojure betrachtet. Subsets von Datenstrukturen – in Clojure oftmals Maps – lassen sich in einer Funktionssignatur mit Destructuring herausstellen (Listing 6). Die Funktion *render-address* benutzt in ihrer Parameterliste hierzu assoziatives Destructuring, angezeigt durch *:keys*. Wird der Funktion eine Map als Parameter übergeben, wird der Wert hinter dem Keyword *:name* an das Symbol *name* gebunden, *:street* an *street* usw. Das bedeutet, dass die Funktion mit jeder Datenstruktur funktioniert, die diese Keys enthält – somit auch in verschiedenen Kontexten der Applikation. Dabei werden für diese Datenstrukturen keinerlei Kopplung und keine übergeordnete Datenstruktur eingeführt, die eine Zusammengehörigkeit suggeriert. Wie man in der untersten Methodensignatur in Listing 5 sieht, lässt sich das nur sehr unbequem in Java umsetzen – das Objekt, das übergeben wird, vor dem Methodenaufruf zu zerlegen, kann schnell in unübersichtlichen Code ausufern. Hält sich die Parameterliste in Grenzen, kann es aber einen gangbaren Weg darstellen, eine Methode ohne Kopplung in verschiedenen Kontexten benutzbar zu machen. Das würde sich wesentlich einfacher gestalten, wenn Java selbst Destructuring anbieten würde. Dass das in statisch typisierten Sprachen möglich ist, zeigen etwa TypeScript und Kotlin. Beim Blick auf die Roadmap für Java erscheint es jedoch gar nicht so unwahrscheinlich, dass die Vorschläge für Pattern Matching in näherer Zeit auf generelles Destructuring ausgedehnt werden könnten.

Libraries und Templates statt Frameworks

Nachdem ich mich mit den Grundlagen von Clojure vertraut gemacht hatte, machte ich mich an die Erstellung meiner ersten Webapplikation. Schnell musste ich feststellen, dass ein Rundum-sorglos-Framework im Stil von Spring, Quarkus, dem Play Framework oder auch JEE nicht existierte. Vielmehr schien es üblich zu sein, verschiedene Libraries zu einer Applikation zu kombinieren. Was zunächst mühsam und nach viel Arbeit klingt, hat einen entscheidenden Vorteil: Eine Library greift nicht in die Struktur des Codes ein, wie es ein Framework mit-

Listing 6

```
;; Beispiel für assoziatives Destructuring
(defn render-address [{:keys [name street zipCode city]]
  (format "%s \n %s \n %s %s" name street zipCode city))

;; Die überzähligen Werte werden ignoriert
(render-address {:name "Harry Hirsch"
                 :street "Hirschstr. 4"
                 :zipCode "35781"
                 :city "Hirschhausen"
                 :age 41
                 :height 181}
```

unter tut. Es gibt keine Konfiguration durch Konvention, es werden keine Annotations an bestimmten Stellen des Codes erwartet, Entwickler werden nicht zu einem anämischen Datenmodell gezwungen, nur weil sie ein Framework benutzen, das erwartet, dass eine Klasse vollständige Getter und Setter hat. Dabei haben sie stets freie Hand in der Strukturierung ihres Programms. Da es in Clojure generell üblich ist, größere Systeme aus mehreren kleinen Subsystemen zu erstellen, wirkt es nach einiger Zeit recht natürlich, keine übergreifenden Konzepte über den Code zu spannen. Um dabei nicht die ersten zwei Wochen der Programmierung mit dem Zusammenstellen und Verdrahten dieser Bibliotheken zu verbringen, gibt es verschiedene Templatemechanismen, ähnlich z. B. zum Spring Boot Starter. Möchte man z. B. eine Applikation mit dem Luminus Stack [2] erzeugen (der als Framework bezeichnet wird, de facto aber eine Zusammenstellung von Libraries darstellt), kann man beim Erstellen eines neuen Projekts angeben, welche Libraries man verwenden möchte, und bekommt sie in einem neuen Projekt fertig aufgesetzt als Code generiert. Zur Auswahl stehen etwa unterschiedliche HTTP-Server (Jetty, http-kit, Undertow), Persistenzdienste (H2, MySQL, PostgreSQL, MongoDB, Datomic), Frontend-Technologien und Testtools. Dabei werden die Grenzen zwischen den Libraries nicht verwischt, sodass es einfach ist, einzelne Komponenten auch in stark gewachsenen Codebasen nachträglich auszutauschen, ohne andere, nicht direkt betroffene Teile zu schädigen. Auch verlieren so aufgebaute Applikationen einen großen Teil ihrer „Magie“. Die Logik steckt vollständig im Code und nicht in Konfigurationen, Aspekten und automatisch erzeugten Proxy-Klassen.

In der Java-Welt existieren durchaus Technologien, die sich auf eine ähnliche Art und Weise nutzen lassen. Eclipse vert.x [3] sieht sich etwa ebenfalls als Tookit – nicht als Framework – und bietet Entwicklern bei der Strukturierung ihres Programms ähnliche Freiheitsgrade. Einen Templating-Mechanismus, der die Auswahl der im Stack vorhandenen Komponenten ermöglicht, bietet vert.x ebenfalls auf seiner Webseite an. Gleichzeitig muss klar sein, dass die gängigen, eher meinungsstarken Java-Frameworks durchaus ihre Berechtigung haben. Durch die erprobte Verzahnung von Komponenten lässt sich viel Entwicklungszeit einsparen, die vorgegebenen Strukturen passen für viele Applikationen und Architekturen ohnehin sehr gut. Für das eigene Verständnis von Architektur und für Spezialfälle kann es jedoch sehr lehrreich sein, sich ab und an von den vorgegebenen Pfaden abzuwenden.

Zusammenfassung

Die großen Stärken von Clojure lassen sich nicht uneingeschränkt auf Java übertragen – schließlich ist es eine andere Sprache mit anderen Paradigmen. Gleichzeitig sollte die Begeisterung für Clojure auch nicht so sehr auf den eigenen Java-Code überspringen, dass Java Entwickler vom Code und den Konzepten überrascht sind. Niemand würde zum Beispiel ernsthaft vorschlagen, dass Java-Entwickler nur noch Maps anstelle von Objekten verwenden und anfangen, sämtliche Logik in statische Methoden zu packen. Trotzdem können die Methodiken, die sich in Clojure bewährt haben, Java-Entwicklern dabei helfen, aus einem etwas anderen Blickwinkel auf den eigenen Code zu schauen und lange verwendete, eingefahrene Muster aufzubrechen. Dabei sind die hier vorgestellten Vorschläge – Vermeiden von anämischen Datenmodellen, Single Responsibility Principle, Entkopplung von Daten in unterschiedlichen Domänen – keinesfalls Alleinstellungsmerkmale von Clojure, sondern werden auch in der reinen Java-Welt als gute Praxis verstanden und kommuniziert. Trotzdem lässt sich in vielen größeren Java-Projekten beobachten, dass sie, auch wegen der Verwendung von vorgegebenen Frameworks, oftmals nicht zum Einsatz kommen. Der Blick über den Tellerrand auf eine Sprache, die diese Prinzipien durch ihr Design vorgibt, kann den eigenen Blick hier oft noch schärfen.

Tim Zöller ist Gründer und IT-Berater der Firma lambdaschmiede GmbH in Freiburg im Breisgau. Er entwickelt seit 12 Jahren Software für die JVM, bloggt über Java, Clojure und die Welt und spricht auf Konferenzen über seine Erfahrungen.

Links & Literatur

[1] https://www.youtube.com/watch?v=VSdnJDO-xdg

[2] https://github.com/luminus-framework

[3] https://vertx.io

Native Images für Spring-Boot-Anwendungen erzeugen

Spring Native Hands-on

Mit dem neuen Projekt Spring Native können Spring-Boot-Anwendungen von der GraalVM-Native-Image-Technologie Gebrauch machen und auch für existierende Spring-Boot-Anwendungen Start-up-Zeiten im Millisekundenbereich erzielen. Der Artikel zeigt, wie das funktioniert, wie weit Spring Native schon ist, und wie man die Technologie für eigene Spring-Boot-Anwendungen einsetzen kann.

von Martin Lippert

Die Vorteile der GraalVM-Native-Image-Technologie klingen verlockend: Start-up-Zeiten im Millisekundenbereich und ein deutlich reduzierter Verbrauch an Ressourcen (vor allem Speicher) – wer möchte das nicht?

Wie in Stephan Rauhs und Karine Vardanyans Artikel über die Technologie in dieser Ausgabe erläutert wird, kommt diese Technologie mit einer Reihe von Einschränkungen daher. Reflection funktioniert beispielsweise in einem Native Image nur, wenn der Compiler darüber informiert wird, für welche Elemente (Klassen, Methoden, Attribute) er die Reflection-Informationen zur Compile-Zeit erzeugen und im Binary hinterlegen muss. Ähnliches gilt für Proxys, zusätzliche Ressourcen, JNI-Aufrufe und Dynamic Class Loading. Andere Techniken, wie zum Beispiel *invokedynamic*, funktionieren in einem Native Image grundsätzlich nicht.

Insofern kann es eine erhebliche Herausforderung sein, eine existierende Java-Anwendung in ein Native Image zu kompilieren. Zum einen muss der Code der eigenen Anwendung frei von nicht unterstützten Techniken sein, und zum anderen müssen passende Konfigurationsdateien erstellt werden, um beispielsweise Reflection zu ermöglichen. Gleiches gilt natürlich auch für alle von der eigenen Anwendung genutzten Libraries.

Was ist mit Spring-Boot-Anwendungen?

Auch für Spring-Boot-Anwendungen gilt: Sie lassen sich mit der GraalVM-Native-Image-Technologie in native Anwendungen kompilieren. Allerdings verwendet das Spring Framework viele der eben genannten Technologien relativ ausgiebig, sodass es mitunter mühsam werden kann, die nötigen Konfigurationen für den Compiler manuell zu erstellen. Grundsätzlich ist das aber möglich.

Was ist Spring Native?

Das Spring-Native-Projekt [1] ermöglicht es Entwicklern, Spring-Boot-Anwendungen mit der GraalVM-Native-Image-Technologie in Executable Binaries zu kompilieren, ohne dass die nötigen Konfigurationsdateien manuell erstellt werden müssen oder die Anwendung speziell angepasst werden muss. Im Idealfall lassen sich also bestehende Spring-Boot-Anwendungen ausschließlich durch wenige zusätzliche Build-Instruktionen zu Native Executables kompilieren (Kasten: „In drei einfachen Schritten zur fertigen Anwendung“).

Ob es Sinn ergibt, jede Spring-Boot-Anwendung zu einem Native Executable zu kompilieren, anstatt die Anwendung in einer JVM laufen zu lassen, sei einmal dahingestellt. Diese Entscheidung hat weniger mit Spring Boot selbst zu tun als vielmehr mit dem Einsatzkontext der Anwendung.

Erste Schritte mit Spring Native

Wie beginnt man neue Spring-Boot-Projekte? Natürlich auf https://start.spring.io (bzw. den entsprechenden Wizards in der eigenen Lieblings-IDE).

In drei einfachen Schritten zur fertigen Anwendung

1. Projekt auf https://start.spring.io erzeugen (Spring Web | Spring Native) und auspacken.
2. *./mvnw spring-boot:build-image* (Build ausführen, Native Image wird kompiliert, Container-Image wird erzeugt, benötigt nur Docker)
3. *docker run --rm -p 8080:8080 demo:0.0.1-SNAPSHOT* (Beispielanwendung starten)

Als Beispiel wähle ich hier die Starter Spring Web, Spring Boot Actuator und eben Spring Native aus. Das generierte Projekt hat dann drei verschiedene Komponenten in der *pom.xml*-Datei, die speziell für Spring Native hinzugefügt wurden:

1. eine zusätzliche Dependency (Listing 1)
2. ein Build-Plug-in, das zusätzliche Informationen zur Build-Zeit erzeugt (Listing 2)
3. ein Build-Plug-in, um ein Container-Image zu erzeugen (Listing 3)

Konfigurationen automatisch erzeugen

Die zusätzliche Dependency *spring-native* (aus Listing 1) beinhaltet vor allem die Spring-spezifische Erweiterung für den GraalVM-Native-Image-Compiler. Diese Erweiterung wird automatisch vom GraalVM-Native-Image-Compiler als Teil des Native-Image-Build-Prozesses ausgeführt.

Das neuartige Spring-AOT-Plug-in für den Build erzeugt die für das Native Image nötigen Konfigurationen automatisch während des Build-Vorgangs. Inhaltlich analysiert diese Build-Erweiterung die zu kompilierende Anwendung auf verwendete Spring-Komponenten und -Annotationen. Je nachdem, welche Spring-Bibliotheken und -Annotationen in der Anwendung verwendet werden, erzeugt Spring Native die passenden Konfigurationsdateien für den GraalVM-Native-Image-Compiler, sodass diese nicht manuell erstellt werden müssen.

Darüber hinaus kann die Spring-AOT-Erweiterung auch mit vielen Schaltern konfiguriert werden, um das Native Image noch genauer auf die eigenen Bedürfnisse zuzuschneiden. Beispielsweise lassen sich diverse Features von Spring komplett ausschalten und somit der dafür benötigte Code komplett aus dem Native Image entfernen.

Der etwas in die Jahre gekommene Support für Spring-XML-Config-Dateien ist ein gutes Beispiel dafür. Verwendet die Anwendung überhaupt keine Spring-XML-Config-Dateien, kann mit dieser Option der komplette XML-Support von Spring inkl. der dazu benötigten Dependencies gar nicht erst in das Native Image hineinkompiliert werden.

Die Spring-AOT-Erweiterung erlaubt es darüber hinaus, dem Native-Image-Support eigene sogenannte Hints mitzugeben. Diese „Hinweise“ geben dem Spring-Native-Support genaue Informationen darüber mit, welche Zusatzinformationen (beispielsweise zu Reflection) benötigt werden – sollten diese nicht automatisch identifiziert werden können.

Ein Beispiel dafür sind eigene Klassen, auf die zum Beispiel eine Library per Reflection zugreift, um sie in JSON zu transformieren.

Listing 1

```
<dependency>
  <groupId>org.springframework.experimental</groupId>
  <artifactId>spring-native</artifactId>
  <version>0.9.2</version>
</dependency>
```

Listing 2

```
<plugin>
  <groupId>org.springframework.experimental</groupId>
  <artifactId>spring-aot-maven-plugin</artifactId>
  <version>0.9.2</version>
  <executions>
    <execution>
      <id>test-generate</id>
      <goals>
        <goal>test-generate</goal>
      </goals>
    </execution>
    <execution>
      <id>generate</id>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Container-Images mit Native Executables

Spring Boot bringt schon seit einigen Versionen ein Maven-Build-Plug-in mit, welches automatisch ein Container-Image für die gebaute Spring-Anwendung erzeugt. Dieses Maven-Build-Plug-in nutzt im Hintergrund die Paketo Buildpacks [2], um aus kompilierten

Spring-Boot-Anwendungen fertige Container-Images zu erzeugen.

Dieses Maven-Build-Plug-in (*spring-boot-maven-plugin*) kann so konfiguriert werden, dass vollautomatisch der GraalVM-Native-Image-Compiler verwendet und ein Native Executable erzeugt wird, welches dann in das Container-Image gelegt wird (anstatt eines JREs und den JAR-Dateien der Dependencys und der Anwendung selbst) – siehe Listing 3.

Ein großer Vorteil dieser Buildpack-basierten Methode ist, dass auf der lokalen Maschine kein passendes GraalVM SDK und keine Native-Image-Erweiterung installiert werden muss. Es reicht aus, die entsprechende Konfiguration (Listing 3) in die *pom.xml*-Datei zu integrieren und den Build auszuführen:

```
./mvnw spring-boot:build-image
```

Das Buildpack bringt das nötige GraalVM SDK automatisch mit. Das Resultat ist ein relativ kleines Container-Image. Es enthält weder ein vollständiges JRE noch die kompletten JAR-Dateien, sondern hauptsächlich das Binary der Anwendung.

Die eigentliche Größe des Binarys und dessen Speicherverbrauch im Betrieb hängt stark davon ab, wie gut und exakt zugeschnitten der Native-Image-Compiler konfiguriert wird. Je mehr Reflection-Informationen man beispielsweise konfiguriert, desto größer wird auch das Binary und desto mehr Speicher verbraucht es. Es kann sich also durchaus lohnen, möglichst wenig und möglichst genaue Reflection-Informationen zu konfigurieren, anstatt pauschal einfach alles.

Das gleiche gilt auch für die Erreichbarkeit von Code. Je genauer der Native-Image-Compiler analysieren kann, welcher Code nicht gebraucht wird, desto mehr Code wird er bei der Kompilierung des Binarys entfernen und desto weniger Ressourcen wird das Binary im Betrieb verbrauchen.

Sobald der Build das Container-Image mit dem Native Binary erzeugt hat, können wir den Container per Docker starten:

```
docker run --rm -p 8080:8080 rest-service:0.0.1-SNAPSHOT
```

Im Logoutput werden wir sehen: Die Spring-Boot-Anwendung startet innerhalb des Containers in wenigen Millisekunden.

Native Images lokal erzeugen

Ein Native Executable für eine Spring-Boot-Anwendung lässt sich auch ohne Buildpacks erzeugen. Wie im Artikel über die Native-Image-Technologie beschrieben, benötigt man dazu ein GraalVM SDK mit installierter Native Image Extension.

Listing 3

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <builder>paketobuildpacks/builder:tiny</builder>
      <env>
        <BP_NATIVE_IMAGE>true</BP_NATIVE_IMAGE>
      </env>
    </image>
  </configuration>
</plugin>
```

Listing 4

```
<profiles>
  <profile>
    <id>native-image</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.graalvm.nativeimage</groupId>
          <artifactId>native-image-maven-plugin</artifactId>
          <version>21.0.0.2</version>
          <configuration>
            <!-- The native image build needs to know the entry point to
                                                    your application -->
            <mainClass>com.example.restservice.RestServiceApplication
                                                    </mainClass>
          </configuration>
          <executions>
            <execution>
              <goals>
                <goal>native-image</goal>
              </goals>
              <phase>package</phase>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

Listing 5

```
<plugins>
  <!-- ... -->
  <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
      <classifier>exec</classifier>
    </configuration>
  </plugin>
</plugins>
```

Anschließend lässt sich das GraalVM-Maven-Plug-in dem Build hinzufügen und passend für den Native-Image-Compiler konfigurieren (Listing 4). Zusätzlich sollte man in diesem Profil das Standardverhalten des Spring-Boot-Maven-Plug-ins leicht verändern (Listing 5), um einen Konflikt mit dem Repackaged JAR des Standard-Spring-Boot-Maven-Plug-ins zu vermeiden.

Auch für das lokal erzeugte Native Image muss die bereits erwähnte Spring Native Dependency ergänzt werden (Listing 1) sowie die Spring-AOT-Erweiterung (Listing 2). Diese beiden Erweiterungen im Build sind also für beide Varianten (Buildpack und lokaler Build) wichtig und sinnvoll.

Der Build wird dann (im hier beschriebenen Beispiel) ausgeführt mit:

```
./mvnw -Pnative-image package
```

Daraufhin wird im *target*-Directory das Native Executable abgelegt, das direkt ausgeführt werden kann:

```
./target/demo
```

Dieser lokale Native-Image-Compile-Schritt läuft direkt auf der eigenen Maschine ab und verwendet das *native-image*-Kommando des lokal installierten GraalVM-SDKs. Führt man diesen Build also beispielsweise auf einer Windows-Maschine aus, wird ein Windows Binary erzeugt. Das ist ein bedeutender Unterschied zur Buildpack-basierten Native-Image-Kompilierung. Das Buildpack erzeugt ein Linux-basiertes Container-Image, in dem das Native Image erzeugt wird und per Docker-Runtime ausgeführt werden kann.

Die Roadmap

Die nächsten Schritte für das Spring-Native-Projekt sind zum einen, stetig weiter den Ressourcenverbrauch über die unterschiedlichsten Projekte und Bibliotheken zu reduzieren. Aktuell lassen sich zwar schon recht viele Spring-Boot-Starter-Module mit Spring Native verwenden, aber nicht alle sind schon komplett auf Speicherverbrauch und Performance optimiert. Hier liegt noch einige Arbeit vor dem Spring-Team.

Darüber hinaus arbeitet eine Reihe von Projekten daran, möglichst viel von Spring Native zu unterstützen und automatisch zur Build-Zeit zu erzeugen. Auch hier sind viele Verbesserungen zu erwarten.

Nicht zuletzt werden mit den nächsten Releases auch kontinuierlich mehr Spring-Boot-Starter-Module und deren Dependencys unterstützt werden. Die aktuelle Liste der unterstützten Module kann man in der Dokumentation einsehen [3].

Fazit

Spring Native kann für Spring-Boot-Entwickler zu einem echten Gamechanger werden. Mit Spring Native werden Entwickler von Spring-Boot-Anwendungen in die Lage versetzt, alle Vorteile der GraalVM-Native-Image-Technologie zu nutzen, ohne die Spring-Boot-Anwendungen speziell dafür zu modifizieren oder gar auf ein anderes Framework zu portieren. Existierende und bereits seit Jahren in der Entwicklung und im Einsatz befindliche Spring-Boot-Anwendungen können mit Spring Native von der neuen GraalVM-Native-Image-Technologie profitieren – und so unter Umständen erhebliche Ressourcen einsparen.

Ohne Frage, das Spring-Native-Projekt steht noch ziemlich am Anfang. Es lassen sich noch nicht alle Spring Boot Starter damit nutzen und auch von den unterstützten Projekten sind noch nicht alle komplett für diesen Einsatz optimiert. Aber die Arbeit an dem Projekt geht mit großen Schritten voran und das Ziel ist extrem vielversprechend.

Martin Lippert arbeitet bei VMware im Spring Engineering Team an Entwicklungswerkzeugen und IDEs für Spring und Spring Boot. Darüber hinaus beschäftigt er sich mit dem Thema Sustainability, besonders im Rahmen der Softwareentwicklung.

Links & Literatur

[1] Spring Native: https://github.com/spring-projects-experimental/spring-native

[2] Paketo Buildpacks und Spring Boot: https://spring.io/blog/2021/01/04/ymnnalft-easy-docker-image-creation-with-the-spring-boot-maven-plugin-and-buildpacks + https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-container-images-buildpacks

[3] https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/#support-spring-boot

Weiterentwicklung des Agilen Manifests

Agile ist tot. Lang lebe Modern Agile!

Das Agile Manifest ist mittlerweile 20 Jahre alt und seitdem unverändert. Was einerseits für gute Ideen spricht, läuft Gefahr, nicht mehr zu aktuellen Entwicklungen zu passen. Zudem gibt es zunehmend Kritik an dem, was heute alles als „Agile" verkauft wird. Modern Agile versucht, darauf eine angepasste, neue Antwort zu geben – oder vielmehr neue Fragen zu formulieren, die uns besser beim Finden möglicher, für uns passender Lösungen helfen.

von Thomas Much

Vor ziemlich genau 20 Jahren, im Februar 2001, trafen sich auf einer Skihütte in Snowbird, Utah (USA), 17 gestandene Softwareentwickler – darunter so bekannte Namen wie Kent Beck, Alistair Cockburn, Martin Fowler, Andrew Hunt, Ron Jeffries, Robert C. Martin und Jeff Sutherland – und formulierten ihre Ideen, wie Software modern entwickelt werden sollte. Das „Agile Manifest" (genauer: Das „Manifest für agile Softwareentwicklung") war geboren [1]. Die vier Werte bzw. Leitsätze dürften heute vielen bekannt sein [2]. Agilisten schätzen

- Individuen und Interaktionen mehr als Prozesse und Werkzeuge,
- funktionierende Software mehr als umfassende Dokumentation,
- Zusammenarbeit mit dem Kunden mehr als Vertragsverhandlung,
- das Reagieren auf Veränderung mehr als das Befolgen eines Plans.

Alle genannten Punkte (auch Dokumentation etc.) sind wichtig, aber die jeweils erstgenannten Werte gelten als wertvoller, weil sie Flexibilität und Pragmatismus fördern. Entsprechend sollten die jeweils zweitgenannten Werte nicht übertrieben werden, weil ansonsten starre Prozesse und zu viel Bürokratie zügige Reaktion auf sich ändernde Kundenanforderungen verhindern.

Dazu wurden dann noch zwölf Prinzipien formuliert, die die Werte etwas genauer beschreiben. Unter anderem geht es darum,

- wertvolle (nützliche) Software frühzeitig und kontinuierlich zum Kunden auszuliefern,
- dass Fachexperten und Entwickler täglich zusammenarbeiten,
- ein besonderes Augenmerk auf technische Exzellenz und gutes Design zu legen.

Und außerdem um regelmäßige Reflexion über das Arbeiten an sich, um ständig immer besser werden zu können.

Das war revolutionär und ziemlich radikal, denn Software wurde damals – und manchmal leider auch heute noch – häufig nur wenige Male pro Jahr, dafür als große Releases mit vielen Änderungen ausgeliefert. Danach folgten dann jede Menge Troubleshooting, Hotfixes – und als Vorbereitung des nächsten Releases der Kampf darum, welche Änderungen es in die nächste Version schafften und welche leider noch ein paar Monate warten mussten.

Die Ideen des Agilen Manifests erwiesen sich für viele Firmen als hilfreich. So hilfreich, dass bei der jährlichen „State of Agile"-Umfrage seit 2016 ca. 95 % aller

Befragten angeben, dass ihre Firma bzw. Organisation agile Entwicklungsmethoden einsetzt [3].

Liest man allerdings, was einige der Erstunterzeichner des Agilen Manifests davon halten, was heute aus ihrer Idee geworden ist, wird man stutzig. Ron Jeffries beispielsweise schreibt, dass Entwickler „Agile“ aufgeben sollten [4]. Genauer: Entwickler sollten aufhören, das zu tun, was heutzutage „Agile“ genannt wird. Wir sollen uns von dem Begriff lösen. Warum das? Was bzw. wie war „Agile“ oder vielleicht besser Agilität – denn früher? Denn die Werte und Prinzipien des Agilen Manifests sind immer noch richtig, nur die Umsetzung ist oft toxisch für alle Beteiligten.

Agilität damals

Trotz der damaligen Einfachheit und Radikalität war den Autoren des Manifests daran gelegen, dass Softwareentwicklung nicht im Chaos versinkt, sondern geordnet stattfindet. Man wollte einen Rahmen finden, an dem sich die leichtgewichtigen Vorgehensweisen der damaligen Zeit orientieren konnten.

Und leichtgewichtige Vorgehensweisen gab es in den 1990er Jahren einige. Extreme Programming (XP) ist sicherlich eine der bekanntesten, weil es nützliche Praktiken wie testgetriebene Entwicklung, Continuous Integration, Refactoring und Story Cards vereinte. Aber es gab beispielsweise auch Crystal, eine ganze Methodenfamilie, die für Projekte mit einer unterschiedlich großen Anzahl von Beteiligten auch ein unterschiedliches Vorgehen vorschlug. Oder Feature-driven Development, die Dynamic Systems Development Method oder das Adaptive Software Development. Es gab also eine Vielzahl von unterschiedlichen Vorgehensweisen, passend für verschiedene Situationen und Projekte, weil ein einziges Vorgehen unmöglich alle Anforderungen und Wünsche abdecken kann, wenn es gleichzeitig möglichst konkrete Hilfestellungen geben soll.

Damals entstand auch Scrum, das mittlerweile wohl bekannteste Rahmenwerk. So bekannt, dass viele es heute synonym für agiles Vorgehen verwenden.

Und heute?

Entsprechend ist es heute oft ein Problem, wenn Firmen und Teams unreflektiert Scrum einsetzen. Bitte nicht falsch verstehen: Scrum kann vernünftig umgesetzt werden und unglaublich hilfreich sein, wenn es zum Team und zur Aufgabe passt. Viel zu oft aber findet man einfach nur Cargo Cult vor, d. h., man macht Scrum, weil irgendwer mal irgendwo in einem anderen Team oder Unternehmen gesehen hat, dass es dort ganz gut funktionierte. Hat man dabei aber nicht verstanden, warum gewisse Rituale durchgeführt werden und welche Voraussetzungen notwendig sind, fällt es einem vermutlich schwer zu erkennen, wann man aus dem vorgegebenen Rahmen ausbrechen und sich bzw. sein Vorgehen weiterentwickeln sollte.

Oft werden Begriffe und Rollen aus Scrum nur verwendet, um den vorhandenen Prozess plakativ neu zu benennen und sich zugleich möglichst wenig verändern zu müssen. Nur kein Risiko eingehen, nichts Neues ausprobieren. Entscheidungen sollen safe sein, damit ich nicht zur Verantwortung gezogen werden kann, wenn Probleme auftauchen – doch genau das ist bei Veränderungen völlig normal. Das zementiert einmal getroffene Entscheidungen und schafft neue Prozesse. Frameworks zum Skalieren der noch gar nicht vorhandenen Agilität bringen Komplexität statt Flexibilität und Leichtgewichtigkeit. All das widerspricht komplett den ursprünglichen Ideen des agilen Manifests.

Die Folge? Weil im Wesentlichen nur neue Etiketten drauf sind, läuft der gleiche alte, träge Prozess weiter. Und alle wundern sich, dass sich nicht wirklich etwas ändert. Man merkt das meist daran, dass die neuen Ideen

und Rituale dauerhaft Fremdkörper bleiben: Die Daily Stand-up Meetings sind langweilig, die Retrospektiven nicht offen und ehrlich, die Schätz- und Planungsmeetings fühlen sich sinnfrei und wie Zeitverschwendung an. Wir sprechen dann auch von agilem Theater – alle spielen mit, das ist mal ganz lustig anzuschauen. Aber dauerhaft möchte man das nicht erleben. Wir sind Softwareentwickler, keine Schauspieler.

Schlimmer noch ist das sogenannte „Dark Agile". Denn echte Agilität verlangt Transparenz über die Arbeit, über Probleme und deren Lösung. Wenn das aber als Kontrollinstrument missbraucht wird, um noch mehr Druck auszuüben und die Mitarbeiter zu Überstunden zu zwingen oder bei Problemen Schuldige zu suchen und zu bestrafen, dann ist das nicht nur kontraproduktiv, sondern extrem schädlich. Mitarbeiter werden dadurch krank, brennen aus und verstecken sich schließlich hinter Prozessen und Bürokratie, wo sie möglichst unangreifbar sind. Für Unternehmen ist das der schleichende Weg in den Niedergang, weil dann niemand mehr Probleme offen benennen wird.

Das alles hat Gegenentwürfe provoziert. Ein schönes Beispiel ist die Programmer Anarchy von Fred George [5]: Lasst die Entwickler doch einfach mal machen, vielleicht auch herumfrickeln. Aber dann haben wir etwas, das wir deployen, messen bzw. beobachten und anschließend bewerten können. Und wenn es nicht gut ist, wird es geändert bzw. neu gemacht. Das war vor knapp zehn Jahren innovativ, provokativ, und aus Sicht von uns Entwicklern irgendwie cool. Und es enthielt viele Ideen und Wahrheiten, wie man Aufgaben in akzeptabler Zeit zur letztendlichen Zufriedenheit aller Beteiligten fertigbekommt.

Das Problem dabei: Die Idee war wieder extrem und radikal. Und das mögen Manager häufig nicht. Also wird eine gute Idee aus Angst oder Unwissenheit abgelehnt. Oder weil sie zu leicht missverstanden werden kann (Keine Dokumentation mehr! Keine Unit-Tests mehr!). Damit teilt die Programmer Anarchy das gleiche Schicksal wie Extreme Programming. Dinge geeignet zu benennen ist sehr, sehr schwer.

Und nicht so radikale, aber trotzdem gute und nützliche Ansätze wie das Software Crafting sind im agilen Hype leider etwas untergegangen.

Wenn also Agilität im Sinne des agilen Manifests so schwer zu greifen ist und die anderen Ansätze auf Ablehnung stoßen oder ignoriert werden – wie setzen wir uns dann Ziele, die alle Beteiligten einfacher verstehen können und aus eigenem Antrieb erreichen wollen?

Modern Agile

Warum so eine lange Bestandsaufnahme? Weil man damit die Beweggründe besser versteht, warum sich in den letzten Jahren immer mehr Menschen Gedanken gemacht haben, wie man den Fokus zurück auf die guten und richtigen Grundideen des agilen Manifests bringt bzw. wie man diese neu und besser greifbar formuliert für die Software- und Produktentwicklung in den 2020er Jahren.

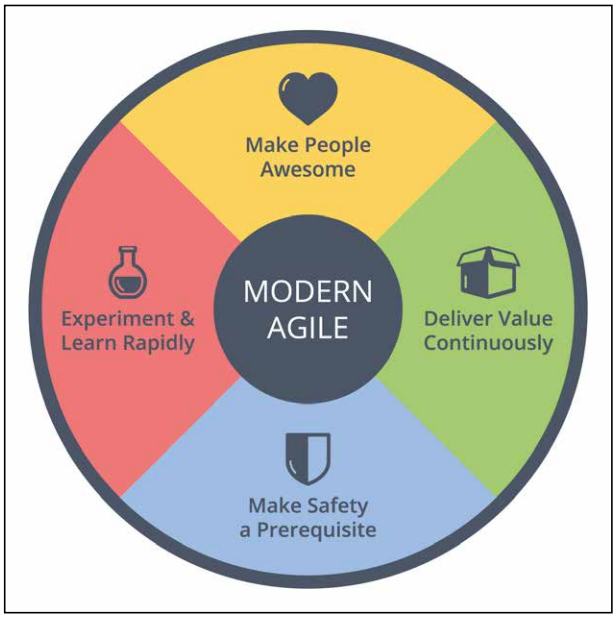


Abb. 1: Das Rad mit den Leitprinzipien von Modern Agile

Und so formulierten 2015 Joshua Kerievsky und sein Team von Industrial Logic die Ideen zu Modern Agile [6], [7] und entwickelten sie seitdem mit einer ganzen Community und als Open Source weiter [8]. Modern Agile wurde wieder radikal auf ganz wenige Kernelemente vereinfacht. Und es ist keine neue Vorgehensweise oder Methodik, sondern eher eine Hilfestellung, um über den eigenen Zustand und das eigene Vorgehen nachzudenken und dann gezielt anzupassen. Modern Agile definiert vier Guiding Principles, die wir uns im Folgenden genauer anschauen werden:

- Make People Awesome
- Make Safety a Prerequisite
- Deliver Value Continuously
- Experiment & Learn Rapidly

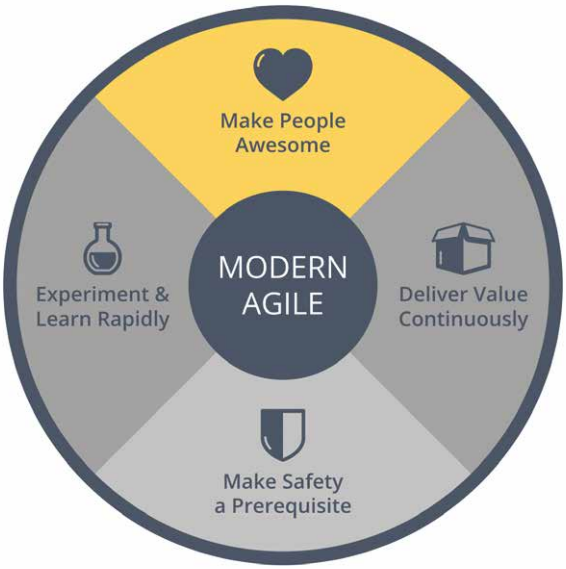
Im Deutschen könnten wir das mit „Leitprinzipien" übersetzen, aber wir müssen aufpassen, dass wir das nicht zu starr als strenge Regeln sehen. Vielleicht wäre „Grundideen" oder „Leitplanken" die bessere Übersetzung.

Die Leitplanken von Modern Agile sind bewusst sehr allgemein gehalten, dabei aber einfach, verständlich, griffig und in vielen Bereichen einsetzbar. Sie sind also nicht nur rein für die Softwareentwicklung gedacht, sondern für die gesamte Produktentwicklung und auch für ganz andere Arbeitsbereiche. Aber, und das ist für uns wiederum wichtig, es passt hervorragend dazu, wie Software modern und erfolgreich entwickelt und ausgeliefert wird.

Während diese Prinzipien auf den ersten Blick etwas zu einfach, fast trivial wirken mögen, steckt doch sehr viel dahinter, wenn man genauer darüber nachdenkt. Meiner Erfahrung nach wirken diese vier Prinzipien, wenn man sie als Fragen an sich, sein Team bzw. seine Firma versteht, sehr inspirierend und regen zum regelmäßigen Reflektieren an. Und weil sie so allgemeingültig sind, haben wir plötzlich eine gemeinsame Orientierung

für alle Beteiligten, vom Fachbereich über uns Softwareentwickler bis hin zum Kunden. Was also steckt hinter den Leitprinzipien (**Abb. 1**)?

Make People Awesome



Awesome? Das ist so ein Begriff der amerikanischen Westküste, der sehr unterschiedlich verstanden werden kann, hier aber natürlich überaus positiv gemeint ist.

Mach die Menschen um dich herum ... großartig! Brillant! Begeistert! Begeisternd! Alle Beteiligten sollen sich bei der Arbeit am bzw. der Benutzung vom Produkt großartig, verstanden und wertgeschätzt fühlen. Wenn wir das hinbekommen, wird so etwas wie Commitment und Loyalität, die die Unternehmensführung sich von Mitarbeitern bzw. Kunden wünscht, fast automatisch folgen. Bei der Entstehung oder Weiterentwicklung eines coolen Produkts möchte ich mitarbeiten. Ein cooles Produkt will ich unbedingt benutzen und kaufe es auch gerne.

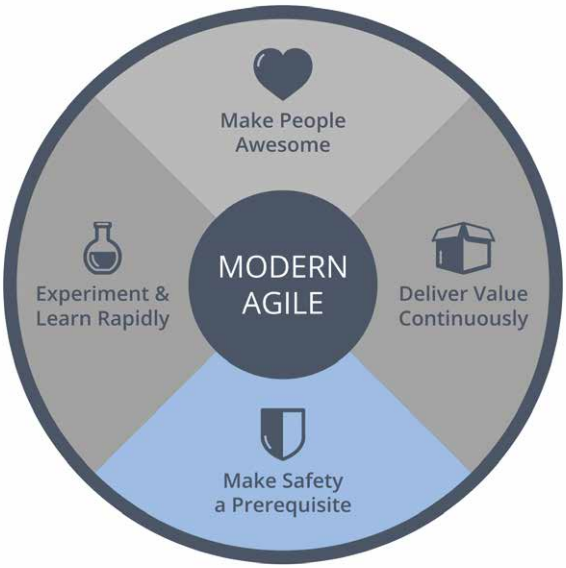
Bei „Make People Awesome“ geht es nicht um Spaß und Wohlfühlen. Ein Tischkicker oder eine aktuelle PlayStation im Büro, freie Getränke, Obst, das ist alles wichtig und gut. Aber irgendwann wird es als Selbstverständlichkeit erwartet, um kontinuierlich Geistesarbeit leisten zu können –wozu auch kreative und gesunde Pausen gehören. Nein, bei „Awesome“ geht es eher um Sinnhaftigkeit und Wertschätzung. Dass ich mit meiner Arbeit etwas bewegen kann. Dass ich ernst genommen werde mit meinen Ideen. Das macht dann auch Spaß und führt zu guten Produkten. Und das merken unsere Kunden.

Wie viele Chefs sagen „Ich nehme meine Mitarbeiter ernst“, aber wenn man die Mitarbeiter fragt, bekommt man ein ganz anderes Ergebnis? Wie viele Kunden fühlen sich von Händlern oder Dienstleistern nicht ernst genommen? Vor allem, falls es mal – unvermeidlich – Probleme gibt? Dagegen ein positives Beispiel: Nach 30 Jahren habe ich die Klangdaten für meinen alten Synthesizer noch auf der Herstellerwebseite als kostenlosen Download bekommen und konnte damit den Synthi wiederbeleben. Da merkt man, dass jemand stolz auf seine Produkte ist! Als Kunde bin ich begeistert und kaufe dort gerne wieder ein.

Denn worum es bei „Awesome“ auf jeden Fall geht, ist. dass sich alle großartig fühlen: Entwickler, Fachbereiche, das Management und natürlich die Anwender bzw. Kunden. Wenn wir nur faule Kompromisse eingehen, weil unsere Bürokratie und unsere Prozesse in realistischer Zeit nichts anderes zulassen, merken das die Kunden. Mit den Ergebnissen solch schlechter Kompromisse mögen wir ja selber nicht arbeiten – jeder von uns hat garantiert ein Produkt oder eine Webseite im Kopf, die hierzu passt. Wenn wir es aber schaffen, Kompromisse einzugehen, mit denen alle gut leben können, werden auch unsere Kunden merken, dass dort etwas mit Hingabe und der nötigen Sorgfalt gebaut wurde. Selbst wenn das Produkt dann noch nicht alles kann, aber das Vorhandene nützlich und gut ist. Während all das stark mit der äußeren Qualität (die, die der Kunde sofort sieht) zu tun hat, werden wir im Folgenden noch sehen, dass wir diese Qualität nur dann kontinuierlich gewährleisten können, wenn auch die innere Qualität unseres Produkts stimmt (das, was man als Kunde nicht oder nicht sofort sieht, sondern was vor allem uns Entwickler bezüglich Wartbarkeit und Änderbarkeit betrifft).

Warum stellt dieses Prinzip eine Weiterentwicklung des Agilen Manifests dar? Dort heißt es „Zusammenarbeit mit dem Kunden“. Diese Zusammenarbeit darf aber, wie oben beschrieben, nicht aus schlechten Kompromissen bestehen. Alle Seiten müssen sich großartig bei dem fühlen, was man zusammen erarbeitet. „Make People Awesome“ fasst genau das in Worte.

Make Safety a Prerequisite



Mach Sicherheit zur Grundvoraussetzung! Das kann die technische Sicherheit sein, dazu später mehr. Vor

allem aber geht es um psychologische, emotionale Sicherheit.

Ich muss mich trauen, offen und ehrlich zu sprechen. Ich muss das Vertrauen in meine Kollegen und das Zutrauen in mich selbst haben, dass meine Ideen und Vorschläge ehrlich und sachlich gehört und nicht als persönlicher Angriff verstanden werden. Ich muss entsprechend auch immer davon ausgehen, dass es die anderen nur gut meinen und mich nicht persönlich angreifen wollen. Wir müssen als Team Risiken abwägen und eingehen können, ohne unkritisch und leichtsinnig zu werden. Und weil wir Menschen in all dem nicht besonders gut sind, müssen wir diese Art der Kommunikation üben. Wenn das klappt, dann arbeiten wir ohne Angst zusammen, dann können wir zusammen unglaublich produktiv sein.

Psychologische Sicherheit ist die Grundvoraussetzung für erfolgreiche Hochleistungsteams – nicht in dem Sinne, dass sich alle totarbeiten, sondern dass das Team kontinuierlich Wertvolles liefert. Das hat Google in einer großangelegten, zweijährigen Studie mit über 180 Google-Teams herausgefunden. Im Ergebnis war es relativ egal, *wer* in einem Team arbeitete. Wichtiger war, *wie* das Team zusammenarbeitete [9]. Fünf Punkte haben sich dabei als relevant herausgestellt:

- Können wir uns aufeinander verlassen?
- Ist unsere Arbeit strukturiert, haben wir Klarheit über unsere Ziele, Rollen etc.?
- Erkenne ich den Sinn meiner Arbeit?
- Ist unsere Arbeit wichtig, hat sie öffentliche (Aus-) Wirkung?
- Am allerwichtigsten aber hat sich die psychologische Sicherheit herausgestellt. Ohne sie wurden die anderen vier Punkte irrelevant, ohne sie waren die Teams nicht erfolgreich.

Als typisches Negativbeispiel, das ich so schon häufiger erlebt habe, erklärt ein Abteilungsleiter: „Hier reden alle offen und ehrlich miteinander!“ Aber wenn man ein bisschen beobachtet, kann man Tuscheln und Gerede auf dem Flur hören, das verstummt, sobald der Abteilungsleiter in die Nähe kommt. Auf dem Flur wird über Schiefstände gelästert, in den Meetings scheint aber alles eitel Sonnenschein zu sein. Dort wird nur das gesagt, was der Abteilungsleiter hören möchte.

Das ist Konfliktvermeidung. Oder, vielleicht sogar gut gemeint vom Abteilungsleiter, ein Kuschelkurs, der die Angestellten watteweich in ihre Komfortzone einpackt. Damit werden wir uns aber kaum trauen, Neues auszuprobieren oder abschätzbare Risiken einzugehen. Das wird nur funktionieren, wenn ich ohne Angst aus meiner Komfortzone herauskommen mag. Angstfreiheit erkennen wir unter anderem daran, dass

- offene und ehrliche Gespräche geführt werden, auch wenn der Chef anwesend ist,
- es sachliche Suchen nach Fehlerursachen gibt, ohne Schuldzuweisungen und Fingerpointing,
- über die Sache gestritten wird, der Streit nicht ins Persönliche geht,
- alle Beteiligten Respekt und Wertschätzung füreinander zeigen, auch in herausfordernden und unklaren Situationen.

Wenn wir also merken, dass irgendeine Form von Angst in unserem Team existiert, sollten wir die Ursachen herausfinden, weil dem Team ansonsten die psychologische Sicherheit abhandenkommen (oder gar nicht erst aufgebaut werden) kann. Und wenn ein Angstklima längere Zeit anhält, sind die Mitarbeiter schnell überfordert, werden krank und kündigen innerlich – oder, insbesondere die besseren, auch explizit. Es gibt aber auch Ängste unserer Kunden, beispielsweise die Angst,

- ein Feature einzusetzen: Geht dabei die Formatierung im Dokument kaputt?
- Daten zu verlieren: Wenn ich das Musikstreaming aktiviere, wird dann meine lokale Musik gelöscht?
- alleingelassen zu werden: Wann erreiche ich den Support? Wo finde ich die Kontaktdaten?

Kundenängste sind oft ein Zeichen, dass an anderer Stelle im Projekt bzw. Unternehmen die Leitplanken von Modern Agile durchbrochen werden.

Trotz der Radikalität war den Autoren des Manifests daran gelegen, dass Softwareentwicklung nicht im Chaos versinkt.

Wie schafft man psychologische (emotionale) Sicherheit? Wie bereits angedeutet, geht es um die Art unserer Kommunikation. Google fing bei den Meetings an, die Kommunikation sicher zu machen, was sich dann nach und nach positiv auf andere Situationen auswirkte. Darauf aufbauend schlägt Modern Agile vor, mit Vereinbarungen für Meetings zu starten. Eine ziemlich brauchbare Liste solcher Vereinbarungen kann man sich herunterladen [10], aber natürlich sollen und dürfen wir sie an unsere Bedürfnisse anpassen.

Oder wir sorgen dafür, dass Konflikte gar nicht lange schwelen und dann irgendwann eskalieren, sondern dass wir Probleme sofort klären. Modern Agile schlägt dafür „Stop Work Authority“-Karten vor [11], die jedes Teammitglied hochhalten kann, wenn Gesundheit, Geld, Beziehungen, Reputation oder Informationen in Gefahr sind oder wenn unsere Zeit verschwendet zu werden droht – ganz im Sinne der Andon-(„Signal“-) Schnur im Toyota Production System (TPS), wo jeder Mitarbeiter das Fließband durch Ziehen einer Schnur

anhalten kann, wenn ein Problem auftritt, das er nicht allein innerhalb der Taktzeit lösen kann [12]. So können alle Kollegen, die für die Lösung des Problems erforderlich sind, mithelfen. Ziel ist, Probleme möglichst früh zu erkennen und zu korrigieren. Das Ziehen der Schnur ist kein Eingeständnis einer falschen Aktion, sondern bewahrt die Firma vor späterer Gefahr, z. B. hohen Kosten. Eigentlich kennen wir das genauso auch aus der Softwareentwicklung: Änderungen (und Fehlerkorrekturen) werden umso teurer, je später sie erfolgen. Modern Agile möchte uns dafür sensibilisieren, so früh wie möglich auf Probleme zu reagieren. Das ist ungewohnt, das muss man sich trauen, und das geht am besten durch Üben.

Wer die „Stop Work Authority"-Karten zu ungewohnt findet, kann erstmal damit starten, Probleme in der Zusammenarbeit, die im Laufe des Tages aufgetreten sind, möglichst zeitnah zu besprechen. Warum müssen wir damit zwei Wochen bis zur nächsten Retrospektive warten? Eine kurze, fokussierte „Mini-Retro" z. B. als gemeinsamer Tagesabschluss hilft dabei, gleich am nächsten Tag besser zusammenarbeiten zu können. Ich nutze das beispielsweise in zahlreichen Teams am Ende eines Mob-Programming-Blocks (oder -Tages), damit wir nicht nur die Aufgabe an sich lösen, sondern immer auch die diversen kleinen Missverständnisse und unterschiedlichen Wünsche klären. Das schafft Vertrauen und Sicherheit.

In den notwendigen Diskussionen sollten wir Konflikte als zusammenarbeitende Kollegen angehen, nicht als Gegner. Also: „Wie können wir das Problem für uns beide gut lösen?" Und nicht: „Du hast das für mich unbrauchbar umgesetzt." Vorwürfe und Kritik sollten wir durch Neugierde ersetzen: Den anderen ehrlich verstehen wollen, das Problem ehrlich verstehen wollen. Und Unterstützung anbieten, statt den anderen abzukapseln.

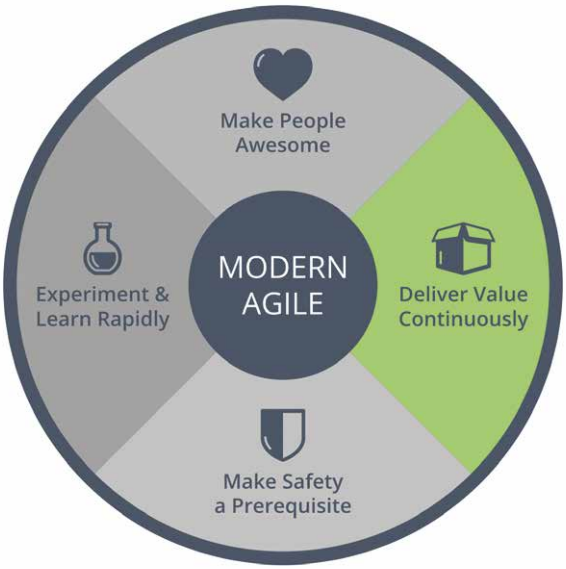
Technische Sicherheit –auch wenn das im Englischen eher mit „Security" bezeichnet wird – kann zur psychologischen Sicherheit beitragen, z. B. wenn ich keine Angst mehr habe,

- zu deployen – weil es genug automatisierte Qualitätsschranken in unseren Pipelines gibt.
- ein neues Feature zu aktivieren – weil ich die Nutzung und Funktionalität durch (fachliches) Monitoring beobachten kann; und weil ich das Feature bei Problemen schnell wieder deaktivieren kann.
- vor Hackerangriffen und Systemausfällen, weil regelmäßig wir Penetrationstests und Chaos Engineering durchführen, vielleicht sogar teilweise automatisiert.
- vor heute noch unbekannten Problemen, weil wir mit Pre-Mortems regelmäßig bekannte und unbekannte Situationen üben und dadurch Vertrauen in unsere Systeme und unsere Urteilskraft bekommen.

Technische Sicherheit betrifft auch Datenschutz und Verschlüsselung und wird daher von Initiativen wie „Security by Design" und DevSecOps als Grundvoraussetzung angesehen. Das ist übrigens auch die Sicherheit, die Kunden wahrnehmen und die sie – richtig umgesetzt – begeistert. Mein Lieblingsbeispiel hier zurzeit sind Cookie-Banner, die einfach und klar gestaltet sind und die nicht nerven.

Und verglichen mit dem Agilen Manifest? Dort ist von „Individuen und Interaktionen" die Rede. Individuen und Interaktionen können aber egoistisch und – absichtlich oder unabsichtlich – negativ sein. Psychologische Sicherheit als Grundvoraussetzung macht hier eine klarere Vorgabe, wie wir interagieren sollten.

Experiment & Learn Rapidly



Die spannende Frage ist, was die beiden oben beschriebenen Prinzipien mit Agilität zu tun haben. Das klingt alles eher nach gesundem Menschenverstand (wobei man auch damit vorsichtig sein muss). Auf jeden Fall scheint das Obige doch auch für „normale" Projekte gut zu sein? Ja, definitiv. Aber (modern) agil wird es durch die dritte Leitplanke „Experimentiere und lerne zügig".

Wie sieht es denn bisher oft aus, wenn ohne Hypothesen, Experimente, Auswertungen und die daraus resultierenden Anpassungen der Produktentwicklung gearbeitet wird? Ein leider nicht ganz fiktives Beispiel: Ein Fachbereich erarbeitet ein riesiges Lastenheft, es wird anschließend ewig daran entwickelt – und beim Big-Bang-Release stellt man dann fest, dass die Hälfte des Lastenhefts bereits obsolet ist und die Kunden von den übrigen Features viele gar nicht nutzen (hoffen wir, dass die relevanten wenigstens gut funktionieren).

Hier wird viel Zeit damit vergeudet, Unnötiges oder Falsches zu tun. Aber wie können wir den Kundenerwartungen besser gerecht werden, bzw. das realisieren, was der Kunde wirklich benötigt? Das Problem dabei ist, dass Kunden (Anwender) nicht sonderlich gut darin sind, Anforderungen im Voraus zu beschreiben. Wenn

wir ihnen aber ein teilweise fertiges Feature zeigen und es sie benutzen lassen, können sie uns sofort sagen, was daran gut ist und was nicht. Oder aber wir beobachten die Kunden durch fachliches Monitoring dabei, welche Features wie stark genutzt werden, und passen mit diesen Erkenntnissen die weitere Planung an. Wir machen also vom Verhalten der Kunden bei der Benutzung eines ausreichend nutzbaren Features abhängig, ob dieses Feature künftig verstärkt entwickelt, umgebaut oder im Zweifel sogar zurückgebaut wird, um die Codebasis nicht mit ungenutzten Features vollzumüllen. Gewisse Punkte können wir also im Voraus gar nicht vernünftig planen, sondern sollten besser Experimente definieren, mit denen die Planung im Verlauf konkretisiert werden kann.

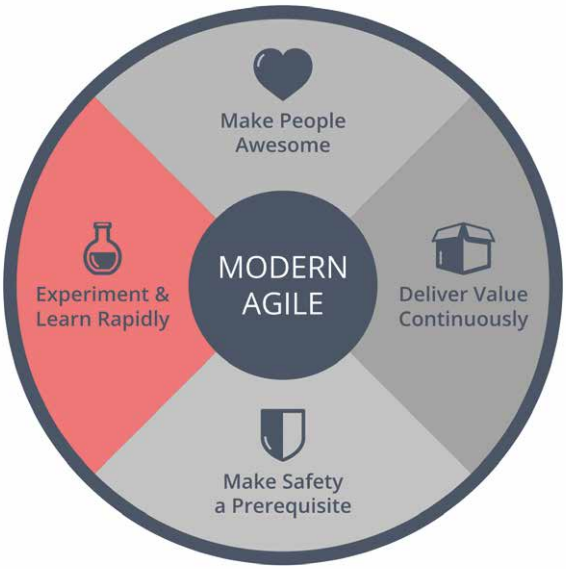
Ein Experiment kann sein, ob ein Feature überhaupt angenommen wird. Oder wir verproben zwei verschiedene Lösungen gegeneinander (A/B-Tests). Und die Experimente müssen auch nicht immer alle Kunden betreffen, sondern wir könnten sie auf definierte oder zufällige Testgruppen beschränken.

Es ist aber ein riesiger Unterschied, ob wir in der Entwicklung auf Experimente vorbereitet sind oder aber nicht. Wenn wir unsere Features mit Hypothesen, dazu passenden Experimenten und deren Auswertung weiterentwickeln – und immer die normale Möglichkeit akzeptieren, dass ein Experiment ganz anders verläuft als erwartet –, dann ist das ein viel klareres und risikoärmeres Vorgehen als Anforderungen zu definieren, diese dann exakt umzusetzen und anschließend zu hoffen, dass alles passt (oder Hotfixes einzuspielen). Und wenn ich (und der Fachbereich) zudem den Verlauf des Experiments beobachten kann, gibt das Sicherheit. Weil man das Experiment jederzeit abbrechen kann, wenn es unerwartet problematisch verläuft.

Wir können aber nicht nur mit den Features experimentieren. Wir können auch mit der eigenen Arbeitsweise experimentieren und schnelleres Lernen lernen! Wie schon erwähnt, machen wir am Ende von Mob-Programming-Sessions gerne eine Mini-Retro, um regelmäßig auf die Stunden intensiver Zusammenarbeit zurückzublicken und uns in kleinen Schritten zu verbessern. Das kann man auch erreichen, indem man etwas Funktionierendes bewusst abändert, z. B. Anzahl und Dauer der Pausen. Gestaltet man das als bewusstes Experiment und wertet es in einer der nächsten Mini-Retros gemeinsam aus, ist man entweder positiv überrascht, dass man auch anders gut arbeiten kann, oder aber man sieht, dass die Änderung nicht so gut funktioniert und nimmt sie wieder zurück. Solche Experimente und Mini-Retros kann man natürlich auch dann durchführen, wenn man noch kein Mob-Programming einsetzt.

Das Agile Manifest spricht von „Reagieren auf Veränderung“. Aber wie sollen wir reagieren? Modern Agile wird wieder konkreter – das Lernen aus den Ergebnissen unserer Experimente zeigt uns den Weg.

Deliver Value Continuously



Kontinuierlich wertvolle Ergebnisse liefern – das betrifft natürlich das Produkt, das wir zum Kunden ausliefern. Erst dann bekommen wir wirklich Feedback, ob das, was wir gebaut haben, nützlich für den Kunden ist. Auch Erkenntnisse können wertvolle Ergebnisse sein, ebenso wie Verbesserungen (z. B. der Codebasis durch Refactorings). Wichtig ist, dass es kontinuierlich geschieht. Nicht viermal pro Jahr ein Deployment oder einmal pro Monat ein großes Refactoring oder einmal pro Jahr eine „Qualitätsoffensive“. Besser häufiger und bevorzugt in kleinen Schritten. Retrospektiven dann, wenn wir sie brauchen. Deployments automatisch bei jedem Commit.

Was hindert uns daran? Wenn wir uns diese Frage beantworten, können wir uns Schritt für Schritt auf den Weg machen. Denn wir werden vermutlich nicht sofort von Quartalsreleases auf kontinuierliche Deployments umstellen. Aber wir können nach und nach die Blockaden beseitigen. Kann ich einfach und sicher deployen, oder ist dafür viel Formalismus notwendig? Habe ich Angst, Code zu ändern, weil ich die Nebenwirkungen nicht abschätzen kann?

Wird das kontinuierliche Liefern durch unser Vorgehen bzw. Prozess unterstützt? Beispiel Scrum: Sind Zwei-bis-vier-Wochen-Iterationen eine Verbesserung für uns bzw. immer noch gut genug? Oder sollten wir Ergebnisse besser immer dann schon ausliefern, wenn sie fertig sind? Helfen uns die üblichen Meetings wie Planning, Estimation und Review sowie das strikte Beachten einer Definition of Ready (DoR) und Definition of Done noch dabei, einen geordneten Ablauf und Rhythmus in unsere Softwareentwicklung zu bekommen? Oder haben wir in den vergangenen Monaten und Jahren genug durch sie gelernt und könnten ihre Inhalte und Ziele viel fließender, kontinuierlicher in unseren Arbeitsalltag einbauen?

Und was uns häufig ausbremst: Fehlendes Wissen und Können. Wir versuchen, das Erlernte, das implizite Wissen, das die einzelnen Teammitglieder erlangt haben,

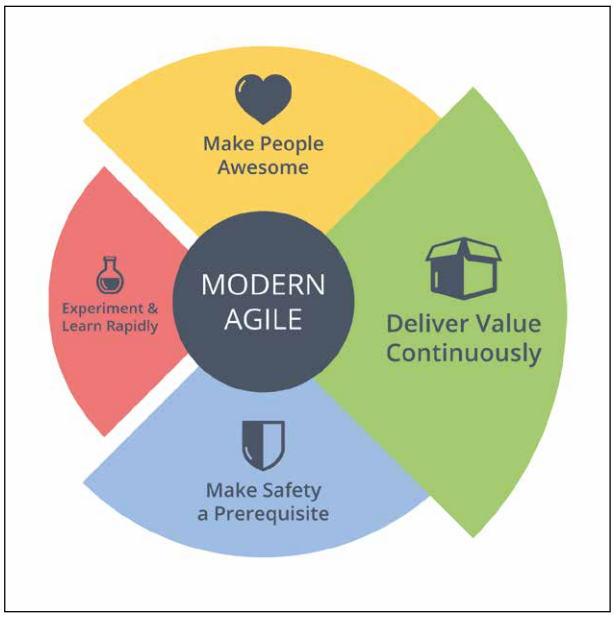


Abb. 2: Eine Unwucht kann leicht zu Holpern und Problemen führen

nachträglich im Team zu verteilen und z. B. als Dokumentation schriftlich weiterzugeben. Aber während gewisse Dinge gut dokumentiert werden können (fachliche Übersichten und solche für die Architektur), ist es unglaublich schwierig, Können auf diese Weise weiterzugeben. Und wenn wir dann für den Anderen einspringen müssen oder uns gegenseitig unterstützen wollen, stellen wir fest, dass wir vielleicht theoretisches Wissen haben, dieses aber nicht praktisch in unserer Systemlandschaft einsetzen können.

Was hilft uns dabei, kontinuierlicher Wertvolles liefern zu können? Technische und menschliche Aspekte. Wir werden das Bauen und Ausliefern unserer Software nach und nach automatisieren (z. B. mit Pipelines), um über Continuous Delivery (mit manuellen Auslösern für das Deployment) zu Continuous Deployments (vollautomatische Auslieferung) zu gelangen. Und selbst, wenn wir Letzteres nicht komplett erreichen, werden wir auf dem versuchten Weg dorthin viele wertvolle Erkenntnisse erlangt haben, wie wir unsere Arbeitsweise verbessern können. Und wir sollten versuchen, das Wissen und Können gemeinsam aufzubauen, statt es hinterher mühsam zu verteilen. Das ist auch der Grund, warum im agilen Umfeld so viel mit Pair- und zunehmend mit Mob-Programming gearbeitet wird: Wir verhindern Wissenssilos in den Köpfen, denn wir sind *ein* Team mit N Kollegen, nicht N Teams aus Einzelnen [13]! Und wir machen „Pull“ statt „Push“, ganz im Sinne der Lean-Prinzipien: Wir machen erst gemeinsam Dinge fertig, bevor wir zig neue Dinge parallel anfangen. Nebenbei passt das zur Limitierung vom „Work in Progress“ (WIP) bei Kanban. Es erzeugt einen „Flow“ von Wertvollem (statt großer Arbeitspakete, die viel zu selten fertig werden, und dann mit dem Risiko großen Scheiterns).

„Deliver Value Continuously“ geht nur bei kontinuierlichem Fokus auf innere Qualität – und das ist der Anknüpfungspunkt zu den Praktiken des Extreme Programming bzw. vom Software Crafting (Georg Berky geht in seinem Artikel „Cruft oder Craft“ genauer darauf ein). Es geht bei unserer Arbeit nicht nur um Featureentwicklung, sondern auch um Bug Fixes, regelmäßige Standardaufgaben, Unterstützung anderer Teams, Wartungsarbeiten im Altsystem, Refaktorisierung, Lernen usw. Und das kontinuierlich und in kleinen Schritten.

Das Agile Manifest stellte „funktionierende Software“ in den Fokus, was damals eine wichtige Verbesserung war. Modern Agile entwickelt dieses Konzept nun so weiter, dass wir kontinuierlich Wertvolles ausliefern müssen. Natürlich Software, aber eben nicht nur.

Läuft das Rad rund?

Wer nun das Gefühl hat, doch eigentlich die Leitplanken von Modern Agile schon zu verwenden, aber unsicher ist, ob die eigene Umsetzung gut genug ist, kann sich anhand des Modern-Agile-Rads die Frage stellen: „Läuft das Rad rund?“ (**Abb. 2**). Probleme entstehen meist dann, wenn die einzelnen Bereiche nicht ausbalanciert bzw. ausgewogen sind, wenn das Rad eine „Unwucht“ hat – dann wird das Vorgehen bzw. der Entwicklungsprozess holpern. Ein paar Beispiele:

- Wenn das Ausliefern technisch hervorragend klappt, aber keine Sicherheit vorhanden ist, haben wir vielleicht Angst, die interessanten und relevanten Experimente durchzuführen, denn ein „fehlgeschlagenes“ Experiment könnte uns zum Vorwurf gemacht werden.
- Wenn das Ausliefern schnell geht, aber viel schneller, als wir lernen können: Ist das nicht Verschwendung? Was machen wir mit den Ergebnissen? Es kann gefährlich werden, wenn wir mit den Kunden experimentieren, ohne angemessen auf deren Rückmeldungen zu reagieren. Dann bleiben die Kunden im schlimmsten Fall auf der (Deployment-)Strecke.
- Wenn wir ein perfektes Klima der Sicherheit haben, aber nichts ausliefern können: Wem nützt das? Die Kunden sind davon sicher nicht beeindruckt. Und für uns Entwickler ist das auch ziemlich frustrierend, denn wir wollen Software entwickeln, die auch wirklich genutzt wird.

Gerade am Anfang werden wir vermutlich zahlreiche Unwuchten finden. Aber auch später, wenn eigentlich alles schon mal gut lief, werden immer wieder Bereiche in ein Ungleichgewicht kommen. Modern Agile ist dafür ein Problemsensor, ein Messinstrument, das uns hilft, aufkommende Probleme rechtzeitig zu entdecken und gegenzusteuern. Wenn uns die Prinzipien von Modern Agile mehr Probleme zeigen als uns lieb ist, sollten wir das als Chance sehen, die Probleme Schritt für Schritt zu lösen. Denn immerhin wissen wir dann von unseren Baustellen und erfahren nicht erst dann davon, wenn alles zu spät (und das Produkt bzw. die Firma kaputt) ist.

Wenn ein Prinzip deutlich stärker ausgeprägt ist als die anderen drei, kann es eventuell sinnvoll sein, dieses Prinzip ein bisschen zurückzufahren, z. B. wenn wir schon viel mehr experimentieren, als unser Team bzw. Produkt derzeit verkraftet. Denn wir sollten lieber mit einem kleinen, dafür runden Rad beginnen und dieses dann nach und nach größer werden lassen, als dass wir zu früh ein großes Rad haben, das ständig holpert.

Programmierpraktiken und Umsetzung

Modern Agile hat bewusst vier recht allgemeine Prinzipien gewählt. Aber für die Umsetzung dieser Ideen brauchen wir konkretere Hilfestellungen, beispielsweise in Form von (Programmier-)Praktiken. Anders als das Extreme Programming gibt Modern Agile aber keine Programmierpraktiken vor. Das ist zum einen sinnvoll, weil die Ideen von Modern Agile in möglichst vielen Bereichen anwendbar sein sollen. Zum anderen ändern sich Praktiken im Laufe der Zeit, während die Grundideen (Prinzipien) eher gleichbleiben.

Dennoch muss Modern Agile natürlich in jedem Bereich konkret ausgestaltet werden. In der Softwareentwicklung sind derzeit die folgenden Praktiken hilfreich:

- *Continuous Deployments:* Wir streben die vollständig automatisierte Auslieferung als Ziel an. Ob wir das wirklich erreichen oder – wie bei Continuous Delivery – noch ein paar manuelle Schritte beibehalten, ist fast nebensächlich. Denn allein durch den Versuch, unsere Auslieferungskette so weit wie möglich zu automatisieren, werden wir sehr viel kontinuierlicher, als wir das anfangs für möglich gehalten haben.
- *Trunk-based Development mit Feature-Toggles (Feature-Flags):* Alle Entwickler arbeiten auf dem Haupt-Branch, und die separaten Features werden hinter Schaltern versteckt, mit denen wir die Features einschalten können (z. B. ist ein Feature in der Testumgebung schon aktiv und wird in der Produktionsumgebung erst nach der fachlichen Abnahme eingeschaltet). Damit entkoppeln wir Deployment- und Releasezeitpunkt, was uns wiederum kontinuierlicher ausliefern lässt.
  Es ist eine ewige Diskussion, ob man Branches verwendet oder nicht, und man kann das nicht abschließend klären. Aber Branches führen häufig zu späterer Integration von Features in den Haupt-Branch, während Trunk-based Development frühestmögliche Integration ist – was uns frühzeitiges Feedback liefert. Man kann auch mit kurzlebigen Branches einigermaßen kontinuierlich arbeiten, aber die Gefahr und Versuchung sind vorhanden, den Branch länger als nötig offenzuhalten. Trunk-based Development zwingt uns dazu, frühestmöglich und kontinuierlich zu integrieren.
- Aber: Warum sollten Entwickler überhaupt separat an Features in separaten Branches arbeiten? Wir haben schon gesehen, dass wir das Wissen und Können zur Lösung einer Aufgabe besser gemeinsam aufbauen, daher sind Pair- und vor allem Mob-Programming wichtige Programmierpraktiken für Modern Agile. Und wenn man Features ohnehin zusammen entwickelt, ist Trunk-based Development fast die natürliche Konsequenz.

Ansonsten sind natürlich weiterhin alle Programmierpraktiken hilfreich, die man vom Extreme Programming und aus dem Software Crafting kennt (Test-driven Development, Refactorings, Clean Code etc.)

Modern Agile betrifft aber eben nicht nur die reine Programmierung, sondern die Zusammenarbeit, das Vorgehen und die Produktentwicklung. Für all diese Bereiche wurden auf der Modern-Agile-Webseite Tipps gesammelt, die man dort als „Cheat Sheets“ herunterladen kann [14]. Und in der „Modern Agile Show“ (verfügbar auf YouTube und als Podcast [15]) diskutiert Joshua Kerievsky mit zahlreichen Gästen die einzelnen Aspekte von Modern Agile.

Spannenderweise gibt es meines Wissens immer noch kein Buch, das sich Modern Agile explizit widmet. Aber es gibt zahlreiche Bücher, die die einzelnen Aspekte der Modern-Agile-Prinzipien im Detail behandeln. Drei Bücher möchte ich hervorheben:

- „Release It“ [16] ist der Klassiker (aktualisiert in einer neuen Auflage) zur technischen Umsetzung einer stabilen und kontinuierlichen Auslieferung.
- „Testing Business Ideas“ [17] beschreibt das Prinzip „Experiment & Learn Rapidly“ und berücksichtigt dabei auch die Softwareentwicklung.
- „Sooner Safer Happier“ [18] kümmert sich um die „Business Agility“, ist also auf Organisationslevel angesetzt und beschreibt Aspekte wie Kulturwandel, Flow, technische Exzellenz, Lernen etc. Damit ist es das umfassendste Buch, das vieles von dem beschreibt, was hinter den Prinzipien von Modern Agile

steckt. Durch das prägnante Kürzel #BVSSH („Better Value Sooner Safer Happier“) umgeht es auch den überladenen Begriff „Agile“.

Fazit

Modern Agile entwickelt das Agile Manifest weiter und macht die Ideen fit für die kommende Dekade. Dabei wird nicht das Rad neu erfunden, sondern Bewährtes gut zusammengestellt. So kann Modern Agile auf jahrelange Erfahrung zurückgreifen, die mit anderen Denkansätzen gesammelt wurde. Beispielsweise passen die beiden Achsen des Modern-Agile-Rads hervorragend zu den zwei Säulen des „Toyota Way“ [19], der großen Einfluss in der Lean-Community hat – zum einen zur Säule „Respect for People“ durch die zwei Leitprinzipien „Make People Awesome“ und „Make Safety a Prerequisite“, zum anderen zur Säule „Continuous Improvement“ durch „Experiment & Learn Rapidly“ und „Deliver Value Continuously“. Wenn sich ganz unterschiedliche Bereiche und Branchen ähnlich ausrichten, kann das so verkehrt nicht sein.

Warum es wichtig ist, Altbewährtes an die Zeit anzupassen und für die Zukunft fit zu machen, hat einen einfachen Grund: Viele von uns kennen Lean, Kanban, Scrum, Extreme Programming etc. – aber wir sollten es nicht als bekannt voraussetzen. Jedes Jahr kommen zunehmend mehr neue, unerfahrene Softwareentwickler und Projektleiter auf den Markt (bzw. in die Firmen). Wenn sie Leitplanken an die Hand bekommen, mit denen sie besser lernen können, moderne Software zu entwickeln, ist uns letztlich allen geholfen – weil wir ihre Kollegen, Chefs, Mitarbeiter oder Kunden sind. Und die Leitplanken von Modern Agile sind gleichsam einfach, verständlich und realistisch. Das macht Mut, sie umzusetzen.

Ein Vorgehen mit den Leitplanken von Modern Agile ermöglicht also das, was viele Firmen derzeit versuchen (Achtung, Buzzwordgefahr):

- Innovativ sein,
- sich selbst immer wieder neu erfinden,
- einen Kulturwandel für die Digitalisierung schaffen und
- alle dabei mitnehmen.

Das geht eben nur, wenn wir Begeisterung mit Sicherheit schaffen und wenn auch die technischen Voraussetzungen für die Kontinuität gegeben sind.

Aber: Das ist nicht einfach, weil es kein Patentrezept gibt. Weil jedes Team, jede Abteilung, jede Firma individuelle Anpassungen für sich finden muss. Und das immer wieder neu. So ein Umdenken kann man gerade in größeren Unternehmen nicht in zwei oder drei Jahren umsetzen – und vor allem nicht „fertig“ bekommen. Das wird nicht „ausgerollt“. Der neue Ansatz muss gemeinsam über Jahre gelebt werden, von entscheidenden Personen (auch von solchen möglichst weit „oben“) *vor*gelebt werden. Das ist Aufwand. Aber wer nicht nur Standardaufgaben erledigen, sondern auch Neues bauen muss, für den kann sich das schnell lohnen. Und es ist auch ganz im Sinne der Präambel des alten agilen Manifests: „Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen.“ [2]

Alles entwickelt sich weiter, auch die Ideen zu gelungener Agilität. Und wer weiß, ob wir in ein paar Jahren noch von Modern Agile sprechen. Namen sind Schall und Rauch. Vielleicht sollten wir uns wirklich komplett von dem Begriff „Agile“ lösen. Vielleicht nennen wir die Ideen demnächst „(Better Value) Sooner Safer Happier“ [18]. Aber die Fragen, die uns Modern Agile mit auf den Weg gibt, die sind gut und werden uns sicher noch eine ganze Zeit begleiten.

Thomas Much ist freiberuflicher Softwareentwickler und Coach für agile Programmierpraktiken (Pair Programming, Mob Programming, TDD etc.). Er lebt mit seiner Familie in Hamburg und twittert unter @thmuch.

Links & Literatur

[1] https://agilemanifesto.org/history.html

[2] https://agilemanifesto.org/iso/de/manifesto.html

[3] https://stateofagile.com

[4] https://ronjeffries.com/articles/018-01ff/abandon-1/

[5] https://jaxenter.de/scrum-programmer-anarchy-agile-77081

[6] https://www.industriallogic.com/blog/modern-agile/

[7] https://www.agilealliance.org/resources/videos/modern-agile/

[8] https://modernagile.org

[9] https://rework.withgoogle.com/blog/five-keys-to-a-successful-google-team/

[10] https://modernagile.org/goodies/ma-meetingAgreements.zip

[11] https://modernagile.org/goodies/swa.zip

[12] https://blog.toyota.co.uk/andon-toyota-production-system

[13] https://github.com/joshbruce/modern-agile-cutler/blob/master/one-sheet.pdf

[14] https://modernagile.org/#cheatsheets

[15] https://modernagile.org/#learnMore

[16] Nygard, Michael T.: „Release It!“: Pragmatic Programmers, 2018

[17] Bland, David J.; Osterwalder, Alex: „Testing Business Ideas“; Wiley, 2020

[18] Smart, Jonathan et al.: „Sooner Safer Happier“; IT Revolution, 2020

[19] https://www.toyota-europe.com/world-of-toyota/this-is-toyota/the-toyota-way

Observability bewahrt Kontinuität und fördert Wachstum

Das A und O

Klassisches Monitoring reicht für den Geschäftserfolg von Unternehmen schon längst nicht mehr aus und ist ein Tech-Segment im Wandel. Das neue Zauberwort lautet „Observability". Damit sollen Firmen abteilungsübergreifend umfassende Echtzeiteinsichten erhalten. Dadurch lassen sich Probleme schneller identifizieren und beheben sowie Arbeitsprozesse und Produkteinführungen optimieren und verkürzen.

von Klaus Kurz

Die Entwicklung neuer Produkte und Funktionen, während bestehende Systeme gleichzeitig zuverlässig arbeiten sollen, stellt Unternehmen oft vor Herausforderungen. Die steigende Nachfrage nach digitalen Diensten während der Corona-Pandemie hat gezeigt, wie notwendig System-Observability und neue Betriebskulturen sind, um zuverlässiges Wachstum zu ermöglichen.

Unternehmen strotzen vor Ideen, doch die Umsetzung erweist sich oft als Herausforderung. Um in einem digitalen Kontext neue Erfahrungen, Dienstleistungen oder Produkte zu schaffen, muss unterstützender Softwarecode in großem Umfang neu geschrieben oder müssen völlig neue Systeme eingerichtet werden. Beide Optionen bergen das Risiko von Betriebsproblemen. Wachsen Organisationen schnell, stehen sie vor ähnlichen Herausforderungen. Die Systeme werden angesichts der großen Datenmengen und der zunehmenden Belastungen stark beansprucht. Technologieteams müssen umfangreiche Änderungen an wesentlichen Systemen vornehmen, was zusätzliche Systemlatenzen und Ausfälle verursachen kann.

Solche Änderungen können problematische Auswirkungen haben, wie die Gefahr von neuen Fehlern, Engpässen und Ausfällen. In vielen Unternehmen ändern Entwickler den Code nach Bedarf, testen ihn und geben ihn an die Betriebsteams weiter. Das Betriebspersonal wiederum ist ganz auf Validierung und Kontinuität ausgerichtet und versteht möglicherweise nicht einmal alle Gedanken hinter dem neuen Code. So kann die interne Dynamik innerhalb von Organisationen schnell zu Komplikationen führen und die Innovation oder das Wachstum in ihren Bahnen stoppen.

Einfluss von COVID-19

Die Corona-Pandemie hat viele dieser Probleme erheblich verschärft, weil Unternehmen ihre Geschäftstätigkeit in großem Umfang auf digitale Kanäle verlagert haben. Fortschrittliche Betreiber erlebten einen enormen Anstieg der Nachfrage nach Onlinediensten, während dem stationären Handel keine andere Wahl blieb, als für Web und Mobile zu werben und sich darauf zu verlassen. Als Folge sind Änderungen an Software- und Server-Set-ups sowie an ganzen Technologiearchitekturen auf der strategischen Agenda nach oben geschossen.

Mit diesem Druck wurden Systemschwächen schnell aufgedeckt. Für viele Unternehmen war es eine Offenbarung, wie gut oder schlecht ihre Systeme laufen und wie schwierig es sein kann, die Ursachen zu finden. Unternehmen sahen sich mit der Notwendigkeit konfrontiert, bessere oder neue digitale Kundenerlebnisse zu schaffen oder eine Verdreifachung des Webverkehrs zu bewältigen.

Der Weg zur Lösung

Unternehmen stehen unter dem Druck, zu verstehen, wo ihre Schwächen im Bereich Betrieb und Skalierbarkeit liegen. Auf der Suche nach einer Lösung wenden sich viele Organisationen der Observability (**Abb. 1**) zu, einem neuen Ansatz, um ein vollständiges Bild des Systembetriebs zu erhalten – einschließlich dessen, was passiert, wenn Code geändert oder aktualisiert wird

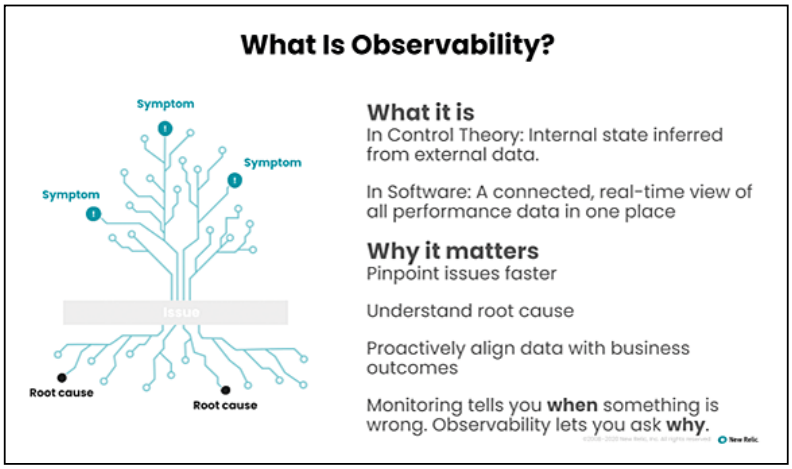


Abb. 1: Was ist Observability?

Abb. 2: Wie sieht es bei Ihnen aus?

oder neue Anwendungen in Betrieb genommen werden. Observability greift typischerweise eine Reihe von Problemen auf, die gerne übersehen werden, z. B. Probleme, die durch komplexe Systeme verursacht werden, die fast nahtlos funktionieren und nur marginale Probleme haben. Solche Ungenauigkeiten neigen dazu, sich zu akkumulieren und zu größeren Ausfällen zu führen.

Observability besteht aus drei Schlüsselelementen: Telemetrie, Kontextualisierung und Verständnis. Telemetrieagenten sammeln Daten jeder Komponente des digitalen Diensts eines Unternehmens. Im Kontext werden die Daten angereichert und korreliert, um ein vollständiges Verständnis des Systemverhaltens aufzubauen. Mit Hilfe dieser Informationen setzt die Technologie die Visualisierung ein, sodass Unternehmen die Daten verstehen, sie abfragen und darauf basierend wissen, wie sie Verbesserungen schnell umsetzen können.

Je komplexer der Aufbau, desto wichtiger ist es, die Vorgänge im Detail zu verstehen. New Relic One, die größte und umfassendste Observability-Plattform der Branche, liefert diese detaillierten Einblicke. Um das Maximum aus einer Hochleistungsmaschine herauszuholen, muss man verstehen, was sie tut. Genau das ist es, was Observability Unternehmen ermöglicht, indem sie einen umfassenden Überblick über die wichtigsten Daten schafft. So können Probleme schnell gelöst und Innovationen in einem rasanten Tempo vorgenommen werden, ohne an Zuverlässigkeit einzubüßen.

Aufbau einer neuen Kultur

Da Unternehmen diese Anpassungen vornehmen und in einem sich schnell verändernden Umfeld erfolgreich sein wollen, gibt es einen zunehmenden Bedarf nach kulturellem Wandel, um die Innovation und Robustheit der Systeme zu unterstützen. Dafür müssen Unternehmen sich auf Dienstleistungen und Nutzererfahrungen konzentrieren, die sich auf qualitativ hochwertige Daten stützen.

Für Unternehmen gibt es eine Reifekurve, wenn sie lernen, besser zu verstehen, wie ihre Systeme funktionieren und wo Veränderungen am nötigsten sind (**Abb. 2**). Observability wird i. d. R. zuerst von Entwicklungs-, Betriebs-, IT- und Infrastrukturteams genutzt, die basierend auf den Daten fundierte Entscheidungen treffen und klar darstellen können, welchen Beitrag sie zur Erreichung der Geschäftsziele leisten. Dadurch werden der Mehrwert und die Vorteile auch für andere Abteilungen deutlich. Oft machen sich dann auch andere Teams Observability zunutze.

Zunehmend setzen Produktions-, Marketing- und Finanzteams Observability ein, um die Systemfunktionalität mit Geschäftszielen wie Nutzung von Dienstleistungen oder Produkten und ihren Kosten zu korrelieren. In den fortschrittlichsten Unternehmen greift sogar der Geschäftsführer auf die Dashboards zu, um schnell Einblicke in die tägliche Leistung zu erhalten.

Observability in der Praxis

Etwa 94 Prozent der etabliertesten Unternehmen sehen Observability als einen Schlüsselaspekt der Entwicklung. Sie integrieren Endnutzer-Performance-Daten mit der Systemleistung, um die Auswirkungen von Änderungen zu verstehen [1]. Ermöglicht wird diese Arbeit durch maschinelles Lernen und künstliche Intelligenz, die menschliche Prozesse erweitern, Rauschen reduzieren und wichtige Schlussfolgerungen ableiten.

Für alle Unternehmen bietet Observability einen genauen Blick auf ihre eigenen Plattformen und schafft Klarheit darüber, wo Verbesserungen vorgenommen werden können und wie Probleme gelöst werden müssen. Zu den Organisationen, die mit der Observability-Software von New Relic arbeiten, gehört beispielsweise Delivery Hero. Das 2013 gegründete Onlinebestellportal für Essen bearbeitet rund zwei Millionen Bestellungen pro Tag. Dabei vertraut das Unternehmen auf New Relic, um Vorgänge in IT-Infrastruktur und -Applikationen im Blick zu behalten. Zu Beginn galt das Monitoring nur der Anwendungsverfügbarkeit und -performance. Inzwischen sind auch geschäftskritische KPIs Teil des Monitorings, wie etwa die Anzahl der Restaurants nach Gebiet oder die Anzahl der Bestellungen pro Land oder Plattform. Delivery Hero hat mit New Relic One Echt-

zeiteinsichten in Performanceprobleme und kann diese so schnell beheben, ehe sie sich auf die Kundenzufriedenheit auswirken. Die Dashboards wurden dafür mit Bezug auf Dutzende Unternehmenskennzahlen entwickelt und geben auch während kritischer Migrationen Einsichten in Echtzeit. Dadurch kann Delivery Hero seine Infrastruktur in einer Zeit schnellen Wachstums und zahlreicher Akquisitionen stets ausbauen und optimieren. Das ist nicht nur gut für die Kundenzufriedenheit, sondern auch für die Geschäftszahlen: Die unterstützte Kubernetes-Cluster-Optimierung senkte die laufenden Kosten um 71 Prozent:

Für alle Organisationen bietet Observability einen scharfen Blick in ihre eigenen Plattformen. Sie schafft Klarheit darüber, wo Verbesserungen vorgenommen werden können und wie Probleme gelöst werden müssen. Jede Organisation, die nach Innovation und schnellem Wachstum strebt und gleichzeitig die Zuverlässigkeit aufrechterhalten und die Erfahrungen der Benutzer verbessern möchte, kann von der hohen Sichtbarkeit und den umsetzbaren Erkenntnissen erheblich profitieren.

New Relic One

Mitte 2020 hat New Relic seine Plattform New Relic One strategisch weiterentwickelt, um sie noch kompakter und intuitiver zu gestalten, damit Unternehmen perfektionierte Software entwickeln können. Dabei lag der Fokus auf Full-Stack Observability, Applied Intelligence und einer Telemetry Data Platform.

- Die Full-Stack Observability unterstützt Unternehmen durch eine kompakte und konsolidierte Analyse und Problembehebung des gesamten Softwarestacks – egal ob APM, Infrastruktur, Logs oder digitales Kundenerlebnis und unabhängig von Entwicklungsmethode und -umgebung. So sind derzeit neun Softwaresprachen und über 300 Cloud- und Softwareintegrationen beinhaltet.
- Die Telemetry Data Platform ist eine zentrale Informationsquelle für alle Daten zu operativen Zusammenhängen. Damit können Petabytes aller Arten von Anwendungs- und Infrastrukturdaten erfasst und visualisiert werden. Eine Bewertung der Daten erfolgt durch Alerts.
- Applied Intelligence unterstützt dabei, Anomalien proaktiv zu erkennen und zu analysieren. Zudem kann damit die „Alert Fatigue“ reduziert und Probleme nach Relevanz priorisiert sowie Incidents schneller diagnostiziert und adressiert werden.

Preisstruktur und Verfügbarkeit

Observability als Tool sollte keinem Entwickler vorenthalten bleiben. Deshalb stellt New Relic allen Interessierten eine kostenlose Version der neuen Plattform zur Verfügung. Darin enthalten sind unbegrenzte Basic-Nutzer der Telemetry Data Platform und insgesamt 100 Gigabyte monatlicher Datenerfassung unabhängig von der Quelle. Darüber hinaus hat ein Standardnutzer kostenlos Zugriff auf Full-Stack Observability. Im Bereich der Applied Intelligence sind 100 Millionen monatliche proaktive Detection-Transaktionen und 1 000 monatliche Incident-Intelligence-Ereignisse gratis. Erst wenn dieses Leistungsvolumen ausgeschöpft ist, kosten weitere Gigabytes erfasster Daten auf der Telemetry Data Platform je 0,25 US$. Zusätzliche Transaktionen oder Ereignisse im Bereich der Applied Intelligence sowie Nutzerlizenzen für die Full-Stack Observability können ebenso bezogen werden. Um herauszufinden, wie Observability für bessere Software, zuverlässige Kontinuität und schnelle Innovation genutzt werden kann, lohnt sich ein Blick auf [2].

Klaus Kurz ist seit Herbst 2019 Director Solutions Consulting Central Europe bei New Relic. Die Projekte, bei denen Kunden auf Klaus Kurz und die Experten in seinem Team setzen, reichen von Cloud-Migration über Cloud-native bis hin zu DevOps und AIOps. Ziel ist dabei stets, das Nutzererlebnis der Kunden zu verbessern, die Transparenz von IT-Prozessen zu steigern und ein lückenloses End-to-End-Monitoring zu ermöglichen.

Links & Literatur

[1] https://newrelic.com/more-perfect-software/more-perfect-software/

[2] https://newrelic.com/de

EnterpriseTales

Wie Cloud-Computing ein Umdenken der Entwickler einfordert

von Lars Kölpin-Freese

Cloud-Computing ist ein Trend, der Auswirkungen für Entwickler hat. Statt dass Serversysteme wie gewohnt implementiert werden können, sind Systeme, die für die Cloud entwickelt werden, aufgrund der Skalierbarkeit häufig verteilte Systeme. Und das hat Implikationen, die wir uns in dieser Ausgabe der Kolumne anschauen wollen.

Cloud-Computing ist ein nicht wegzudenkender Trend in der heutigen Zeit. Der Begriff Cloud-Computing ist so facettenreich, dass man eigentlich gar nicht per se von Cloud-Computing sprechen kann. Plattform as a Service, Function as a Service, Kubernetes, Software as a Service – alles Terminologien, die unter den Begriff des Cloud-Computing fallen können. Geht ein Unternehmen in die Cloud, ist es also gar nicht so eindeutig, was eigentlich gemeint ist.

Es lässt sich dennoch eine Gemeinsamkeit festhalten: Cloud-Computing ist fast immer mit Elastizität verbunden, sei es beim Hoch-, sei es beim Runterskalieren von Hard- oder Software. Nicht umsonst ist das Versprechen von Cloud-Computing schon seit jeher: „Pay only what you use" – der Traum eines jeden Unternehmers. Das gleiche Versprechen gilt auch für die Betreiber der Software. „Scale up as you go", lautet hier häufig das Versprechen.

Nur ein wichtiger Faktor wird in der Gleichung weitestgehend außen vorgelassen, nämlich der, der die Elastizität überhaupt möglich macht – der Entwickler der Software. Denn Cloud-Computing kann seine vollen Vorteile nicht entfachen, wenn die Software es nicht zulässt.

Gerade für die Entwickler ist das Ziel, wenn sie Applikationen entwickeln, die „für die Cloud gemacht sind", vor allem eines: eine möglichst hohe Skalierbarkeit der Software zu gewährleisten. Sprich: Wenn viele Nutzer die Applikation gleichzeitig nutzen, können schnell neue Instanzen hochgefahren werden. Nimmt die Last wieder ab, können die Instanzen wieder freigelassen werden. Gehen Entwickler in die Cloud, so bedeutet das nicht selten, dass gewisse Skalierungskonzepte der Infrastrukturebene bereits Eingang in die eigene Systemarchitektur finden müssen: das strikte Trennen und Denken in zustandsbehafteten und zustandslosen Systemen. Denn nur durch diese Trennung kann das erhoffte und versprochene horizontale Skalieren der Cloud Realität werden.

Doch was bedeutet es, wenn wir sagen: „Wir denken in zustandsbehafteten und zustandslosen Systemen?" Betrachten wir zunächst ein zustandsloses Beispiel. Klassische, formularbasierte Geschäftsanwendungen basieren meistens auf dem Request-Response-Modell des HTTP-Protokolls. Sendet der Browser des Benutzers also Daten, z. B. das Anlegen einer Entität zum Server, so verbindet sich dieser wiederum mit einer Datenbank und speichert die Daten entsprechend. Je nachdem, ob die Anfrage erfolgreich war oder nicht, sendet der Server eine Antwort. Welcher Server genau diese Anfrage beantwortet hat, ist dabei zunächst egal. Das gilt allerdings nur, wenn alle benötigten Informationen für die Anfrage auch zustandslos sind. Das ist ein Grund, warum tokenbasierte Authentifizierungsmethoden, die von JSON Web Tokens (JWT) Gebrauch machen, so großen Anklang finden. Denn diese Methoden machen die Server zustandslos, indem sie den Zustand (das Token) auf den Client verlagern und auch dort speichern. Genau dieses zustandslose Modell erlaubt das Skalieren der Server auf eine oder eben zwanzig Instanzen.

Anders sieht es aus, wenn der Client auf die Verbindung zu einem bestimmten Server angewiesen ist, er also zustandsbehaftet ist. Ein Paradebeispiel für zustandsbehaftete Anwendungen sind Anwendungen, die WebSockets nutzen bzw. das System, das oft damit verbunden ist. Häufig sind solche Systeme mit einem Presence-System gekoppelt. Ein Presence-System ist ein Feature einer Software, das einigen aus Chatsystemen bekannt ist. Dabei ist ein Presence-System dafür zuständig, Informationen zu spei-

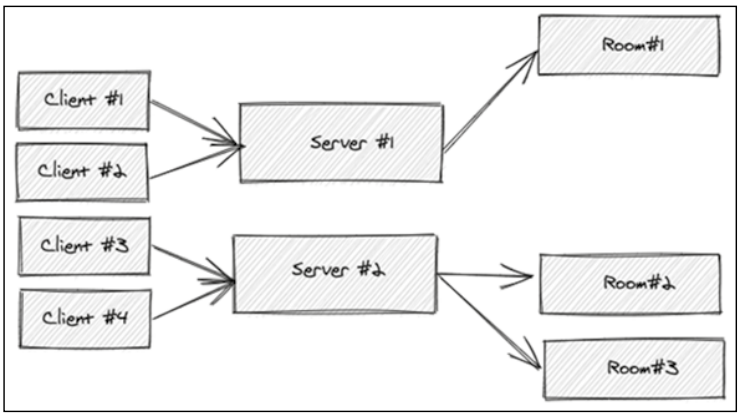


Abb. 1: Zustandsbehaftetes Presence-System

chern, z. B. darüber, in welchen virtuellen Räumen sich Benutzer befinden oder ob diese online oder offline sind. Diese Information muss anschließend in diesem System abruf- und aktualisierbar sein. Die einfachste Möglichkeit, ein solches System umzusetzen, wird in **Abbildung 1** gezeigt. Die Implementierung eines Presence-Servers, der die Verbindungen der WebSockets pflegt, würde dazu zunächst die Daten der Räume in-Memory selbst pflegen und dabei in den jeweiligen Lebenszyklus-Methoden *onConnect* und *onDisconnect* Benutzer hinzufügen und wieder löschen, sofern ein Client die Verbindung verliert oder abbricht. Das birgt aber Probleme: Es ist lediglich eine einzelne Serverinstanz für die komplette Untermenge an Räumen zuständig. Das bedeutet, dass sich alle Clients immer genau mit dieser einen Instanz verbinden müssen, da andere Instanzen keine Chance hätten, über die existierenden Räume informiert zu werden.

Deshalb setzen solche Systeme häufig auf Sticky Sessions. Sticky Sessions sind eine Möglichkeit, bereits auf der Clientseite zu bestimmen, welcher Server die Anfrage eines Clients entgegennehmen soll. Diese werden beispielsweise durch *query*-Parameter oder Cookies abgebildet. Angewandt auf das System in der **Abbildung 1** bedeutet das, dass sich die Clients jeweils mit einem konkreten Server verbinden. Dazu muss das System einen Routing-Key bereitstellen. Dieser könnte für diesen Anwendungsfall z. B. die ID des Raums sein. Das System würde dann garantieren, dass alle Clients desselben Raumes mit dem gleichen Server verbunden sind. Beispielsweise würden sich Client #1 und Client #2, die sich im Raum #1 befinden immer mit dem Server #1 verbinden.

Ein solcher Ansatz löst zwar das Problem des gemeinsamen Zustands (der durch Räume repräsentiert wird), führt aber das Problem der Load Imbalance ein. Gibt es Räume, die eventuell besonders viele Verbindungen pflegen, so sind diese Serverinstanzen eher überlastet, während die Instanzen mit wenig verbundenen Nutzern Rechenkraft verschwenden. Zusätzlich besteht ein weiteres Problem: Fällt diese Instanz weg, sind alle Räume der kompletten Instanz mit ihren Nutzern verloren. Außerdem gilt gerade im Cloud-Umfeld: Alles ist jederzeit ersetzbar. Im Umfeld von Kubernetes gibt es beispielsweise das Konzept der Rolling Deployments. Hier werden bei einem neuen Deployment alte Pods (= Instanzen des Presence-Systems) komplett heruntergefahren, während gleichzeitig neue hochgefahren werden. Es würden also mit jedem Deployment alle aktiven Räume unserer Applikation verloren gehen. Ein Problem, wenn man bedenkt, dass mehrere Deployments am Tag keine Seltenheit sind. Auch ist es mit diesem Ansatz sehr schwer, Funktionen umzusetzen, die die Informationen über die Gesamtheit der Räume kennen müssen.

Betrachtet man das Problem aus einer abstrakten Perspektive, dann wird klar, dass es sich bei einem Presence-System eigentlich um eine zustandsvolle Applikation, quasi eine Echtzeitdatenbank, handelt. Und solche sollten von den zustandslosen Applikationen getrennt und separat skaliert werden. Das ist ein Grund, warum die aus dem Node.js-Umfeld bekannte Bibliothek socket.io [1] einen Adapter für Redis und das im Java-Umfeld bekannte Spring Framework für die WebSocket-Verbindungen ein RabbitMQ-Relay [2] bereitstellt. Sowohl Redis als auch RabbitMQ sind auf Zustandshaltung und dessen Anwendungsfälle ausgelegt. Beide Systeme können beispielsweise in einem High-Availability-Modus ausgeführt werden. Das heißt, es ist möglich, dass beide ihre Daten auf verschiedenen Knoten replizieren, um Datensicherheit, auch bei Ausfällen einzelner Knoten, zu garantieren. Faktoren, die der Ansatz aus **Abbildung 1**

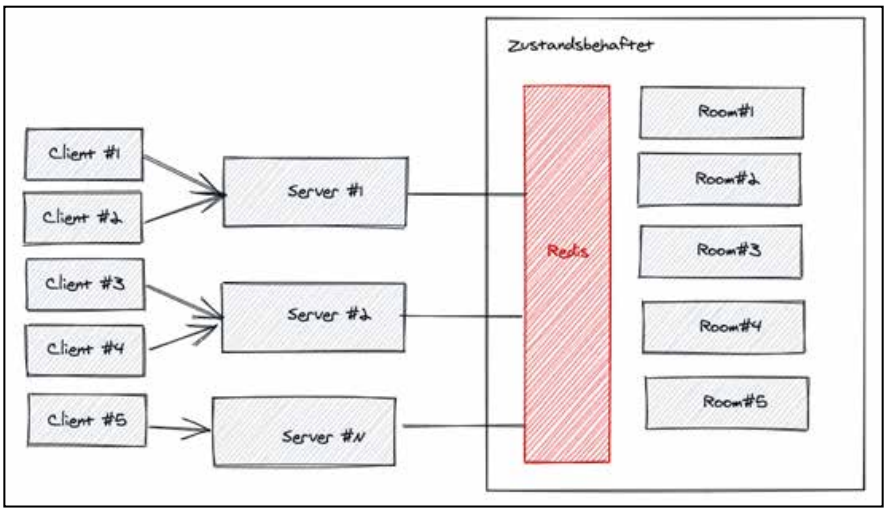


Abb. 2: Ein skalierbares Presence-System

bis dato nicht berücksichtigt hat. Auch ist das Skalieren von Redis und RabbitMQ mit den entsprechenden Plugins wesentlich einfacher als die Eigenentwicklung eines eigenen Systems. Sowohl RabbitMQ als auch Redis können als Message Queue fungieren, die die Cross-Instanz-Kommunikation übernimmt.

Zur Lösung der angeführten Probleme des Presence-Systems bedeutet das also, dass ein zusätzliches System benötigt wird, das die reine Zustandshaltung der Räume übernimmt. Eine Architektur wie in **Abbildung 2** ist deshalb förderlich.

Statt die Daten und Zustände der Nutzer in den Räumen wie zuvor von einzelnen Instanzen zu verwalten, ist es sinnvoll, diese Aufgabe an ein weiteres System, wie z. B. Redis, zu delegieren. Redis fungiert als Datenbank und Vermittler zwischen den Instanzen. Alle Instanzen der entwickelten Presence-System-Applikation registrieren sich dazu am externen Redis-System. Da es sich bei Redis um eine Datenbank handelt, gibt es standardmäßig Skalierungsfunktionalitäten wie asynchrone Replikation, Clustering und Sharding gratis dazu. Dadurch wird vor allem erreicht, dass das Presence-System komplett von der Datenhaltung losgelöst ist und ganz anders skalieren kann. Durch die neue Architektur kann sich jeder Client mit jedem beliebigen Server verbinden und wäre voll funktionsfähig. Natürlich bringt auch das Implikationen mit sich. Dadurch, dass ein weiteres, drittes System eingeführt wird, gibt es natürlich eine weitere Systemgrenze und Systemgrenzen müssen fast immer versioniert werden. Wird also z. B. zwischen zwei Versionen ein Attribut eines Raumes hinzugefügt, geändert oder gelöscht, so müssen die neuen Deployments immer eine Art Migrationspfad bzw. Abwärtskompatibilität bereitstellen, um einen Parallelbetrieb mit alten Versionen zu gewährleisten. Das ist in der Entwicklung natürlich teurer als das bloße Neustarten einer Applikation im Betrieb.

Eigentlich hat der Kunde nur mal eben schnell ein simples Echtzeitsystem gefordert und heraus kam ein relativ komplexes System, was dem zustandslosen Deployment geschuldet war. Das hat mich zu der Erkenntnis gebracht: Hinter Cloud-Computing steckt häufig mehr als das bloße Anmieten von Infrastruktur – obwohl das definitiv das Hauptaugenmerk ist. Sehr viel mehr aber bewegt die Entwicklung von Applikationen, die vorrangig „für die Cloud“ entwickelt werden, den Entwickler zum Denken in verteilten, skalierbaren Systemen.

Und das bringt neue Herausforderungen für die Entwickler. Dabei müssen gerade diese darauf achten, ihre Systeme möglichst zustandslos zu implementieren und durch geeignete zustandsbehaftete Systeme zu ergänzen. Denn gerade das Skalieren und Warten von zustandsbehafteten Systemen sind eine extrem komplexe Aufgabe. Sharding, Replikations- sowie High-Availability-Mechanismen sind Problemfelder, über die sich bereits viele Personen die Köpfe zerbrochen haben und deren Wissens man sich bedienen sollte. Natürlich heißt das nicht, dass solche Systeme immer so komplex entwickelt werden müssen und die „einfache Lösung“ nicht ausreicht. Entwickler sollten sich meiner Meinung nach aber der Implikationen bewusst sein, was es heißt, wenn ein System zustandsbehaftet ist.

Lars Kölpin-Freese ist Enterprise-Entwickler bei der OPEN KNOWLEDGE GmbH in Oldenburg. Zu seinen Leidenschaften gehören moderne Frontend-Technologien, vor allem im Zusammenspiel mit modernen Cloud-Architekturen.

Links & Literatur

[1] http://socket.io

[2] https://www.rabbitmq.com

Mit Module Federation und Web Components Angular und React gemeinsam nutzen

Architekturvielfalt durch Microfrontends

Module Federation erlaubt das Laden separat bereitgestellter Microfrontends und das Teilen von Abhängigkeiten. Web Components erlauben zusätzlich den parallelen Einsatz verschiedener Frameworks und Versionen. Die Kombination von beidem erhöht jedoch auch die Bundle Size und macht einige Workarounds notwendig.

von Manfred Steyer

Microfrontends bringen – zumindest theoretisch – die Vorteile von Microservices in den Client: Einzelne autarke Teams entwickeln und deployen ihre Codestrecken unabhängig voneinander und treffen auch ihre eigenen Entscheidungen. Das geht sogar so weit, dass jedes Team über die zu nutzenden Technologien, darunter Bibliotheken und Frameworks sowie deren Versionen, selbst entscheidet. Allerdings führt die Nutzung unterschiedlicher Frameworks und Versionen im Frontend zu Herausforderungen: Zum einen müssen sie in den Browser geladen werden und erhöhen somit die Anzahl an Bytes, die über die Leitung gehen. Zum anderen müssen die verschiedenen Frameworks und Versionen, die eigentlich gar nichts voneinander wissen, zum Zusammenspiel bewegt werden. Es gibt allerdings eine Lösung für dieses Problem, indem man die Module Federation mit Web Components kombiniert. Letztere basieren auf unterschiedlichen Angular-Versionen und auf React. Welche Konsequenzen dieser Ansatz mit sich bringt, beschreibe ich am Ende dieses Artikels. Der Quellcode des dazu verwendeten Beispiels findet sich unter [1].

Das Beispiel zur Ausgangssituation

Um zu veranschaulichen, wie sich mit der Kombination von Module Federation und Web Components verschiedene Frameworks, bzw. verschiedene Framework-Versionen, kombinieren lassen, kommt die in **Abbildung 1** gezeigte Anwendung zum Einsatz.

Dieser Screenshot lässt schon erkennen, dass die Beispielanwendung sowohl Angular als auch React kombiniert. Tatsächlich kombiniert sie aber ebenfalls noch zwei verschiedene Angular-Versionen (**Abb. 2**).

Interessant hieran ist auch, dass sich jeweils zwei Anwendungen eine Angular-Version teilen. Dies lässt sich, wie weiter unten beschrieben, mit dem Standardverhalten von Module Federation erreichen. Damit die einzelnen Frameworks und die unterschiedlichen Framework-Versionen zusammenspielen, wurden sie als Web Components veröffentlicht.

Web Components mit Module Federation bereitstellen

Das vom Angular-Team entwickelte Projekt @angular/elements erlaubt das Bereitstellen von Angular-Komponenten als Framework-unabhängige Web Components. Da auch eine Angular-Anwendung lediglich eine Komponente ist, die wiederum aus zahlreichen weiteren Komponenten besteht, lassen sich damit auch ganze Anwendungen in Web Components umwandeln. Um @angular/elements zu nutzen, ist das gleichnamige Paket zu installieren:

```
ng add @angular/elements
```

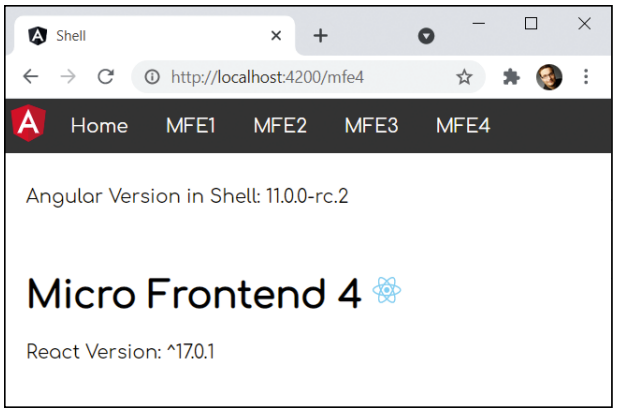


Abb. 1: Beispiel mit mehreren Frameworks und Versionen

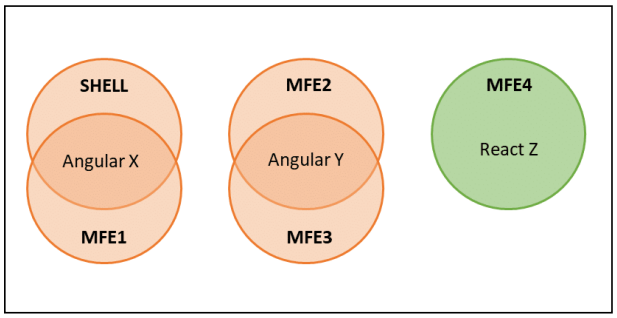


Abb. 2: Teilen von Frameworks und Versionen

Listing 1

```
[...]
import { createCustomElement } from '@angular/elements';
[...]

@NgModule({
  [...]
  declarations: [ [...], AppComponent ],
  bootstrap: [] // No bootstrap components!
})
export class AppModule {
  constructor(private injector: Injector) { }

  ngDoBootstrap() {
    const ce = createCustomElement(AppComponent, {injector: this.injector});
    customElements.define('mfe1-element', ce);
  }
}
```

Listing 2

```
new ModuleFederationPlugin({
  name: "mfe1",
  filename: "remoteEntry.js",
  exposes: {
    './web-components': './src/bootstrap.ts',
  },
  shared: ["@angular/core", "@angular/common", "@angular/router"]
})
```

Anschließend können Sie seine Methode *createCustomElement* nutzen, um eine Angular-Komponente als Web Component zu verpacken. Listing 1 übernimmt diese Aufgabe in der Methode *ngDoBootstrap* des *AppModules*.

Angular ruft *ngDoBootstrap* auf, wenn das *AppModule* keine Bootstrap-Komponente aufweist. In diesem Fall sind die Autor:innen selbst für die Bootstrapping-Logik verantwortlich. Neben der Angular-Komponente bekommt *createCustomElement* auch den aktuellen Injector übergeben, um die Komponente mit dem Dependency-Injection-Mechanismus von Angular zu verbinden. Die Methode *customElements*.define wird durch den zugrunde liegenden Browserstandard definiert und registriert die Web Component unter dem Elementnamen *mfe1-element*. Insofern lässt sie sich nun mit jedem beliebigen Framework mittels *<mfe1-element></mfe1-element>* aufrufen. Inputs und Outputs bildet Angular Elements übrigens auf gleichnamige Eigenschaften und Events ab.

Um diese Web Component nun der Shell über Module Federation anzubieten, müssen wir das Paket @angular-architects/module-federation [2] installieren:

```
ng add @angular-architects/module-federation
```

Dieser Aufruf generiert unter anderem eine *webpack.config.js* mit der Konfiguration für Module Federation. Mit dieser Konfiguration wird uns das Veröffentlichen von Dateien für die Shell erlaubt. Listing 2 stellt zum Beispiel unter dem relativen Pfad *./web-components* den Inhalt der Datei *./src/bootstrap.ts* bereit.

Die Datei *bootstrap.ts* wurde vom Paket @angular-architects/module-federation generiert. Es kümmert sich um das Bootstrapping des *AppModule*. Da wir im *AppModule* auch die Web Component veröffentlichen, bekommt die Shell darauf Zugriff. Normalerweise platziert das CLI den Quellcode für Boostrapping in der Datei *main.ts*. Aus technischen Gründen verschiebt *@angular-architects/module-federation* ihren Inhalt jedoch nach *bootstrap.ts*.

Außerdem definiert die gezeigte Konfiguration den Namen des Microfrontends (*name*) sowie den Namen des von Module Federation generierten Remote Entry Points mit Metadaten zum Microfrontend (*filename*). Diese Datei gilt es in die Shell zu laden, sodass sie alle nötigen Informationen zur Nutzung des jeweiligen Microfrontends hat.

Unter *shared* finden sich jene npm-Pakete, die es mit der Shell und anderen Microfrontends zu teilen gilt. Bevor Module Federation eine Abhängigkeit teilt, prüft es, ob die angebotenen und benötigten Versionen kompatibel zueinander sind. Dazu verwendet es die Einträge aus der Datei *package.json*. Sind sie nicht zueinander kompatibel, lädt Module Federation standardmäßig beide Versionen. Durch dieses Standardverhalten ergibt sich die in **Abbildung 2** dargestellte Situation.

React und Web Components

Im Gegensatz zu Angular bietet React leider keine offizielle Unterstützung für Web Components. Allerdings finden sich Communityprojekte, die sich dieser Aufgabe angenommen haben. Als Alternative können Sie auch manuell eine native Web Component erstellen und darin eine React-Komponente rendern (Listing 3).

Diese Web Component lässt sich mit Module Federation veröffentlichen. Im Fall von React können Sie jedoch die *webpack.config.js* direkt um die gezeigten Einträge erweitern. Ein Paket für den Brückenschlag wie @angular-architects/module-federation ist hier also nicht notwendig.

Listing 3

```
// Native Web Component extending Browser's HTMLElement
class Mfe4Element extends HTMLElement {
  connectedCallback() {
    ReactDOM.render(<App/>, this);
  }
}

customElements.define('mfe4-element', Mfe4Element);
```

Listing 4

```
[...]
new ModuleFederationPlugin({
  remotes: {
    'mfe1': "mfe1@http://localhost:4201/remoteEntry.js",
    [...]
  },
  shared: ["@angular/core", "@angular/common", "@angular/router"]
})
[...]
```

Listing 5

```
RouterModule.forRoot([
  { [...], component: WrapperComponent,
    data: { importName: 'mfe1', elementName: 'mfe1-element' }},
  { [...], component: WrapperComponent,
    data: { importName: 'mfe2', elementName: 'mfe2-element' }},
  [...]
])
```

Dynamic Federation

Falls Sie das Registrieren der Microfrontends in der *webpack.config.js* zu sehr einschränkt, können Sie auch auf Dynamic Federation [3] setzen. In diesem Fall können Sie auf den Abschnitt *remote* verzichten und geben stattdessen die Eckdaten für das Laden des Microfrontends zur Laufzeit bekannt. Das Paket @angular-architects/module-federation bietet dafür die Methode *loadRemoteModule* an:

```
await loadRemoteModule({
  remoteEntry: 'http://localhost:4201/remoteEntry.js',
  remoteName: 'mfe1',
  exposedModule: './web-components
})
```

Web Components mit Module Federation laden

Da im hier präsentierten Beispiel auch die Shell auf Angular basiert, bekommt sie das Paket @angular-architects/module-federation installiert. In ihrer *webpack.config.js* sind nun die einzelnen Microfrontends als *remotes* zu registrieren (Listing 4).

Diese Einträge bilden Pfade (hier *mfe1*) auf die Namen der veröffentlichten Microfrontends (hier ebenfalls *mfe1*) und deren Remote Entry Points (hier *http://localhost:4201/remoteEntry.js*) ab. Außerdem sind auch hier die zu teilenden Bibliotheken zu hinterlegen. Anschließend kann die Shell mit einem dynamischen *import* die Microfrontends laden:

```
await import('mfe1/web-components');
```

Die davon registrierten Web Components lassen sich danach genauso wie andere HTML-Elemente behandeln und dynamisch in die Seite einfügen:

```
const element = document.createElement('mfe1-element');
document.body.appendChild(element);
```

Um mit dem Low-Level-Code für das Laden und Instanziieren von Web Components nicht ständig konfrontiert zu werden, versteckt das hier besprochene Beispiel ihn in einer Wrapper-Komponente. Dabei handelt es sich um eine Angular-Komponente, die die nötigen Eckdaten über die Routenkonfiguration erhält (Listing 5).

Dieser Wrapper ist auch notwendig, weil der Angular-Router nur Angular-Komponenten und keine Web Components aktivieren kann.

Bewertung

Die bisher beschriebene Lösung erfüllt die eingangs festgelegten Anforderungen. Sie kommt jedoch nicht ohne Nachteile. Deswegen möchte ich den restlichen Artikel zur Bewertung der Vor- und Nachteile dieses Ansatzes nutzen.

Laden von Microfrontends

Das Laden von Microfrontends gestaltet sich sehr einfach. Dank Dynamic Federation kann sich die Shell zur Laufzeit über verfügbare Microfrontends informieren und sie bei Bedarf laden. Das Schöne dabei ist, dass Angular hiervon gar nichts mitbekommt. Aus der Sicht unseres führenden Frameworks findet hier lediglich Lazy Loading statt. Wir können es also prinzipiell so nutzen, wie es gedacht ist, und kommen ohne komplizierte zusätzliche Meta-Frameworks aus.

Teilen von Abhängigkeiten und Bundle Size

Das Teilen von Abhängigkeiten zwischen separat kompilierten Microfrontends ist wohl eines der mächtigsten Möglichkeiten von Module Federation. Sofern zwei oder mehr Microfrontends eine kompatible Version derselben Abhängigkeit verwenden, lädt Module Federation diese nur einmal. Damit Module Federation weiß, welches Microfrontend welche Abhängigkeiten und Versionen benötigt, sind vorab deren Remote Entry Points in die Shell zu laden. Die zu teilenden Abhängigkeiten können jedoch nicht mit Tree Shaking optimiert werden. Diese Technik bedeutet ja, dass der Build-Prozess sämtliche nicht benötigten Bestandteile einer Bibliothek entfernt. Allerdings weiß der Build-Prozess nicht, wie andere erst zur Laufzeit geladene Microfrontends eine geteilte Abhängigkeit nutzen. Deswegen deaktiviert Module Federation für sämtliche geteilte Abhängigkeiten das Tree Shaking. Dazu kommt, dass gerade in unserem Szenario auch mehrere Frameworks und Versionen geladen werden müssen. Das erhöht zusätzlich die Anzahl an Bytes, die über die Leitung wandern müssen. Inwieweit das kritisch ist, gilt es von Fall zu Fall abzuschätzen. Während die Bundle Size bei öffentlichen Webauftritten maßgeblich den Erfolg beeinflussen kann, ist sie bei internen Anwendungen häufig weniger wichtig.

Web Components

Dank Web Components lassen sich die einzelnen Frameworks und Versionen voreinander verbergen. Web Components funktionieren auch mittlerweile in allen modernen Browsern. Wer noch Internet Explorer 11 unterstützen muss, kann sich mit Polyfills behelfen. Die sind zwar nicht perfekt, man kann sich damit jedoch arrangieren. Da wir Web Components aber nicht direkt mit dem Router nutzen können, benötigen wir eine generische Wrapper-Komponente. Diese verbirgt auch die nicht so schönen technischen Details des dynamischen Erzeugens von Web Components.

Zusammenspiel verschiedener Router

Router gehen in der Regel davon aus, dass sie allein über die Seite bestimmen können. Da wir nun mehrere Single Page Applications in den Browser laden, müssen wir deren Router zum Zusammenspiel bewegen. Dazu sind ein paar Workarounds notwendig. Beispielsweise müssen wir nun die Router wissen lassen, welcher Teil des URLs für sie bestimmt ist. Nehmen wir als Beispiel den URL */mfe1/a*. Das erste Segment, *mfe1*, signalisiert dem Router der Shell, dass das Microfrontend 1 zu laden ist. Das zweite Segment, */a*, teilt hingegen dem Router von Microfrontend 1 mit, welche Route er aktivieren soll. Die hier verwendete Lösung, die sich unter [1] findet, verwendet statt Pfaden sogenannte UrlMatcher. Dabei handelt es sich um Funktionen, die dem Router sagen, ob eine bestimmte Route zu aktivieren ist. Der Matcher *startsWith* legt beispielsweise fest, dass eine Route aktiviert wird, wenn der URL mit einem bestimmten Segment beginnt:

```
{ matcher: startsWith('mfe1'), component: WrapperComponent, data: {
                importName: 'mfe1', elementName: 'mfe1-element' }}
```

Alternativ dazu bestimmt ein *endsWith('a')*, dass eine Route dann zu aktivieren ist, wenn der URL mit dem Segment */a* endet.

Abgesehen davon kommt es vor, dass sich ein Angular-Router in einer verzögert geladenen Web Component gar nicht für den aktuellen URL zuständig fühlt, zumal Router in der Regel beim Programmstart geladen werden. In diesem Fall benötigt dieser Router eine Sondereinladung. Diese ist bei jeder URL-Änderung auszustellen. Hierzu kann die Anwendung auf das *popstate* Event horchen und das Routing mit der Methode *navigateByUrl* des Angular-Routers anfordern. Wie bei allen Workarounds empfiehlt es sich, diese hinter ein paar Hilfskonstrukten zu verstecken, damit Entwickler:innen nicht ständig damit konfrontiert sind.

Workarounds für Angular und Zone.js

Gerade Angular bringt noch zwei weitere Herausforderungen mit sich. Eine besteht in der Tatsache, dass Angular-Anwendungen jeweils für eine Plattform gestartet werden. Die wohl häufigste Plattform führt Angular im Browser aus. Für Server-side Rendering existiert eine weitere Plattform. Leider darf jede Plattform nur einmal instanziiert werden. Wenn jedoch eine Angular-Version von mehreren Microfrontends geteilt wird, würden diese mit dem standardmäßig vom CLI generierten Code mehrere Instanzen davon erzeugen. Ein ähnliches Problem ergibt sich beim Einsatz von Zone.js. Diese Abhängigkeit von Angular sollte auch nur ein einziges Mal instanziiert werden, um Probleme bei der Change Detection zu vermeiden.

Die Lösung ist für beide Fälle dieselbe: Wir müssen die erzeugten Plattform- sowie Zone.js-Instanzen anderen Microfrontends über den globalen Namensraum zur Verfügung stellen. Den hierzu notwendigen Workaround implementiert die hier diskutierte Lösung [1] in den Dateien *bootstrap.ts* und *app.component.ts*.

Fazit

Module Federations erlauben das Laden separat bereitgestellter Microfrontends zur Laufzeit. Es erlaubt auch das Teilen von Abhängigkeiten und bringt Strategien zum Umgang mit Konflikten mit sich. Das alles gestaltet sich für uns Entwickler:innen äußerst geradlinig. Aus Sicht der Frameworks wie Angular oder React gestaltet sich alles wie normales Lazy Loading. Das macht den Einsatz komplexer zusätzlicher Meta-Frameworks unnötig.

Durch das Hinzuziehen von Web Components lassen sich auch unterschiedliche Frameworks und Versionen voreinander verbergen. Das erhöht natürlich die Bundle Size und kann sich somit auf die (Start-)Performance der Anwendung auswirken. Außerdem macht dieses Vorgehen einige Workarounds notwendig. Das erhöht wiederum die Komplexität der Lösung.

Eine Selbstbeschränkung auf eine Major-Version des führenden Frameworks – sofern das möglich ist – sollte Vorteile bringen. Ist das aber im gegebenen Fall zu restriktiv, können sich Teams mit den hier aufgezeigten Maßnahmen mehr Freiheiten „erkaufen“. Wie immer im Bereich der Softwarearchitektur gilt es somit auch hier, die vorherrschenden Architekturanforderungen zu kennen und die möglichen Ansätze vor deren Hintergrund zu bewerten.

Manfred Steyer ist Trainer und Berater mit Fokus auf Angular, Google Developer Expert und Trusted Collaborator im Angular-Team. Er schreibt für O'Reilly, das deutsche Java Magazin und Heise Developer. Unter www.ANGULARarchitects.io bieten er und sein Team Angular-Schulungen und Beratung in Deutschland, Österreich und der Schweiz an.

Links & Literatur

[1] https://github.com/manfredsteyer/multi-framework-microfrontend

[2] https://www.npmjs.com/package/@angular-architects/module-federation

[3] https://www.angulararchitects.io/aktuelles/dynamic-module-federation-with-angular/

Einsatz in einer medizinischen Java-Anwendung

Neuroph und DL4J

In diesem Artikel möchten wir zeigen, wie neuronale Netze, speziell das Multilayer Perzeptron zweier Java Frameworks, für die Erkennung von Blutkörperchen in Bildern verwendet werden können.

von Dr. Valentin Steinhauer und Dr. Larissa Steinhauer

Bei mikroskopischen Blutbilduntersuchungen erfolgt unter anderem eine Analyse der sechs Typen von weißen Blutkörperchen. Zu diesen zählen: Neutrophile, Eosinophile, Basophile Granulozyten, Monozyten und Lymphozyten. Anhand der Anzahl, Reife und Verteilung dieser weißen Blutkörperchen erhält man wertvolle Hinweise auf mögliche Erkrankungen. Hier werden wir uns jedoch nicht auf die Handhabung der Blutausstriche, sondern auf die Erkennung der Blutkörperchen konzentrieren.

Für die beschriebenen Tests wurde das Bresser-Trino-Mikroskop mit einem Mikrookular verwendet und mit einem Computer (HP Z600) verbunden. Zur Bildanalyse wurde das von uns in diesem Artikel vorgestellte Programm verwendet. Die Software basiert auf neuronalen Netzen unter Verwendung der Java Frameworks Neuroph [1] und Deep Learning for Java (DL4J) [2]. Die Färbung der Ausstriche für das Mikroskop wurden mit Löffler-Lösung gemacht.

Trainingsdaten

Für das Training der neuronalen Netze wurden die Bilder (Images) der Blutkörperchen zentriert, in das Format gray scala umgewandelt und normalisiert. Nach der Vorbereitung sahen die Bilder aus, wie in **Abbildung 1** gezeigt.

Für das Training wurde ein Datensatz von 663 Bildern mit 6 Labels – *ly*, *bg*, *eog*, *mo*, *sng*, *seg* – zusammengestellt. Für Neuroph wurden die in Listing 2 gezeigten *imageLabel* gesetzt.

Danach sieht das Verzeichnis für die Inputdaten aus wie in **Abbildung 2**.

Für DL4J setzt sich das Verzeichnis für die Inputdaten (your data location) anders zusammen (**Abb. 3**).

Listing 1

```
List<String> imageLabels = new ArrayList();
  imageLabels.add("ly");
  imageLabels.add("bg");
  imageLabels.add("eog");
  imageLabels.add("mo");
  imageLabels.add("sng");
  imageLabels.add("seg");
```

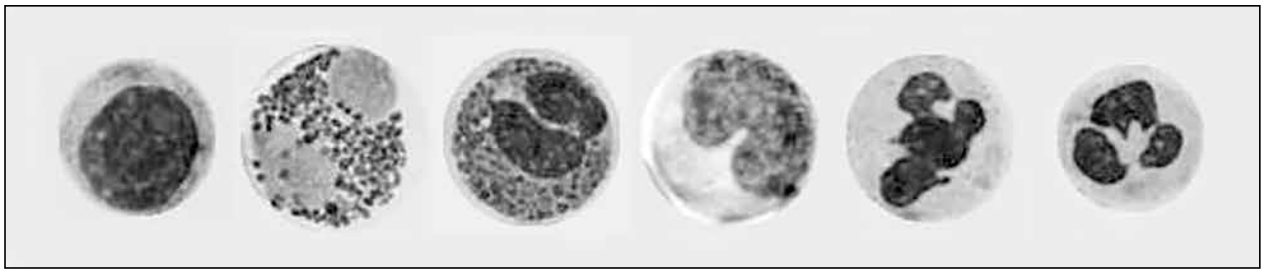


Abb. 1: Die JPG-Bilder weisen eine Größe von 100 x 100 Pixel und zeigen (von links nach rechts) Lymphozyt (ly), Basophile (bg), Eosinophile (eog), Monozyt (mo), stabkerniger (junger) Neutrophil (sng), segmentkerniger (reifer) Neutrophile (seg); die Zelltypen wurden für das Training der neuronalen Netze verwendet

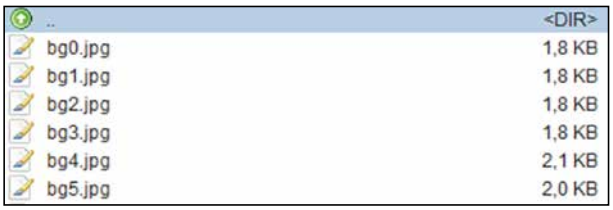


Abb. 2: Das Verzeichnis für die Inputdaten

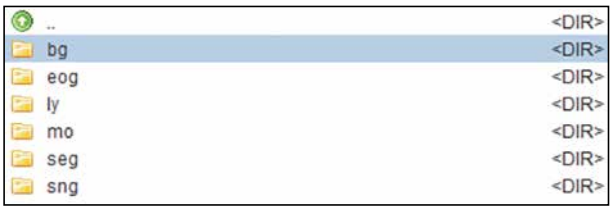


Abb. 3: Verzeichnis für die Inputdaten für DL4J

Listing 2

```
<dependency>
  <groupId>org.neuroph</groupId>
  <artifactId>neuroph-core</artifactId>
  <version>2.96</version>
</dependency>
<dependency>
  <groupId>org.neuroph</groupId>
  <artifactId>neuroph-imgrec</artifactId>
  <version>2.96</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

Listing 3

```
private static final double LEARNINGRATE = 0.05;
private static final double MAXERROR = 0.05;
private static final int HIDDENLAYERS = 13;

//Offenes Netz
Map<String, FractionRgbData> map;
 try {
   map = ImageRecognitionHelper.getFractionRgbDataForDirectory(new
                                    File(imageDir), new Dimension(10, 10));
   dataSet = ImageRecognitionHelper.createRGBTrainingSet(image-
                                                        Labes, map);
   // create neural network
   List<Integer> hiddenLayers = new ArrayList<>();
   hiddenLayers.add(HIDDENLAYERS);/
   NeuralNetwork nnet = ImageRecognitionHelper.createNewNeuralNet-
            work("leukos", new Dimension(10, 10), ColorMode.COLOR_RGB,
              imageLabels, hiddenLayers, TransferFunctionType.SIGMOID);
   // set learning rule parameters
   BackPropagation mb = (BackPropagation) nnet.getLearningRule();
   mb.setLearningRate(LEARNINGRATE);
   mb.setMaxError(MAXERROR);
   nnet.save("leukos.net");
 } catch (IOException ex) {
   Logger.getLogger(Neuroph.class.getName()).log(Level.SEVERE, null, ex);
 }
```

Die meisten Bilder im Datensatz stammen aus eigenen Aufnahmen. Es gab aber auch Bilder von offenen und freien Internetquellen. Zudem enthielt der Datensatz die Bilder mehrfach, da diese jeweils auch um 90, 180 und 270 Grad gedreht und gespeichert wurden.

Neuroph-MLP-Netz

Die wichtigsten Abhängigkeiten für das Neuroph-Projekt in *pom.xml* sind die in Listing 2 gezeigten.

Listing 4

```
HashMap<String, Double> output;
String fileName = "leukos112.seg";
NeuralNetwork nnetTest = NeuralNetwork.createFromFile("leukos.net");
// get the image recognition plugin from neural network
ImageRecognitionPlugin imageRecognition = (ImageRecognitionPlugin)
                        nnetTest.getPlugin(ImageRecognitionPlugin.class);
output = imageRecognition.recognizeImage(new File(fileName);
```

Listing 5

```
<dependency>
  <groupId>org.deeplearning4j</groupId>
  <artifactId>deeplearning4j-core</artifactId>
  <version>1.0.0-beta4</version>
</dependency>
<dependency>
  <groupId>org.nd4j</groupId>
  <artifactId>nd4j-native-platform</artifactId>
  <version>1.0.0-beta4</version>
</dependency>
```

Listing 6

```
protected static int height = 100;
protected static int width = 100;
protected static int channels = 1;
protected static int batchSize = 20;

protected static long seed = 42;
protected static Random rng = new Random(seed);
protected static int epochs = 100;
protected static boolean save = true;
```

Listing 7

```
//DataSet
  String dataLocalPath = "your data location";
  ParentPathLabelGenerator labelMaker = new ParentPathLabelGenerator();
  File mainPath = new File(dataLocalPath);
  FileSplit fileSplit = new FileSplit(mainPath, NativeImageLoader.ALLOWED_FORMATS, rng);
  int numExamples = toIntExact(fileSplit.length());
  numLabels = Objects.requireNonNull(fileSplit.getRootDir().listFiles(File::isDirectory)).length;
  int maxPathsPerLabel = 18;
  BalancedPathFilter pathFilter = new BalancedPathFilter(rng, labelMaker, numExamples, numLabels,
                                                                          maxPathsPerLabel);
  //training – Test teilen
  double splitTrainTest = 0.8;
  InputSplit[] inputSplit = fileSplit.sample(pathFilter, splitTrainTest, 1 - splitTrainTest);
  InputSplit trainData = inputSplit[0];
  InputSplit testData = inputSplit[1];

//Offenes Netz
MultiLayerNetwork network = lenetModel();
network.init();
ImageRecordReader trainRR = new ImageRecordReader(height, width, channels, labelMaker);
trainRR.initialize(trainData, null);
DataSetIterator trainIter = new RecordReaderDataSetIterator(trainRR, batchSize, 1, numLabels);
  scaler.fit(trainIter);
  trainIter.setPreProcessor(scaler);
  network.fit(trainIter, epochs);
```

Listing 8

```
private MultiLayerNetwork lenetModel() {
  /*
    * Revisde Lenet Model approach developed by ramgo2 achieves slightly above random
    * Reference: https://gist.github.com/ramgo2/833f12e92359a2da9e5c2fb6333351c5
  */
  MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(seed)
    .l2(0.005)
    .activation(Activation.RELU)
    .weightInit(WeightInit.XAVIER)
    .updater(new AdaDelta())
    .list()
    .layer(0, convInit("cnn1", channels, 50, new int[]{5, 5}, new int[]{1, 1}, new int[]{0, 0}, 0))
    .layer(1, maxPool("maxpool1", new int[]{2, 2}))
    .layer(2, conv5x5("cnn2", 100, new int[]{5, 5}, new int[]{1, 1}, 0))
    .layer(3, maxPool("maxool2", new int[]{2, 2}))
    .layer(4, new DenseLayer.Builder().nOut(500).build())
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
      .nOut(numLabels)
      .activation(Activation.SOFTMAX)
      .build())
    .setInputType(InputType.convolutional(height, width, channels))
    .build();

  return new MultiLayerNetwork(conf);

}
```

Es wurde ein Multilayer-Perzeptron mit den in Listing 3 gezeigten Parametern gesetzt.

Beispiel

Die Implementierung eines Tests kann aussehen wie in Listing 4 gezeigt.

Client

Für die grafische Zellerkennung wurde eine einfache SWING-Oberfläche entwickelt. Ein Beispiel für die Erkennung eines Lymphozyten ist in **Abbildung 4** zu sehen.

DL4J-MLP-Netz

Die wichtigsten Abhängigkeiten für das DL4J-Projekt in *pom.xml* werden in Listing 5 gezeigt.

Auch hier wurde ein Multilayer-Perzeptron mit den in Listing 6 gezeigten Parametern [3] eingesetzt.

LeNet Model

Bei diesem Model [4] handelt es sich um eine Art vorwärts gerichtetes neuronales Netzwerk für Image Processing (Listing 8).

Beispiel

Ein Test des Lenet Models könnte wie in Listing 9 dargestellt aussehen.

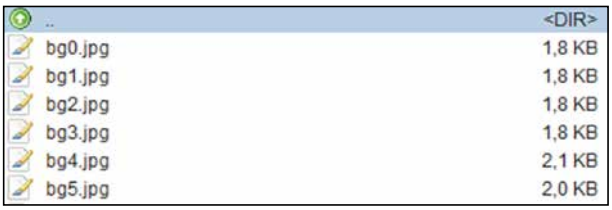


Abb. 4: Das Programm erkennt einen Lymphozyten und hebt ihn hervor

Ergebnisse

Nach einigen Testdurchläufen erhält man das in Tabelle 1 dargestellte Resultat.

Wie man sehen kann, sind die Ergebnisse bei der Verwendung von Neuroph etwas besser als die von DL4J. Man muss aber beachten, dass die Resultate sehr von der Qualität der Inputdaten und der Topologie des Netzes abhängen. Wir planen, dieses Thema zeitnah weiter zu untersuchen.

Mit diesen Ergebnissen konnten wir jedoch schon jetzt zeigen, dass die Bilderkennung mit gleich zwei fundierten und möglicherweise komplementären Java Frameworks für medizinische Zwecke eingesetzt werden kann.

Listing 9

```
trainIter.reset();
DataSet testDataSet = trainIter.next();
List<String> allClassLabels = trainRR.getLabels();
int labelIndex;
int[] predictedClasses;
String expectedResult;
String modelPrediction;
int n = allClassLabels.size();
System.out.println("n = " + n);
for (int i = 0; i < n; i = i + 1) {
  labelIndex = testDataSet.getLabels().argMax(1).getInt(i);
  System.out.println("labelIndex=" + labelIndex);
  INDArray ia = testDataSet.getFeatures();
  predictedClasses = network.predict(ia);
  expectedResult = allClassLabels.get(labelIndex);
  modelPrediction = allClassLabels.get(predictedClasses[i]);
  System.out.println("For a single example that is labeled " +
expectedResult + " the model predicted " + modelPrediction);
}
```

Leukos	Neuroph	DL4J
Lymphozyten (*ly*)	87	85
Basophilen (*bg*)	96	63
Eosinophilen (*eog*)	93	54
Monozyten (*mo*)	86	60
Stabkerniger Neutrophilen (*sng*)	71	46
Segmentkerniger Neutrophilen (*seg*)	92	81

Tabelle 1: Ergebnisse der Leukozytenzählung (N-Erfolg/N-Proben in %)

Danksagung

An dieser Stelle möchten wir uns bei Herrn A. Klinger (Geschäftsführung Devoteam GmbH Deutschland) und bei Frau M. Steinhauer (Bioinformatikerin) für ihre Unterstützung bedanken.

Dr. Valentin Steinhauer war bei Devoteam GmbH in Weiterstadt tätig. Er verfügt über mehrjährige Erfahrung aus Softwareprojekten als Coach, Trainer, Architekt, Teamleiter und Entwickler.

Dr. Larissa Steinhauer ist bei Devoteam GmbH in Weiterstadt tätig. Sie verfügt über mehrjährige Erfahrung aus Softwareprojekten als Coach, Testerin, Entwicklerin, Trainer, Projektleiterin.

Links & Literatur

[1] Neuroph: http://neuroph.sourceforge.net

[2] Deep Learning for Java – DL4J: https://deeplearning4j.org/

[3] Schnelleinstieg in Deeplearning4j – Teil 2: Training und Verwendung: https://jaxenter.de/deeplearning4j-teil-2-79884

[4] LeNet: https://en.wikipedia.org/wiki/LeNet